Vertica Documentation

Getting started

Launch Vertica on major cloud providers and on-premises. Test drive Vertica with the Community Edition, and review new features.

Setup
Quickstart
Vertica Community Edition (CE)
New features

Ecosystem

Integrate Vertica with your environment.

Supported platforms
Architecture
Security & authentication
Containerized Vertica
Client connections
Management Console

Working with data

Transform and analyze data with the power of Vertica.

Data exploration
Data load
Unstructured data
Data analysis
Machine learning
Export

Administration

Perform common database administrator tasks.

Configure the database
Users and privileges
Start and stop the database
Backup and restore
Upgrade
Projections
Analyze workloads

Customize Vertica

Add scripts, SQL functions, and external libraries that automate tasks and extend Vertica functionality.

Custom SQL functions
External procedures
Stored procedures
Install external libraries
Develop external libraries

Reference

Comprehensive references, including SQL elements and SDKs to extend Vertica.

SQL reference

SDK/API reference

Glossary

Published on 2025-08-21.

Supported platforms

Welcome to Vertica Analytics Platform Supported Platforms. This document describes platform support for the various components of Vertica 12.0.x.

In this section

- Vertica server and Management Console
- Client drivers support
- Vertica SDKs
- FIPS 140-2 supported platforms
- Eon on-premises storage
- Vertica on Amazon Web Services
- Containerized environments
- Virtualized environments
- Hadoop integrations
- Apache Kafka integrations
- Apache Spark integrations
- Linux volume manager (LVM)
- End-of-support notices

Vertica server and Management Console

Operating systems and versions

OpenText supports the Vertica Analytic Database 12.0.x running on the following 64-bit operating systems and versions on x86_x64 architecture.

In general, OpenText provides support for the Vertica Analytic Database, not its host operating system, hardware, or other environmental elements. However, OpenText makes an effort to ensure the success of its customers on recent versions of the following popular operating systems for the x86_64 architecture.

When there are multiple minor versions supported for a major operating system release, OpenText recommends that you run Vertica on the latest minor version listed in the supported versions list. For example, if you run Vertica on a Red Hat Enterprise Linux 7.x release, OpenText recommends you upgrade to or be running the latest supported RHEL 7.x release.

Platform	Processor	Supported Versions	Known Issues

Red Hat Enterprise Linux / CentOS Important Vertica does not support or recommend in-place upgrades from one major version to another major version. For example, you cannot perform in-place upgrades from RHEL/CentOS 7.x to RHEL/CentOS 8.x. For information on how to upgrade to Red Hat Enterprise Linux 7, see Upgrading your operating system on nodes in your Vertica cluster . For information on changes to the operating system for Red Hat Enterprise Linux 7, see the Red Hat Enterprise Linux 7 documentation . Vertica support for CentOS is based on testing done on Red Hat Enterprise Linux. Vertica continues to support CentOS, but testing and troubleshooting will be performed with the associated RedHat version.	x86_64	7.x: all with known issues 8.x: all with known issues Important 6.x is no longer supported .	**8.x** : There are some circumstances where you cannot create a cluster from Management Console due to an issue with the private key file. To create R extensions, manually install the libgfortran4 package. Download the applicable package from the CentOS Linux and Stream releases page .
SUSE Linux Enterprise Server	x86_64	12 SP2 and higher 15.x: all	
openSUSE	x86_64	42.3	
Amazon Linux	x86_64	2.0 on Amazon Machine Instances (AMIs)	
Oracle Enterprise Linux (Red Hat compatible kernels only)	x86_64	6.x: all 7.x: all	
Debian Linux	x86_64	8.5, 8.9 10.x: with known issues	**10.2** : On Vertica 9.3 and higher, you cannot restart the Management Console using the MC Interface in your browser. To restart the Management Console, enter one of the following commands: • systemctl restart vertica-consoled
Ubuntu	x86_64	14.04 LTS and higher with known issues	**16.x/18.x** : On Vertica 9.3 and higher, you cannot restart the Management Console using the MC Interface in your browser. To restart the Management Console, enter one of the following commands: • systemctl restart vertica-consoled • /etc/init.d/vertica-consoled restart

Recommended storage format types

Choose the storage format type based on deployment requirements. Vertica recommends the following storage format types where applicable:

- ext3
- ext4
- NFS for backup
- XFS
- Amazon S3 Standard, Azure Blob Storage, or Google Cloud Storage for communal storage and related backup tasks when running in Eon Mode

> **Note**
> For the Vertica I/O profile, the ext4 file system is considerably faster than ext3.

The storage format type at your backup and temporary directory locations must support fcntl lockf (POSIX) file locking.

You can view the file systems in use on your nodes by querying the system table STORAGE_USAGE .

Vertica users have successfully deployed other file systems, Vertica cannot guarantee or desired outcomes on all storage format types. In certain support situations, you may be asked to migrate to a recommended storage format type to help with troubleshooting or to fix an issue.

Vertica Analytic Database supports Linux Volume Manager (LVM) on all supported operating systems. Your LVM version must be 2.02.66 or later, and must include device-mapper version 1.02.48 or later. For information on requirements and restrictions, see the section, Vertica Support for LVM.

Network address family support

Vertica server supports IPv4 and IPv6 network addresses for both internal and external communications. The database cluster uses IPv4 by for internal communications by default. You can choose to have the cluster use IPv6 for its internal communications when you install Vertica and create the cluster.

Vertica supports using IPv6 to identify nodes in the database cluster. However, AWS DNS resolution does not support IPv6. To have a cluster in AWS that uses IPv6, use the IPv6 IP addresses instead of using host names when installing Vertica and forming the cluster.

Currently, Vertica does not support using IPv6 on Google Cloud Platform or Microsoft Azure.

The MC currently does not support IPv6. If your Vertica database uses IPv6 for internal communications, the MC will not be able to connect to or manage the database. The MC must communicate with the database cluster using its own internal network addresses.

Supported browsers for Management Console

Vertica Analytic Database 12.0.x Management Console is supported on the following web browsers:

- Chrome
- Firefox
- Microsoft Edge

Vertica server and Management Console compatibility

Management Console (MC) 12.0.x is compatible with all supported Vertica server versions.

Client drivers support

Vertica provides JDBC, ODBC, OLE DB, Python, vsql, and ADO.NET client drivers. Download the latest drivers from Vertica Client Drivers . Choose from drivers for the following platforms:

Platform	Drivers	See also
Linux/UNIX	ODBC, JDBC, Python, vsql clients	Installing the ODBC client driver
Windows	ODBC, ADO.NET, OLE DB client drivers, the vsql client, Microsoft Connectivity Pack, Visual Studio plug-in	Windows client driver installer
macOS (including M1 and M2 processors)	ODBC, vsql clients	Installing the ODBC client driver

Cross-platform	JDBC client driver .jar file available for installation on all platforms	

To view a list of driver and server version compatibility, see Client driver and server version compatibility .

ADO.NET and OLE DB drivers

The ADO.NET and OLE DB drivers are supported on the following platforms:

Platform	Processor	Supported Versions	.NET Requirements
Microsoft Windows	x86 (32-bit)	Windows 10	Microsoft .NET Framework 4.6 or later service packs
Microsoft Windows	x64 (64-bit)	Windows 10	
Microsoft Windows Server	x64 (64-bit)	2016 2019	

JDBC driver

All non-FIPS JDBC drivers are supported on any Java 5-compliant platform or later (Java 5 is the minimum).

ODBC driver

Vertica Analytic Database provides both 32-bit and 64-bit ODBC drivers. Vertica 12.0.x ODBC drivers are supported on the following platforms:

Platform	Processor	Supported Versions	Driver Manager
Microsoft Windows	x86 (32-bit)	Windows 10	Microsoft ODBC MDAC 2.8
Microsoft Windows	x64 (64-bit)	Windows 10	
Microsoft Windows Server	x64 (64-bit)	2016 2019	
Red Hat Enterprise Linux / CentOS	x86_64	7.0, 7.3 or later	iODBC 3.52.6 or later unixODBC 2.3.0 or later DataDirect 5.3, 6.1, or later
FIPS-compliant Red Hat Enterprise Linux	x86_64	8.1 or later	
SUSE Linux Enterprise	x86_64	12 SP2, 12 SP3, 12 SP4	
openSUSE	x86_64	42.3	
Oracle Enterprise Linux (Red Hat compatible kernel only)	x86_64	7.3 or later	
Ubuntu	x86_64	14.04 LTS, 16.04 LTS, 18.04 LTS, 19.1	
Amazon Linux	x86_64	2	
Debian Linux	x86_64	8.5, 8.9, 10	
macOS	x86_64, M1, M2	10.15 or later	

vsql client

The Vertica vsql client is included in all client packages. It is not available as a separate download. The vsql client is supported on the following platforms:

Operating System	Processor	Supported Versions
Microsoft Windows	x86, x64	Windows 2016, 2019 Windows 10
Red Hat Enterprise Linux / CentOS	x86, x64	7.x: all 8.x: all
FIPS-compliant Red Hat Enterprise Linux	x86_64	8.1 or later
SUSE Linux Enterprise	x86_64	12: SP2 or later
openSUSE	x86, x64	42.3
Oracle Enterprise Linux (Red Hat compatible kernels only)	x86, x64	6.7 or later 7.x: all
Ubuntu	x86, x64	14.04 LTS, 16.04 LTS, 18.04 LTS, 19.1
Debian Linux	x86, x64	8.5, 8.9
macOS	x86, x64, M1, M2	10.15 or later
Amazon Linux	x86, x64	2

In this section

- Perl driver requirements
- Python driver requirements

Perl driver requirements

To use Perl with Vertica, you must install the Perl driver modules (DBI and DBD::ODBC) and a Vertica ODBC driver on the machine where Perl is installed. The following table lists the Perl versions supported with Vertica 12.0.x.

Later versions of Perl (5.10 and above), DBI, and DBD::ODBC might also work.

Perl Version	Perl Driver Modules	ODBC Requirements
• 5.8 • 5.10	• DBI driver version 1.609 • DBD::ODBC version 1.22	See Client drivers support .

Python driver requirements

To use Python with Vertica, you must install either:

- The vertica-python client.
- The pyodbc module.

For details, see Installing Python client drivers .

The following table lists compatible versions of Python, the Python drivers, and ODBC.

Python Version	Python Driver Module	ODBC Requirements
2.4.6	pyodbc 2.1.6	See Client drivers support .

2.7.x	Vertica Python Client (Linux only)
2.7.3	pyodbc 3.0.6
3.3.4	pyodbc 3.0.7

Vertica SDKs

This section details software requirements for running User Defined Extensions (UDxs) developed using the Vertica SDKs.

C++ SDK

The Vertica cluster does not have any special requirements for running UDxs written in C++.

Java SDK

Your Vertica cluster must have a Java runtime installed to run UDxs developed using the Vertica Java SDK. Vertica has tested the following Java Runtime Environments (JREs) with this version of the Vertica Java SDK:

- Oracle Java Platform Standard Edition 6 (version number 1.6)
- Oracle Java Platform Standard Edition 7 (version number 1.7)
- Oracle Java Platform Standard Edition 8 (version number 1.8)
- OpenJDK 6 (version number 1.6)
- OpenJDK 7 (version number 1.7)
- OpenJDK 8 (version number 1.8)

Python SDK

The Vertica Python SDK does not require any additional configuration or header files.

R language pack

The Vertica R Language Pack provides version 3.5 of the R runtime and associated libraries for interfacing with Vertica. You install the R Language Pack on the Vertica server.

FIPS 140-2 supported platforms

Vertica uses a certified OpenSSL FIPS 140-2 cryptographic module to meet the security standards set by the National Institute of Standards and Technology (NIST) for Federal Agencies in the United States or other countries. Vertica links with the version of OpenSSL on the system to perform cryptographic operations at run time. When operating in FIPS mode, Vertica relies on the operating system's FIPS configuration to ensure a FIPS-certified version of OpenSSL is present in the environment.

Vertica has been tested in FIPS mode using OpenSSL 1.1.1k on the following operating systems:

- Red Hat Enterprise Linux 8.5
- SUSE Linux Enterprise Server 15

Vertica supports FIPS mode on FIPS-compliant operating system versions that are equal to or higher than the tested version. For example, Vertica supports only FIPS-compliant versions of Red Hat Enterprise Linux 8.5 and higher. Vertica does not support FIPS mode with 7.x versions of Red Hat Enterprise Linux or CentOS.

FIPS-enabled Vertica requires the following:

- A user-generated certificate signed by an approved Certificate Authority.
- TLS 1.2 to support the server-client connection for a FIPS-enabled system.

Supported drivers

Vertica supports the following client drivers for FIPS-compliance:

- vsql
- ODBC
- JDBC

Important
FIPS-enablement is not supported in the Management Console.

For more information see Federal information processing standard .

Eon on-premises storage

Vertica supports the following storage platforms for Vertica Eon Mode running on-premises.

Pure Storage FlashBlade

Vertica supports communal storage on Pure Storage FlashBlade version 3.0.0 and later. See Create an Eon Mode database on-premises with FlashBlade for more information.

Vertica does not support the use of Vertica Management Console or admintools to administer data located on Pure Storage hardware.

For information on configuring Pure Storage, refer to support.purestorage.com .

MinIO

Vertica supports communal storage on MinIO version 2018-12-27T18:33:08Z and later. See Create an Eon Mode database on-premises with MinIO for more information.

> **Caution**
> In Eon Mode, Vertica relies on the communal storage platform to manage data safety and integrity. For production use, always use MinIO in a distributed mode cluster. This mode provides high availability and data integrity protection. See the Distributed MinIO Quickstart Guide for instructions on configuring MinIO in distributed mode.
>
> To ensure MinIO consistency guarantees, MinIO must be configured for read-after-write and list-after-write consistency. For details, see the MinIO documentation .

Vertica does not support the use of Vertica Management Console or admintools to administer data located on MinIO.

See the MinIO website for more information about MinIO.

HDFS

Vertica supports communal storage on HDFS when accessed through WebHDFS. See Create an Eon Mode database on-premises with HDFS for more information.

For HDFS, Vertica does not support the following:

- The MapR distribution of HDFS, which is accessed through an NFS mount point and not through WebHDFS.
- Using Vertica Management Console or admintools to administer data located on HDFS.
- Cloudera (CDH) versions 5.x in Eon Mode.
- The copycluster vbr backup and restore utility for communal storage on HDFS.

Other validated object storage

The preceding sections detail storage platforms and versions that Vertica engineering tests under specific performance and load thresholds. In addition to these storage platforms, the Vertica Partner Engineering team validates object storage platforms that meet strict performance requirements.

For details, see On-Premises .

Vertica on Amazon Web Services

For information about deploying Vertica on Amazon Web Services (AWS), see Vertica on Amazon Web Services .

AWS instance types

Vertica supports a range of AWS instance types to deploy cluster hosts or MC hosts on AWS. See Supported AWS instance types for a complete list of supported instance types.

Amazon machine images

Vertica provides tested and pre-configured Amazon Machine Images (AMIs) to deploy cluster hosts or MC hosts on AWS. The Vertica AMI allows users to configure their own storage using the officially supported version of Vertica Analytic Database for AWS.

See Choose a Vertica AMI Operating Systems for a list of operating systems currently available in Vertica AMIs.

Consider the following when using the Vertica AMI:

- Vertica develops AMIs on a slightly different schedule than the product release schedule. The AMIs for Vertica releases are available sometime following the initial release of Vertica software.
- Each Vertica AMI comes pre-configured with default resource limit settings .
- Amazon does not support using 32-bit binaries on Amazon Linux 2.0 AMIs. Therefore, you cannot use the Vertica 32-bit client libraries on these AMIs.

IPv6 support

Vertica supports using IPv6 to identify nodes in the database cluster. However, AWS DNS resolution does not support IPv6. To have a cluster in AWS that uses IPv6, use the IPv6 IP addresses instead of using host names when installing Vertica and forming the cluster.

Containerized environments

Vertica supports running in any containerized environment that conforms to the performance requirements for vioperf , vnetperf , and vcpuperf .

As Vertica extends our support and deployment in containerized environments including Kubernetes, we cannot test and certify all possible configurations. However, OpenText makes an effort to ensure the success of its customers on recent versions of supported operating systems for the x86_64 architecture.

Vertica tests containers running on Docker. When the underlying hardware, OS, and container are configured correctly, the database system performs well. In some circumstances, there is a minor performance difference for queries made against a cold- or partially-populated depot when accessing communal storage.

Because your Vertica support contract covers Vertica products only, if you choose to run Vertica on a container configuration and you experience an issue that might not be caused by Vertica products, the Vertica Support team might ask you to reproduce the issue in a different environment, or engage with the support resources for your containerization technology.

For guidelines on how to provision and size your Kubernetes resources for Vertica deployments, see Recommendations for Sizing Vertica Nodes and Clusters in the Vertica Knowledge Base.

> **Note**
> If your Kubernetes cluster is in the cloud or on a managed service, each Vertica node must operate in the same availability zone.

VerticaDB operator and Vertica server version support

The VerticaDB operator supports Vertica server versions 11.0.0 and higher.

Container orchestration version support

Component	Supported Version
Kubernetes	1.21.1 and higher
Helm	3.5.0 and higher

Communal storage support

Containerized Vertica on Kubernetes supports the following public and private cloud providers:

- Amazon Web Services S3
- S3-compatible storage, such as MinIO
- Google Cloud Storage
- Azure Blob Storage
- Hadoop File Storage

Managed Kubernetes services support

Vertica supports the following managed Kubernetes services:

- Amazon Elastic Kubernetes Service (EKS)
- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)

Cluster management platform support

Vertica supports the Vertica DB operator and Vertica on Kubernetes environment on Red Hat OpenShift versions 4.8 and higher.

Virtualized environments

Vertica supports running in any virtualized environment that conforms to the performance requirements for vioperf , vnetperf , and vcpuperf .

Vertica does not support VM Snapshot.

Important
Vertica does not support suspending or migrating virtual machines while Vertica is running. A virtual machine that is suspended or migrated will in all likelihood be marked as DOWN to the Vertica cluster, reducing the overall performance of the cluster, or in a worst-case scenario, cause the cluster to crash.

Vertica has tested VMware, and when the underlying hardware is configured correctly, VMWare performs well. Customers have also deployed other virtualization configurations successfully. If you choose to run Vertica on a different virtualization configuration and you experience an issue, the Vertica Support team may ask you to reproduce the issue using a bare-metal environment to aid in troubleshooting. Depending on the details of the case, the Support team may also ask you to enter a support ticket with your virtualization vendor.

Guidelines for hypervisor and virtual machine configuration

There are many enterprise-grade hypervisors available on the market today, most of which support Linux-based virtual machines (VMs) in support of Vertica. When selecting and configuring your virtual environment, refer to the following guidelines.

- Do not over-subscribe the physical resources (CPU, memory, and network) of the hosting hardware. Many hypervisors allow you to take advantage of scaling out solutions by over-subscribing resources, for example, deploying more virtual CPUs than are physically installed in the host hardware. However, this type of deployment has a negative performance effect on a Vertica cluster.
- Configure the hypervisor to run low-latency, high-performance applications. This means that you should disable power-saving features and CPU frequency scaling on the hypervisor hardware because these technologies contribute to latency in the applications.
- Choose an operating system for the Vertica VMs that is supported by Vertica and by the hypervisor you are using. For some hypervisors, different operating systems may perform better than others. Vertica recommends that you investigate the options with your hypervisor vendor.
- Configure attached storage for high I/O performance. A virtualized Vertica node requires the same amount of disk I/O performance as a non-virtualized one. Vertica recommends that customers use the vioperf utility to validate the actual performance throughput being achieved on each VM.
- If you are providing storage using a shared storage device, make sure to validate disk I/O performance on the cluster as a whole to ensure that the shared resource(s) do not create a bottleneck. To achieve this validation, run the vioperf utility on all the cluster nodes simultaneously to determine the maximum disk I/O performance that can be achieved on each VM during times of heavy I/O load.
- Memory recommendations for Vertica running in a virtualized environment are no different than running in a non-virtualized environment. Vertica recommends that you allocate 8 GB of memory per virtual core. Again, do not over-subscribe the memory available in the hypervisor, because this creates contention for the physical resources, causes negative performance impacts, and possibly crashes the VMs.
- Networking requirements for a virtualized Vertica cluster are the same as for a non-virtualized cluster. Each node in the cluster must be able to communicate with all the other nodes, and latency in those communications can have a negative effect on cluster performance. When you are running multiple virtual machines on a single host server, the network communication is very fast. This occurs because the network traffic is virtualized in the memory space of the hypervisor and never leaves the physical server. However, if the cluster expands beyond a single host, the physical networking of that host can become a bottleneck for the cluster. If you are deploying in a virtual environment, that environment has a robust networking infrastructure that can provide the necessary connection speeds between physical hosts. In most cases, there will be multiple 10 GBE networking connections. Use the vnetperf utility to validate actual network performance speeds between nodes in your Vertica cluster.
- When deploying multiple Vertica VMs per physical host, the fewer the better. The goal of virtualization is to consolidate workloads to reduce overall hardware footprints. However, running multiple Vertica VMs on the same host can place the Vertica cluster in a situation where a single hardware failure can take down multiple nodes in a cluster, and perhaps even the cluster itself. Vertica recommends that when you virtualize a Vertica cluster, spread the VMs across as many physical hosts as possible, with an ideal goal of having one Vertica VM per physical host.
- While virtual networking can be very robust, Vertica has found that UDP broadcast traffic that is used in the spread daemon can be unreliable in most virtual environments, especially when those environments are spread across more than one physical host. In order for Vertica to function effectively in a virtualized environment, use the `--point-to-point` flag when you execute the `/opt/vertica/sbin/install_vertica` script. This flag configures the spread daemons to communicate directly with one another.

Hadoop integrations

OpenText supports Vertica 12.0.x with the following Hadoop distributions. OpenText expects Vertica to work with subsequent Hadoop distributions, and tests these later distributions as soon as practical.

Distribution	Supported Versions	Important Notes

Cloudera (CDH)	• 5.11 and higher* • 6.x	You cannot use versions 5.x in Eon Mode.
HortonWorks Data Platform (HDP)	• 2.4 and higher* • 3.0	

* Vertica is phasing out support for this platform. See End-of-support notices for more information.

You must apply patches for the following issues: HDFS-8855 and HDFS-8696. See your Hadoop vendor documentation for further instructions.

MapR versions 5.2 and later are expected to work. You cannot use MapR in Eon Mode.

Apache Kafka integrations

You can use Vertica with the Apache Kafka message broker. For more information on Kafka integration, refer to Apache Kafka integration .

Kafka versions

Vertica has been tested with different versions of Apache Kafka. The following table lists the Kafka versions that each Vertica version supports:

Apache Kafka Versions	Vertica Versions
2.0, 2.1, 2.2.1, 2.4.1	9.3.1 and higher
2.0, 2.1	9.3.0 and higher
1.0, 1.1, 2.0	9.2.1 and higher
0.11, 1.0, 1.1	9.1.1 and higher

Avro schema registry versions

The Vertica integration for Apache Kafka has been tested with the Avro schema registry distributed with Confluent 3.3.1 and 4.0.0. For more information about Confluent, see the Confluent website .

Java versions

The data streaming job scheduler uses the Vertica JDBC library to connect to the target database, and requires Java 8 or later.

Apache Spark integrations

You can use the Vertica Connector for Apache Spark to transfer data between Vertica and Apache Spark. The following table shows the versions Apache Spark and Scala the Connector supports as well as the name of the Spark Connector JAR file to use for each combination:

Apache Spark Version	Scala Version	Spark Connector JAR file
2.0*	2.11	vertica-spark2.0_scala2.11.jar
2.1*	2.11	vertica-spark2.1_scala2.11.jar
2.2	2.11	vertica-spark2.1_scala2.11.jar
2.3	2.11	vertica-spark2.1_scala2.11.jar
2.4.1	2.11	vertica-spark2.1_scala2.11.jar
2.4.1	2.12	vertica-spark2.4-3.0_scala2.12.jar
3.0	2.12	vertica-spark2.4-3.0_scala2.12.jar

* Vertica is phasing out support for this Apache Spark version. See End-of-support notices for more information.

Notes

- A Spark Connector JAR file can support multiple versions of Spark. For example, vertica-spark2.1_scala2.11.jar supports Spark 2.1, 2.2, 2.3, and 2.4.1.
- Vertica recommends you always use the version of the Spark Connector shipped with your version of the Vertica server. When you upgrade your Vertica server, you should also upgrade your version of the Spark Connector.

For more information on Apache Spark integration, refer to Apache Spark integration .

Linux volume manager (LVM)

Vertica 12.0.x supports Linux Volume Manager (LVM) on all supported operating systems.

LVM version supported

Vertica supports LVM version 2.02.66 or later, and must include device-mapper version 1.02.48 or later.

LVM configuration notes

In configuring LVM:

- When you create logical volumes with the lvcreate command, use the readahead option to set the read ahead sector count to greater than 2048 KB.
- You can use the default settings for all other LVM options.

LVM restrictions

The following limitations apply to LVM support:

- You cannot have physical drives shared across several nodes.
- Vertica supports linear logical volumes only. Vertica does not support striped or mirrored logical volumes.
- Vertica supports extending logical volumes (lvextend), but not reducing the size of a logical volume.
- Vertica recommends frequent backups.
- Vertica does not support LVM backup and restore, such as LVM snapshot and merge. Use the Vertica backup utility, vbr.
- Vertica does not support LVM space reclamation because space reclamation is duplicated when reducing the size of a logical volume.
- Vertica does not support LVM migration. Use Vertica Copy operations.
- Vertica does not support LVM high availability. Use Vertica high availability capabilities.
- Vertica does not support LVM RAID. Configure RAID at the disk controller level.

End-of-support notices

These end-of-support notices apply to specific client, Linux, Hadoop, and Kafka distributions.

End-of-support notices

Vertica no longer supports the following client platforms and server distributions:

- AIX (all releases)
- Amazon Linux 2017.09
- Debian 7.6, 7.7
- HP-UX (all releases)
- macOS 10.10
- Red Hat Enterprise Linux/CentOS 6.x
- SUSE 11SP3
- Ubuntu 12.04

New features

This guide briefly describes the new features introduced in the most recent releases of Vertica and provides references to detailed information in the documentation set.

For known and fixed issues in the most recent release, see the Vertica Release Notes .

In this section

- New and changed in Vertica 12.0.4
- New and changed in Vertica 12.0.3
- New and changed in Vertica 12.0.2

- New and changed in Vertica 12.0.1
- New and changed in Vertica 12.0.0
- Deprecated and removed functionality

New and changed in Vertica 12.0.4

In this section

- admintools
- Client connectivity
- Client drivers
- Complex types
- Containers and Kubernetes
- Database management
- dc_catalog_truncations improvements
- Directed queries
- Documentation updates
- Eon Mode
- Apache Kafka integration
- Loading data
- Machine learning
- Partitioning
- Performance improvements
- Resource management
- SDK updates
- Security and authentication
- Stored procedures
- Upgrade
- User-defined extensions

admintools

Setting re_ip timeout

You can configure how long re_ip executes a given task before it times out, by editing the setting of prepare_timeout_sec in admintools.conf . By default, this parameter is set to 7200 (seconds).

Client connectivity

Set idle timeouts during connection initialization

You can set the IdleTimeoutInitializingConnectionsMs configuration parameter to change the length of time that the server waits for an idle client during each step in connection initialization, before a session is created.

Client hostnames

The hostnames of clients (as reported by their operating systems) are now recorded in the CLIENT_OS_HOSTNAME column of the following tables:

- SESSIONS
- CURRENT_SESSION
- USER_SESSIONS

Client drivers

ODBC and vsql: support for M1 and M2 processors

The ODBC and vsql client drivers now support Apple M1 and M2 processors. For details, see Client drivers support .

Complex types

UNNEST function

The new UNNEST function expands one or more array columns. UNNEST is similar to EXPLODE, with two key differences:

- UNNEST returns element values but not positions.
- UNNEST automatically expands all arrays that are passed to it.

EXPLODE performance optimization

The EXPLODE function translates a single row of input into multiple rows of output, one per collection element. There is, therefore, usually no benefit to partitioning the input. Previously, a query using EXPLODE required an OVER clause and Vertica partitioned the input before calling EXPLODE. Now, you can skip this partitioning step by using the skip_partitioning function parameter and omitting the OVER clause.

CROSS JOIN EXPLODE

You can use the output of EXPLODE as if it were a relation by using CROSS JOIN EXPLODE in a query, as in the following example:

```
=> ALTER SESSION SET UDPARAMETER FOR ComplexTypesLib skip_partitioning = true;

=> SELECT student, score FROM tests
  CROSS JOIN EXPLODE(scores) AS t (pos, score)
  ORDER BY score DESC;
 student | score
---------+-------
 Sam     |   98
 Sam     |   97
 Bob     |   92
 Tom     |   91
 Sam     |   85
 Tom     |   82
 Bob     |   79
 Bob     |   78
 Tom     |   75
 Tom     |   68
(10 rows)
```

For details, see Joining on Results .

Containers and Kubernetes

Server-side encryption (SSE)

Veritca on Kubernetes supports all three types of S3 SSE:

- SSE-3
- SSE-KMS
- SSE-C

For details, see Configuring communal storage and Custom resource definition parameters .

Set configuration parameters

You can set configuration parameters in the Vertica server with the communal.additionalConfig parameter.

For details, see Custom resource definition parameters .

Set depot volume type

You can store the depot in an emptyDir volume with the local.depotVolume parameter. When you use an emptyDir volume, the local.storageClass and local.requestSize parameters only apply to data stored in PersistentVolumes.

For details about the custom resource parameters, see Custom resource definition parameters . For details about volume types, see the Kubernetes documentation .

EventTrigger custom resource definition (CRD)

The EventTrigger CRD extends the Kubernetes Job workload resource to watch a VerticaDB custom resource (CR) for a state change. When a condition on the VerticaDB CR changes to a specified status, the EventTrigger runs a Job.

For details, see EventTrigger CRD .

Use underscore character in subcluster name

You can now use the underscore character ("_") in the subcluster name. For details about subcluster naming conventions, see Subclusters on Kubernetes .

Vertica image with no private keys

Vertica on Kubernetes provides an image that does not contain static SSH keys to authenticate internal communication between the pods. This requires that you provide SSH keys to the custom resource (CR). You can conceal your SSH keys in a Secret , and pass that Secret to the CR with the sshSecret field.

For details about the image and passing SSH keys to the CR, see the following:

- Vertica images
- Custom resource definition parameters
- VerticaDB CRD

Custom name label

You can customize your VerticaDB CRD application's name with the app.kubernetes.io/name label:

```
spec:
  labels:
    app.kubernetes.io/name: app-name
```

If you do not add a custom name label, then the default value is vertica . For additional details about the name label, see the Kubernetes documentation .

Delete PVCs after VerticaDB delete

When you delete a VerticaDB custom resource, the VerticaDB operator deletes any PVCs that are associated with that instance. For details about Vertica on Kubernetes persistent storage, see Containerized Vertica on Kubernetes .

Database management

Amazon simple notification service (SNS) notifiers

You can now send messages to SNS topics with notifiers . For details, see Configuring reporting for the simple notification service (SNS) .

dc_catalog_truncations improvements

DC_CATALOG_TRUNCATIONS Improvements

After a cluster restart (normal or crash), there is no indication how many transactions were rolled back by the system. New columns epoch_before_truncation and epoch_after_truncation have been added to the dc_catalog_truncations tables to track where the workload needs to be restarted to catchup the missing transactions.

Directed queries

INSERT...SELECT hints

You can now annotate the query of an INSERT...SELECT statement with directed query hints.

Documentation updates

Data exploration

When data is not well-specified in a schema, developing a table definition so you can load the data can be challenging. Vertica provides tools you can use to explore data and refine a table definition. These tools and processes are now described in a new top-level section of the documentation, Data exploration .

Eon Mode

Subcluster sandboxing

Sandboxing enables you to spin-off a secondary subcluster from an existing cluster, resulting in two mutually isolated clusters that share the same data but do not interfere with each other. After sandbox creation , you can perform standard database operations and queries, such as creating new tables or loading libraries, without affecting the other cluster. For example, dropping a table in the sandboxed subcluster does not drop the table in the main cluster, and vice versa.

Sandboxes support many use cases, including the following:

- Testing a new version of Vertica without needing to spin-up a new cluster and reload data.
- Trying out features without compromising the consistency of the main cluster.
- Sharing data with another team by giving them access to a sandboxed subcluster.

After removing the sandbox and performing necessary cleanup tasks, the subcluster can rejoin the main cluster.

Apache Kafka integration

kafka_conf_secret parameter

The kafka_conf_secret parameter conceals sensitive data that you must pass to rdkafka to configure the Vertica and Kafka integration. Vertica does not log or store values that you pass in kafka_conf_secret.

This option is available on tools and functions that use the kafka_conf parameter.

For details, see Directly setting Kafka library options and Apache Kafka integration .

Loading data

Optimization for external tables on object stores

When an external table uses data that is stored on an object store (S3, GCS, or Azure), and that data has many levels of partitioning, performance can be affected. The ObjectStoreGlobStrategy configuration parameter allows you to read and prune partitions using a different strategy, improving performance when queries are selective and there are many partition directory levels. With this setting, fewer calls are made to the object store API. For details, see Partitions on Object Stores .

Iceberg support

You can now define external tables backed by Parquet data stored by Apache Iceberg . An Iceberg table consists of data files and metadata describing the schema. Unlike other external tables, an Iceberg external table need not specify column definitions (DDL). The information is read from Iceberg metadata at query time. For certain data types you can adjust column definitions, for example to specify VARCHAR sizes. See CREATE EXTERNAL TABLE ICEBERG .

Machine learning

Model versioning

Model versioning provides an infrastructure to track and manage the status of registered models in a database. The versioning infrastructure supports a collaborative environment where multiple users can submit candidate models for individual applications. dbadmin and users with the MLSUPERVISOR role can manage and change the status of all registered models, including which models are currently in production.

Vertica provides the following model versioning functionality:

- REGISTER_MODEL : Registers a model and adds it to the versioning environment.
- CHANGE_MODEL_STATUS : Changes the status of a registered model.
- REGISTERED_MODELS : Lists registered models in the database.
- MODEL_STATUS_HISTORY : Tracks the status history of registered models.

For details about the model versioning environment and an in-depth example, see Model versioning .

Poisson regression

Vertica now supports the Poisson regression algorithm to model count data. Poisson regression offers an alternative to linear regression or logistic regression and is useful when your desired prediction range is non-negative real numbers or integers.

Autoregresssive integrated moving average (ARIMA) support

Vertica now supports autoregressive integrated moving average (ARIMA) models for time series analytics. ARIMA models combine the abilities of AUTOREGRESSOR and MOVING_AVERAGE models by making future predictions based on both preceding time series values and errors of previous predictions. You can use the ARIMA function to create and train an ARIMA model and the PREDICT_ARIMA function to make predictions.

For an in-depth example that trains and makes predictions with an ARIMA model, see ARIMA model example .

Partitioning

Window partitioning for 1:N transform functions using PARTITION ROW

Previously, window partitioning could only be used for 1:1 analytic functions. Now, the PARTITION ROW syntax can be used for window partitioning 1:N transform functions.

Performance improvements

Refreshing partitioned projections

In previous releases, new and updated projections were not available to queries before they were completely refreshed with all table data. This delayed availability of projections to process query requests.

Now, when you refresh the projections of a partitioned table—for example, a new projection of a table with existing data—the refresh operation first loads data from the partition with the highest range of keys. After this partition is refreshed, Vertica begins to refresh the partition with next highest partition range. This process continues until all projection partitions are refreshed. In the meantime, while the refresh operation is in progress, projection partitions that have completed the refresh process become available to process query requests.

Resource management

Default user resource pool

The new configuration parameter DefaultResourcePoolForUsers lets you configure the default user resource pool that is assigned to new users, including LDAP users. For details, see User resource allocation .

SDK updates

One-to-many UDTFs and PARTITION ROW

Single-phase user-defined transform functions (UDTFs) written in C++ and Python now support the isExploder class property, which indicates whether the UDTF performs a one-to-many transform. If this property is set to True within the getTransformFunctionProperties factory method, the function uses an OVER(PARTITION ROW) clause by default and thereby removes the need to specify an OVER clause when calling the UDTF. Users can also include any expression in SELECT queries that call a one-to-many UDTF.

For more information about this class property and one-to-many UDTFs, see Partitioning options for UDTFs and TransformFunctionFactory class . See Python example: explode for an in-depth example detailing a one-to-many UDTF.

Security and authentication

OAuth

JSON web token validation

Vertica can now validate OAuth tokens without contacting the identity provider by verifying the signer of the OAuth token. To enable this feature, create an authentication record that uses ' oauth ' and set the authentication parameters to use JWT validation. For details, see Configuring OAuth authentication .

Just-in-time user provisioning

Just-in-time (JIT) user provisioning is the act of automatically configuring an authenticated user and their roles based on information provided by the identity provider. When a client authenticates to Vertica with an OAuth token the authentication record enables JIT user provisioning, Vertica automatically creates the user (if the user does not already exist in Vertica). Additionally, if Keycloak is the identity provider, Vertica grants to the users their roles as specified by Keycloak.

For example, if a client presents an OAuth token to authenticate as user Alice with role director , and Alice does not exist in Vertica, Vertica automatically creates the user Alice , grants to her the authentication record. This configuration also uses Keycloak as the identity provider, so Vertica grants to her the director role as a default role .

For details, see Just-in-time user provisioning

JDBC: custom truststore

You can use a custom truststore to secure the connection between the JDBC client and the identity provider. This is useful if your identity provider's certificate is not signed by a well-known certificate authority.

The path to the truststore and its password are specified with the following connection properties:

- oauthtruststorepath
- oauthtruststorepassword

If oauthtruststorepath is not specified, the JDBC client uses the default truststore.

For details on this and other connection properties, see JDBC connection properties . For details on configuring OAuth, see Configuring OAuth authentication .

Stored procedures

Scheduled execution

You can now schedule the automatic execution of stored procedures. This can be used to automate various tasks, like logging database activity, revoking privileges, or creating roles. For details, see Scheduled execution .

Viewing overloaded stored procedures

You can now view the source code of a particular implementation of an overloaded stored procedure with EXPORT_OBJECTS . For details, see Exporting objects .

Upgrade

Upgrade to any Vertica version higher than your current version

Previously, Vertica upgrades were incremental, requiring you to upgrade to each major and minor Vertica version in your upgrade path. You can now skip intermediate version and upgrade directly to any version of Vertica higher than your current version.

User-defined extensions

Redactable user-defined extension and session parameters

User-defined extension and session parameters that meet any of the following criteria are automatically redacted from logs and system tables like QUERY_REQUESTS :

- Named "secret" or "password"
- End with "_secret" or "_password"

New and changed in Vertica 12.0.3

In this section

- Client connectivity
- Containers and Kubernetes
- Machine learning
- Partitioning
- Query optimization
- Security and authentication
- SQL functions and statements

Client connectivity

Client label environment variable

You can now set a label for the vsql client with the VSQL_CLIENT_LABEL environment variable. For details, see vsql environment variables .

Containers and Kubernetes

TLS certificate authentication to prometheus metrics

In some environments, you might need to access Prometheus metrics from an external client. Vertica provides the prometheus.tlsSecret Helm chart parameter to configure a role-based access control (RBAC) proxy sidecar to authenticate requests with user-provided TLS certificates.

For details, see Helm chart parameters and Prometheus integration .

Set readiness probe on container

The readinessProbeOverride custom resource definition (CRD) parameter overrides settings for the default readiness probe so that you can fine-tune when the Vertica pod is ready to accept traffic.

For details, see Custom resource definition parameters .

IRSA profile authentication to Amazon EKS

You can authenticate to Amazon Elastic Kubernetes Service (EKS) with IAM roles for service accounts (IRSA). For details, see Configuring communal storage

Operator scheduling rules for helm chart

You can control which node the operator pod is scheduled on with the following Helm chart parameters:

- affinity
- nodeSelector
- priorityClassName
- tolerations

For details, see Helm chart parameters .

Set liveness and startup probes on container

The livenessProbeOverride and startupProbeOverride custom resource definition (CRD) parameters override settings for the corresponding default probes. These parameters fine-tune how the container and the Vertica process within the container indicate their state to other objects in the StatefulSet.

For details, see Custom resource definition parameters .

Override pod-level security context

The podSecurityContext custom resource definition (CRD) parameter can elevate pod-level privileges so that you can perform privileged actions, such as setting sysctl commands in the pod.

For details, see Custom resource definition parameters . For additional details about pod-level privileges, see the Kubernetes documentation .

Machine learning

Extract predictor importance of XGBoost models

The new XGB_PREDICTOR_IMPORTANCE function measures the importance of the predictors in XGB_CLASSIFIER and XGB_REGRESSOR models.

K-prototypes

You can now use the k-prototypes clustering algorithm to cluster mixed data into different groups based on similarities between data points.

New MLSUPERVISOR role

A new predefined role MLSUPERVISOR can delegate DBADMIN privileges for ML model management to other users.

Partitioning

New drop partitions (d) lock

DROP_PARTITIONS now sets an exclusive D lock on the target table. This lock is only compatible with I-lock operations, so table load operations such as INSERT and COPY are allowed during the drop operation. All other operations that require a table lock, such as UPDATE or DELETE, are blocked until the drop partition operation is complete and the D-lock is released.

Query optimization

Disabling projections

ALTER PROJECTION now supports a new DISABLE parameter, which marks a projection as unavailable to the optimizer.

Security and authentication

Authentication filtering

Users can now specify the authentication record they want to authenticate with by providing the credentials required by that record. Previously, users could only authenticate with the highest priority authentication record or the records to which it fell through . For details, see Authentication filtering .

SQL functions and statements

LIKE support for ANY/ALL

LIKE operators now support the keywords ANY and ALL, which let you specify multiple patterns to test against a string expression. For example, the following query qualifies the case-insensitive ILIKE with ALL to find all customer names that contain the strings media and ever :

```
=> SELECT DISTINCT (customer_name) FROM customer_dimension
    WHERE customer_name ILIKE ALL ('%media%','%ever%') ORDER BY customer_name;
 customer_name
---------------
 Evermedia
(1 row)
```

For details, see LIKE ANY/ALL Usage .

New and changed in Vertica 12.0.2

In this section

- Client connectivity
- Client drivers
- Complex types

- Containers and Kubernetes
- Documentation updates
- Eon Mode
- Installation and upgrade
- Licensing and auditing
- Loading data
- Machine learning
- Management Console
- Query optimization
- Security and authentication
- SQL functions and statements
- Vertica on the cloud

Client connectivity

Client labels

You can now set the client label for vsql connections at the start of the session with the --label option. Client sessions and their labels are listed in the SESSIONS system table. This option supplements the existing SET_CLIENT_LABEL function, which sets the client label for existing sessions.

JDBC database name

The database name parameter in DriverManager.getConnection() is now optional.

Client drivers

OAuth: disable hostname validation

You can now disable hostname validation of the OAuth identity provider with the boolean oauthvalidatehostname in OAuthJsonConfig.

OAuthValidateHostname is set to true by default, requiring the identity provider's certificate to list the IP or hostname of the identify provider. For details, see JDBC connection properties and ODBC DSN connection properties .

ODBC request timeout

You can now specify the timeout for requests sent by ODBC clients with the ConnectionTimeout parameter. For details on this and other parameters, see ODBC DSN connection properties .

Complex types

Array search

ARRAY_FIND and CONTAINS can now search for elements that satisfy a predicate by using Lambda functions . The new FILTER function applies a predicate to an array and returns an array containing only elements that satisfy the predicate. See Searching and Filtering .

Parquet export

EXPORT TO PARQUET now supports the ARRAY and ROW types in any combination.

Containers and Kubernetes

IAM profile authentication to Amazon EKS

You can authenticate to Amazon Elastic Kubernetes Service (EKS) with an Identity and Access Management (IAM) profile. For details, see Configuring communal storage .

TLS certificate management for helm charts

The VerticaDB operator can generate and manage TLS certificates with the webhook.certSource Helm chart parameter. The operator can generate certificates internally, or it can manage existing certificates.

For details about setting the webhook.certSource parameter, see Helm chart parameters .

For details about managing TLS certificates for Helm charts, see Installing the Vertica DB operator .

CA bundle management for helm charts

You can include your certificate authority (CA) bundle in the Secret provided to the webhook.tlsSecret Helm chart parameter. This parameter stores the CA bundle with its TLS certificate and keys.

Previously, you had to use the webhook.caBundle parameter to provide access to the CA bundle. This parameter is now deprecated.

For details, see Helm chart parameters .

Custom catalog path

You can create a custom catalog path in the container filesystem to store the catalog in a different location than the local data. Set the local.catalogPath Custom Resource Definition (CRD) parameter to define the custom catalog location.

For details, see Custom resource definition parameters .

Role and RoleBinding creation

The skipRoleAndRoleBindingCreation Helm chart parameter determines whether the Helm chart creates Roles and RoleBindings when you install the Helm chart. When set to true, you can install the Helm chart as a user that does not have privileges to create Roles or RoleBindings.

For details about the parameter, see Helm chart parameters .

For details about granting operator privileges, see Installing the Vertica DB operator .

Documentation updates

New setup section

Documentation about setting up Vertica in on-premises and cloud environments has been combined into a new Setup section. The section contains three parts:

- Plan your setup : Information to help choose a Vertica environment that fits your requirements and available resources.
- Set up Vertica on the cloud : Instructions for deploying Vertica on a cloud environment.
- Set up Vertica on-premises : Instructions for installing Vertica in an on-premises environment.

Information about setting up Vertica in Kubernetes remains in Containerized Vertica .

Eon Mode

RESHARDING_EVENTS system table

The RESHARDING_EVENTS system table provides information about historic and ongoing resharding operations, such as the current status of a resharding process and the timestamps for resharding events.

Installation and upgrade

Install binary packages in parallel

The new --parallel-no-prompts flag tells the install script to install the Vertica binaries on hosts in parallel rather than one at a time. Using this flag reduces the time it takes to install Vertica, especially in large clusters. See --parallel-no-prompts .

Licensing and auditing

Improved handling of AutoPass licenses

Vertica now rejects AutoPass licenses on builds that don't have Autopass License Server and returns an error code and notice, which details why the license failed to parse.

Loading data

Warning on unmatched data fields

The JSON and Avro parsers now produce warnings if the data being loaded contains fields that are not part of the table definition. New fields are logged in the UDX_EVENTS table and can be added to the target table definition using ALTER TABLE .

For more information, see Schema evolution and the parser reference pages, FJSONPARSER and FAVROPARSER .

Machine learning

Autoregression models support the yule-walker algorithm

You can now train autoregression models with the Yule-Walker algorithm. The AUTOREGRESSOR training function provides a method parameter by which you can specify either the Ordinary Least Squares or Yule-Walker training algorithm.

Jaro-winkler distance

You can now calculate the Jaro Distance and Jaro-Winkler Distance between two strings.

Management Console

Localize user interface text

You can localize user interface (UI) text using language files in `/opt/vconsole/temp/webapp/resources/i18n/lang` . The language files include the `locales.json` file, a list of supported languages, and a resource bundle, a directory that stores files that contain UI text strings that you can translate. By default, Vertica provides resource bundles for the following:

- American English
- Simplified Chinese

For details, see Localizing user interface text .

Support for r6i and c6i instance types on AWS

Management Console supports the following c6i and r6i EBS instance types on Amazon Web Services (AWS) for both Eon and Enterprise Mode:

- c6i.4xlarge
- c6i.8xlarge
- c6i.12xlarge
- c6i.16xlarge
- r6i.4xlarge
- r6i.8xlarge
- r6i.12xlarge
- r6i.16xlarge

You can select these instance types when you perform any of the following operations with Management Console:

- Provision a database
- Revive a database
- Scale up a database
- Add a new subcluster

For details about EBS volume defaults, see Eon Mode volume configuration defaults for AWS and Enterprise Mode volume configuration defaults for AWS .

Scale down and terminate subclusters with stopped nodes

You can scale down or terminate subclusters with stopped nodes in the following cloud environments:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure

For details, see Scaling subclusters in MC and Terminating subclusters in MC .

Query optimization

Interpolate next value

Previously, event series joins allowed you to directly compare values from two series using INTERPOLATE PREVIOUS . This feature has now been expanded, allowing you to use INTERPOLATE NEXT as well.

Security and authentication

TLS configuration support

Notifiers

You can now specify a TLS Configuration for CREATE NOTIFIER and ALTER NOTIFIER . This parameter supersedes the existing TLSMODE, CA BUNDLE, and CERTIFICATE notifier parameters.

To maintain backward compatibility, Vertica automatically generates TLS Configurations for notifiers that use the deprecated parameters.

CONNECT TO VERTICA

You can now specify a TLS Configuration for CONNECT TO VERTICA . This parameter supersedes the existing TLSMODE PREFER parameter.

SQL functions and statements

DROP TLS CONFIGURATION

You can now drop custom TLS Configurations.

Inverse hyperbolic functions

Vertica now supports the following functions:

- ACOSH
- ASINH
- ATANH

Directed query enhancements

The meta-function SAVE_PLANS lets you create multiple optimizer-generated directed queries from the most frequently executed queries. New columns have been added to the system table DIRECTED_QUERIES to store SAVE_PLANS-generated metadata.

Several directed query statements— ACTIVATE DIRECTED QUERY , DEACTIVATE DIRECTED QUERY , and DROP DIRECTED QUERY —now support WHERE clauses that can resolve to multiple directed queries. For example:

```
DEACTIVATE DIRECTED QUERY WHERE save_plans_version = 21;
```

Vertica on the cloud

Vertica AMIs support IMDSv2

Vertica AMIs can now use the AWS Instance Metadata Service Version 2 (IMDSv2) to authenticate to AWS services, including S3.

New and changed in Vertica 12.0.1

In this section

- admintools
- Client drivers
- Complex types
- Containers and Kubernetes
- Machine learning
- Management Console
- Performance improvements
- S3 object store
- Security and authentication
- SQL functions and statements
- Users and privileges

admintools

stop_db and stop_subcluster graceful shutdown

When you run the admintools commands stop_db and stop_subcluster on an Eon Mode database, the default behavior is now to gracefully shut down the specified subclusters. Both commands provide override options that force immediate shutdown of the target subclusters.

For examples, see Stopping the database and Stopping a Subcluster .

Fast startup

The admintools command start_db has a new --fast option for starting Eon Mode databases. When invoked on an unencrypted database, Vertica uses startup information from cluster_config.json to expedite the startup process.

Client drivers

OAuthJsonConfig precedence

The connection parameters oauthjsonconfig (ODBC) and OAuthJsonConfig (JDBC) now take precedence over the deprecated OAuth parameters:

- OAuthAccessToken/oauthaccesstoken
- OAuthRefreshToken/oauthrefreshtoken
- OAuthClientId/oauthclientid
- OAuthClientSecret/oauthclientsecret
- OAuthTokenUrl/oauthtokenurl
- OAuthDiscoveryUrl/oauthdiscoveryurl
- OAuthScope/oauthscope

Complex types

Add new fields to complex type columns

Previously, Vertica did not allow you to change the fields of complex type columns in native tables. Now, you can add new fields to an existing complex type column using ALTER TABLE . See Adding a new field to a complex type column for details.

Field aliases in ROW literals

In ROW literals, you can now name individual fields using AS. The following two statements are equivalent:

```
=> SELECT ROW('Amy' AS name, 2 AS id, false AS current) AS student;

=> SELECT ROW('Amy', 2, false) AS student(name, id, current);
```

Containers and Kubernetes

Elevate security context privileges for core files

The securityContext custom resource definition parameter can elevate Vertica server container privileges so that you can perform privileged actions, such as create core files that contain information about the Vertica server process.

For details, see Custom resource definition parameters and Metrics gathering . For additional details about security context, see the Kubernetes documentation .

Resize persistent volumes automatically on managed Kubernetes services

When you use a storageClass that allows volume expansion and you allocate depot storage with a percentage of the total disk space, the operator automatically adjusts the size of the PV and the depot when you update spec.local.requestSize .

For details, see Custom resource definition parameters .

Override helm chart object naming convention

The nameOverride Helm chart parameter sets the naming prefix for all objects that the Helm chart creates.

For details, see Helm chart parameters .

Java 8 support in the Vertica server image

The Vertica server image includes the Java 8 OpenJDK so that you can develop Java UDx and use the Java runtime in your Vertica environment.

For details about Java UDx development, see Java SDK . For a detailed list of Vertica images, see Containerized Vertica .

Machine learning

Isolation forest

The APPLY_IFOREST function now supports a contamination parameter that specifies the approximate ratio of data points in the training data that are labeled as outliers. The function uses this contamination value to calculate an anomaly score threshold, a minimum limit that determines whether an input data point is marked as an outlier. You can set contamination instead of setting a threshold explicitly with the threshold parameter.

Management Console

Eon Mode root volume encryption on AWS

On Eon Mode on Amazon Web Services (AWS), the root volume is encrypted by default when you perform any of the following operations with Management Console:

- Provision a database
- Revive a database
- Scale up a database
- Add a new subcluster

For details, see Amazon Web Services in MC and Subclusters in MC .

Support gp3 volume type on AWS

Vertica supports the gp3 volume type as the default volume type for root and EBS volumes on Amazon Web Services (AWS). The gp3 volume type is more cost-efficient and has better performance for input/output operations per second (IOPS) than the gp2 volume type. For details, see the Amazon gp3 product overview documentation .

Vertica uses gp3 volumes as the default volume type for the following operations in both Enterprise Mode and Eon Mode:

- Create a database
- Revive a database
- Add a subcluster
- Scale up a subcluster

> **Note**
> If the existing subcluster uses gp2 volumes, the scale-up operation adds gp2 volumes to the existing subcluster.

Additionally, the Vertica CloudFormation Template (CFT) uses the gp3 volume as the default volume type.

For details about Vertica on AWS, see the following:

- Vertica on Amazon Web Services
- Amazon Web Services in MC
- Subclusters in MC

AWS cluster creation i4i instance type support

Management Console now supports three AWS i4i instance types for database cluster creation. For configuration details, see Enterprise Mode volume configuration defaults for AWS and Eon Mode volume configuration defaults for AWS .

Performance improvements

REFRESH_COLUMNS

Refreshing a SET USING column where SET USING is set to a constant value such as null has been optimized. Compared to previous releases, calling REFRESH_COLUMNS in REBUILD mode now performs significantly faster when it repopulates a SET USING column with constant values.

One exception applies: this optimization does not apply to SET USING subqueries such as SELECT null.

Backup/restore

Better manifest processing and garbage collection have significantly improved performance of vbr backup and restore operations.

S3 object store

Server-side encryption (SSE)

Vertica now supports all three types of S3 server-side encryption: SSE-S3, SSE-KMS, and SSE-C. See S3 object store .

Security and authentication

Custom TLS configurations

You can now create custom TLS Configurations and assign them to the following database parameters to secure different connection types:

- ServerTLSConfig
- HttpsTLSConfig
- LDAPLinkTLSConfig
- LDAPAuthTLSConfig
- InternodeTLSConfig

For details, see Security parameters .

InternodeTLSConfig TLSMODE

You can now set the TLSMODE for internode encryption with ALTER TLS CONFIGURATION .

For details on data channel TLS, see Data channel TLS .

SQL functions and statements

CREATE and DROP TLS configurations

You can now create and drop TLS Configurations.

REFRESH_COLUMNS updates

The REFRESH_COLUMNS function now supports a Boolean flag that specifies whether to split ROS containers if the range of partition keys to refresh spans multiple containers or part of a single container. By default, this flag is set to true: split ROS containers as needed.

Users and privileges

Expanded cryptographic privileges

Keys, certificates, and TLS Configurations now support granting and revoking privileges.

Superusers have limited privileges on cryptographic objects that they do not own. For details, see Database Object Privileges .

Cryptographic keys and certificates

You can now grant / revoke USAGE, DROP, and ALTER privileges on cryptographic keys.

You cannot explicitly grant privileges on certificates; rather, privileges on certificates comes from ownership or privileges on the underlying cryptographic key. For details, see GRANT (key) .

TLS configurations

You can now grant / revoke DROP and USAGE privileges on TLS Configurations.

New and changed in Vertica 12.0.0

In this section

- Client connectivity
- Client drivers
- Containers and Kubernetes
- Database management
- Diagnostic tools
- Documentation updates
- Eon Mode
- Loading data
- Machine learning
- Management Console
- Projections
- SDK updates
- Security and authentication
- SQL functions and statements
- Users and privileges
- Vertica on the cloud

Client connectivity

Keepalive and heartbeat protocol settings

These new parameters let you adjust TCP keepalive and heartbeat protocol settings at the database and session levels.

- KeepAliveIdleTime: The time, in seconds, before the first TCP keepalive probe is sent to ensure that the client is still connected.
- KeepAliveProbeInterval: The time interval, in seconds, between keepalive probes.
- KeepAliveProbeCount: The number of consecutive keepalive probes that must be unacknowledged by the client before the client connection is considered lost and closed.

For details, see Managing client connections .

Client drivers

Node.js client driver

The vertica-nodejs client driver is now available .

Centralized OAuth configuration

To simplify OAuth configuration, you can now set OAuth parameters with a single JSON string with the new oauthjsonconfig (ODBC) and OAuthJsonConfig (JDBC) parameters.

To preserve existing configurations, the old parameters take precedence over the new JSON parameter, but the new JSON parameter is the recommended configuration method, and the old parameters have been deprecated .

JDBC DataSource user property

You can now get and set the user property of DataSource with getUser() and setUser() , respectively. For details, see JDBC API.

ODBC client driver enhancements

A number of enhancements have been made to the ODBC client driver to better conform to the ODBC specification. These changes may cause regressions in existing ODBC client programs:

- If an application calls SQLSetDescField to set any field other than the following, the record becomes unbound:
  - SET_DESC_COUNT
  - The deferred fields :
    - SQL_DESC_DATA_PTR
    - SQL_DESC_INDICATOR_PTR
    - SQL_DESC_OCTET_LENGTH_PTR
- SQLSetDescField must be called in the sequence order specified in the Microsoft documentation .
- If an application wants to set the precision or scale field for a numeric or date type, it must use SQLSetDescField to explicitly set those fields rather than relying on SQLBindParameter . For details, see the Microsoft documentation .
- If an application calls SQLGetData , the StrLen_or_IndPtr can return the actual length of the data available, SQL_NO_TOTAL , or SQL_NULL_DATA . If the data is truncated, Vertica returns SQL_NO_TOTAL . When checking the length of the available buffer, you must consider all cases, including SQL_NO_TOTAL and SQL_NULL_DATA . For details, see the Microsoft documentation on SQLGetData and Getting Long Data .
- If an application calls SQLExecute , the response will always attempt to set the SQLSTATE , including the status and other metadata.
- Many exception messages have been improved with more detailed error information.

JDBC: CallableStatement

The Vertica JDBC client driver now supports stored procedures through CallableStatement . For details, see Executing queries through JDBC .

Containers and Kubernetes

Spread encryption on Vertica database

You can enable Spread encryption on the Vertica database. Use the encryptSpreadComm custom resource parameter to set the EncryptSpreadComm security parameter for the database.

For details, see Custom resource definition parameters .

Prometheus operator integration

Vertica on Kubernetes integrates with the Prometheus operator to facilitate adoption in environments that currently use the Prometheus operator.

For details, see Prometheus integration .

Database management

Disk quota for schemas and tables

You can set disk quotas for schemas, individual tables, or both. Disk quotas are checked by most user operations that increase storage size. Quotas do not affect recovery, rebalancing, or Tuple Mover operations.

In Eon Mode, disk usage is an aggregate of all disk space used by all shards for the schema or table. This value is computed for primary subscriptions only. In Enterprise Mode, disk usage is the sum of disk space used by all storage containers on all nodes for the schema or table. This sum excludes buddy projections but includes all other projections.

The DISK_QUOTA_USAGES system table records quotas and current usage for objects that have quotas.

For details, see Disk quotas .

Diagnostic tools

Scrutinize

scrutinize now collects the full time zone by running the following command:

```
timedatectl | grep "Time zone"
```

Output is directed to:

```
scrutinize-output-dir/VerticaScrutinize.timestamp/node-name/context/Commands/timezone
```

For example, after running scrutinize:

```
$ cat ./VerticaScrutinize.20220513110429/v_vmart_node0001/context/Commands/timezone
    Time zone: America/New_York (EDT, -0400)
```

Documentation updates

Several organizational changes have been made to make information easier to find:

- Top-level topics have been re-ordered to place similar information together. The SQL Reference has been moved near the bottom with the other reference documentation. The lists of configuration parameters have been moved into the SQL Reference.
- In the SQL Reference, functions and meta-functions have been combined under Functions, and the alphabetical list includes all of them. (Previously, the alphabetical list included only meta-functions.) Meta-functions are labeled as such on their reference pages.
- The documentation about client drivers has been refactored.
- The information that was in the Concepts Guide has been placed with the other documentation on those topics. For example, the section about projections is now with the other documentation about projections.
- Several duplicate topics were removed, and several collections of short related topics have been consolidated.

Eon Mode

Re-shard an Eon Mode database

The new RESHARD_DATABASE function allows you to change the number of shards in an Eon Mode database. Previously, the shard count was set during database creation and could not be altered.

For more information about re-sharding, see Change the number of shards in the database.

Graceful shutdown

If you want to drain a subcluster's client connections before shutting it down, you can now gracefully shut down subclusters using SHUTDOWN_WITH_DRAIN. The function allows existing connections on the subcluster to continue their work until either all connections close or a user-specified timeout is reached. When one of these conditions is met, the function proceeds to shut down the subcluster.

For more information about the graceful shutdown process, see Graceful Shutdown.

Loading data

Partitioned file paths

Data files are sometimes partitioned in the file system using the directory structure. Partitioning moves values out of the raw data, where they have to be included for each row, and into the directory structure, saving disk space. Partitioning can also improve query performance by allowing entire directories to be skipped.

Previously, only the Parquet and ORC parsers could take advantage of partitioned file paths. Now COPY supports partitioned file paths for all parsers using the new PARTITION COLUMNS option. The hive_partition_cols parameter for the Parquet and ORC parsers is deprecated.

The hive_partition_cols parameter has the following behavior changes from previous releases:

- Nested partition directories must appear in consistent order in the file system. The following path pattern is invalid:
  ```
  /data/created=2022-01-01/region=north
  /data/region=south/created=2022-01-02
  ```
- If the column value cannot be parsed from the directory name, COPY rejects the path instead of treating the value as null.
- If the path is missing a declared partition column, COPY always returns an error. Previously, if do_soft_schema_match_by_name was true in the Parquet parser, the parser filled the column with nulls.
- Partition columns are no longer required to be the last columns in the table definition.

See Partitioned file paths.

Machine learning

Isolation forest

Vertica now supports the isolation forest (iForest) algorithm. iForest is an unsupervised outlier detection algorithm that you can use for applications such as fraud detection and system health monitoring, or for removing outliers from your data. You can use the IFOREST function to train an iForest model and the APPLY_IFOREST function to apply the model to an input relation.

For an extended discussion on the iForest algorithm, see Isolation Forest.

Option to build LINEAR_REG and LOGISTIC_REG models without calculating intercept

You now have the option of training linear and logistic regression models without calculating an intercept. See LINEAR_REG and LOGISTIC_REG for details.

Management Console

Edit custom alerts

You can edit the following properties of an existing custom alert:

- Alert Name
- SQL Query
- Variable values

For details, see Custom alerts .

Revive an Eon Mode database on Microsoft Azure

Management Console supports reviving Eon Mode database clusters on Microsoft Azure.

Alternate filesystem layouts for reviving and scaling on AWS

On Amazon Web Services (AWS), you can use alternate filesystem layouts for the depot, temp, and catalog Vertica directories for the following actions:

- Reviving a database
- Creating a subcluster
- Scaling a subcluster

For details, see Amazon Web Services in MC and Subclusters in MC .

Projections

Partition range support for aggregate projections

Partition Ranges can now be specified for live aggregate and top-K projections.

SDK updates

Polymorphic aggregate functions

User-defined aggregate functions (UDAFs) can now be polymorphic. A polymorphic function can accept any number and type of arguments. For information on how to write a polymorphic function, see Creating a polymorphic UDx .

Complex types in the Python SDK

You can now read and write complex types in user-defined extensions written in Python. The complex types support includes arrays, rows, and combinations of both. For details, see Arguments and return values . For examples of a Python UDx utilizing complex types, see Python example: matrix multiplication and Python example: complex types JSON parser .

Security and authentication

Generic authentication errors

To improve security, Vertica no longer issues the following error when a user attempts to connect and authenticate:

```
Invalid username or password
```

In addition, Vertica no longer issues method-specific error messages, and all authentication failures will result in the same error message:

```
authentication failed for username "name"
```

This change may affect clients that decide to retry connection attempts based on the type of connection error code. One such example is vsql , which would previously attempt a plaintext connection if TLS failed (and TLSMODE was ENABLE), but not if vsql received an error code reporting incorrect user credentials.

Because the new, generic message and error code does not specify the cause for the failure, vsql cannot distinguish between a authentication error as a result of an invalid TLS configuration or invalid user credentials, and it will attempt to establish a plaintext connection in both cases.

Default authentication records

Vertica now creates three default authentication records and grants them to the public role. These authentication records have a priority of -1 , so all user-created authentication records take priority over these default records.

In previous versions of Vertica, if no authentication records were enabled, the following implicit authentication records were applied:

- Users without a password were authenticated with the trust method. This implicit authentication record continues to apply in 12.0.0 for users without a password.
- Users with a password were authenticated with the password method. This implicit authentication record has been removed in favor of the default records.

Fallthrough authentication

You can now allow authentication records to fall through to the next record (in order of priority) upon failure. For details, see Fallthrough authentication .

Upgrade behavior

In previous versions, this fallthrough behavior was only available for ident (could fall through to any other authentication method) and ldap (could only fall through to other ldap methods) authentication methods and this behavior could not be disabled. As of Vertica 12.0.0, this behavior is disabled by default for all new authentication records, including ident and ldap .

To preserve the behavior of existing databases that depended on ident 's fallthrough Vertica automatically enables fallthrough for ident authentication records on upgrade if all of the following are true:

- The database contains an ident authentication record.
- The *ident* authentication record has the highest priority .
- The database contains another user-defined authentication record.

Again, in previous versions, ldap records only fell through to other ldap records and skipped records that used other methods. This is no longer the case; ldap records are fallthrough-compatible with many other methods .

Therefore, to replicate old fallthrough behavior for ldap , your ldap records must be consecutive (in priority order) to fallthrough to each other.

SQL functions and statements

INFER_TABLE_DDL supports JSON

You can now use the INFER_TABLE_DDL function to produce candidate table definitions from JSON files. Because a JSON file does not have an explicit schema, the function inspects the data itself. Because JSON data can vary from record to record or file to file, the function can return more than one candidate definition. In the following example, differences are highlighted:

```
=> SELECT INFER_TABLE_DDL ('/data/*.json'
    USING PARAMETERS table_name='restaurants', format='json',
max_files=3, max_candidates=3);
WARNING 0:  This generated statement contains one or more float types which might lose precision
WARNING 0:  This generated statement contains one or more varchar/varbinary types which default to length 80

                INFER_TABLE_DDL
-------------------------------------------------------------------------------
 Candidate matched 1/2 of total files(s):
  create table "restaurants"(
    "cuisine" varchar,
    "location_city" Array[varchar],
    "menu" Array[Row(
      "item" varchar,
      "price" float
    )],
    "name" varchar
  );
Candidate matched 1/2 of total files(s):
  create table "restaurants"(
    "cuisine" varchar,
    "location_city" Array[varchar],
    "menu" Array[Row(
      "items" Array[Row(
        "item" varchar,
        "price" numeric
      )],
      "time" varchar
    )],
    "name" varchar
  );

(1 row)
```

Immutable tables

Immutable tables are insert-only tables in which existing data cannot be modified, regardless of user privileges. Updating row values and deleting rows are prohibited. Certain changes to table metadata—for example, renaming tables—are also prohibited, in order to prevent attempts to circumvent these restrictions.

You set an existing table to be immutable with ALTER TABLE :

```
ALTER TABLE table SET IMMUTABLE ROWS;
```

For details, see Immutable tables .

Users and privileges

LDAP link: user groups as default roles

By default, synchronizing LDAP users through the LDAP Link service automatically grants roles (derived from their LDAP groups) to the users. However, these are not default roles and therefore must be enabled manually with SET ROLE before they take effect.

The new LDAPLinkAddRolesAsDefault parameter (disabled by default) makes these roles default roles automatically:

```
=> ALTER DATABASE DEFAULT SET LDAPLinkAddRolesAsDefault = 1;
```

For details on this and other LDAP Link parameters, see LDAP link parameters .

Vertica on the cloud

AWS i4i, r6i, and c6i instance types now supported

Vertica now supports three i4i, two r6i, and one c6i EC2 instance types for use in database clusters. See Supported AWS instance types .

Deprecated and removed functionality

Vertica retires functionality in two phases:

- **Deprecated** : Vertica announces deprecated features and functionality in a major or minor release. Deprecated features remain in the product and are functional. Published release documentation announces deprecation on this page. When users access this functionality, it may return informational messages about its pending removal.
- **Removed** : Vertica removes a feature in a major or minor release that follows the deprecation announcement. Users can no longer access the functionality, and this page is updated to verify removal (see History , below). Documentation that describes this functionality is removed, but remains in previous documentation versions.

Deprecated

The following functionality was deprecated and will be removed in future versions:

Release	Functionality	Notes
12.0.4	Visual Studio 2012, 2013, and 2015 plug-ins and the Microsoft Connectivity Pack	
12.0.3	ADO.NET driver support for .NET 3.5	
	prometheus.createServiceMonitor Helm chart parameter	
	webhook.caBundle Helm chart parameter	Use webhook.tlsSecret instead. For details, see Helm chart parameters .
12.0.2	JDBC 4.0 and 4.1 support	
	Support for Visual Studio 2008 and 2010 plug-ins	
	CA bundles	Use TLS configurations instead.
	The following parameters for CREATE NOTIFIER and ALTER NOTIFIER : • TLSMODE • CA BUNDLE • CERTIFICATE	Use the TLS CONFIGURATION parameter instead. For details, see Creating a Kafka notifier .
	The TLSMODE PREFER parameter for CONNECT TO VERTICA .	Use the TLS CONFIGURATION parameter instead.
	cert-manager for Helm chart TLS configuration.	Use webhook.certSource parameter to generate certificates internally or provide custom certificates. See Helm chart parameters .
	webhook.caBundle Helm chart parameter .	Provide the certificate authority (CA) bundle in a Secret to webhook.tlsSecret .
12.0.0	hive_partition_cols parameter for PARQUET and ORC parsers	Use COPY...PARTITION COLUMNS instead. See Partitioned file paths .
	The following ODBC / JDBC OAuth parameters: • OAuthAccessToken/oauthaccesstoken • OAuthRefreshToken/oauthrefreshtoken • OAuthClientId/oauthclientid • OAuthClientSecret/oauthclientsecret • OAuthTokenUrl/oauthtokenurl • OAuthDiscoveryUrl/oauthdiscoveryurl • OAuthScope/oauthscope	Replaced by OAuthJsonConfig (ODBC) and oauthjsonconfig (JDBC).

Removed

The following functionality was removed:

Release	Functionality	Notes
12.0.4	ADO.NET driver support for .NET 3.5	The ADO.NET driver now requires the Microsoft .NET Framework 4.6 or higher. The Vertica Client Drivers and Tools for Windows prompts you to install .NET 4.6 if your system does not meet this requirement.
12.0.3	Support for Visual Studio 2008 and 2010 plug-ins	
	vsql support for macOS 10.12-10.14	vsql now requires macOS 10.15 or higher.
12.0.1	Internet Explorer 11	Internet Explorer 11 web browser is retired and out of support as of June 15, 2022.
12.0.0	macOS 10.12	ODBC and vsql client driver support
	AWS Library (S3EXPORT)	Replaced by File export .

History

The following functionality or support has been deprecated or removed as indicated:

Functionality	Component	Deprecated in:	Removed in:
Visual Studio 2012, 2013, and 2015 plug-ins and the Microsoft Connectivity Pack.	Client drivers	12.0.4	
vsql support for macOS 10.12-10.14	Client drivers		12.0.3
CA bundles	Security	12.0.2	
The following parameters for CREATE NOTIFIER and ALTER NOTIFIER : • TLSMODE • CA BUNDLE • CERTIFICATE	Security	12.0.2	
The TLSMODE PREFER parameter for CONNECT TO VERTICA .	Security	12.0.2	
JDBC 4.0 and 4.1 support	Client drivers	12.0.2	
Support for Visual Studio 2008 and 2010 plug-ins	Client drivers	12.0.2	12.0.3
Internet Explorer 11 support	Management Console		12.0.1
ODBC support for macOS 10.12-10.14	Client drivers		12.0

The following ODBC / JDBC OAuth parameters: • OAuthAccessToken/oauthaccesstoken • OAuthRefreshToken/oauthrefreshtoken • OAuthClientId/oauthclientid • OAuthClientSecret/oauthclientsecret • OAuthTokenUrl/oauthtokenurl • OAuthDiscoveryUrl/oauthdiscoveryurl • OAuthScope/oauthscope	Client drivers	12.0	
hive_partition_cols parameter for PARQUET and ORC parsers	Server	12.0	
The following ODBC / JDBC OAuth parameters: • OAuthAccessToken/oauthaccesstoken • OAuthRefreshToken/oauthrefreshtoken • OAuthClientId/oauthclientid • OAuthClientSecret/oauthclientsecret • OAuthTokenUrl/oauthtokenurl • OAuthDiscoveryUrl/oauthdiscoveryurl • OAuthScope/oauthscope	Client drivers	12.0	
INFER_EXTERNAL_TABLE_DDL function	Server	11.1.1	
Admission Controller Webhook image	Kubernetes	11.0.1	11.0.2
Admission Controller Helm chart	Kubernetes	11.0.1	
Shared DATA and DATA,TEMP storage locations	Server	11.0.1	
DESIGN_ALL option for EXPORT_CATALOG()	Server	11.0	
HDFSUseWebHDFS configuration parameter and LibHDFS++	Server	11.0	
INFER_EXTERNAL_TABLE_DDL (path, table) syntax	Server	11.0	11.1.1
AWS library functions: • AWS_GET_CONFIG • AWS_SET_CONFIG • S3EXPORT • S3EXPORT_PARTITION	Server	11.0	12.0
Vertica Spark connector V1	Client	11.0	
admintools db_add_subcluster --is-secondary argument	Server	11.0	
Red Hat Enterprise Linux/CentOS 6.x	Server	10.1.1	11.0
STRING_TO_ARRAY(array,delimiter) syntax	Server	10.1.1	
Vertica JDBC API com.vertica.jdbc.kv package	Client Drivers	10.1	
ARRAY_CONTAINS function	Server	10.1	

Client-server TLS parameters: • SSLCertificate • SSLPrivateKey • SSLCA • EnableSSL LDAP authentication parameters: • tls_key • tls_cert • tls_cacert • tls_reqcert LDAPLink and LDAPLink dry-run parameters: • LDAPLinkTLSCACert • LDAPLinkTLSCADir • LDAPLinkStartTLS • LDAPLinkTLSReqCert	Server	10.1	11.0
MD5 hashing algorithm for user passwords	Server	10.1	
Reading structs from ORC files as expanded columns	Server	10.1	11.0
vbr configuration section [S3] and S3 configuration parameters	Server	10.1	
flatten_complex_type_nulls parameter to the ORC and Parquet parsers	Server	10.1	11.0
System table WOS_CONTAINER_STORAGE	Server	10.0.1	11.0.2
skip_strong_schema_match parameter to the Parquet parser	Server	10.0.1	10.1
Specifying segmentation on specific nodes	Server	10.0.1	
DBD meta-function DESIGNER_SET_ANALYZE_CORRELATIONS_MODE	Server	10.0.1	11.0.1
Meta-function ANALYZE_CORRELATIONS	Server	10.0	
Eon Mode meta-function BACKGROUND_DEPOT_WARMING	Server	10.0	
Reading structs from Parquet files as expanded columns	Server	10.0	10.1
Eon Mode meta-functions: • SET_DEPOT_PIN_POLICY • CLEAR_DEPOT_PIN_POLICY	Server	10.0	10.1
vbr configuration parameter SnapshotEpochLagFailureThreshold	Server	10.0	
Array-specific functions: • array_min • array_max • array_sum • array_avg	Server	10.0	10.1
DMLTargetDirect configuration parameter	Server	10.0	
HiveMetadataCacheSizeMB configuration parameter	Server	10.0	10.1

MoveOutInterval	Server	10.0	
MoveOutMaxAgeTime	Server	10.0	
MoveOutSizePct	Server	10.0	
Windows 7	Client Drivers		9.3.1
DATABASE_PARAMETERS admintools command	Server	9.3.1	
Write-optimized store (WOS)	Server	9.3	10.0
7.2_upgrade vbr task	Server	9.3	
DropFailedToActivateSubscriptions configuration parameter	Server	9.3	10.0
--skip-fs-checks	Server	9.2.1	
32-bit ODBC Linux and OS X client drivers	Client	9.2.1	9.3
Vertica Python client	Client	9.2.1	10.0
macOS 10.11	Client	9.2.1	
DisableDirectToCommunalStorageWrites configuration parameter	Server	9.2.1	
CONNECT_TO_VERTICA meta-function	Server	9.2.1	9.3
ReuseDataConnections configuration parameter	Server	9.2.1	9.3
Network interfaces (superseded by network addresses)	Server	9.2	
Database branching	Server	9.2	10.0
KERBEROS_HDFS_CONFIG_CHECK meta-function	Server	9.2	
Java 5 support	JDBC Client	9.2	9.2.1
Configuration parameters for enabling projections with aggregated data: • EnableExprsInProjections • EnableGroupByProjections • EnableTopKProjections • EnableUDTProjections	Server	9.2	
DISABLE_ELASTIC_CLUSTER()	Server	9.1.1	11.0
eof_timeout parameter of KafkaSource	Server	9.1.1	9.2
Windows Server 2012	Server	9.1.1	
Debian 7.6, 7.7	Client driver	9.1.1	9.2.1
IdolLib function library	Server	9.1	9.1.1
SSL certificates that contain weak CA signatures such as MD5	Server	9.1	
HCatalogConnectorUseLibHDFSPP configuration parameter	Server	9.1	

S3 UDSource	Server	9.1	9.1.1
HCatalog Connector support for WebHCat	Server	9.1	
partition_key column in system tables STRATA and STRATA_STRUCTURES	Server	9.1	10.0.1
Vertica Pulse	Server	9.0.1	9.1.1
Support for SQL Server 2008	Server	9.0.1	9.0.1
SUMMARIZE_MODEL meta-function	Server	9.0	9.1
RestrictSystemTable parameter	Server	9.0.1	
S3EXPORT multipart parameter	Server	9.0	
EnableStorageBundling configuration parameter	Server	9.0	
Machine Learning for Predictive Analytics package parameter key_columns for data preparation functions.	Server	9.0	9.0.1
DROP_PARTITION meta-function, superseded by DROP_PARTITIONS	Server	9.0	
Machine Learning for Predictive Analytics package parameter owner .	Server	8.1.1	9.0
Backup and restore --setupconfig command	Server	8.1	9.1.1
SET_RECOVER_BY_TABLE meta-function. Do not disable recovery by table.	Server	8.0.1	
Column rebalance_projections_status.duration_sec	Server	8.0	
HDFS Connector	Server	8.0	9.0
Prejoin projections	Server	8.0	9.2
Administration Tools option --compat21	Server	7.2.1	
admin_tools -t config_nodes	Server	7.2	11.0.1
Projection buddies with inconsistent sort order	Server	7.2	9.0
backup.sh	Server	7.2	9.0
restore.sh	Server	7.2	9.0
copy_vertica_database.sh	Server	7.2	
JavaClassPathForUDx configuration parameter	Server	7.1	
ADD_LOCATION meta-function	Server	7.1	
bwlimit configuration parameter	Server	7.1	9.0
vbr configuration parameters retryCount and retryDelay	Server	7.1	11.0
EXECUTION_ENGINE_PROFILE counters: file handles, memory allocated	Server	7.0	9.3
EXECUTION_ENGINE_PROFILES counter memory reserved	Server	7.0	

MERGE_PARTITIONS() meta-function	Server	7.0	
krb5 client authentication method **Note** Use the Kerberos gss method for client authentication, instead of krb5.	All clients	7.0	
range-segmentation-clause	Server	6.1.1	9.2
scope parameter of meta-function CLEAR_PROFILING	Server	6.1	
Projection creation type IMPLEMENT_TEMP_DESIGN	Server, clients	6.1	

Getting started

Welcome to Getting Started. This guide walks you through the process of configuring a Vertica Analytics Platform database and running typical queries.

For short tutorial on how to install Vertica, create a database, and load data, see the Quickstart guide .

In this section

- Using this guide
- Quickstart guide
- Vertica community edition (CE)
- Vertica interfaces
- Introducing the VMart example database
- Installing and connecting to the VMart example database
- Querying data
- Backing up and restoring the database
- Using Database Designer to create a comprehensive design
- Restoring the status of your host
- Appendix: VMart example database schema, tables, and scripts

Using this guide

This guide shows how to set up a Vertica database and run simple queries that perform common database tasks.

Who should use this guide?

This guide targets anyone who wants to learn how to create and run a Vertica database. This guide requires no special knowledge at this point, although a rudimentary knowledge of basic SQL commands is useful when you begin to run queries.

For short tutorial on how to install Vertica, create a database, and load data, see the Quickstart guide .

What you need

The examples in this guide require one of the following:

- Vertica installed on one host or a cluster of hosts. Vertica recommends a minimum of three hosts in the cluster.
- Vertica installed on a virtual machine (VM).

For further instructions about installation, see Installing Vertica .

Accessing your database

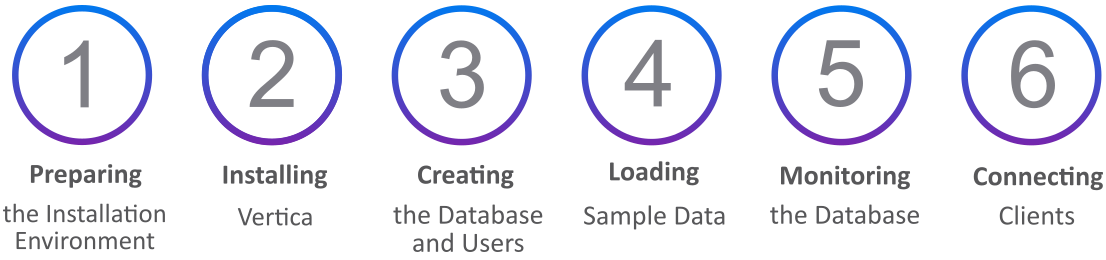
You access your database with an SSH client or the terminal utility in your Linux console, such as vsql. Throughout this guide, you use the following user interfaces:

- Linux command line (shell) interface
- Administration Tools
- vsql client interface
- Management Console

Quickstart guide

This section contains a short guide to setting up an installation environment for Vertica, loading data, and using various client drivers.

Examples in the documentation use $ to denote a terminal prompt and => to denote a vsql prompt.



In this section

- Preparing the installation environment
- Installing Vertica
- Creating a database and users
- Loading sample data
- Monitoring the database
- Connecting clients

Preparing the installation environment

Before installing Vertica, you must configure your environment.

To run Vertica Enterprise on-premises, follow the numbered instructions below.

To run the Vertica in a Virtual Machine instead, see Vertica community edition (CE) .

1. Copy the installation file to your home directory. The example shows an rpm file for CentOS/RHEL, but you may have a deb file for Debian.

   ```
   $ scp vertica-10.1.0.x86_64.RHEL6.rpm /~
   ```

2. Identify the IP address of the current node.

   ```
   $ ipconfig -a
   ```

   If the ipconfig command is not found in your path, you can try running it directly using the paths /sbin/ipconfig or /usr/sbin/ipconfig . If neither of those work, use the ip command:

   ```
   $ ip a
   ```

   > **Note**
   > The previous commands can return multiple addresses. For example, if your system is configured to use both IPv4 and IPv6 addressing, the commands will list two addresses, one for each address family. In this case, you must determine which address you want to use.

3. Ensure your packages are up to date. Run the command based on your distribution.
   On CentOS and RedHat:

   ```
   $ sudo yum update -y
   ```

   On openSUSE:

   ```
   $ sudo zypper up
   ```

   On Debian and Ubuntu:

   ```
   $ sudo apt-get update && sudo apt-get upgrade
   ```

4. Set swappiness to 0 (recommended).

   ```
   $ sudo systemctl vm.swappiness=0
   ```

5. Verify that SELinux is running in permissive mode or is disabled.

   ```
   $ sudo setenforce 0
   ```

6. Disable the system firewall.

```
$ sudo systemctl mask firewalld
$ sudo systemctl disable firewalld
$ sudo systemctl stop firewalld
```

7. Install Vertica .

Installing Vertica

1. To install from the binary, run the command based on your distribution.
   On CentOS, RedHat, and openSUSE:
   ```
   $ sudo rpm -Uvh vertica-10.1.0.x86_64.RHEL6.rpm
   ```
   On Debian and Ubuntu:
   ```
   $ sudo dpkg -i vertica-10.1.0.x86_64.deb
   ```
2. Run the installation script. The following command specifies the localhost, the rpm, a database admin, and home directory.
   ```
   $ sudo /opt/vertica/sbin/install_vertica -s localhost -r vertica-10.1.0.x86_64.RHEL6.rpm
   -u dbadmin -g dbadmin -d /home/dbadmin -p vertica -L -Y
   ```
3. Switch to the newly created dbadmin user.
   ```
   $ su dbadmin
   ```
4. Run admintools and accept the EULA and operating license.
   ```
   $ admintools
   ```
5. Creating a database and users .

Creating a database and users

The admintools utility included in the installation provides a number of administrative functions. The following steps show how to create a database and users with this utility.

1. View the status of your cluster. It should return an empty table.
   ```
   $ admintools -t view_cluster
   ```
   ```
    DB | Host | State
   ----+------+-------
   ```
2. Create a database called "vdb" in the home directory with the password "vertica". This command also sets the plaintext password "vertica" for both the database and dbadmin.
   ```
   $ admintools -t create_db --data_path=/home/dbadmin --catalog_path=/home/dbadmin --database=vdb --password=vertica --hosts=localhost
   ```
3. Run vsql and enter "vertica" at the password prompt.
   ```
   $ vsql
   ```
4. Create a user named "Mike" with the password "inventor."
   ```
   => CREATE USER Mike IDENTIFIED BY 'inventor';
   ```
5. Grant the USAGE permission on the public schema.
   ```
   => GRANT USAGE ON SCHEMA PUBLIC TO Mike;
   ```
6. Load sample data .

Loading sample data

Vertica offers several solutions for loading files with structured and unstructred data, and from several formats.

Creating a sample data file

Create a sample CSV file called cities.csv with the following contents and save it to /home/dbadmin/cities.csv.

```
City,State,Zip,Population
Boston,MA,02108,694583
Chicago,IL,60601,2705994
Seattle,WA,98101,744955
Dallas,TX,75201,1345047
New York,NY,10001,8398748
```

Loading structured data from a file

1. Run vsql.
```
$ vsql
```
2. Create the cities table.
```
=> CREATE TABLE cities (
city       varchar(20),
state      char(2),
zip        int,
population int
);
```
3. Use the COPY statement to load the data from the cities.csv file. The following command logs exceptions and rejections in the home directory.
```
=> COPY cities FROM LOCAL '/home/dbadmin/cities.csv' DELIMITER ',' NULL '' EXCEPTIONS '/home/dbadmin/cities_exceptions.log'
REJECTED DATA '/home/dbadmin/cities_rejections.log';
```
4. Review the rejections log for what data was excluded. Here, the header was excluded.
```
$ cat /home/dbadmin/cities_rejections.log

City,State,Zip,Population
```
5. Review the exceptions for details on the error. In this case, the header failed Vertica's integer data type verification.
```
$ cat /home/dbadmin/cities_exceptions.log

COPY: Input record 1 has been rejected (Invalid integer format 'Zip' for column 3 (zip)).
Please see /home/dbadmin/cities_rejections.log, record 1 for the rejected record. This record was record 1 from cities.csv
```
6. To fix this, add SKIP 1 to the original COPY statement. This excludes the first row.
```
=> COPY cities FROM LOCAL '/home/dbadmin/cities.csv' DELIMITER, 'NULL'
EXCEPTIONS '/home/dbadmin/cities_exceptions.log'
REJECTED DATA '/home/dbadmin/cities_rejections.log' SKIP 1;
```

Loading unstructured data with flex tables

To load data from another source, Vertica uses Flex tables. Flex tables simplify data loading by allowing you to load unstructured or "semi-structured" data without having to create a schema or column definitions.

Supported formats include:

- Avro Data
- CEF
- CSV
- Delimited
- JSON

1. Create a table called cities_flex. Notice how it does not include column names or data types.
```
=> CREATE FLEXIBLE TABLE cities_flex();
```
2. Load the CSV file into the table.
```
=> COPY cities_flex FROM '/source/cities.csv' PARSER FDELIMITEDPARSER (delimiter=',');
```
3. Query the cities_flex table, specifying the column names from the original CSV file.
```
=> SELECT city, state FROM cities_flex;
```

Monitoring the database

This page includes a collection of general-purpose SQL statements useful for monitoring your database.

Check disk space

Check disk space used by tables.

```
=> SELECT projection_schema, anchor_table_name, to_char(sum(used_bytes)/1024/1024/1024,'999,999.99')
as disk_space_used_gb FROM
projection_storage
GROUP by projection_schema, anchor_table_name ORDER by
disk_space_used_gb desc limit 50;
```

Check total disk space used.

```
=> SELECT to_char(sum(used_bytes)/1024/1024/1024,'999,999.99') AS gb FROM projection_storage;
```

Check the amount of free disk space.

```
=> SELECT to_char(sum(disk_space_free_mb)/1024,'999,999,999') AS
disk_space_free_gb, to_char(sum(disk_space_used_mb)/1024,'999,999,999') AS
disk_space_used_gb FROM disk_storage;
```

Adjust data types

Change the Zip and Population columns from VARCHAR to INT.

```
=> UPDATE cities_flex_keys set data_type_guess='int' WHERE key_name='Zip';
=> UPDATE cities_flex_keys set data_type_guess='int' WHERE key_name='Population';
=> COMMIT;
```

Refresh the cities_flex_view with the new data types

```
=> SELECT build_flextable_view('cities_flex');
```

Materialize the flex table

Materialize the flex table and all columns into a persistent Vertica table.

```
=> CREATE TABLE cities AS SELECT * from cities_flex_view;
```

View user and role information

View user information.

```
=> SELECT user_name, is_super_user, resource_pool, memory_cap_kb, temp_space_cap_kb, run_time_cap FROM users;
```

Identify users.

```
=> SELECT * FROM user_sessions;
```

View queries by user.

```
=> SELECT * FROM query_profiles WHERE user_name ILIKE '%dbadmin%';
```

View roles.

```
=> SELECT * FROM roles;
```

View database information

View resource pool assignments.

```
=> SELECT user_name, resource_pool FROM users;
```

View table information.

```
=> SELECT table_name, is_flextable, is_temp_table, is_system_table, count(*) FROM tables GROUP by 1,2,3,4;
```

View projection information.

```
=> SELECT is_segmented, is_aggregate_projection, has_statistics, is_super_projection, count(*) FROM projections GROUP by 1,2,3,4,5;
```

View update information.

```
=> SELECT substr(query, 0, instr(query, '')+1) count(*) FROM (SELECT transaction_id, statement_id, upper(query::varchar(30000)) as query FROM query_profiles
WHERE regexp_like(query,'^\s*update\s','i')) sq GROUP BY 1 ORDER BY 1;
```

View active events.

```
=> SELECT * FROM active_events WHERE event_problem_description NOT ILIKE '%state to UP';
```

View backups.

```
=> SELECT * FROM database_backups;
```

View disk storage.

```
=> SELECT node_name, storage_path, storage_usage, storage_status, disk_space_free_percent FROM disk_storage;
```

View long-running queries

```
=> SELECT query_duration_us/1000000/60 AS query_duration_mins, table_name, user_name, processed_row_count AS rows_processed,
substr(query,0,70) FROM query_profiles
ORDER BY query_duration_us DESCLIMIT 250;
```

View sizes and counts of Read Optimized Store (ROS) containers.

```
=> SELECT node_name, projection_name, sum(ros_count), sum(ros_used_bytes) FROM projection_storage GROUP BY 1,2 HAVING sum(ros_count) >= 50
ORDER BY 3 DESC LIMIT 250;
```

View license information

View license consumption.

```
=> SELECT GET_COMPLIANCE_STATUS();
```

View how the database complies with your license.

```
=> SELECT AUDIT('');
```

Audit the database to check if it complies with raw storage allowance of your license.

```
=> SELECT AUDIT_LICENSE_SIZE;
```

Compare storage size of database the database and your license.

```
=> SELECT /*+(license_utilization)*/
audit_start_timestamp,
database_size_bytes / (1024^3) AS database_size_gb,
license_size_bytes / (1024^3) AS license_size_gb, usage_percent
FROM v_catalog.license_audits ORDER BYaudit_start_timestamp DESC LIMIT 30;
```

Connecting clients

Vertica supports several third-party clients. A list of Vertica client drivers can be found here .

Connecting to DbVisualizer

1. Download the DbVisualizer client application .
2. Create a database. Database Menu -> Create Database Connection.
3. Specify a name for the connection.
4. In the "Driver (JDBC)" field, specify Vertica.
5. In the "Database Server" field, specify an IP address.
6. In the "Database Port" field, specify a port number.
7. In the "Database Name" field, specify a database name.
8. In the "Database Userid" field, specify a username.
9. In the "Database Password" field, specify a password.
10. Use the "ping" function to test the connection.

Connecting to tableau

1. Download Tableau .

2. Open Tableau Desktop.
3. Select Server Connection.
4. Select Vertica as the server type.
5. Set the Server IP.
6. Set the Port to "vdb".
7. Sign into the database.

Vertica community edition (CE)

The Vertica Community Edition (CE) is a free, limited license that Vertica provides users so that they can get a hands-on introduction to the platform. It allows you to deploy up to three nodes using a maximum of 1TB of data.

As part of the CE license, you agree to the collection of some anonymous, non-identifying usage data. This data lets Vertica understand how customers use the product, and helps guide the development of new features. None of your personal data is collected. For details on what is collected, see the Community Edition End User License Agreement .

Vertica provides two options to use the Community Edition:

- CE container image. Container images require a container engine such as Docker Desktop. For installation details, see the official Docker documentation .
- Vertica Community Edition Virtual Machine (Vertica CE VM), which is available for download on the Vertica website .

CE container image

The CE image is a single-node, lightweight alternative to the Vertica CE VM. Vertica provides two options to get the CE container image:

- Pull the image from the Vertica DockerHub registry .
- Build a custom CE image using the one-node-ce GitHub repository . This requires the free CE trial RPM .

The CE container environment includes the following:

- VMart example database
- admintools
- vsql
- Developer libraries

Important
Use the Vertica CE container image with the following limitations:

- There is a two-day verification period before CE licenses are issued.
- CE images expire after 1 year.

For more information about Vertica licensing, see Managing licenses .

Vertica CE VM

The Vertica CE VM is a preconfigured Linux environment that includes:

- Vertica Community Edition with the VMart example database
- Management Console
- admintools
- vsql
- A tutorial that guides you through a series of common tasks

> **Note**
> The Vertica CE VM is not supported for production use.

For a preview of the tutorial included in the Vertica CE VM, see the Vertica CE VM User Guide .

To download and install the Vertica CE VM, follow the instructions in the Vertica CE VM Installation Guide .

Vertica interfaces

Vertica provides tools to perform administrative tasks quickly and easily:

- Management Console (MC) provides a unified view of your Vertica cluster through a browser connection.
- Administration Tools provides a simple graphical user interface for you to perform certain tasks such as starting and stopping a database, running Database Designer, and more.

The following sections provide detailed information about both tools.

Management Console

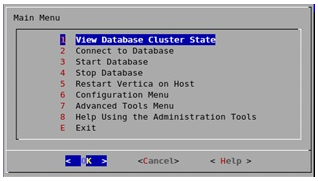
Management Console (MC) is the Vertica in-browser monitoring and management tool. Its graphical user interface provides a unified view of your Vertica database operations. Through user-friendly, step-by-step screens, you can create, configure, manage, and monitor your Vertica databases and their associated clusters. You can use MC to operate your Vertica database in Eon Mode or in Enterprise Mode. You can use MC to provision and deploy a Vertica Eon Mode database.

For detailed instructions, see Management Console .

Administration tools

If possible, always run the Administration Tools using the database administrator account (dbadmin) on the administration host.

When you run Administration Tools, the **Main Menu** dialog box appears with a dark blue background and a title on top. The screen captures used in this documentation set are cropped down to the dialog box itself, as shown in the following screenshot.



The Administration Tools interface responds to mouse clicks in some terminal windows, but it might respond only to keystrokes. The following table is a quick reference to keystroke usage in the Administration Tools interface:

Key	**Action**
Return	Run selected command.
Tab	Cycle between **OK, Cancel, Help** , and menu.
Up/Down Arrow	Move cursor up and down in menu, window, or help file.
Space	Select item in list.
Character	Select corresponding command from menu.

For details, see Using the Administration Tools in the Administrator's Guide.

After your first login

The first time you log in as the database administrator and run the Administration Tools, complete the following steps:

1. Accept the end-user license agreement (EULA) to proceed.
   A window displays, requesting the location of the license key file you downloaded from the OpenText website. The default path is /tmp/vlicense.dat .
2. Enter the absolute path to your license key and select **OK** .
3. To return to the command line, select **Exit** and click **OK** .

Introducing the VMart example database

Vertica ships with a sample multi-schema database called the VMart Example Database, which represents a database that might be used by a large supermarket (VMart) to access information about its products, customers, employees, and online and physical stores. Using this example, you can create, run, optimize, and test a multi-schema database.

The VMart database contains the following schema:

- public (automatically created in any newly created Vertica database)
- store
- online_Sales

VMart database location and scripts

If you installed Vertica from the RPM package, the VMart schema is installed in the /opt/vertica/examples/VMart_Schema directory. This folder contains the following script files that you can use to get started quickly. Use the scripts as templates for your own applications.

Script/file name	Description
vmart_count_data.sql	SQL script that counts rows of all example database tables, which you can use to verify load.
vmart_define_schema.sql	SQL script that defines the logical schema for each table and referential integrity constraints.
vmart_gen.cpp	Data generator source code (C++).
vmart_gen	Data generator executable file.
vmart_load_data.sql	SQL script that loads the generated sample data to the corresponding tables using COPY.
vmart_ queries.sql	SQL script that contains concatenated sample queries for use as a training set for the Database Designer.
vmart_query_##.sql	SQL scripts that contain individual queries; for example, vmart_query_01 through vmart_query_09.sql
vmart_schema_drop.sql	SQL script that drops all example database tables.

For more information about the schema, tables, and queries included with the VMart example database, see the Appendix .

Installing and connecting to the VMart example database

Follow the steps in this section to create the fully functioning, multi-schema VMart example database to run sample queries. The number of example databases you create within a single Vertica installation is limited only by the disk space available on your system. However, Vertica strongly recommends that you start only one example database at a time to avoid unpredictable results.

Vertica provides two options to install the example database:

- Quick Installation Using a Script : This option lets you create the example database and start using it immediately. Use this method to bypass the schema and table creation processes and start querying immediately.
- Advanced Installation . The advance option is an advanced-but-simple example database installation using the Administration Tools interface. Use this method to better understand the database creation process and practice creating a schema, creating tables, and loading data.

> **Note**
> Both installation methods create a database named VMart. If you try both installation methods, you need to drop the VMart database you created (see Restoring the Status of Your Host) or create the subsequent database with a new name. However, Vertica strongly recommends that you start only one example database at a time to avoid unpredictable results
>
> This tutorial uses Vertica-provided queries, but if you create your own design and use your own queries, you can follow the same set of procedures.

In this section

- Quick installation using a script
- Advanced installation

Quick installation using a script

The script you need to perform a quick installation is located in /opt/vertica/sbin and is called install_example . This script creates a database on the default port (5433), generates data, creates the schema and a default superprojection, and loads the data. The folder also contains a delete_example script, which stops and drops the database.

1. In a terminal window, log in as the database administrator.
   ```
   $ su dbadmin
   Password: (your password)
   ```

2. Change to the /examples directory.
   $ cd /opt/vertica/examples
3. Run the install script:
   $ /opt/vertica/sbin/install_example VMart

After installation, you should see the following:

```
[dbadmin@localhost examples]$ /opt/vertica/sbin/install_example VMart
Installing VMart example example database
Mon Jul 22 06:57:40 PDT 2013
Creating Database
Completed
Generating Data. This may take a few minutes.
Completed
Creating schema
Completed
Loading 5 million rows of data. Please stand by.
Completed
Removing generated data files
Example data
```

The example database log files, ExampleInstall.txt and ExampleDelete.txt , are written to /opt/vertica/examples/log .

To start using your database, continue to Connecting to the Database in this guide. To drop the example database, see Restoring the Status of Your Host in this guide.

Advanced installation

To perform an advanced-but-simple installation, set up the VMart example database environment and then create the database using the Administration Tools or Management Console.

> **Note**
> If you installed the VMart database using the quick installation method, you cannot complete the following steps because the database has already been created.
>
> To try the advanced installation, drop the example database (see Restoring the Status of Your Host on this guide) and perform the advanced Installation, or create a new example database with a different name. However, Vertica strongly recommends that you install only one example database at a time to avoid unpredictable results.

The advanced installation requires the following steps:

In this section

- Step 1: setting up the example environment
- Step 2: creating the example database
- Step 3: connecting to the database
- Step 4: defining the database schema
- Step 5: loading data

Step 1: setting up the example environment

1. Stop all databases running on the same host on which you plan to install your example database.
   If you are unsure if other databases are running, run the Administration Tools and select **View Cluster State** . The State column should show DOWN values on pre-existing databases.
   If databases are running, click **Stop Database** in the **Main Menu** of the Administration Tools interface and click **OK** .
2. In a terminal window, log in as the database administrator:
   ```
   $ su dbadmin
   Password:
   ```
3. Change to the /VMart_Schema directory.
   ```
   $ cd /opt/vertica/examples/VMart_Schema
   ```

Do not change directories while following this tutorial. Some steps depend on being in a specific directory.

4. Run the sample data generator.

```
$ ./vmart_gen
```

5. Let the program run with the default parameters, which you can review in the README file.

```
Using default parameters
datadirectory = ./
numfiles = 1
seed = 2
null = ' '
timefile = Time.txt
numfactsalesrows = 5000000
numfactorderrows = 300000
numprodkeys = 60000
numstorekeys = 250
numpromokeys = 1000
numvendkeys = 50
numcustkeys = 50000
numempkeys = 10000
numwarehousekeys = 100
numshippingkeys = 100
numonlinepagekeys = 1000
numcallcenterkeys = 200
numfactonlinesalesrows = 5000000
numinventoryfactrows = 300000
gen_load_script = false
Data Generated successfully !
Using default parameters
datadirectory = ./
numfiles = 1
seed = 2
null = ' '
timefile = Time.txt
numfactsalesrows = 5000000
numfactorderrows = 300000
numprodkeys = 60000
numstorekeys = 250
numpromokeys = 1000
numvendkeys = 50
numcustkeys = 50000
numempkeys = 10000
numwarehousekeys = 100
numshippingkeys = 100
numonlinepagekeys = 1000
numcallcenterkeys = 200
numfactonlinesalesrows = 5000000
numinventoryfactrows = 300000
gen_load_script = false
Data Generated successfully !
```

6. If the vmart_gen executable does not work correctly, recompile it as follows, and run the sample data generator script again.

```
$ g++ vmart_gen.cpp -o vmart_gen
$ chmod +x vmart_gen
$ ./vmart_gen
```

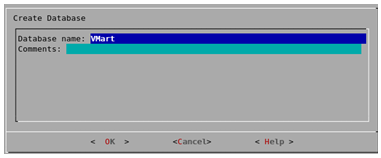
Step 2: creating the example database

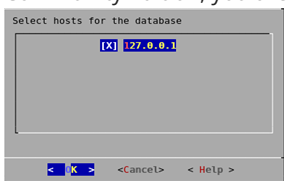
To create the example database: use the Administration Tools or Management Console, as described in this section.

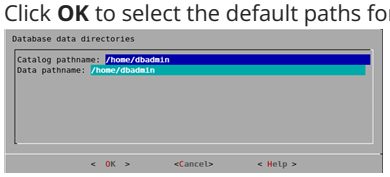
Creating the example database using the administration tools

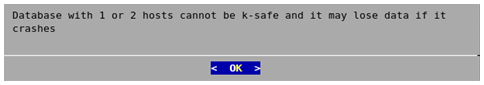
In this procedure, you create the example database using the Administration Tools. To use the Management Console, go to the next section.

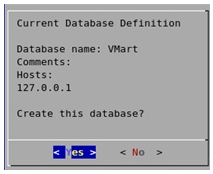
> **Note**
> If you have not used Administration Tools before, see Vertica interfaces .

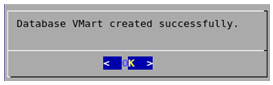
1. Run the Administration Tools.
   ```
   $ /opt/vertica/bin/admintools
   ```
   or simply type admintools
2. From the Administration Tools **Main Menu** , click **Configuration Menu** and click **OK** .
3. Click **Create Database** and click **OK** .
4. Name the database VMart and click **OK** .
   

5. Click **OK** to bypass the password and click **Yes** to confirm.
   There is no need for a database administrator password in this tutorial. When you create a production database, however, always specify an administrator password. Otherwise, the database is permanently set to trust authentication (no passwords).
6. Select the hosts you want to include from your Vertica cluster and click **OK** .
   This example creates the database on a one-host cluster. Vertica recommends a minimum of three hosts in the cluster. If you are using the Vertica Community Edition, you are limited to three nodes.
   

7. Click **OK** to select the default paths for the data and catalog directories.
   

   - Catalog and data paths must contain only alphanumeric characters and cannot have leading space characters. Failure to comply with these restrictions could result in database creation failure.
   - When you create a production database, you'll likely specify other locations than the default. See Prepare Disk Storage Locations in the Administrator's Guide for more information.
8. Since this tutorial uses a one-host cluster, a K-safety warning appears. Click **OK** .
   

9. Click **Yes** to create the database.
   

   During database creation, Vertica automatically creates a set of node definitions based on the database name and the names of the hosts you selected and returns a success message.
10. Click **OK** to close the **Database VMart created successfully** message.
    

Creating the example database using Management Console

In this procedure, you create the example database using Management Console. To use the Administration Tools, follow the steps in the preceding section.

> **Note**
> To use Management Console, the console should already be installed and you should be familiar with its concepts and layout. For details, see

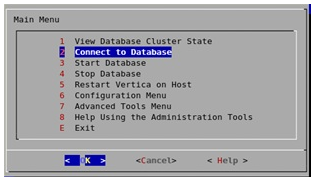
> Management Console .

1. Connect to Management Console and log in.
2. On the Home page, click **Infrastructure** to go to the Databases and Clusters page.
3. Click to select the appropriate existing cluster and click **Create Database** .
4. Follow the on-screen wizard, which prompts you to provide the following information:
   - Database name, which must be between 3–25 characters, starting with a letter, and followed by any combination of letters, numbers, or underscores.
   - (Optional) database administrator password for the database you want to create and connect to.
   - IP address of a node in your database cluster, typically the IP address of the administration host.
5. Click **Next** .

Step 3: connecting to the database

Regardless of the installation method you used, follow these steps to connect to the database.

1. As dbadmin , run the Administration Tools.
   $ /opt/vertica/bin/admintools
   or simply type admintools .
2. If you are already in the Administration Tools, navigate to the Main Menu page.
3. Select **Connect to Database** , click **OK** .



   To configure and load data into the VMart database, complete the following steps:
   - Step 4: defining the database schema
   - Step 5: loading data

   If you installed the VMart database using the Quick Installation method, the schema, tables, and data are already defined. You can choose to drop the example database (see Restoring the Status of Your Host in this guide) and perform the Advanced Installation, or continue straight to Querying Your Data in this guide.

Step 4: defining the database schema

The VMart database installs with sample scripts with SQL commands that are intended to represent queries that might be used in a real business. The vmart_define_schema.sql script runs a script that defines the VMart schema and creates tables. You must run this script before you load data into the VMart database.

This script performs the following tasks:

- Defines two schemas in the VMart database schema: *online_sales* and *store* .
- Defines tables in both schemas.
- Defines constraints on those tables.

```
Vmart=> \i vmart_define_schema.sql
CREATE SCHEMA
CREATE SCHEMA
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
ALTER TABLE
CREATE TABLE
CREATE TABLE
ALTER TABLE
CREATE TABLE
ALTER TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
ALTER TABLE
```

Step 5: loading data

Now that you have created the schemas and tables, you can load data into a table by running the vmart_load_data.sql script. This script loads data from the 15 .tbl text files in /opt/vertica/examples/VMart_Schema into the tables that vmart_design_schema.sql created.

It might take several minutes to load the data on a typical hardware cluster. Check the load status by monitoring the vertica.log file, as described in Monitoring Log Files in the Administrator's Guide.

```
VMart=> \i vmart_load_data.sql
Rows Loaded
-------------
1826
(1 row)
Rows Loaded
-------------
60000
(1 row)
Rows Loaded
-------------
250
(1 row)
Rows Loaded
-------------
1000
(1 row)
Rows Loaded
-------------
50
(1 row)
Rows Loaded
-------------
50000
(1 row)
Rows Loaded
-------------
10000
(1 row)
Rows Loaded
-------------
100
(1 row)
Rows Loaded
-------------
100
(1 row)
Rows Loaded
-------------
1000
(1 row)
Rows Loaded
-------------
200
(1 row)
Rows Loaded
-------------
5000000
(1 row)
Rows Loaded
-------------
300000
(1 row)
VMart=>
```

Querying data

The VMart database installs with sample scripts that contain SQL commands that represent queries that might be used in a real business. Use basic SQL commands to query the database, or try out the following command. Once you're comfortable running the example queries, you might want to write your own.

> **Note**
> The data that your queries return might differ from the example output shown in this guide because the sample data generator is random.

Type the following SQL command to return the values for five products with the lowest fat content in the Dairy department. The command selects the fat content from Dairy department products in the product_dimension table in the public schema, orders them from low to high and limits the output to the first five (the five lowest fat contents).

```
VMart => SELECT fat_content
      FROM ( SELECT DISTINCT fat_content
           FROM product_dimension
           WHERE department_description
           IN ('Dairy') ) AS food
      ORDER BY fat_content
      LIMIT 5;
```

Your results will be similar to the following:

```
 fat_content
-------------
80
81
82
83
84
(5 rows)
```

The preceding example is from the vmart_query_01.sql file. You can execute more sample queries using the scripts that installed with the VMart database or write your own. For a list of the sample queries supplied with Vertica, see Appendix: VMart example database schema, tables, and scripts .

Backing up and restoring the database

Important
Inadequate security on backups can compromise overall database security. Be sure to secure backup locations and strictly limit access to backups only to users who already have permissions to access all database data.

Vertica supplies a comprehensive utility, vbr , that lets you back up and restore a full database, as well as create backups of specific schema or tables. You should back up your database regularly and before major or destructive operations.

All vbr operations rely on a configuration file that describes your database, backup locations, and other parameters. Typically you use the same configuration file for both the backup and restore operations. To create your first configuration file, copy one of the sample files for backup listed in Sample vbr configuration files . Edit the copy to specify a snapshot (backup) name, your database details, and where to back up. The comments in the sample file guide you.

The following example shows a full backup:

```
$ vbr -t backup --config full-backup.ini
Starting backup of database VTDB.
Participating nodes: v_vmart_node0001, v_vmart_node0002, v_vmart_node0003, v_vmart_node0004.
Snapshotting database.
Snapshot complete.
Approximate bytes to copy: 2315056043 of 2356089422 total.
[==================================================] 100%
Copying backup metadata.
Finalizing backup.
Backup complete!
```

By default, there is no screen output other than the progress bar.

You can restore the entire database or selected schemas and tables. You can also use vbr to replicate data from one database to another or to copy an entire cluster. For more information about vbr , see Backing up and restoring the database .

Using Database Designer to create a comprehensive design

Vertica Database Designer:

- Analyzes your logical schema, sample data, and, optionally, your sample queries.
- Creates a physical schema design (a set of projections) that can be deployed automatically or manually.
- Does not require specialized database knowledge.
- Can be run and rerun any time for additional optimization without stopping the database.
- Uses strategies to provide optimal query performance and data compression.

Use Database Designer to create a comprehensive design, which allows you to create projections for all tables in your database.

You can also use Database Designer to create an incremental design , which creates projections for all tables referenced in the queries you supply.

You can create a comprehensive design with Database Designer using Management Console or through Administration Tools. You can also choose to run Database Designer programmatically .

In this section

- Running Database Designer with Management Console
- Running Database Designer with administration tools

Running Database Designer with Management Console

In this tutorial, you'll create a comprehensive design with Database Designer through the Management Console interface. If, in the future, you have a query that you want to optimize, you can create an enhanced (incremental) design with additional projections. You can tune these projections specifically for the query you provide.

> **Note**
> To run Database Designer outside Administration Tools, you must be a dbadmin user. If you are not a dbadmin user, you must have the DBDUSER role assigned to you and own the tables for which you are designing projections. For details, see Database Designer access requirements .

You can choose to create the design manually or use the Management Console wizard, as described below.

Important
Set your browser so it does not cache pages. If a browser caches pages, you might be unable to see the new design added.

Follow these steps to to create a comprehensive design with the Management Console wizard:

1. Log in to Management Console.
2. Verify that your database is up and running.
3. Choose the database for which you want to create the design. You can find the database under the **Recent Databases** section or by clicking **Existing Infrastructure** to reach the Databases and Clusters page.
   The database overview page opens.
4. At the bottom of the screen, click the **Design** button.
5. In the **New Design** dialog box, enter the design name.
6. Click **Wizard** to continue.
7. Create an initial design. For **Design Type** , select **Comprehensive** and click **Next** .
8. In the **Optimization Objective** window, select **Balance Load and Performance** to create a design that is balanced between database size and query performance. Click **Next** .
9. Select the schemas. Because the VMart design is a multi-schema database, select all three schemas (public, store, and online_sales) for your design in the **Select Sample Data** window. Click **Next** .
   If you include a schema that contains tables without data, the design could be suboptimal. You can choose to continue, but Vertica recommends that you deselect the schemas that contain empty tables before you proceed.
10. Choose the K-safety value for your design. The K-Safety value determines the number of buddy projections you want Database Designer to create.
11. Choose Analyze Correlations Mode. Analyze Correlations Mode determines if Database Designer analyzes and considers column correlations when creating the design.
    - **Ignore:** When creating a design, ignore any column correlations in the specified tables.
    - **Consider existing:** Consider the existing correlations in the tables when creating the design. If you set the mode to 1, and there are no existing correlations, Database Designer does not consider correlations.

- **Analyze missing:** Analyze column correlations on tables where the correlation analysis was not previously performed. When creating the design, consider all column correlations (new and existing).
- **Analyze all:** Analyze all column correlations in the tables and consider them when creating the design. Even if correlations exist for a table, reanalyze the table for correlations.

Click **Next** .

12. Submit query files to Database Designer in one of two ways:
    - Supply your own query files by selecting the **Browse** button.
    - Click **Use Query Repository** , which submits recently executed queries from the QUERY_REQUESTS system table.

    Click **Next** .
13. In the **Execution Options** window, select any of the following options:
    - **Analyze statistics** : Select this option to run statistics automatically after design deployment, so Database Designer can make better decisions for its proposed design.
    - **Auto-build** : Select this option to run Database Designer as soon as you complete the wizard. This option only builds the proposed design.
    - **Auto-deploy** : Select this option for auto-build designs that you want to deploy automatically.
14. Click **Submit Design** . The Database Designer page opens:
    - If you chose to automatically deploy your design, Database Designer executes in the background.
    - If you did not select the **Auto-build** or **Auto-deploy** options, you can click **Build Design** or **Deploy Design** on the Database Designer page.
15. In the **My Designs** pane, view the status of your design:
    - When the deployment completes, the **My Design** pane shows **Design Deployed** .
    - The event history window shows the details of the design build and deployment.

To run Database Designer with Administration Tools, see Running Database Designer with administration tools in this guide.

Running Database Designer with administration tools

In this procedure, you create a comprehensive design with Database Designer using the Administration Tools interface. If, in the future, you have a query that you want to optimize, you can create an enhanced (incremental) design with additional projections. You can tune these projections specifically for the query you provide. See Incremental Design for more information.

Follow these steps to create the comprehensive design using Database Designer in Administration Tools:

1. If you are not in Administration Tools, exit the vsql session and access Administration Tools:
    - Type \q to exit vsql.
    - Type admintools to access the Administration Tools Main Menu.
2. Start the database for which you want to create a design.
3. From the **Main Menu** , click **Configuration Menu** and then click **OK** .
4. From the **Configuration Menu** , click **Run Database Designer** and then click **OK** .
5. When the **Select a database for design** dialog box opens, select **VMart** and then click **OK** .
   If you are prompted to enter the password for the database, click **OK** to bypass the message. Because no password was assigned when you installed the VMart database, you do not need to enter one now.
6. Click **OK** to accept the default directory for storing Database Designer output and log files.
7. In the **Database Designer** window, enter a name for the design, for example, vmart_design , and click **OK** . Design names can contain only alphanumeric characters or underscores. No other special characters are allowed.
8. Create a complete initial design. In the **Design Type** window, click **Comprehensive** and click **OK** .
9. Select the schemas. Because the VMart design is a multi-schema database, you can select all three schemas (online_sales, public, and store) for your design. Click **OK** .
   If you include a schema that contains tables without data, the Administration Tools notifies you that designing for tables without data could be suboptimal. You can choose to continue, but Vertica recommends that you deselect the schemas that contain empty tables before you proceed.
10. In the **Design Options** window, accept all three options and click **OK** .
    The three options are:
    - **Optimize with queries:** Supplying the Database Designer with queries is especially important if you want to optimize the database design for query performance. Vertica recommends that you limit the design input to 100 queries.
    - **Update statistics:** Accurate statistics help the Database Designer choose the best strategy for data compression. If you select this option, the database statistics are updated to maximize design quality.
    - **Deploy design:** The new design deploys automatically. During deployment, new projections are added, some existing projections retained, and any necessary existing projections removed. Any new projections are refreshed to populate them with data.
11. Because you selected the **Optimize with queries** option, you must enter the full path to the file containing the queries that will be run on your database. In this example, it is:

    ```
    /opt/vertica/examples/VMart_Schema/vmart_queries.sql
    ```

    The queries in the query file must be delimited with semicolons (;). The last query must end with a semicolon (;).

12. Choose the K-safety value you want and click **OK** . The design K-Safety determines the number of buddy projections you want Database Designer to create.
    If you create a comprehensive design on a single node, you are not prompted to enter a K-safety value.
13. In the **Optimization Objective** window, select **Balanced query/load performance** to create a design that is balanced between database size and query performance. Click **OK** .
14. When the informational message displays, click **Proceed** .
    Database Designer automatically performs these actions:
    - Sets up the design session.
    - Examines table data.
    - Loads queries from the query file you provided (in this example, /opt/vertica/examples/VMart_Schema/vmart_queries.sql).
    - Creates the design.

    Deploys the design or saves a SQL file containing the commands to create the design, based on your selections in the Desgin Options window.
    Depending on system resources, the design process could take several minutes. You should allow this process to complete uninterrupted. If you must cancel the session, use Ctrl+C.
15. When Database Designer finishes, press **Enter** to return to the Administration Tools menu. Examine the steps taken to create the design. The files are in the directory you specified to store the output and log files. In this example, that directory is /opt/vertica/examples/VMart_Schema . For more information about the script files, see About Database Designer .

For additional information about managing your designs, see Creating a database design in the Administrator's Guide.

Restoring the status of your host

When you finish the tutorial, you can restore your host machines to their original state. Use the following instructions to clean up your host and start over from scratch.

Stopping and dropping the database

Follow these steps to stop and/or drop your database. A database must be stopped before it can be dropped.

1. If connected to the database, disconnect by typing \q .
2. In the Administration Tools **Main Menu** dialog box, click **Stop Database** and click **OK** .
3. In the **Select database to stop** window, select the database you want to stop and click **OK** .
4. After stopping the database, click **Configuration Menu** and click **OK** .
5. Click **Drop Database** and click **OK** .
6. In the **Select database to drop** window, select the database you want to drop and click **OK** .
7. Click **Yes** to confirm.
8. In the next window type yes (lowercase) to confirm and click **OK** .

Alternatively, use the delete_example script, which stops and drops the database:

1. If connected to the database, disconnect by typing \q .
2. In the Administration Tools **Main Menu** dialog box, select **Exit** .
3. Log in as the database administrator.
4. Change to the /examples directory.
   ```
   $ cd /opt/vertica/examples
   ```
5. Run the delete_example script.
   ```
   $ /opt/vertica/sbin/delete_example Vmart
   ```

Uninstalling Vertica

See Uninstall Vertica .

Optional steps

You can also choose to:

- Remove the dbadmin account on all cluster hosts.
- Remove any example database directories you created.

Appendix: VMart example database schema, tables, and scripts

This appendix provides detailed information about the VMart example database's schema, tables, and scripts.

The VMart example database contains three different schemas:

- public
- store
- online_sales

The term “schema” has several related meanings in Vertica:

- In SQL statements, a schema refers to named namespace for a logical schema.
- Logical schema refers to a set of tables and constraints.
- Physical schema refers to a set of projections.

Tables identifies the three schemas and all the data tables in the VMart database. Each schema contains tables that are created and loaded during database installation. See the schema maps for a list of tables and their contents:

- Public schema map
- Store schema map
- online_sales schema map

Sample scripts describes the sample scripts that contain SQL commands that represent queries that might be used in a real business using a VMart-like database. Once you’re comfortable running the example queries, you might want to write your own.

In this section

- Tables
- Public schema map
- Store schema map
- online_sales schema map
- Sample scripts

Tables

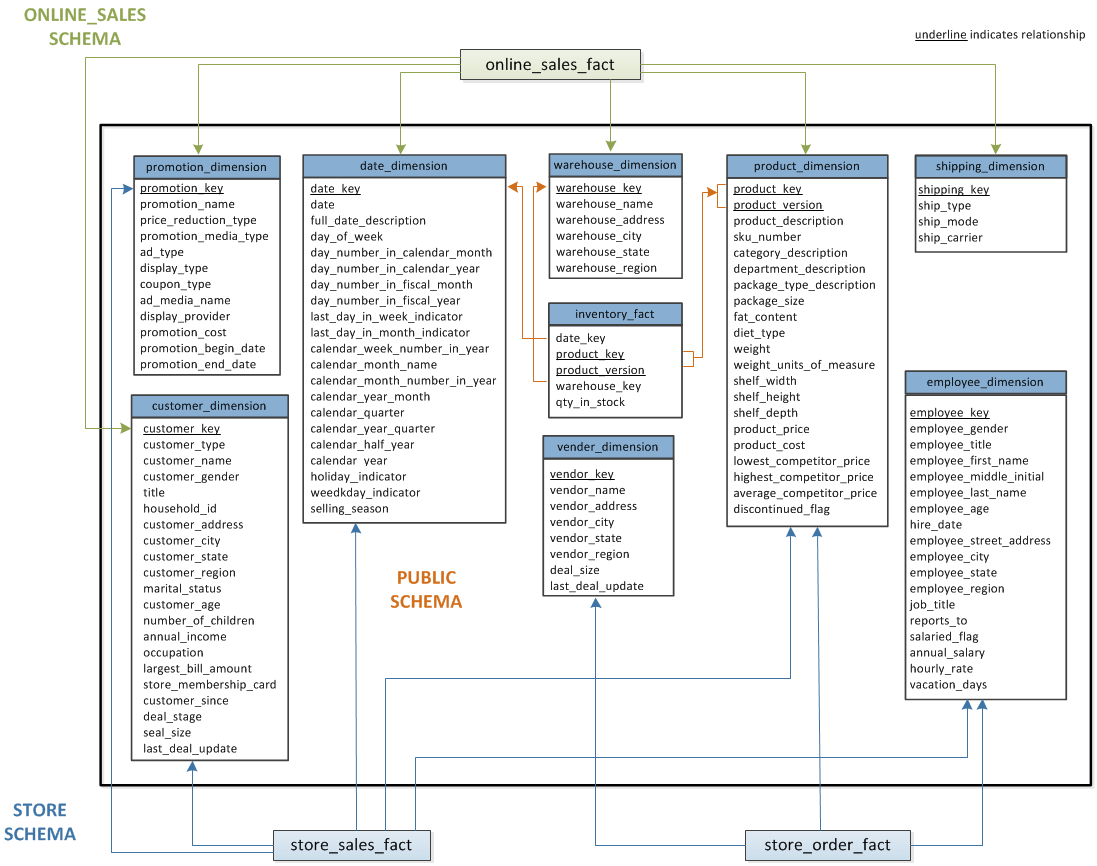
The three schemas in the VMart database include the following tables:

public Schema	**store Schema**	**online_sales Schema**
inventory_fact	store_orders_fact	online_sales_fact
customer_dimension	store_sales_fact	call_center_dimension
date_dimension	store_dimension	online_page_dimension
employee_dimension		
product_dimension		
promotion_dimension		
shipping_dimension		
vendor_dimension		
warehouse_dimension		

Public schema map

The public schema is a snowflake schema. The following graphic illustrates the public schema and its relationships with tables in the online_sales and store schemas.

The subsequent subsections describe database tables.



In this section

- inventory_fact
- customer_dimension
- date_dimension
- employee_dimension
- product_dimension
- promotion_dimension
- shipping_dimension
- vendor_dimension
- warehouse_dimension

inventory_fact

This table contains information about each product in inventory.

Column Name	Data Type	NULLs
date_key	INTEGER	No
product_key	INTEGER	No
product_version	INTEGER	No
warehouse_key	INTEGER	No
qty_in_stock	INTEGER	No

customer_dimension

This table contains information about all the retail chain's customers.

Column Name	Data Type	NULLs

customer_key	INTEGER	No
customer_type	VARCHAR(16)	Yes
customer_name	VARCHAR(256)	Yes
customer_gender	VARCHAR(8)	Yes
title	VARCHAR(8)	Yes
household_id	INTEGER	Yes
customer_address	VARCHAR(256)	Yes
customer_city	VARCHAR(64)	Yes
customer_state	CHAR(2)	Yes
customer_region	VARCHAR(64)	Yes
marital_status	VARCHAR(32)	Yes
customer_age	INTEGER	Yes
number_of_children	INTEGER	Yes
annual_income	INTEGER	Yes
occupation	VARCHAR(64)	Yes
largest_bill_amount	INTEGER	Yes
store_membership_card	INTEGER	Yes
customer_since	DATE	Yes
deal_stage	VARCHAR(32)	Yes
deal_size	INTEGER	Yes
last_deal_update	DATE	Yes

date_dimension

This table contains information about dates. It is generated from a file containing correct date/time data.

Column Name	Data Type	NULLs
date_key	INTEGER	No
date	DATE	Yes
full_date_description	VARCHAR(18)	Yes
day_of_week	VARCHAR(9)	Yes
day_number_in_calendar_month	INTEGER	Yes

day_number_in_calendar_year	INTEGER	Yes
day_number_in_fiscal_month	INTEGER	Yes
day_number_in_fiscal_year	INTEGER	Yes
last_day_in_week_indicator	INTEGER	Yes
last_day_in_month_indicator	INTEGER	Yes
calendar_week_number_in_year	INTEGER	Yes
calendar_month_name	VARCHAR(9)	Yes
calendar_month_number_in_year	INTEGER	Yes
calendar_year_month	CHAR(7)	Yes
calendar_quarter	INTEGER	Yes
calendar_year_quarter	CHAR(7)	Yes
calendar_half_year	INTEGER	Yes
calendar_year	INTEGER	Yes
holiday_indicator	VARCHAR(10)	Yes
weekday_indicator	CHAR(7)	Yes
selling_season	VARCHAR(32)	Yes

employee_dimension

This table contains information about all the people who work for the retail chain.

Column Name	Data Type	NULLs
employee_key	INTEGER	No
employee_gender	VARCHAR(8)	Yes
courtesy_title	VARCHAR(8)	Yes
employee_first_name	VARCHAR(64)	Yes
employee_middle_initial	VARCHAR(8)	Yes
employee_last_name	VARCHAR(64)	Yes
employee_age	INTEGER	Yes
hire_date	DATE	Yes
employee_street_address	VARCHAR(256)	Yes
employee_city	VARCHAR(64)	Yes

employee_state	CHAR(2)	Yes
employee_region	CHAR(32)	Yes
job_title	VARCHAR(64)	Yes
reports_to	INTEGER	Yes
salaried_flag	INTEGER	Yes
annual_salary	INTEGER	Yes
hourly_rate	FLOAT	Yes
vacation_days	INTEGER	Yes

product_dimension

This table describes all products sold by the department store chain.

Column Name	Data Type	NULLs
product_key	INTEGER	No
product_version	INTEGER	No
product_description	VARCHAR(128)	Yes
sku_number	CHAR(32)	Yes
category_description	CHAR(32)	Yes
department_description	CHAR(32)	Yes
package_type_description	CHAR(32)	Yes
package_size	CHAR(32)	Yes
fat_content	INTEGER	Yes
diet_type	CHAR(32)	Yes
weight	INTEGER	Yes
weight_units_of_measure	CHAR(32)	Yes
shelf_width	INTEGER	Yes
shelf_height	INTEGER	Yes
shelf_depth	INTEGER	Yes
product_price	INTEGER	Yes
product_cost	INTEGER	Yes
lowest_competitor_price	INTEGER	Yes

highest_competitor_price	INTEGER	Yes
average_competitor_price	INTEGER	Yes
discontinued_flag	INTEGER	Yes

promotion_dimension

This table describes every promotion ever done by the retail chain.

Column Name	Data Type	NULLs
promotion_key	INTEGER	No
promotion_name	VARCHAR(128)	Yes
price_reduction_type	VARCHAR(32)	Yes
promotion_media_type	VARCHAR(32)	Yes
ad_type	VARCHAR(32)	Yes
display_type	VARCHAR(32)	Yes
coupon_type	VARCHAR(32)	Yes
ad_media_name	VARCHAR(32)	Yes
display_provider	VARCHAR(128)	Yes
promotion_cost	INTEGER	Yes
promotion_begin_date	DATE	Yes
promotion_end_date	DATE	Yes

shipping_dimension

This table contains information about shipping companies that the retail chain uses.

Column Name	Data Type	NULLs
shipping_key	INTEGER	No
ship_type	CHAR(30)	Yes
ship_mode	CHAR(10)	Yes
ship_carrier	CHAR(20)	Yes

vendor_dimension

This table contains information about each vendor that provides products sold through the retail chain.

Column Name	Data Type	NULLs

vendor_key	INTEGER	No
vendor_name	VARCHAR(64)	Yes
vendor_address	VARCHAR(64)	Yes
vendor_city	VARCHAR(64)	Yes
vendor_state	CHAR(2)	Yes
vendor_region	VARCHAR(32)	Yes
deal_size	INTEGER	Yes
last_deal_update	DATE	Yes

warehouse_dimension

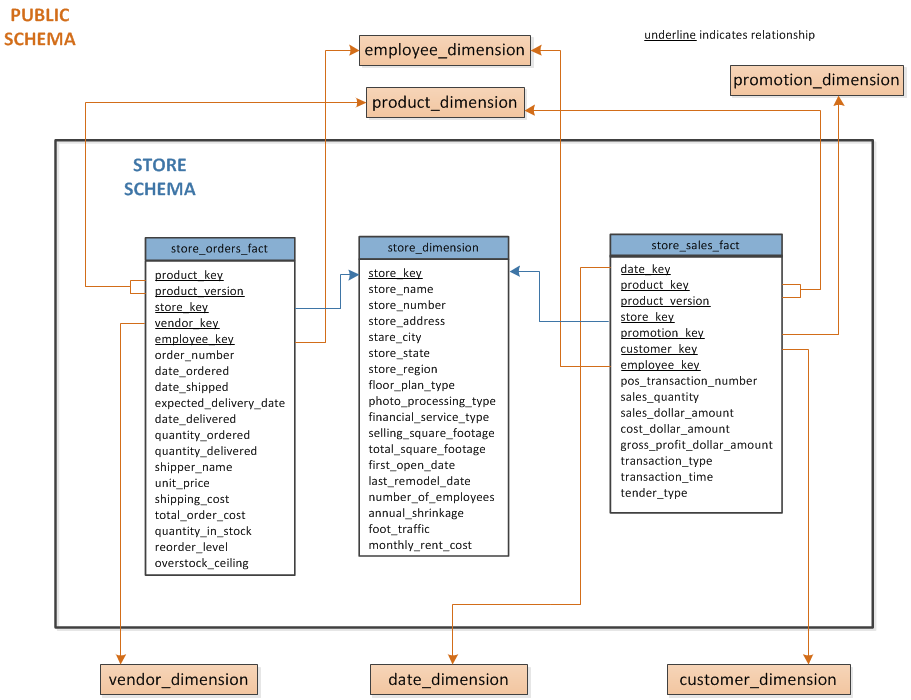
This table provides information about each of the chain's warehouses.

Column Name	Data Type	NULLs
warehouse_key	INTEGER	No
warehouse_name	VARCHAR(20)	Yes
warehouse_address	VARCHAR(256)	Yes
warehouse_city	VARCHAR(60)	Yes
warehouse_state	CHAR(2)	Yes
warehouse_region	VARCHAR(32)	Yes

Store schema map

The store schema is a snowflake schema that contains information about the retail chain's bricks-and-mortar stores. The following graphic illustrates the store schema and its relationship with tables in the public schema.

The subsequent subsections describe database tables.



In this section

- store_orders_fact
- store_sales_fact
- store_dimension

store_orders_fact

This table contains information about all orders made at the company's brick-and-mortar stores.

Column Name	Data Type	NULLs
product_key	INTEGER	No
product_version	INTEGER	No
store_key	INTEGER	No
vendor_key	INTEGER	No
employee_key	INTEGER	No
order_number	INTEGER	No
date_ordered	DATE	Yes
date_shipped	DATE	Yes
expected_delivery_date	DATE	Yes
date_delivered	DATE	Yes
quantity_ordered	INTEGER	Yes
quantity_delivered	INTEGER	Yes
shipper_name	VARCHAR(32)	Yes

unit_price	INTEGER	Yes
shipping_cost	INTEGER	Yes
total_order_cost	INTEGER	Yes
quantity_in_stock	INTEGER	Yes
reorder_level	INTEGER	Yes
overstock_ceiling	INTEGER	Yes

store_sales_fact

This table contains information about all sales made at the company's brick-and-mortar stores.

Column Name	Data Type	NULLs
date_key	INTEGER	No
product_key	INTEGER	No
product_version	INTEGER	No
store_key	INTEGER	No
promotion_key	INTEGER	No
customer_key	INTEGER	No
employee_key	INTEGER	No
pos_transaction_number	INTEGER	No
sales_quantity	INTEGER	Yes
sales_dollar_amount	INTEGER	Yes
cost_dollar_amount	INTEGER	Yes
gross_profit_dollar_amount	INTEGER	Yes
transaction_type	VARCHAR(16)	Yes
transaction_time	TIME	Yes
tender_type	VARCHAR(8)	Yes

store_dimension

This table contains information about each brick-and-mortar store within the retail chain.

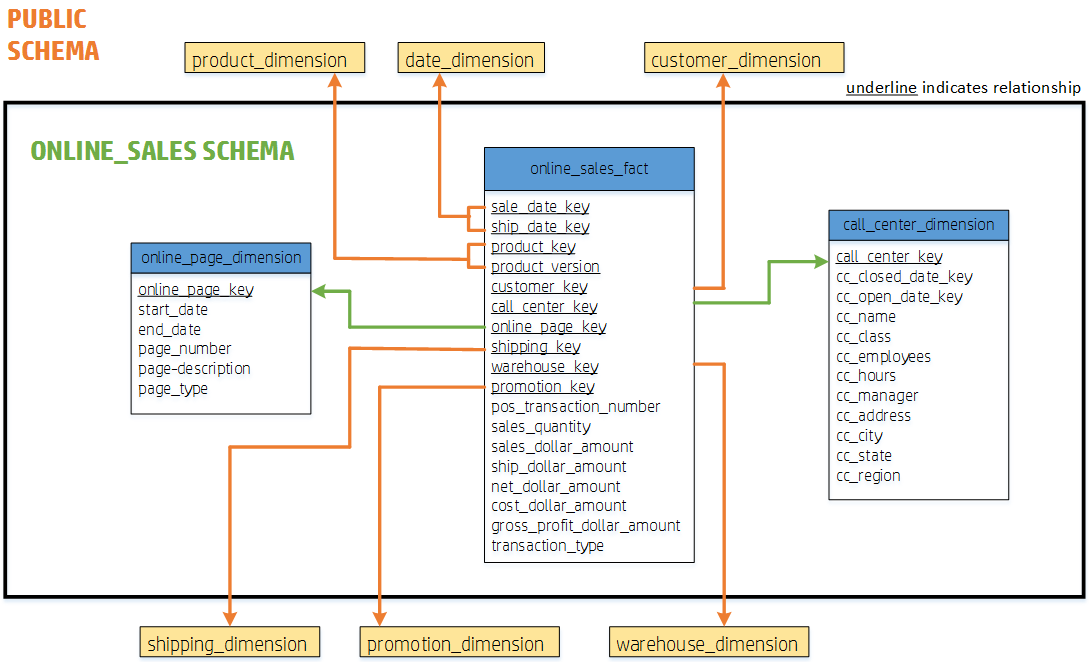
Column Name	Data Type	NULLs
store_key	INTEGER	No
store_name	VARCHAR(64)	Yes

store_number	INTEGER	Yes
store_address	VARCHAR(256)	Yes
store_city	VARCHAR(64)	Yes
store_state	CHAR(2)	Yes
store_region	VARCHAR(64)	Yes
floor_plan_type	VARCHAR(32)	Yes
photo_processing_type	VARCHAR(32)	Yes
financial_service_type	VARCHAR(32)	Yes
selling_square_footage	INTEGER	Yes
total_square_footage	INTEGER	Yes
first_open_date	DATE	Yes
last_remodel_date	DATE	Yes
number_of_employees	INTEGER	Yes
annual_shrinkage	INTEGER	Yes
foot_traffic	INTEGER	Yes
monthly_rent_cost	INTEGER	Yes

online_sales schema map

The online_sales schema is a snowflake schema that contains information about the retail chains. The following graphic illustrates the online_sales schema and its relationship with tables in the public schema.

The subsequent subsections describe database tables.



In this section

- online_sales_fact
- call_center_dimension
- online_page_dimension

online_sales_fact

This table describes all the items purchased through the online store front.

Column Name	Data Type	NULLs
sale_date_key	INTEGER	No
ship_date_key	INTEGER	No
product_key	INTEGER	No
product_version	INTEGER	No
customer_key	INTEGER	No
call_center_key	INTEGER	No
online_page_key	INTEGER	No
shipping_key	INTEGER	No
warehouse_key	INTEGER	No
promotion_key	INTEGER	No
pos_transaction_number	INTEGER	No
sales_quantity	INTEGER	Yes
sales_dollar_amount	FLOAT	Yes
ship_dollar_amount	FLOAT	Yes

net_dollar_amount	FLOAT	Yes
cost_dollar_amount	FLOAT	Yes
gross_profit_dollar_amount	FLOAT	Yes
transaction_type	VARCHAR(16)	Yes

call_center_dimension

This table describes all the chain's call centers.

Column Name	Data Type	NULLs
call_center_key	INTEGER	No
cc_closed_date	DATE	Yes
cc_open_date	DATE	Yes
cc_date	VARCHAR(50)	Yes
cc_class	VARCHAR(50)	Yes
cc_employees	INTEGER	Yes
cc_hours	CHAR(20)	Yes
cc_manager	VARCHAR(40)	Yes
cc_address	VARCHAR(256)	Yes
cc_city	VARCHAR(64)	Yes
cc_state	CHAR(2)	Yes
cc_region	VARCHAR(64)	Yes

online_page_dimension

This table describes all the pages in the online store front.

Column Name	Data Type	NULLs
online_page_key	INTEGER	No
start_date	DATE	Yes
end_date	DATE	Yes
page_number	INTEGER	Yes
page_description	VARCHAR(100)	Yes
page_type	VARCHAR(100)	Yes

Sample scripts

You can create your own queries, but the VMart example directory includes sample query script files to help you get started quickly.

You can find the following sample scripts at this path /opt/vertica/examples/VMart_Schema .

To run any of the scripts, enter

```
=>  \i <script_name>
```

Alternatively, type the commands from the script file manually.

> **Note**
> The data that your queries return might differ from the example output shown in this guide because the sample data generator is random.

In this section

- vmart_query_01.sql
- vmart_query_02.sql
- vmart_query_03.sql
- vmart_query_04.sql
- vmart_query_05.sql
- vmart_query_06.sql
- vmart_query_07.sql
- vmart_query_08.sql
- vmart_query_09.sql

vmart_query_01.sql

```
-- vmart_query_01.sql
-- FROM clause subquery
-- Return the values for five products with the
-- lowest-fat content in the Dairy department
SELECT fat_content
FROM (
  SELECT DISTINCT fat_content
  FROM product_dimension
  WHERE department_description
  IN ('Dairy') ) AS food
  ORDER BY fat_content
  LIMIT 5;
```

Output

```
 fat_content
-------------
          80
          81
          82
          83
          84
(5 rows)
```

vmart_query_02.sql

```
-- vmart_query_02.sql
-- WHERE clause subquery
-- Asks for all orders placed by stores located in Massachusetts
-- and by vendors located elsewhere before March 1, 2003:
SELECT order_number, date_ordered
FROM store.store_orders_fact orders
WHERE orders.store_key IN (
  SELECT store_key
  FROM store.store_dimension
  WHERE store_state = 'MA')
    AND orders.vendor_key NOT IN (
  SELECT vendor_key
  FROM public.vendor_dimension
  WHERE vendor_state = 'MA')
    AND date_ordered < '2012-03-01';
```

Output

```
order_number | date_ordered
-------------+--------------
       53019 | 2012-02-10
      222168 | 2012-02-05
      160801 | 2012-01-08
      106922 | 2012-02-07
      246465 | 2012-02-10
      234218 | 2012-02-03
      263119 | 2012-01-04
       73015 | 2012-01-01
      233618 | 2012-02-10
       85784 | 2012-02-07
      146607 | 2012-02-07
      296193 | 2012-02-05
       55052 | 2012-01-05
      144574 | 2012-01-05
      117412 | 2012-02-08
      276288 | 2012-02-08
      185103 | 2012-01-03
      282274 | 2012-01-01
      245300 | 2012-02-06
      143526 | 2012-01-04
       59564 | 2012-02-06
...
```

vmart_query_03.sql

```
-- vmart_query_03.sql
-- noncorrelated subquery
-- Requests female and male customers with the maximum
-- annual income from customers
SELECT customer_name, annual_income
FROM public.customer_dimension
WHERE (customer_gender, annual_income) IN (
  SELECT customer_gender, MAX(annual_income)
  FROM public.customer_dimension
  GROUP BY customer_gender);
```

Output

```
  customer_name   | annual_income
------------------+---------------
 James M. McNulty |        999979
 Emily G. Vogel   |        999998
(2 rows)
```

vmart_query_04.sql

```
-- vmart_query_04.sql
-- IN predicate
-- Find all products supplied by stores in MA
SELECT DISTINCT s.product_key, p.product_description
FROM store.store_sales_fact s, public.product_dimension p
WHERE s.product_key = p.product_key
AND s.product_version = p.product_version AND s.store_key IN (
  SELECT store_key
  FROM store.store_dimension
  WHERE store_state = 'MA')
ORDER BY s.product_key;
```

Output

```
 product_key |          product_description
-------------+-------------------------------------------
1 | Brand #1 butter
1 | Brand #2 bagels
2 | Brand #3 lamb
2 | Brand #4 brandy
2 | Brand #5 golf clubs
2 | Brand #6 chicken noodle soup
3 | Brand #10 ground beef
3 | Brand #11 vanilla ice cream
3 | Brand #7 canned chicken broth
3 | Brand #8 halibut
3 | Brand #9 camera case
4 | Brand #12 rash ointment
4 | Brand #13 low fat milk
4 | Brand #14 chocolate chip cookies
4 | Brand #15 silver polishing cream
5 | Brand #16 cod
5 | Brand #17 band aids
6 | Brand #18 bananas
6 | Brand #19 starch
6 | Brand #20 vegetable soup
6 | Brand #21 bourbon
...
```

vmart_query_05.sql

```
-- vmart_query_05.sql
-- EXISTS predicate
-- Get a list of all the orders placed by all stores on
-- January 2, 2003 for the vendors with records in the
-- vendor_dimension table
SELECT store_key, order_number, date_ordered
FROM store.store_orders_fact
WHERE EXISTS (
   SELECT 1
   FROM public.vendor_dimension
   WHERE public.vendor_dimension.vendor_key = store.store_orders_fact.vendor_key)
   AND date_ordered = '2012-01-02';
```

Output

```
 store_key | order_number | date_ordered
-----------+--------------+--------------
        98 |       151837 | 2012-01-02
       123 |       238372 | 2012-01-02
       242 |       263973 | 2012-01-02
       150 |       226047 | 2012-01-02
       247 |       232273 | 2012-01-02
       203 |       171649 | 2012-01-02
       129 |        98723 | 2012-01-02
        80 |       265660 | 2012-01-02
       231 |       271085 | 2012-01-02
       149 |        12169 | 2012-01-02
       141 |       201153 | 2012-01-02
         1 |        23715 | 2012-01-02
       156 |        98182 | 2012-01-02
        44 |       229465 | 2012-01-02
       178 |       141869 | 2012-01-02
       134 |        44410 | 2012-01-02
       141 |       129839 | 2012-01-02
       205 |        54138 | 2012-01-02
       113 |        63358 | 2012-01-02
        99 |        50142 | 2012-01-02
        44 |       131255 | 2012-01-02
...
```

vmart_query_06.sql

```
-- vmart_query_06.sql
-- EXISTS predicate
-- Orders placed by the vendor who got the best deal
-- on January 4, 2004
SELECT store_key, order_number, date_ordered
FROM store.store_orders_fact ord, public.vendor_dimension vd
WHERE ord.vendor_key = vd.vendor_key
AND vd.deal_size IN (
  SELECT MAX(deal_size)
  FROM public.vendor_dimension)
AND date_ordered = '2013-01-04';
```

Output

```
 store_key | order_number | date_ordered
-----------+--------------+--------------
        45 |       202416 | 2013-01-04
        24 |       250295 | 2013-01-04
       121 |       251417 | 2013-01-04
       198 |        75716 | 2013-01-04
       166 |        36008 | 2013-01-04
        27 |       150241 | 2013-01-04
       148 |       182207 | 2013-01-04
         9 |       188567 | 2013-01-04
       113 |        66017 | 2013-01-04
...
```

vmart_query_07.sql

```
-- vmart_query_07.sql
-- Multicolumn subquery
-- Which products have the highest cost,
-- grouped by category and department
SELECT product_description, sku_number, department_description
FROM public.product_dimension
WHERE (category_description, department_description, product_cost) IN (
  SELECT category_description, department_description,
  MAX(product_cost) FROM product_dimension
  GROUP BY category_description, department_description);
```

Output

```
product_description        |      sku_number       |   department_description
---------------------------+-----------------------+----------------------------------
 Brand #601 steak          | SKU-#601              | Meat
 Brand #649 brooms         | SKU-#649              | Cleaning supplies
 Brand #677 veal           | SKU-#677              | Meat
 Brand #1371 memory card   | SKU-#1371             | Photography
 Brand #1761 catfish       | SKU-#1761             | Seafood
 Brand #1810 frozen pizza  | SKU-#1810             | Frozen Goods
 Brand #1979 canned peaches | SKU-#1979            | Canned Goods
 Brand #2097 apples        | SKU-#2097             | Produce
 Brand #2287 lens cap      | SKU-#2287             | Photography
...
```

vmart_query_08.sql

```
-- vmart_query_08.sql
-- between online_sales_fact and online_page_dimension
SELECT page_description, page_type, start_date, end_date
FROM online_sales.online_sales_fact f, online_sales.online_page_dimension d
WHERE f.online_page_key = d.online_page_key
AND page_number IN
  (SELECT MAX(page_number)
    FROM online_sales.online_page_dimension)
AND page_type = 'monthly' AND start_date = '2012-06-02';
```

Output

```
     page_description     | page_type | start_date | end_date
--------------------------+-----------+------------+-----------
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
Online Page Description #1 | monthly   | 2012-06-02 | 2012-06-11
(12 rows)
```

vmart_query_09.sql

```
-- vmart_query_09.sql
-- Equi join
-- Joins online_sales_fact table and the call_center_dimension
-- table with the ON clause
SELECT sales_quantity, sales_dollar_amount, transaction_type, cc_name
FROM online_sales.online_sales_fact
INNER JOIN online_sales.call_center_dimension
ON (online_sales.online_sales_fact.call_center_key
     = online_sales.call_center_dimension.call_center_key
    AND sale_date_key = 156)
ORDER BY sales_dollar_amount DESC;
```

Output

```
 sales_quantity | sales_dollar_amount | transaction_type |      cc_name
----------------+---------------------+------------------+-------------------
              7 |                 589 | purchase         | Central Midwest
              8 |                 589 | purchase         | South Midwest
              8 |                 589 | purchase         | California
              1 |                 587 | purchase         | New England
              1 |                 586 | purchase         | Other
              1 |                 584 | purchase         | New England
              4 |                 584 | purchase         | New England
              7 |                 581 | purchase         | Mid Atlantic
              5 |                 579 | purchase         | North Midwest
              8 |                 577 | purchase         | North Midwest
              4 |                 577 | purchase         | Central Midwest
              2 |                 575 | purchase         | Hawaii/Alaska
              4 |                 573 | purchase         | NY Metro
              4 |                 572 | purchase         | Central Midwest
              1 |                 570 | purchase         | Mid Atlantic
              9 |                 569 | purchase         | Southeastern
              1 |                 569 | purchase         | NY Metro
              5 |                 567 | purchase         | Other
              7 |                 567 | purchase         | Hawaii/Alaska
              9 |                 567 | purchase         | South Midwest
              1 |                 566 | purchase         | New England
...
```

Architecture

Understanding how Vertica works helps you effectively design, build, operate, and maintain a Vertica database. This section assumes that you are familiar with the basic concepts and terminology of relational database management systems and SQL.

Column storage

Vertica stores data in a column format so it can be queried for best performance. Compared to row-based storage, column storage reduces disk I/O, making it ideal for read-intensive workloads. Vertica reads only the columns needed to answer the query. Columns are encoded and compressed to further improve performance.

Vertica uses a number of different encoding strategies, depending on column data type, table cardinality, and sort order. Encoding increases performance because there is less disk I/O during query execution. In addition, you can store more data in less space.

Compression allows a column store to occupy substantially less storage than a row store. In a column store, every value stored in a projection column has the same data type. This greatly facilitates compression, particularly in sorted columns. In a row store, each value of a row can have a different data type, resulting in a much less effective use of compression. The efficient storage methods that Vertica uses allow you to maintain more historical data in physical storage.

Projections

A *projection* consists of a set of columns with the same sort order, defined by a column to sort by or a sequence of columns by which to sort. Like an index or materialized view in a traditional database, a projection accelerates query processing. When you write queries in terms of the original tables, the query uses the projections to return query results. For more information, see Projections .

Projections are distributed and replicated across nodes in your cluster, ensuring that if one node becomes unavailable, another copy of the data remains available. This redundancy is called *K-safety* .

Automatic data replication, failover, and recovery provide for active redundancy, which increases performance. Nodes recover automatically by querying the system.

Eon and Enterprise Modes

A Vertica database runs in one of two modes: Eon or Enterprise. Both modes can be deployed on-premises or in the cloud. In Eon Mode, compute and storage are separate; the database uses shared communal storage, and you can add or remove nodes or subclusters based on demand. In Enterprise Mode, each database node has a share of the data and queries are distributed to take advantage of data locality. Understanding the differences between these modes is key. See Eon vs. Enterprise Mode .

In this section

- Eon vs. Enterprise Mode
- Eon Mode concepts
- Enterprise Mode concepts

Eon vs. Enterprise Mode

A Vertica database runs in one of two modes: Eon or Enterprise. Both modes can be deployed on-premises or in the cloud. Understanding the difference between these two modes is key. If you are deploying a Vertica database, you must decide which mode to run it in early in your deployment planning. If you are using an already-deployed Vertica database, you should understand how each mode affects loading and querying data.

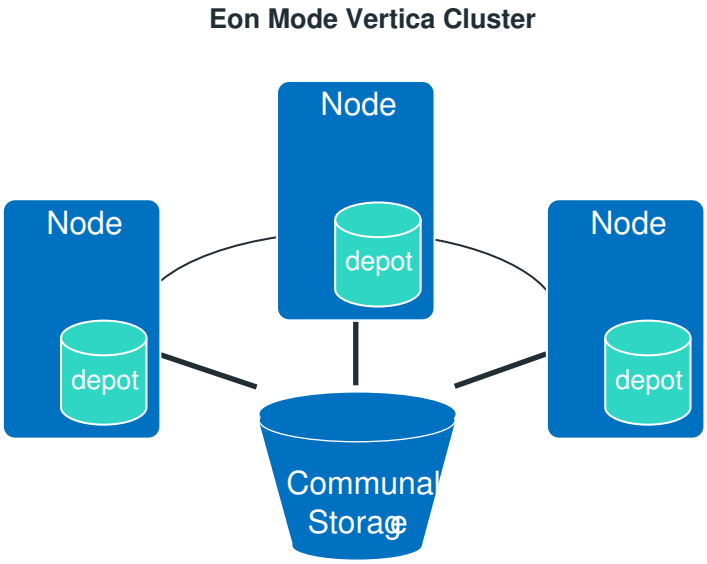
Vertica databases running in Eon and Enterprise modes store their data differently:

- Eon Mode databases use communal storage for their data.
- Enterprise Mode databases store data locally in the file system of nodes that make up the database.

These different storage methods lead to a number of important differences between the two modes.

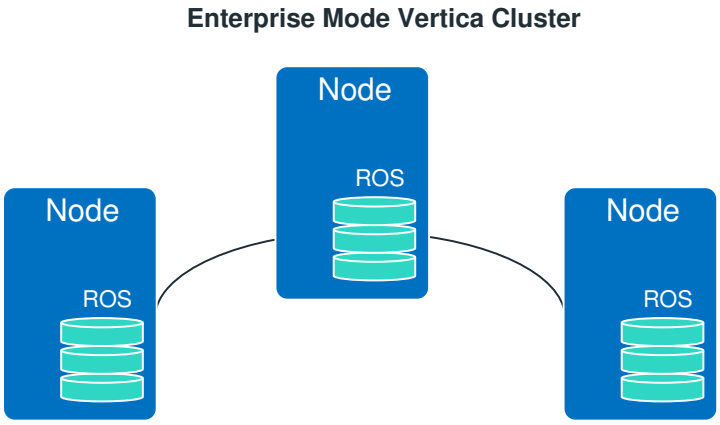
Storage overview

Eon Mode stores data in a shared object store called communal storage:



When deployed in a cloud environment, Vertica stores its data in a cloud-based storage container, such as an AWS S3 bucket. When deployed on-premises, Vertica stores data in a locally-deployed object store, such as a Pure Storage FlashBlade appliance. Separating the persistent data storage from the compute resources (the nodes that load data and process queries) provides flexibility.

Enterprise Mode stores data across the filesystems of the database nodes:



Each node is responsible for storing and processing a portion of the data. The data is co-located on the nodes in both cloud-based and on-premises databases to provide resiliency in the event of node failure. Having the data located close to the computing power offers a different set of advantages. When a node is added to the cluster, or comes back online after being unavailable, it automatically queries other nodes to update its local data.

Key advantages of each mode

The different ways Eon Mode and Enterprise Mode store data give each mode an advantage in different environments. The following table summarizes these differences. For details, see Eon vs. Enterprise Mode .

Chief advantages of...	**Where database mode is...**	
	Eon	**Enterprise**
Cloud	• Easily scaled up or down to meet changing workloads and reduce costs. • Workloads can be isolated to a subcluster of nodes. • Virtually no limits on database size. Most cloud providers offer essentially unlimited data storage (for a price).	Works in most cloud platforms. Eon Mode works in specific cloud providers.
On-premises	• Workloads can be isolated to a subset of nodes called a subcluster. • Can increase storage without adding nodes (and, if the object store supports hot plugging, without downtime).	No additional hardware needed beyond the servers that make up the database cluster.

Note
You can migrate an Enterprise Mode database to Eon with the meta-function MIGRATE_ENTERPRISE_TO_EON . For details on using this meta-function, see Migrating an enterprise database to Eon Mode .

Performance

Eon Mode and Enterprise Mode databases have roughly the same performance in the same environment when properly configured.

An Eon Mode database typically enables caching data from communal storage on a node's local *depot* , which the node uses to process queries. With depot caching enabled, query performance on an Eon Mode database is equivalent to an Enterprise Mode database, where each node stores a portion of the database locally. In both cases, nodes access locally-stored data to resolve queries.

To further improve performance, you can enable depot *warming* on an Eon Mode database. When depot warming is enabled, a node that is undergoing startup preemptively loads its depot with frequently queried and pinned data. When the node completes startup and begins to execute queries, its depot already contains much of the data it needs to process those queries. This reduces the need to fetch data from communal storage, and expedites query performance accordingly.

Query performance in an Eon Mode database is liable to decline if its depot is too small. A small depot increases the chance that a query will require data that is not in the depot. That results in nodes having to retrieve data from communal storage more frequently.

Note
When comparing a cloud-based Eon Mode database to an on-premises Enterprise Mode database, performance differences are typically due to the overall performance impact of a shared cloud-based virtual environment compared to on-premises dedicated hardware. An Enterprise Mode

database running in the same cloud would have the same performance as the Eon Mode, in most cases.

Installation

An Eon Mode database must have an object store to store its data communally. An Enterprise Mode database does not require any additional storage hardware beyond the storage installed on its nodes. Depending on the environment you've chosen for your Vertica database (especially if you are installing on-premises), the need to configure an object store may make your installation a bit more complex.

Because Enterprise Mode does not need additional hardware for data storage, it can be a bit simpler to install. An on-premises Eon Mode install needs additional hardware and additional configuration for the object store that provides the communal storage.

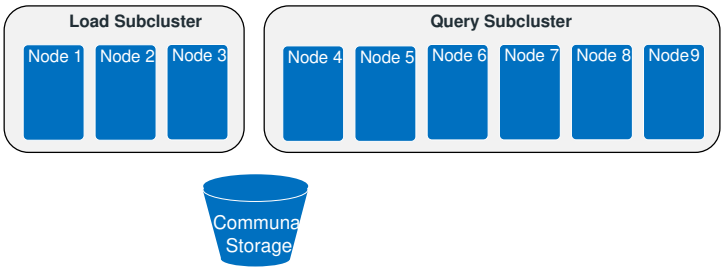
Enterprise Mode is especially useful for development environments because it does not require additional hardware beyond the nodes you use to run it. You can even create a single-node Enterprise Mode database, either on physical hardware or on a virtual machine. You can download a preconfigured single-node Enterprise Mode virtual machine that is ready to run. See Vertica community edition (CE) for more information.

Deploying an Eon Mode database in a cloud environment is usually simpler than an on-premises install. The cloud environments provide their own object store for you. For example, when you deploy an Eon Mode database in Amazon's AWS, you just need to create an S3 bucket for the communal data store. You then provide the S3 URL to Vertica when creating the database. There is no need to install and configure a separate data store.

Deploying an Enterprise Mode database in the cloud is similar to installing one on-premises. The virtual machines you create in the cloud must have enough local storage to store your database's data.

Workload isolation

You often want to prevent intensive workloads from interfering with other potentially time-sensitive workloads. For example, you may want to isolate ETL workloads from querying workloads. Groups of users that rely on real-time analytics can be isolated from groups that are running batched reports.



Eon Mode databases offer the best workload isolation option. It allows you to create groups of nodes called subclusters that isolate workloads. A query only runs on the nodes in a single subcluster. It does not affect nodes outside the subcluster. You can assign different groups of users a different subcluster to use.

In an Eon Mode database, subclusters and scalability work hand in hand. You often add, remove, stop, and start entire subclusters of nodes, rather than scaling nodes individually.

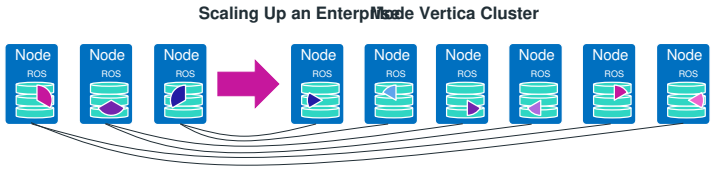
Enterprise Mode does not offer subclusters to isolate workloads. You can use features such as resource pools and other settings to give specific queries priority and access to more resources. However, these features do not truly isolate workloads as subclusters do. See Managing workloads for an explanation of managing workloads using these features.

Scalability

You can scale a Vertica database by adding or removing nodes to meet changing analytic needs. Scalability is usually more important in cloud environments where you are paying by the hour for each node in your database. If your database isn't busy, there is no reason to have underused nodes costing you money. You can reduce the number of nodes in your database during quiet times (weekends and holidays, for example) to save money.

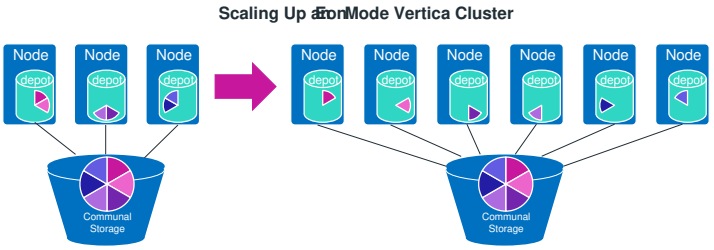
Scalability is usually less important for on-premises installations. There are limited additional costs involved in having nodes running when they are not fully in use.

An Enterprise Mode database scales less efficiently than an Eon Mode one. When an Enterprise Mode database scales, it must re-segment (rebalance) its data to be spread among the new number of nodes.



Rebalancing is an expensive operation. When scaling the database up, Vertica must break up files and physically move a percentage of the data from the original nodes to the new nodes. When scaling down, Vertica must move the data off of the nodes that are being removed and distribute it among the remaining nodes. The database is not available during rebalancing. This process can take 12, 24, or even 36 hours to complete, depending on the size of the database. After scaling up an Enterprise Mode database, queries should run faster because each node is responsible for less data. Therefore, each node has less work to do to process each query. Scaling down an Enterprise Mode database usually has the opposite effect—queries will run slower. See Elastic cluster for more information on scaling an Enterprise Mode database.

Eon Mode databases scale more efficiently because data storage is separate from the computing resources.

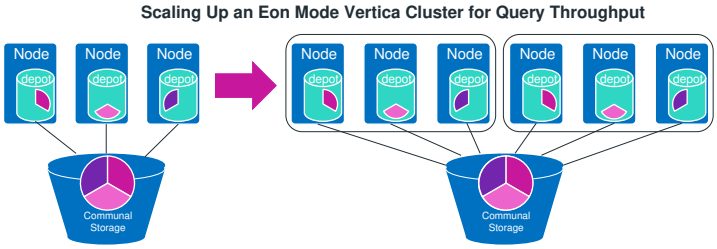


When you scale up an Eon Mode database, the database's data does not need to be re-segmented. Instead, the additional nodes subscribe to preexisting segments (called shards) of data in communal storage. When expanding the cluster, Vertica rebalances the shards assigned to each node, rather than physically splitting the data storage and moving in between nodes. The new nodes prepare to process queries by retrieving data from the communal storage to fill their depots (a local cache of data from the communal storage). The database remains available while scaling and the process takes minutes rather than hours to complete.

> **Note**
> Node subscriptions are slightly more complicated than shown in the previous diagram. To ensure K-safety , each node actually subscribes to a second shard (or shards, if there are more shards than nodes) to act as a backup in case the node subscribing to those shards goes down. See Shards and subscriptions for details.

If the number of shards in the communal storage is equal to or higher than the new number of nodes (as shown in the previous diagram), then query performance improves after expanding the cluster. Each node is responsible for processing less data, so the same queries will run faster after you scale the cluster up.

You can also scale your database up to improve query throughput. Query throughput improves the number of queries processed by your database in parallel. You usually care about query throughput when your workload contains many, shorter-running queries ("dashboard queries"). To improve throughput, add more nodes to your database in a new subcluster . The subcluster isolates queries run by clients connected to it from the other nodes in the database. Subclusters work independently and in parallel. Isolating the workloads means that your database runs more queries simultaneously.



If a subcluster contains more nodes than the number of shards in communal storage, multiple nodes subscribe to the same shard. In this case, Vertica uses a feature called elastic crunch scaling to execute the query faster. Vertica divides the responsibility for the data in each shard between the subscribing nodes. Each node only needs to process a subset of the data in the shard it subscribes to. Having less data to process means that each node usually finishes its part of the query faster. This often translates into the query finishing its executing sooner.

Important
Always make the number of nodes in your Eon Mode subclusters a multiple of the number of shards in the database, or an even divisor of the number of shards. For example, in a six-shard database, your subclusters should have three, six, or twelve shards. Vertica recommends you never have more than two shards per node.

A mismatch between the number of shards and the number of nodes can impact performance. For example, suppose you have a six-shard database. If you expand a subcluster from three to five nodes, one node would still be the only subscriber for two shards. This means that node has to do twice the work of the other nodes in the subcluster during queries. In this case, you see no benefit from adding the two new nodes, because the node subscribing to two shards becomes a bottleneck.

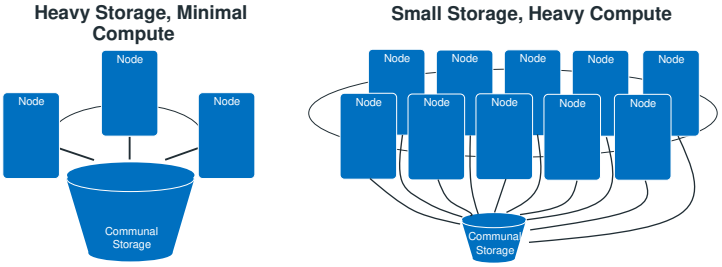
Scaling down an Eon Mode database works similarly. Shutting down entire subclusters reduced your database's query throughput. If you remove nodes from a subcluster, the remaining nodes subscribe to any shards that do not have a subscriber. This process is fast, and the database remains running while it is happening.

Expandability

As you load more data into your database, you may eventually need to expand its data storage. Because Eon Mode databases separate compute from storage, you often expand its storage without changing the number of nodes.

In a cloud environment, you usually do not have a limit on storage. For example, an AWS S3 bucket can store as much data as you want. As long as you are willing to pay for additional storage charges, you do not have to worry about expanding your database's storage.

When you install Eon Mode on-premises, how you expand storage depends on the object store you are using. For example, Pure Storage FlashBlades support hot plugging new blades to add additional storage. This feature lets you expand the storage in your Eon Mode database with no downtime.



In most cases, you usually query a subset of the data in your database (called the working data set). Eon Mode's decoupling of compute and storage let you size your compute (the number of nodes in your database) to the working data set and your desired performance rather than to the entire data set.

For example, if you are performing time series analysis in which the active data set is usually the last 30 days, you can size your cluster to manage 30 days' worth of data. Data older than 30 days simply grows in communal storage. The only reason you need to add more nodes to your Eon Mode database is to meet additional workloads. On the other hand, if you want very high performance on a small data set, you can add as many nodes as you need to obtain the performance you want.

In an Enterprise Mode database, nodes are responsible for storage as well as compute. Because of the tight coupling between compute and storage, the best way to expand storage in an Enterprise Mode database is to add new nodes. As mentioned in the Scalability section, adding nodes to an Enterprise Mode database requires rebalancing the existing data in the database.

Due to the disruption rebalancing causes to the database, you usually expand the storage in an Enterprise Mode database infrequently. When you do expand its storage, you usually add significant amounts of storage to allow for future growth.

Adding nodes to increase storage has the downside that you may be adding compute power to your cluster that isn't really necessary. For example, suppose you are performing time-series analysis that focuses on recent data and your current cluster offers you enough query performance to meet your needs. However, you need to add additional storage to keep historical data. In this case, adding new nodes to your database for additional storage adds computing power you really don't need. Your queries may run a bit faster. However, the slight benefit of faster results probably does not justify the costs of adding more computing power.

Eon Mode concepts

Eon Mode separates the computational processes from the communal storage layer of your database. This separation gives you the ability to store your data in a single location (such as S3 on AWS or Pure Storage) and elastically vary the number of compute nodes connected to that location according to your computational needs. You can adjust the size of your cluster without interrupting analytic workloads, adding or removing nodes as the volume of work changes

The following topics explain how Eon Mode works.

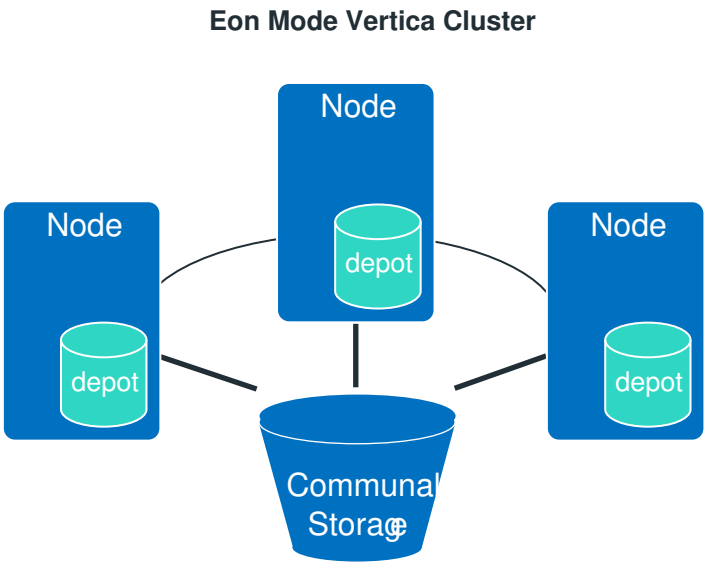
In this section

- Eon Mode architecture
- Shards and subscriptions
- Subclusters
- Elasticity
- Data integrity and high availability in an Eon Mode database
- Stopping, starting, terminating, and reviving Eon Mode database clusters

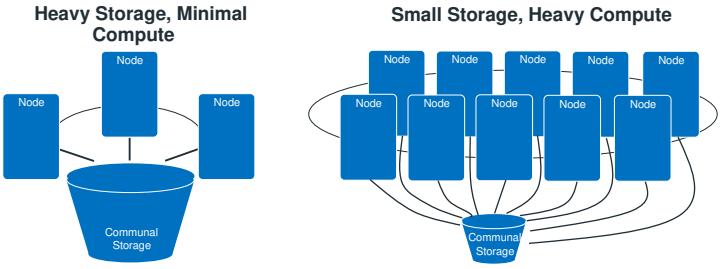
Eon Mode architecture

Eon Mode separates the computational resources from the storage layer of your database. This separation gives you the ability to store your data in a single location. You can elastically vary the number of nodes connected to that location according to your computational needs. Adjusting the size of your cluster does not interrupt analytic workloads.

You can create an Eon Mode database either in a cloud environment, or on-premises on your own systems.



Eon Mode is suited to a range of needs and data volumes. Because compute and storage are separate, you can scale them separately.



Communal storage

Instead of storing data locally, Eon Mode uses a single communal storage location for all data and the catalog (metadata). Communal storage is the database's centralized storage location, shared among the database nodes. Communal storage is based on an object store, such as Amazon's S3 in the cloud or a PureStorage FlashBlade appliance in an on-premises deployment. In either case, Vertica relies on the object store to maintain the durable copy of the data.

Communal storage has the following properties:

- Communal storage in the cloud is more resilient and less susceptible to data loss due to storage failures than storage on disk on individual machines.
- Any data can be read by any node using the same path.
- Capacity is not limited by disk space on nodes.
- Because data is stored communally, you can elastically scale your cluster to meet changing demands. If the data were stored locally on the nodes, adding or removing nodes would require moving significant amounts of data between nodes to either move it off of nodes that are being removed, or onto newly-created nodes.

Communal storage locations are listed in the STORAGE_LOCATIONS system table with a SHARING_TYPE of COMMUNAL.

Within communal storage, data is divided into portions called shards . Shards are how Vertica divides the data among the nodes in the database. Nodes subscribe to particular shards, with subscriptions balanced among the nodes. When loading or querying data, each node is responsible for the data in the shards it subscribes to. See Shards and subscriptions for more information.

Depot storage

A potential drawback of communal storage is its speed, especially in cloud environments. Accessing data from a shared cloud location is slower than reading it from local disk. Also, the connection to communal storage can become a bottleneck if many nodes are reading data from it at once. To improve data access speed, the nodes in an Eon Mode database maintain a local disk cache of data called the depot. When executing a query, the

nodes first check whether the data it needs is in the depot. If it is, then the node finishes the query using the local copy of the data. If the data is not in the depot, the node fetches the data from communal storage, and saves a copy in the depot.

The node stores newly-loaded data in the depot before sending it to communal storage. See Loading Data below for more details.

By default, Vertica sets the maximum size of the depot to be 60% of the total disk space of the filesystem that stores the depot. You can adjust the size of the depot if you wish. Vertica limits the size of the depot to a maximum of 80% of the filesystem that contains it. This upper limit ensures enough disk space for other uses, such as temporary files that Vertica creates during data loads.

Each node also stores a local copy of the database catalog.

Loading data

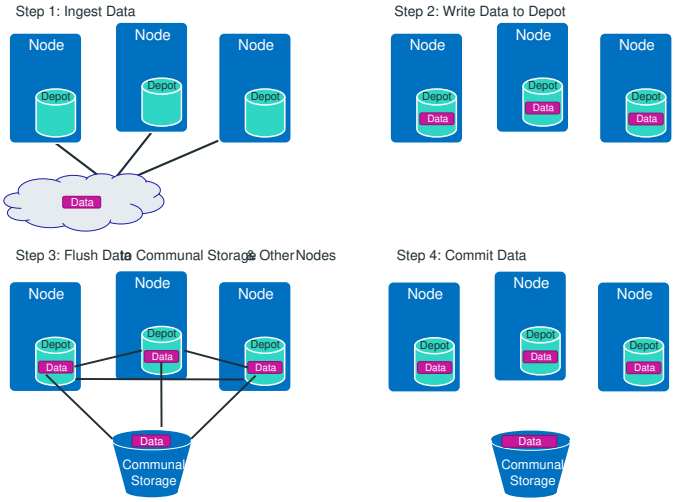
In Eon Mode, COPY statements usually write to read optimized store (ROS) files in a node's depot to improve performance. The COPY statement segments, sorts, and compresses for high optimization. Before the statement commits, Vertica ships the ROS files to communal storage.

Because a load is buffered in the depot on the node executing the load, the size of your depot limits the amount of data you can load in a single operation. Unless you perform multiple loads in parallel sessions, you are unlikely to encounter this limit.

> **Note**
> If your data loads do overflow the amount of space in your database's depot, you can tell Vertica to bypass the depot and load data directly into communal storage. You enable direct writes to communal storage by setting the UseDepotForWrites configuration parameter to 0. See Eon Mode parameters for more information. Once you have completed your large data load, switch this parameter back to 1 to re-enable writing to the depot.

At load time, the participating nodes write files to the depot and synchronously send them to communal storage. The data is also sent to all nodes that subscribe to the shard into which the data is being loaded. This mechanism of sending data to peers at load time improves performance if a node goes down, because the cache of the peers who take over for the down node is already warm. The file compaction mechanism (mergeout) puts its output files into the cache and also uploads them to the communal storage.

The following diagram shows the flow of data during a COPY statement.



Querying data

Vertica uses a slightly different process to plan queries in Eon Mode to incorporate the sharding mechanism and remote storage. Instead of using a fixed-segmentation scheme to distribute data to each node, Vertica uses the sharding mechanism to segment the data into a specific number of shards that at least one (and usually more) nodes subscribes to. When the optimizer selects a projection, the layout for the projection is determined by the participating subscriptions for the session. The optimizer generates query plans that are equivalent to those in Enterprise Mode. It selects one of the nodes that subscribes to each shard to participate in query execution.

Vertica first tries to use data in the depot to resolve a query. When the data in the depot cannot resolve the query, Vertica reads from the communal storage. You could see an impact on query performance when a substantial number of your queries read from the communal storage. If this is the case, then you should consider re-sizing your depot or use depot system tables to get a better idea of what is causing the issue. You can use ALTER_LOCATION_SIZE to change depot size.

Workload isolation and scaling

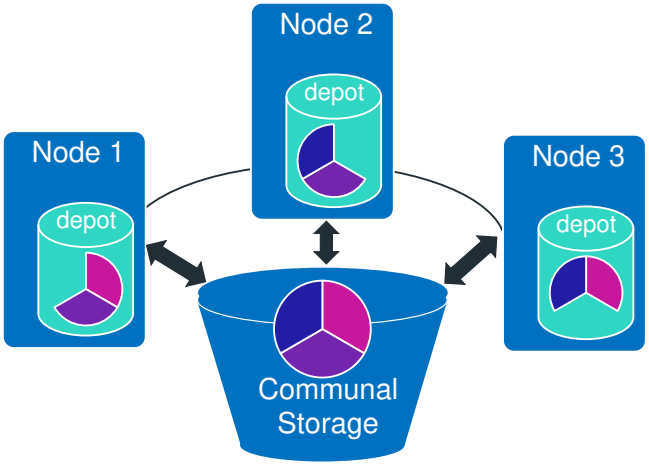
Eon Mode lets you define subclusters that divide up your nodes to isolate workloads from one another. You can also use subclusters to ensure that scaling down your cluster does not result in Vertica going into read-only mode to maintain data integrity. See Subclusters for more information.

Shards and subscriptions

In Eon Mode, Vertica stores data communally in a shared data storage location (for example, in S3 when running on AWS). All nodes are capable of accessing all of the data in the communal storage location. In order for nodes to divide the work of processing queries, Vertica must divide the data between them in some way. It breaks the data in communal storage into segments called shards. Each node in your database subscribes to a subset of the shards in the communal storage location. The shards in your communal storage location are similar to a collection of segmented projections in an Enterprise Mode database.

When K-safety is 1 or higher (high availability), each shard has more than one node subscribing to it in each subcluster. One of the subscribers is responsible for executing queries involving the shard. The other subscribers act as backups. If the main subscriber shuts down or is stopped, then another subscriber takes its place. See Data integrity and high availability in an Eon Mode database for more information.

Eon Mode Vertica Shard Subscriptions (3 Shard Database)



Each shard in the database has a primary subscriber. This subscriber is a primary node that maintains the data in the shard by planning Tuple Mover operations on it. This node can delegate executing these actions to another node in the database cluster. See Tuple mover for more information about these operations. If the primary subscriber node is stopped or goes down, Vertica chooses another primary node that subscribes to the shard as the shard's primary subscriber. If all of the primary nodes that subscribe to a shard go down or are stopped, your database goes into read-only mode to maintain data integrity. Any primary node that is the sole subscriber to a shard is a critical node .

A special type of shard called a replica shard stores metadata for unsegmented projections. Replica shards exist on all nodes.

You define the number of shards when you create your database. For the best performance, the number of shards you choose should be no greater than 2× the number of nodes. At most, you should limit the shard-to-node ratio to no greater than 3:1. MC warns you to take all aspects of shard count into consideration. The number of shards should always be a multiple (or an even divisor) of the number of nodes in your database. See Choosing the Number of Shards and the Initial Node Count for more information.

After database creation, you can change the number of shards in your database with RESHARD_DATABASE . See Change the number of shards in the database for details.

For efficiency, Vertica transfers metadata about shards directly between database nodes. This peer-to-peer transfer applies only to metadata; the actual data that is stored on each node gets copied from communal storage to the node's depot as needed.

Subclusters

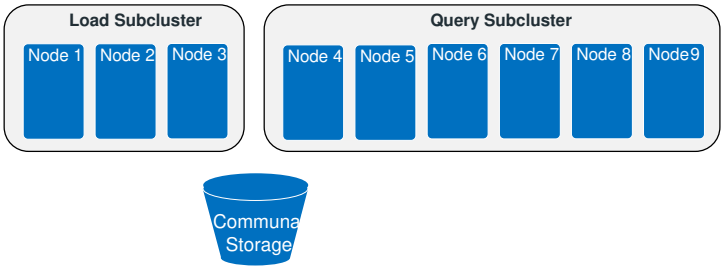
Because Eon Mode separates compute and storage, you can create subclusters within your cluster to isolate work. For example, you might want to dedicate some nodes to loading data and others to executing queries. Or you might want to create subclusters for dedicated groups of users (who might have different priorities). You can also use subclusters to organize nodes into groups for easily scaling your cluster up and down.

Every node in your Eon Mode database must belong to a subcluster. This requirement means your database must always have at least one subcluster. When you create a new Eon Mode database, Vertica creates a subcluster named default_subcluster that contains the nodes you create on database creation. If you add nodes to your database without assigning them to a subcluster, Vertica adds them to the default subcluster. You can choose to designate another subcluster as the default subcluster, or rename default_subcluster to something more descriptive. See Altering subcluster settings for more information.

Using subclusters for work isolation

Database administrators are often concerned about workload management. Intense analytics queries can consume so many resources that they interfere with other important database tasks, such as data loading. Subclusters help you prevent resource issues by isolating workloads from one another.

In Eon Mode, by default, queries only run on nodes in the subcluster that contains the initiator node. For example, consider the two subclusters shown in the following diagram. If you are connected to Node 4, your queries would run on nodes 4 through 9.



Similarly, queries started on Node 1 only run on nodes 1 through 3.

This isolation lets you configure your database cluster to prevent workloads from interfering with each other. You can assign subclusters to specific tasks such as loading data, performing in-depth analytics, and short-running dashboard queries. You can also create subclusters for different groups in your organization, so their workloads do not interfere with one another. You can tune the settings of each subcluster (resource pools, for example) to match their specific workloads.

Subcluster types

There are two types of subclusters: primary and secondary.

Primary subclusters form the core of your Vertica database. They are responsible for planning the maintenance of the data in the communal storage. Your primary subclusters must always be running. If all of your primary subclusters shut down, your database shuts down because it cannot maintain the data in communal storage without a primary subcluster.

Usually, you have just a single primary subcluster in your database. You can choose to have multiple primary subclusters. Additional primary subclusters can make your database more resilient to having primary nodes fail. However, additional primary subclusters make your database less scalable. You usually do not dynamically add or remove nodes from primary subclusters or shut them down to scale your database. In most cases, a single primary subcluster is enough.

Secondary subclusters are designed for dynamic scaling: you add and remove or start and stop these subclusters based on your analytic needs. They are not essential for maintaining your database's data. So, you can easily add, remove, and scale up or down secondary subclusters without impacting the database's ability to run normally.

The nodes in the subcluster inherit their primary or secondary status from the subcluster that contains them; primary subclusters contain primary nodes and secondary subclusters contain secondary nodes.

Subcluster types and elastic scaling

The most important difference between primary and secondary subclusters is their impact on how Vertica determines whether the database is K-Safe and has a quorum . Vertica only considers the nodes in primary subclusters when determining whether all of the shards in the database have a subscribing node. It also only considers primary nodes when determining whether more than half the nodes in the database are running (also known as having a quorum of primary nodes). If either of these conditions is not met, the database goes into read-only mode to prevent data corruption. See Data integrity and high availability in an Eon Mode database for more information about how Vertica maintains data integrity.

Vertica does not consider the secondary nodes when determining whether the database has shard coverage or a quorum of nodes. This fact makes secondary subclusters perfect for managing groups of nodes that you plan to expand and reduce dynamically. You can stop or remove an entire subcluster of secondary nodes without forcing the database into read-only mode.

Minimum subcluster size for K-safe databases

In a K-safe database, subclusters must have at least three nodes in order to operate. Each subcluster tries to maintain subscriptions to all shards in the database. If a subcluster has less than three nodes, it cannot maintain redundant shard coverage where each shard has at least two subscribers in the subcluster. Without redundant coverage, the subcluster cannot continue processing queries if it loses a node. Vertica returns an error if you attempt to rebalance shards in a subcluster with less than three nodes in a K-safe database.

See also

- Eon Mode architecture
- Adding and removing nodes from subclusters
- Altering subcluster settings
- Change the number of shards in the database

- Configuring your Vertica cluster for Eon Mode

Elasticity

Elasticity refers to the ability for you adjust your database to changing workload demands by adding or removing nodes. When your database experiences high demand, you can add new nodes or start stopped nodes to increase the amount of compute available. When your database experiences lower demands (such as during holidays or weekends) you can stop or terminate nodes to save money. You can also gradually add nodes over time as your database demands grow.

All nodes in an Eon Mode database belong to a subcluster. By choosing which subclusters get new nodes, you can affect how the new nodes impact your database. There are two goals you can achieve when adding nodes to your database:

- Improve query throughput: higher throughput means your database processes more queries simultaneously. You often want to improve throughput when you have a workload of "dashboard queries": many relatively short-running queries. In this case, speeding up the processing of individual queries is not as important as having more queries run in parallel.
- Improve query performance: higher query performance means that your complex in-depth analytic queries complete faster.

Scaling for query throughput

To scale for query throughput, add additional nodes to your database in one or more new subclusters . Subclusters independently process queries: a query only runs on the nodes in the subcluster containing the initiator node. By adding one or more subclusters, your database can process more queries at the same time. For the best performance, add the same number of nodes to each new subcluster as there are shards in the database. For example, if you have 6 shards in your database, add 6 nodes to each new subcluster you create.

To take advantage of the improved throughput offered by the new subclusters, clients must connect to them. The best way to ensure your users take advantage of the subclusters you have added for throughput scaling is to create connection load balancing policies that spread client connections across the all nodes in all of these subclusters. See Connection load balancing policies for more information.

Subclusters also organize nodes into groups that can easily be stopped or started together. This feature makes expanding and shrinking your database easier. See Starting and stopping subclusters for details.

Scaling for query performance

To improve the performance of individual queries in a subcluster, add more nodes to it. Queries perform faster when there is more computing power available to process them.

Adding nodes is especially effective if your subcluster has less nodes than there are shards in the database. In this case, nodes are responsible for processing data in multiple shards. When you add more nodes, the newly-added nodes take over responsibility for some of the shards. With less data to process, each node finishes their part of the query faster, resulting in better overall performance. For the best performance, make the number of nodes in the subcluster an even divisor of (or equal to) the number of shards in the database. For example, in a 12-shard database, make the number of nodes in the subcluster 3, 6, or 12.

You can further improve query performance by adding more nodes than there are shards in the database. When nodes outnumber shards, multiple nodes in the subcluster subscribe to the same shard. In this case, when processing a query, Vertica uses a feature called elastic crunch scaling (ECS) to have all of the nodes in the subcluster take part in the query. ECS assigns a subset of the data in each shard to the shard's subscribers. For example, in six-node subcluster in a a three-shard database, each shard has two subscribers. ECS assigns each of the subscribers half of the data in the shard to process during queries. In most cases, with less data to process, the nodes finish executing the query faster. When adding more nodes than shards to a subcluster, make the number of nodes a multiple of the number of shards to ensure an even distribution. For example, in a three-shard database, make the number of nodes in the subcluster 6, 9, 12, and so on.

Using different subclusters for different query types

You do not have to choose one form of elasticity over the other in your database. You can create a group of subclusters to improve query throughput and one or more subclusters that improve query performance. The difference between the two subcluster types is mainly the number of subclusters you create and the number of nodes they contain. To improve throughput, add a multiple subclusters that contain a number of nodes that is equal to or less than the number of shards in the database. The more subclusters you add, the greater the throughput you achieve. To improve query performance, add one or more subclusters where the number of nodes is a multiple of the number of shards in the database.

Once you have created your set of subclusters, you must have clients connect to the correct subcluster for the types of queries they will run. For clients executing frequent, simple dashboard queries, create a connection load balancing policy that connects them to nodes in the throughput scaling subclusters. For clients running more complex analytic queries, create another load balancing policy that connects them to nodes in the performance scaling subcluster.

For details on scaling your Eon Mode database, see Scaling your Eon Mode database .

Data integrity and high availability in an Eon Mode database

The nodes in your Eon Mode database's primary subclusters are responsible for maintaining the data in your database. These nodes (collectively called the database's primary nodes) plan the Tuple Mover mergeout operations that manage the data in the shards. They can also execute these operations if they are the best candidate to do so (see The Tuple Mover in Eon Mode Databases).

The primary nodes can be spread across multiple primary subclusters—they all work together to maintain the data in the shards. The health of the primary nodes is key for your database to continue running normally.

The nodes in secondary subclusters do not plan Tuple Mover operations. They can execute Tuple Mover mergeout operations if a primary node assigns it to them. Your database cluster can lose all of its secondary nodes and still maintain the data in the shards.

Maintaining data integrity the is top goal of your database. If your database loses too many primary nodes, it cannot safely process data. In this case, it goes into read-only mode to prevent data inconsistency or corruption.

High availability (having the database continue running even if individual nodes are lost) is another goal of Vertica. It has several redundancy features to help it prevent downtime. With these features enabled, your database continues to run even if it loses one or more primary nodes.

There are two requirements for the database to continue normal operations: maintaining quorum, and maintaining shard coverage.

Maintaining quorum

The basic requirement for the primary nodes in your Eon Mode database is maintaining a quorum of primary nodes running at all times. To maintain quorum, more than half of the primary nodes (50% plus 1) must be up. For example, in a database with 6 primary nodes, at least 4 of them must be up. If half or more of the primary nodes are down, your database goes into read-only mode to prevent potential data integrity issues. In a database with 6 primary nodes, the database goes into read-only if it loses 3 or more of them. See Read-Only Mode below.

Vertica only counts the primary nodes that are currently part of the database when determining whether the database has quorum. Removing primary nodes cannot result in a loss of quorum. During the removal process, Vertica adjusts the node count to prevent the loss of quorum.

At a minimum, your Eon Mode database must have at least one primary node to function. In most cases, it needs more than one. See Minimum Node Requirements for Eon Mode Database Operation below.

Maintaining shard coverage

In order to continue to process data, your database must be able to maintain the data in its shards. To maintain the data, each shard must have a subscribing primary node that is responsible for running the Tuple Mover on it. This requirement is called having shard coverage. If one or more shards do not have a primary node maintaining its data, your database loses shard coverage and goes into read-only mode (explained below) to prevent possible data integrity issues.

The measure of how resilient your Eon Mode database is to losing a primary node is called its K-safety level. The value K is the number of redundant shard subscriptions your Eon Mode database cluster maintains. It also represents the number of primary nodes in your database that can fail and still be able to run safely. In many cases, your database can lose more than K nodes and still continue to run normally, as long as it maintains shard coverage.

Vertica recommends that your database always have a K-safety value of 1 (K=1). In a K=1 database, each shard has two subscribers: a primary subscriber that is responsible for the shard, and a secondary subscriber that can fill in if the primary subscriber is lost. The primary subscriber is responsible for running the Tuple Mover on the data in the shard. The secondary subscriber maintains a copy of the shard's catalog metadata. so it can fill in if the primary subscriber is lost.

If a shard's primary subscriber fails, the secondary subscriber fills in for it. Because it does not maintain a separate depot for its secondary subscription, the secondary subscriber always directly accesses the shard's data in communal storage. This direct access impacts your database's performance while a secondary subscriber fills in for a primary subscriber. For this reason, always restart or replace a down primary node as soon as possible.

With primary and secondary subscribers in a K=1 database, the loss of a single primary node does not affect the database's ability to process and maintain data. However, if the secondary subscriber fails while standing in for the primary subscriber, your database would lose shard coverage. and be forced to go into read-only mode.

Elastic K-safety

Vertica uses a feature called elastic K-safety to help limit the possibility of shard coverage loss. By default, if either the primary or secondary subscriber to a shard fails, Vertica subscribes an additional primary node to the shard. This subscription takes time to be established, as the newly-subscribed node must get a copy of the shard's metadata. If the shard's sole subscriber fails while the new subscriber is getting the shard's metadata, the database loses shard coverage and can shut down. Once the newly-subscribed node gets a copy of the metadata, it is able to take over maintenance of the shard in case the other subscriber fails. At this point, your database once again has two subscribers for the shard.

Once the down nodes recover and rejoin the subcluster, Vertica removes the subscriptions it added for elastic K-safety. Once all of the nodes rejoin the cluster, the shard subscriptions are the same as they were before the node loss.

With elastic K-safety, your database could manage to maintain shard coverage through the gradual loss of primary nodes, up to the point that it loses quorum. As long as there is enough time for newly-subscribed nodes to gather the shard's metadata, your database is able to maintain shard coverage. However, your database could still be forced into read-only mode due to loss of shard coverage if it lost the primary and secondary subscribers to a shard before a new primary node could complete the process of subscribing to the shard.

> **Note**
> Vertica stops adding new subscriptions when your database gets close to losing quorum. It continues to add new subscriptions if your cluster has more than N ÷ 2 + K + 1 primary nodes up, where N is the total number of primary nodes in the database. For example, if you have 10 primary nodes in your K=1 database, Vertica adds new subscriptions as long as the number of primary nodes that are up is greater than 7 (10 ÷ 2 + 1 + 1). If the number of up primary nodes falls to 6 in this database, adding an additional subscription does not make sense. Losing another primary node would force the database to shut down due to a loss of quorum.

Database read-only mode

If your database loses either quorum or primary shard coverage, it goes into read-only mode. This mode prevents potential data corruption that could result when too many nodes are down or unable to reach other nodes in the database. Read-only mode prevents changes being made to the database that require updates to the global catalog.

DML and DDL in read-only mode

In read-only mode, statements that change the global catalog (such as most DML and DDL statements) fail with an error message. For example, executing DDL statements such as CREATE TABLE while the database is in read-only mode results in the following error:

```
=> CREATE TABLE t (a INTEGER, b VARCHAR);
ERROR 10428:  Transaction commit aborted since the database is currently in read-only mode
HINT:  Commits will be restored when the database restores the quorum
```

DML statements such as COPY return a different error. Vertica stops them from executing before they perform any work:

```
=> COPY warehouse_dimension from stdin;
ERROR 10422:  Running DML statements is not possible in read-only mode
HINT:  Running DMLs will be restored when the database restores the quorum
```

By returning the error early, Vertica avoids performing all of the work required to load data, only to fail when it tries to commit the transaction.

DDL and DML statements that do not affect the global catalog still work. For example, you can create a local temporary table and load data into it while the database is in read-only mode.

Queries in read-only mode

Queries can run on any subcluster that has shard coverage. For example, suppose you have an Eon Mode database with a 3-node primary and a 3-node secondary subcluster. If two of the primary nodes go down, the database loses quorum and goes into read-only mode. The primary subcluster also loses shard coverage, because two of its nodes are down. The remaining node does not have a subscription to at least some of the shards. In this case, queries on the remaining primary node (except for some system table queries) always fail:

```
=> SELECT * FROM warehouse_dimension;
ERROR 9099:  Cannot find participating nodes to run the query
```

The secondary subcluster still has shard coverage so you can execute queries on it.

Monitoring read-only mode

Besides noticing DML and DDL statements returning errors, you can determine whether the database has gone into read-only mode by monitoring system tables:

- The NODES system table has a column named is_readonly that becomes true for all nodes when the database is in read-only mode.

```
=> SELECT node_name, node_state, is_primary, is_readonly, subcluster_name FROM nodes;

     node_name       | node_state | is_primary | is_readonly |  subcluster_name
---------------------+------------+------------+-------------+--------------------
 v_verticadb_node0001 | UP         | t          | t           | default_subcluster
 v_verticadb_node0002 | DOWN       | t          | t           | default_subcluster
 v_verticadb_node0003 | DOWN       | t          | t           | default_subcluster
 v_verticadb_node0004 | UP         | f          | t           | analytics
 v_verticadb_node0005 | UP         | f          | t           | analytics
 v_verticadb_node0006 | UP         | f          | t           | analytics
(6 rows)
```

- When the database goes into read-only mode, every node that is still up in the database records a Cluster Read-only event (event code 20). You can find these events by querying the event monitoring system tables such as ACTIVE_EVENTS :

```
=> \x
Expanded display is on.

=> SELECT * FROM ACTIVE_EVENTS WHERE event_code = 20;
-[ RECORD 1 ]-------------+-------------------------------------------------------------------------
node_name                | v_verticadb_node0001
event_code               | 20
event_id                 | 0
event_severity           | Critical
event_posted_timestamp   | 2021-11-22 15:57:24.514475+00
event_expiration         | 2089-12-10 19:11:31.514475+00
event_code_description   | Cluster Read-only
event_problem_description | Cluster cannot perform updates due to quorum loss and can only be queried
reporting_node           | v_verticadb_node0001
event_sent_to_channels   | Vertica Log
event_posted_count       | 1
-[ RECORD 2 ]-------------+-------------------------------------------------------------------------
node_name                | v_verticadb_node0004
event_code               | 20
event_id                 | 0
event_severity           | Critical
event_posted_timestamp   | 2021-11-22 15:57:24.515022+00
event_expiration         | 2089-12-10 19:11:31.515022+00
event_code_description   | Cluster Read-only
event_problem_description | Cluster cannot perform updates due to quorum loss and can only be queried
reporting_node           | v_verticadb_node0004
event_sent_to_channels   | Vertica Log
event_posted_count       | 1
-[ RECORD 3 ]-------------+-------------------------------------------------------------------------
node_name                | v_verticadb_node0005
event_code               | 20
event_id                 | 0
event_severity           | Critical
event_posted_timestamp   | 2021-11-22 15:57:24.515019+00
event_expiration         | 2089-12-10 19:11:31.515019+00
event_code_description   | Cluster Read-only
event_problem_description | Cluster cannot perform updates due to quorum loss and can only be queried
reporting_node           | v_verticadb_node0005
event_sent_to_channels   | Vertica Log
event_posted_count       | 1
-[ RECORD 4 ]-------------+-------------------------------------------------------------------------
node_name                | v_verticadb_node0006
event_code               | 20
event_id                 | 0
event_severity           | Critical
event_posted_timestamp   | 2021-11-22 15:57:24.515172+00
event_expiration         | 2089-12-10 19:11:31.515172+00
event_code_description   | Cluster Read-only
event_problem_description | Cluster cannot perform updates due to quorum loss and can only be queried
reporting_node           | v_verticadb_node0006
event_sent_to_channels   | Vertica Log
event_posted_count       | 1
```

See Monitoring events .

Recover from read-only mode

To recover from read-only mode, restart the down nodes. Restarting the nodes resolves loss of quorum or loss of primary shard coverage that caused the database to go into read-only mode.

Once the down nodes restart and rejoin the database, Vertica restarts on the nodes that were in read-only mode. This step is necessary to allow the nodes to resubscribe to their shards. During this restart, client connections to these nodes will drop. For example, users connected via vsql to one of the nodes where Vertica is restarting see the message:

```
server closed the connection unexpectedly
    This probably means the server terminated abnormally
    before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
```

Users using vsql to connect to nodes as Vertica restarts see the message:

```
vsql: FATAL 4149:  Node startup/recovery in progress. Not yet ready
to accept connections
```

Once Vertica restarts on the nodes, the database resumes normal operation.

When Vertica sets the K-safety value in an Eon Mode database

When you have three or more primary nodes in your database, Vertica automatically sets the database's K-safety to 1 (K=1). It also automatically configures shard subscriptions so that each node can act as a backup for another node, as described in Maintaining Shard Coverage .

This behavior is different than an Enterprise Mode database, where you must design your database's physical schema to meet several criteria before you can have Vertica mark the database as K-safe. See Difference Between Enterprise Mode and Eon Mode K-safe Designs below for details.

> **Note**
> Databases with less than three primary nodes have no data redundancy (K=0). Vertica recommends you only use a database with less than three primary nodes for testing.

Minimum node requirements for Eon Mode database operation

The K-safety level of your database determines the minimum number of primary nodes it must have:

- When K=0, your database must have at least 1 primary node. Setting K to 0 allows you to have a single-node Eon Mode database. Note that in a K=0 database, the loss of a primary node will result in the database going into read-only mode.
- When K=1 (the most common case), your database must have at least three primary nodes. This number of primary nodes allows Vertica to maintain data integrity if a primary node goes down.
- If you want to manually set the K-safe value to 2 (see Difference Between Enterprise Mode and Eon Mode K-safe Designs below) you must have at least 5 primary nodes.

Vertica prevents you from removing primary nodes if your cluster would fall below the lower limit for your database's K-safety setting. If you want to remove nodes in a database at this lower limit, you must lower the K-safety level using the MARK_DESIGN_KSAFE function and then call REBALANCE_SHARDS .

Critical nodes and subclusters

Vertica designates any node or subcluster in the database whose loss would cause the database to go into read-only mode as critical. For example, in an Eon Mode database, when a primary node goes down, nodes with secondary subscriptions to its shards take over maintaining the shards' data. These nodes become critical. Their loss would cause the database to lose shard coverage and be forced to go into read-only mode.

> **Note**
> When elastic K-safety is enabled (which is the default) Vertica subscribes additional primary nodes to a down primary node's shards. After these nodes finish subscribing by getting a copy of the shard's metadata, they are ready to fill in if the secondary subscriber also goes down. When this happens, the node filling in for the down node is no longer considered critical.

Vertica maintains a list of critical nodes and subclusters in two system tables: CRITICAL_NODES and CRITICAL_SUBCLUSTERS . Before stopping nodes or subclusters, check these tables to ensure the node or subcluster you intend to stop is not critical.

Difference between Enterprise Mode and Eon Mode K-safe designs

In an Enterprise Mode database, you use the MARK_DESIGN_KSAFE function to enable high availability in your database. You call this function after you have designed your database's physical schema to meet all the requirements for K-safe design (often, by running the database designer). If you attempt to mark your database as K-safe when the physical schema does not support the level K-safety you pass to MARK_DESIGN_KSAFE, it returns ar error. See Designing segmented projections for K-safety for more information.

In Eon Mode, you do not need to use the MARK_DESIGN_KSAFE because Vertica automatically makes the database K-safe when you have three or more primary nodes. You can use this function to change the K-safety level of your database. In an Eon Mode database, this function changes how Vertica configures shard subscriptions. You can call MARK_DESIGN_KSAFE with any level of K-safety you want. It only has an effect when you call REBALANCE_SHARDS to update the shard subscriptions for the nodes in your database.

> **Note**
> Usually, you do not use a K-safety value of greater than 1 in a cloud-based Eon Mode database. Adding replacement nodes to a cluster is easy in a cloud environment.

Stopping, starting, terminating, and reviving Eon Mode database clusters

If you do not need your Eon Mode database for a period of time, you can choose to stop or terminate its cluster. Stopping or terminating the cluster saves you money when running in cloud environments.

Stopping and starting a database cluster

When you stop your database cluster, you shut down the nodes in the cluster. Shutting down the cluster is an additional step beyond just shutting down the database. When you shut down the cluster in cloud environments, the node's instances no longer run but are still defined in the cloud platform. You can quickly restart the cluster and database when you need to use it again.

Stopping the database cluster is the best option to use when you will not need it for a short to medium time frame. For example, if no one accesses your database on weekends or holidays, you may consider stopping the cluster.

You save money when you shut down your database cluster in cloud environments. Stopped clusters do not consume expensive CPU resources. Stopped clusters can still cost you money, however. If you configured your nodes with persistent local storage, your cloud provider usually still charges a small amount to maintain that storage space.

Terminating and reviving a database cluster

Terminating a database cluster frees up the resources used by the database cluster's nodes.

On a cloud platform, terminating the database cluster deletes the node's virtual machine instances. The database's data and catalog remain stored in communal storage. You can restart the database by reviving it. When you revive a database, you provision a new database cluster and configure it to use the database's data and metadata stored in communal storage.

> **Note**
> You cannot revive sandboxed subclusters. If you call the revive_db admintools command on a cluster with both sandboxed and unsandboxed subclusters, the nodes in the unsandboxed subclusters start as expected, but the nodes in the sandboxed subclusters remain down. Attempting to revive only a sandboxed subcluster returns an error.

In an on-premises Eon Mode database, terminating the database cluster usually means shutting down the database and then repurposing the hardware that the nodes ran on.

Terminating the database cluster is the best option for when you will not need the database for an extended period (or if you are unsure whether you will ever need the database again). As long as you do not delete the communal storage location, you can get your database running again by reviving it.

To revive a database, you create a new Vertica Eon Mode cluster and configure it to use the database's communal storage location. The easiest way to revive a database in the cloud is to use the Management Console. It provisions a new Eon Mode cluster for you, and then revives the database onto it.

Reviving a database takes longer than starting a stopped database. Even if you use the MC to automate the process, provisioning a new set of nodes takes much longer than just restarting stopped nodes. When the new nodes start for the first time, they must load data from communal storage from scratch.

Terminating the database cluster can save you more money over simply stopping the database when the database's nodes have persistent local storage. Cloud providers usually charge you a small recurring fee for the space consumed by persistent local storage on the nodes.

See also

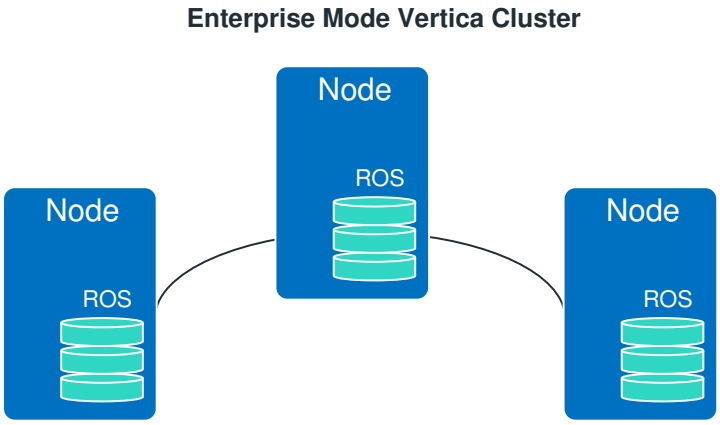
- Reviving an Eon Mode database cluster
- Terminating an Eon Mode database cluster
- Eon Mode architecture
- Adding and removing nodes from subclusters

- Altering subcluster settings

Enterprise Mode concepts

In an Enterprise Mode Vertica database, the physical architecture is designed to move data as close as possible to computing resources. This architecture differs from a cluster running in Eon Mode which is described in Eon Mode concepts .

The data in an Enterprise Mode database is spread among the nodes in the database. Ideally, the data is evenly distributed to ensure that each node has an equal amount of the analytic workload.



Hybrid data store

When running in Enterprise Mode, Vertica stores data on the database in read optimized store (ROS) containers. ROS data is segmented, sorted, and compressed for high optimization. To avoid fragmentation of data among many small ROS containers, Vertica periodically executes a mergeout operation, which consolidates ROS data into fewer and larger containers.

Data redundancy

In Enterprise Mode, each node of the Vertica database stores and operates on data locally. Without some form of redundancy, the loss of a node would force your database to shut down, as some of its data would be unavailable to service queries.

You usually choose to have your Enterprise Mode database store data redundantly to prevent data loss and service interruptions should a node shut down. See K-safety in an Enterprise Mode database for details.

In this section

- K-safety in an Enterprise Mode database
- High availability with projections
- High availability with fault groups

K-safety in an Enterprise Mode database

K-safety sets the fault tolerance in your Enterprise Mode database cluster. The value K represents the number of times the data in the database cluster is replicated. These replicas allow other nodes to take over query processing for any failed nodes.

> **Note**
> K-safety works in a similar manner in an Eon Mode database, with several important differences. See Data integrity and high availability in an Eon Mode database for details.

In Vertica, the value of K can be zero (0), one (1), or two (2). If a database with a K-safety of one (K=1) loses a node, the database continues to run normally. Potentially, the database could continue running if additional nodes fail, as long as at least one other node in the cluster has a copy of the failed node's data. Increasing K-safety to 2 ensures that Vertica can run normally if any two nodes fail. When the failed node or nodes return and successfully recover, they can participate in database operations again.

> **Note**
> If the number of failed nodes exceeds the K value, some the data may become unavailable. In this case, the database is considered unsafe and automatically shuts down. However, if every data segment is available on at least one functioning cluster node Vertica continues to run safely.

Potentially, up to half the nodes in a database with a K-safety of 1 could fail without causing the database to shut down. As long as the data on each failed node is available from another active node, the database continues to run.

> **Note**
> If half or more of the nodes in the database cluster fail, the database automatically shuts down even if all of the data in the database is available from replicas. This behavior prevents issues due to network partitioning.

> **Note**
> The physical schema design must meet certain requirements. To create designs that are K-safe, Vertica recommends using the Database Designer .

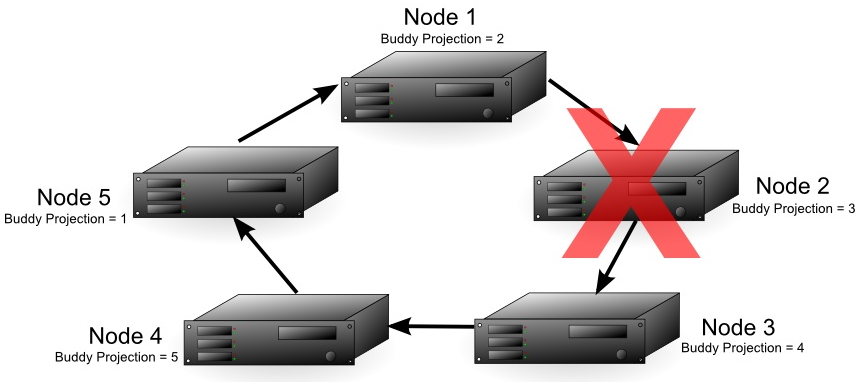
Buddy projections

In order to determine the value of K-safety, Vertica creates buddy projections , which are copies of segmented projections distributed across database nodes. (See Segmented projections and Unsegmented projections .) Vertica distributes segments that contain the same data to different nodes. This ensures that if a node goes down, all the data is available on the remaining nodes.

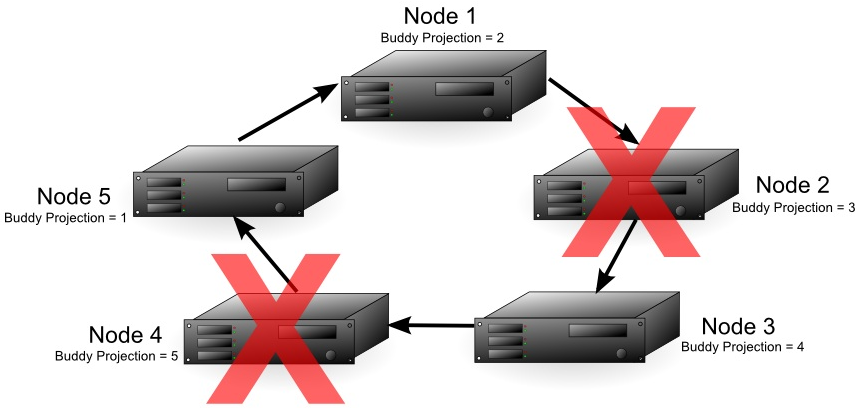
K-Safety example

This diagram above shows a 5-node cluster with a K-safety level of 1. Each node contains buddy projections for the data stored in the next higher node (node 1 has buddy projections for node 2, node 2 has buddy projections for node 3, and so on). If any of the nodes fail, the database continues to run. The database will have lower performance because one of the nodes must handle its own workload and the workload of the failed node.

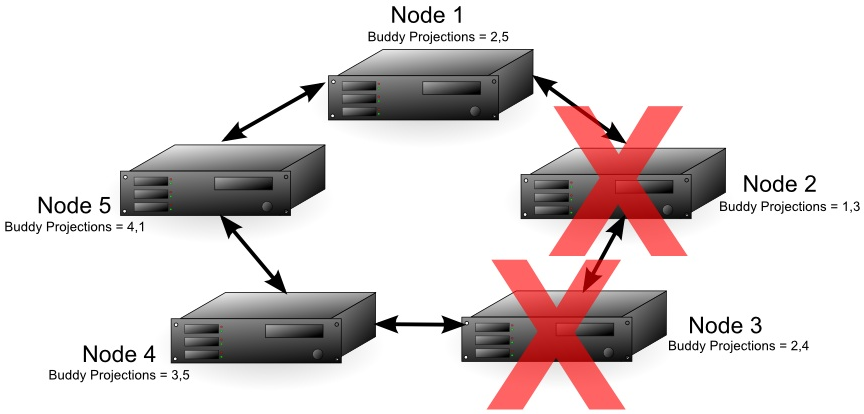
The diagram below shows a failure of Node 2. In this case, Node 1 handles processing for Node 2 since it contains a replica of node 2's data. Node 1 also continues to perform its own processing. The fault tolerance of the database falls from 1 to 0, since a single node failure could cause the database to become unsafe. In this example, if either Node 1 or Node 3 fails, the database becomes unsafe because not all of its data is available. If Node 1 fails,Node 2's data is no longer be available. If Node 3 fails, its data is no longer available, because node 2 is down and could not use the buddy projection. In this case, nodes 1 and 3 are considered critical nodes. In a database with a K-safety level of 1, the node that contains the buddy projection of a failed node, and the node whose buddy projections are on the failed node, always become critical nodes.



With Node 2 down, either node 4 or 5 could fail and the database still has all of its data available. The diagram below shows that if node 4 fails, node 3 can use its buddy projections to fill in for it. In this case, any further loss of nodes results in a database shutdown, since all the nodes in the cluster are now critical nodes. In addition, if one more node were to fail, half or more of the nodes would be down, requiring Vertica to automatically shut down, no matter if all of the data were available or not.



In a database with a K-safety level of 2, Node 2 and any other node in the cluster could fail and the database continues running. The diagram below shows that each node in the cluster contains buddy projections for both of its neighbors (for example, Node 1 contains buddy projections for Node 5 and Node 2). In this case, nodes 2 and 3 could fail and the database continues running. Node 1 could fill in for Node 2 and Node 4 could fill in for Node 3. Due to the requirement that half or more nodes in the cluster be available in order for the database to continue running, the cluster could not continue running if node 5 failed, even though nodes 1 and 4 both have buddy projections for its data.



> **Note**
> Vertica requires that more than half of all nodes in a cluster must always be available; otherwise, it views the database as being in an unsafe state and shuts it down. Thus, in the previous example, the cluster cannot continue running if Node 5 fails, even though nodes 1 and 4 have buddy projections for its data.

Monitoring K-safety

You can access System Tables to monitor and log various aspects of Vertica operation. Use the SYSTEM table to monitor information related to K-safety, such as:

- NODE_COUNT : Number of nodes in the cluster
- NODE_DOWN_COUNT : Number of nodes in the cluster that are currently down
- CURRENT_FAULT_TOLERANCE : The K-safety level

High availability with projections

To ensure high availability and recovery for database clusters of three or more nodes, Vertica:

- Replicates small, unsegmented projections
- Creates buddy projections for large, segmented projections.

Replication (unsegmented projections)

When it creates projections, Database Designer replicates them, creating and storing duplicates of these projections on all nodes in the database.

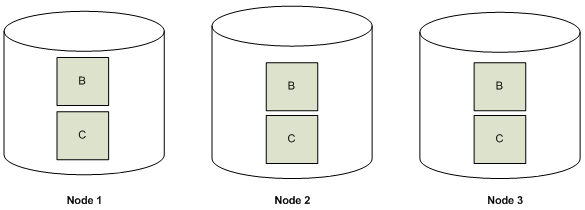
Replication ensures:

- Distributed query execution across multiple nodes.

- High availability and recovery. In a K-safe database, replicated projections serve as buddy projections. This means that you can use a replicated projection on any node for recovery.

> **Note**
> We recommend you use Database Designer to create your physical schema. If you choose not to, be sure to segment all large tables across all database nodes, and replicate small, unsegmented table projections on all database nodes.

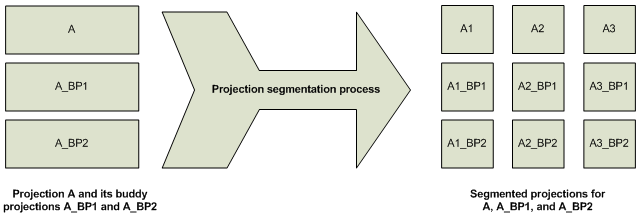
The following illustration shows two projections, B and C, replicated across a three node cluster.



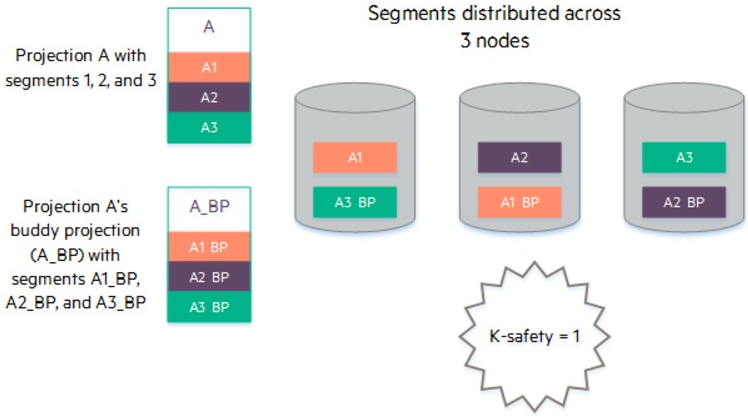
Buddy projections (segmented projections)

Vertica creates *buddy projections* which are copies of segmented projections that are distributed across database nodes (see Segmented projections .) Vertica distributes segments that contain the same data to different nodes. This ensures that if a node goes down, all the data is available on the remaining nodes. Vertica distributes segments to different nodes by using offsets. For example, segments that comprise the first buddy projection (A_BP1) are offset from projection A by one node, and segments from the second buddy projection (A_BP2) are offset from projection A by two nodes.

The following diagram shows the segmentation for a projection called A and its buddy projections, A_BP1 and A_BP2, for a three node cluster.



The following diagram shows how Vertica uses offsets to ensure that every node has a full set of data for the projection.



How result sets are stored

Vertica duplicates table columns on all nodes in the cluster to ensure high availability and recovery. Thus, if one node goes down in a K-Safe environment, the database continues to operate using duplicate data on the remaining nodes. Once the failed node resumes its normal operation, it automatically recovers its lost objects and data by querying other nodes.

Vertica compresses and encodes data to greatly reduce the storage space. It also operates on the encoded data whenever possible to avoid the cost of decoding. This combination of compression and encoding optimizes disk space while maximizing query performance.

Vertica stores table columns as projections. This enables you to optimize the stored data for specific queries and query sets. Vertica provides two methods for storing data:

- Projection segmentation is recommended for large tables (fact and large dimension tables)
- Replication is recommended for the rest of the tables.

High availability with fault groups

Use fault groups to reduce the risk of correlated failures inherent in your physical environment. Correlated failures occur when two or more nodes fail as a result of a single failure. For example, such failures can occur due to problems with shared resources such as power loss, networking issues, or storage.

Vertica minimizes the risk of correlated failures by letting you define fault groups on your cluster. Vertica then uses the fault groups to distribute data segments across the cluster, so the database continues running if a single failure event occurs.

> **Note**
> If your cluster layout is managed by a single network switch, a switch failure can be a single point of failure. Fault groups cannot help with single-point failures.

Vertica supports complex, hierarchical fault groups of different shapes and sizes. You can integrate fault groups with elastic cluster and large cluster arrangements to add cluster flexibility and reliability.

Making Vertica aware of cluster topology with fault groups

You can also use fault groups to make Vertica aware of the topology of the cluster on which your Vertica database is running. Making Vertica aware of your cluster's topology is required when using terrace routing , which can significantly reduce message buffering on a large cluster database.

Automatic fault groups

When you configure a cluster of 120 nodes or more, Vertica automatically creates fault groups around control nodes. *Control nodes* are a subset of cluster nodes that manage spread (control messaging). Vertica places nodes that share a control node in the same fault group. See Large cluster for details.

User-defined fault groups

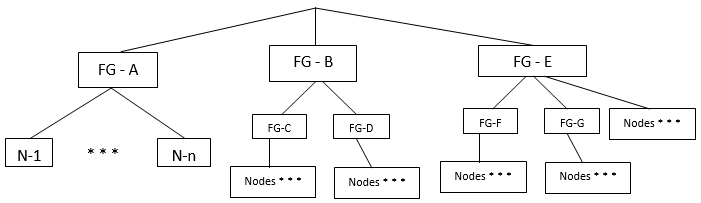
Define your own default groups if:

- Your cluster layout has the potential for correlated failures.
- You want to influence which cluster hosts manage control messaging.

Example cluster topology

The following diagram provides an example of hierarchical fault groups configured on a single cluster:

- Fault group FG–A contains nodes only.
- Fault group FG-B (parent) contains child fault groups FG-C and FG-D . Each child fault group also contain nodes.
- Fault group FG–E (parent) contains child fault groups FG-F and FG-G . The parent fault group FG–E also contains nodes.



How to create fault groups

Before you define fault groups, you must have a thorough knowledge of your physical cluster layout. Fault groups require careful planning.

To define fault groups, create an input file of your cluster arrangement. Then, pass the file to a script supplied by Vertica, and the script returns the SQL statements you need to run. See Fault Groups for details.

Setup

You can create Eon Mode and Enterprise Mode Vertica clusters in cloud and on-premises environments:

- On the cloud: Deploy Vertica clusters running on Amazon Web Services, Microsoft Azure, or Google Cloud Platform. For setup instructions for each cloud provider, see Set up Vertica on the cloud .
- On-premises: Manually install Vertica on your host hardware. For detailed instructions on the manual install process, see Set up Vertica on-premises .

In this section

- Plan your setup
- Set up Vertica on the cloud
- Set up Vertica on-premises
- Upgrading Vertica
- Uninstall Vertica

Plan your setup

Before you get started with Vertica, consider your business needs and available resources. Vertica is built to run in a variety of environments depending on your requirements:

- Infrastructure: Cloud Environment or On-Premises Environment .
- Database Mode: Eon or Enterprise .
- Setup Method (Cloud only): Install Vertica Manually or Deploy Vertica Automatically or Manually .

Choosing an on-premises or cloud environment

You can choose to run Vertica on physical host hardware, or deploy Vertica on the cloud.

Cloud environment

Vertica can run on Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. You might consider running Vertica on cloud resources for any of the following benefits:

- You plan to quickly scale your cluster size up and down to accommodate varying analytic workload. You will provision more computing resources during peak work loads without incurring the same resource costs during low-demand periods. The Vertica database's Eon Mode is designed for this use case.
- You prefer to pay over time (OpEx) for ongoing cloud deployment, rather than the higher up-front cost of buying hardware for on-premises deployment.
- You need to reduce the costs, labor, and expertise involved in maintaining physical on-premises hardware (such as accommodating for server purchases, hardware depreciation, software maintenance, power consumption, floor space, and backup infrastructure).
- You prefer simpler, faster deployment. Deploying on the cloud eliminates the need for more specific hardware expertise during setup. In addition, on cloud platforms such as AWS and GCP, Vertica offers templates that allow you to deploy a pre-configured set of resources on which Vertica and Management Console are already installed, in just a few steps.
- You have very variable workloads and you do not want to pay for idle equipment in a data center when you can simply rent infrastructure when you need it.
- You are a start-up and don't want to build out a data center until your product or service is proven and growing.

If you plan to deploy Vertica on the cloud, see Set up Vertica on the cloud .

On-premises environment

An on-premises environment can provide benefits in cases like the following:

- Your business requirements demand keeping sensitive data on-premises.
- You prefer to pay a higher up-front cost (CapEx) of buying hardware for on-premises deployment, rather than potentially paying a higher long-term total cost of a cloud deployment.
- You cannot rely on continuous access to the internet.
- You prefer end-to-end control over your environment, rather than depending on a third-party cloud provider to store your data.
- You may have already invested in a data center and suitable hardware for Vertica that you want to capitalize on.

If you plan to install Vertica in an on-premises environment, see Set up Vertica on-premises .

Choosing a database mode

You can create a Vertica database in one of two modes: Eon Mode or Enterprise Mode. The mode determines the database's underlying architecture, how the database cluster scales, and how data is loaded. Eon Mode uses communal storage to separate storage from compute and allows you to easily scale compute up or down to meet changing workloads. Enterprise Mode is a share-nothing architecture that's particularly optimized for data local to each node. Database mode does not affect the way you run queries and other everyday tasks while using the database.

For an in-depth explanation of Enterprise Mode and Eon Mode, see Eon vs. Enterprise Mode .

If you choose to create an Eon Mode database in an on-premises environment, see Communal storage for on-premises Eon Mode databases to estimate the amount of storage needed.

Choosing an installation method

After you have decided how you will run Vertica, you can choose which setup method works for your needs.

Install Vertica manually

Manually installing Vertica through the command line works on all platforms. You will first set up a cluster of nodes, then install Vertica.

Manual installation might be right for you if your cluster will have many specific configuration requirements, and you have a database administrator with the expertise to set up the cluster manually on your chosen platform. Manual installation takes more time, but you can configure your cluster to your system's exact needs.

For an on-premises environment, you must install Vertica manually. See Set up Vertica on-premises to get started.

For Amazon AWS, Google Cloud Platform, and Microsoft Azure, you have the option to deploy automatically or install manually. See Set up Vertica on the cloud for information on manual installation on each cloud platform.

> **Note**
> If you manually install Vertica in a cloud environment, you cannot access the Management Console.

Deploy Vertica automatically or manually

Automatic deployment is available on AWS, GCP, and Microsoft Azure. Manual deployment is only available on AWS through Vertica Amazon Machine Images (AMI), which include the Vertica software and the recommended configuration.

Automatic deployment creates a pre-configured environment consisting of cloud resources on which your cluster can run, with Vertica and Management Console already installed. You can enter a few parameters into a template on your chosen platform and be up and running with Vertica.

For cloud-specific deployment information, see Set up Vertica on the cloud .

Set up Vertica on the cloud

This section explains how you can deploy Vertica clusters running on Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP). It assumes that you are familiar with the cloud environment on which you will create your Vertica cluster.

Vertica offers simple, automatic deployment on all three platforms. By setting a few parameters, you can launch a fully functional environment with Vertica and Management Console already installed.

If launching a pre-configured environment doesn't work for your specific needs, you can instead set up your nodes in the cloud and manually install Vertica in order to have more control over your setup. AWS also supports manually deploying a Vertica Amazon Machine Image (AMI) that has Vertica pre-installed and allows for greater control over your environment configuration.

After you set up your environment, you can create a database in either Enterprise Mode or Eon Mode.

Automatic deployment (all cloud platforms)

Vertica offers automatic configuration of resources and quick deployment on the cloud.

AWS

Vertica provides CloudFormation Templates (CFTs) in the AWS Marketplace. You can use a CFT to automatically launch preconfigured AWS resources in minutes, with Vertica and Management Console automatically installed.

Each CFT includes the in-browser Vertica Management Console. When you install Vertica using one of the CFTs, Management Console provides AWS-specific cluster management options, including the ability to quickly create a new cluster and Vertica database.

To deploy Vertica on AWS automatically, see Deploy Vertica using CloudFormation templates .

After deployment, you can create an Eon Mode or Enterprise Mode cluster and database using Management Console.

For more information about the AWS environment, see the official Amazon Web Services documentation .

Google Cloud Platform

For GCP, Vertica provides an automated installer that is available from the Google Cloud Marketplace.

Input a few parameters, and the Google Cloud Launcher will deploy the Vertica solution, including your new database. You can create up to a 16-node cluster. The solution includes the Vertica Management Console as the primary UI to get started.

To deploy Vertica on GCP automatically, see Deploy Vertica from the Google cloud marketplace .

For more information about the GCP environment, see the official Google Cloud Platform documentation .

Microsoft Azure

Vertica offers a fully automated cluster deployment from the Microsoft Azure Marketplace. This solution will automatically deploy a Vertica cluster and create an initial database, allowing you to log in to the Vertica Management Console and start using it after deployment has finished.

To deploy Vertica on Azure automatically, see Deploy Vertica from the Azure Marketplace .

For more information about the Azure environment, see the official Microsoft Azure documentation.

Manual installation and deployment (GCP, azure)

Manual installation might be the right option for you if you have many specific configuration requirements, and have an administrator who is familiar with setting up and maintaining cloud resources in the environment of your choice. Setup and maintenance may take longer and requires more expertise, but you will have more control over how your cluster is configured.

The process of installing Vertica manually on cloud resources is very similar to doing so with on-premises hardware.

If you manually install the Vertica server on cloud hosts, you cannot access Management Console.

Vertica offers cloud-specific manual installation instructions for GCP and Azure. If you want more control over your AWS cluster configuration, see Manual Deployment . Before you install, make sure to refer to the documentation of the platform you are using in order to set up your cloud resources correctly.

Google Cloud Platform

To install Vertica on GCP manually, see the GCP section of the Vertica documentation: Vertica on Google Cloud Platform .

Refer to the official Google Cloud Platform documentation for more detail on setting up your GCP resources.

Microsoft Azure

To install Vertica on Azure manually, see the Azure section of the Vertica documentation: Vertica on Microsoft Azure .

Refer to the official Microsoft Azure documentation for more detail on setting up your Azure resources.

Manual deployment (AWS)

Manual deployment is only available on AWS, which supports Amazon Machine Images (AMI) that include the Vertica software and the recommended configuration. The Vertica AMI acts as a template, requiring fewer configuration steps than a manual installation but still allowing control over your configuration. Vertica provides AMIs for both Management Console and cluster hosts.

For more information about the manual deployment process, see Manually deploy Vertica on AWS .

In this section

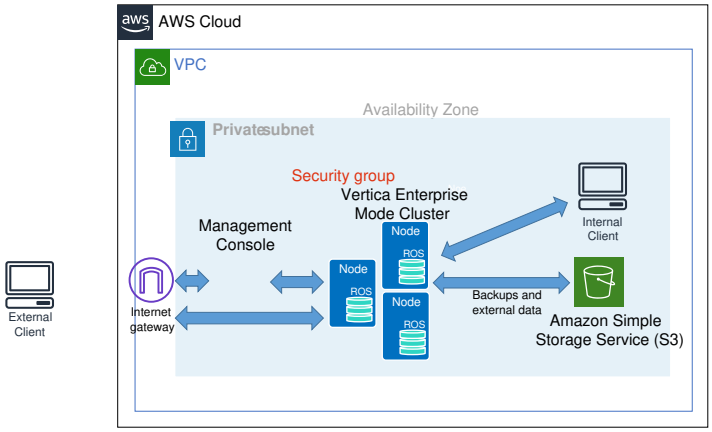
- Vertica on Amazon Web Services
- Vertica on Microsoft Azure
- Vertica on Google Cloud Platform

Vertica on Amazon Web Services

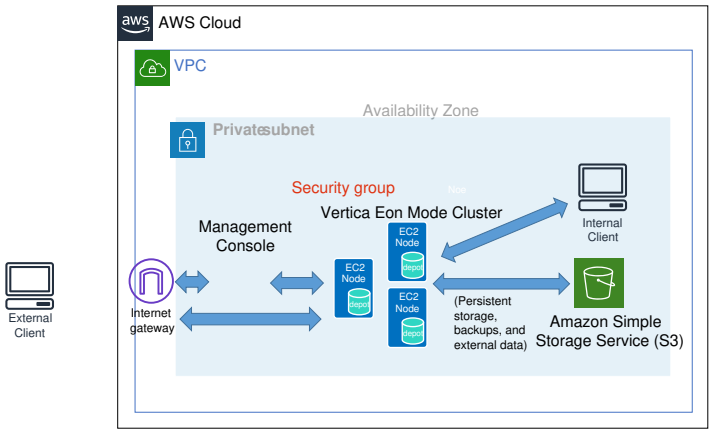
Vertica clusters on AWS can operate on EC2 instances automatically provisioned using a CloudFormation Template (CFT), or manually deployed using Amazon Machine Images (AMIs). For information about these deployment methods, see Deploy Vertica using CloudFormation templates and Manually deploy Vertica on AWS .

You can deploy a Vertica database on AWS running in either Enterprise Mode or Eon Mode. The differences between these two modes lay in their architecture, deployment, and scalability:

- **Enterprise Mode** stores data locally on the nodes in the database.



- **Eon Mode** stores its data in an S3 bucket.



Eon Mode separates the computational processes from the communal storage layer of your database. This separation lets you elastically vary the number of nodes in your database cluster to adjust to varying workloads.

Vertica also supports the following AWS features:

- **Enhanced Networking** : Vertica recommends that you use the AWS enhanced networking for optimal performance. For more information, see Enabling Enhanced Networking on Linux Instances in a VPC in the AWS documentation.
- **Command Line Interface** : Use the Amazon command-line Interface (CLI) with your Vertica AMIs. For more information, see What Is the AWS Command Line Interface? .
- **Elastic Load Balancing** : Use elastic load balancing (ELB) for queries up to one hour. When enabling ELB, configure the timer to 3600 seconds. For more information see Elastic Load Balancing in the AWS documentation.

For more information about Amazon cluster instances and their limitations, see the Amazon documentation .

In this section

In this section

- Supported AWS instance types
- AWS authentication
- Deploy Vertica using CloudFormation templates
- Manually deploy Vertica on AWS

Supported AWS instance types

Vertica supports a range of Amazon Web Services instance types, each optimized for different purposes. Choose the instance type that best matches your requirements. The two tables below list the AWS instance types that Vertica supports for Vertica cluster hosts, and for use in MC. For more information, see the Amazon Web Services documentation on instance types and volumes .

Important

If you plan to use an Amazon Machine Image (AMI) on multiple AWS accounts, make sure to subscribe to the image on all your accounts. This allows you to access an image even when it is delisted from the AWS Marketplace.

Instance types for Vertica cluster hosts

Each Amazon EC2 Instance type natively provides one of the following storage options:

- Elastic Block Store (EBS) provides durable storage: Data files stored on instance persist after instance is stopped.

- Instance Store provides temporary storage: Data files stored on instance are lost when instance is stopped.

Vertica AMIs can use either the Instance Metadata Service Version 1 (IMDSv1) or the Instance Metadata Service Version 2 (IMDSv2) to authenticate to AWS services, including S3.

For more information about storage configuration in AWS, see Configure storage .

> **Note**
> Instance types that support EBS volumes support encrypting.

Optimization	Instance Types Using *Only* EBS Volumes (Durable)	Instance Types Using Instance Store Volumes (Temporary)
General purpose	m4.4xlarge m4.10xlarge m5.4xlarge m5.8xlarge m5.12xlarge	m5d.4xlarge m5d.8xlarge m5d.12xlarge
Compute	c4.4xlarge c4.8xlarge c5.4xlarge c5.9xlarge c6i.4xlarge c6i.8xlarge c6i.12xlarge c6i.16xlarge c6i.24xlarge c6i.32xlarge	c3.4xlarge c3.8xlarge c5d.4xlarge c5d.9xlarge
Memory	r4.4xlarge r4.8xlarge r4.16xlarge r5.4xlarge r5.8xlarge r5.12xlarge r6i.4xlarge r6i.8xlarge r6i.12xlarge r6i.16xlarge r6i.24xlarge r6i.32xlarge	r3.4xlarge r3.8xlarge r5d.4xlarge r5d.8xlarge r5d.12xlarge

Storage		d2.4xlarge d2.8xlarge i3.4xlarge i3.8xlarge i3.16xlarge i3en.3xlarge i3en.6xlarge i3en.12xlarge i4i.4xlarge i4i.8xlarge i4i.16xlarge

Note
By default, the c4.8xlarge, d2.8xlarge, and m4.10xlarge instances have their processor C-states set to a value of 1 in the Vertica AMI. This measure is meant to improve performance by limiting the sleep states that an instance running Vertica uses.

For more information about sleep states, visit the AWS Documentation .

Instance types available for MC hosts

Optimization	Type	Supports EBS Storage (Durable)	Supports Ephemeral Storage (Temporary)
Computing	c4.large	Yes	No
	c4.xlarge	Yes	No
	c5.large	Yes	No
	c5.xlarge	Yes	No

Choosing AWS Eon Mode instance types

When running an Eon Mode database in AWS, choose instance types that support ephemeral instance storage or EBS volumes for your depot, depending on cost and availability. Vertica recommends either r4 or i3 instances for production clusters. It is not mandatory to have an EBS-backed depot, because in Eon Mode, a copy of the data is safely stored in communal storage. However, you must have an EBS-backed catalog for Eon Mode databases.

The following table provides information to help you make a decision on how to pick instances with ephemeral instance storage or EBS only storage. Check with Amazon Web Services for the latest cost per hour.

Important
If you select instances that use instance store, if you then terminate those instances there is the potential for data loss. For Eon mode, MC displays an alert to inform the user of the potential data loss when terminating instances that support instance store.

Storage Type	Instance Type	Pros/Cons
Instance storage	i3.8xlarge	Instance storage offers better performance than EBS attached storage through multiple EBS volumes. Instance storage can be striped (RAIDed) together to increase throughput and load balance I/O. Data stored in instance-store volumes is not persistent through instance stops, terminations, or hardware failures.

EBS-only storage	r4.8xlarge with 600 GB EBS volume attached	Newer instance types from AWS have only the EBS option. In most AWS regions, it's easier to provision a large number of instances. You can terminate an instance but leave the EBS volume around for faster revive. Perserving the EBS will preserve the depot. While some of the cached files might have become stale, they will be ignored and evicted. Much of the cached data will not be stale. It will save time when the node revives and warms its depot. Take advantage of full-volume encryption.

More information

For more information about Amazon cluster instances and their limitations, see Manage Clusters in the Amazon Web Services documentation.

AWS authentication

Amazon defines two ways to control access to AWS resources such as S3: IAM roles and the combination of id, secrets, and (optionally) session tokens. For long-term access to non-communal storage buckets, you should use IAM roles for access control centralization. You do not need to change your application's configuration if you want to change its access settings. You just alter the IAM role applied to your EC2 instances.

However, for one-time tasks like backing up and restoring the database or loading data to and from non-communal storage buckets, you should use an AWS access key .

Vertica uses both of these authentication methods to support different features and use cases:

- An Eon Mode database's access to S3 for communal and catalog storage must always use IAM role authentication. IAM roles are the default access control method for AWS resources. Vertica uses this method if you do not configure the legacy access control session parameters.
- Individual users can read data from S3 storage locations other than the ones Vertica uses for communal storage. For example, users can use COPY to load data into Vertica from an S3 bucket or query an external table stored on S3. If the IAM role assigned to the Vertica nodes does not have access to this external S3 data, the user must set an id, secret, and optionally an access token in session variables to authorize access to it. These session variables override the IAM role set on the server. See S3 parameters for a list of these session parameters.
- Individual users can export data to S3 using file export . File export cannot use IAM authorization. Users who want to export data to S3 must set id, secret, and optionally access token values in session variables.

Important

If the database is running in Eon Mode, using id and secret authentication is more complex. In addition to having access to the external S3 data, any id that a user sets must be authorized to read from and write to the S3 storage locations that Vertica uses to store communal and catalog data. The queries that the user executes uses this id for all storage requests, not just those for accessing external S3 data. If the id does not have access to the catalog and communal storage, the user cannot execute queries.

Configuring an IAM role

To configure an IAM role to grant Vertica to access AWS resources you must:

1. Create an IAM role to allow EC2 instances to access the specific resources.
2. Grant that role permission to access your resources.
3. Attach this IAM role to each EC2 instance in the Vertica cluster.

To see an example of IAM roles for a Vertica cluster, look at the roles defined in one of the Cloud Formation Templates provided by Vertica. You can download these templates from any of the Vertica entries in the Amazon Marketplace . Under each entry's Usage Information section, click the View CloudFormation Template link, then click Download CloudFormation Template.

For more information about IAM roles, see IAM Roles for Amazon EC2 in the AWS documentation.

Deploy Vertica using CloudFormation templates

Vertica provides CloudFormation Templates (CFTs) on the AWS Marketplace that allow you to get a cluster up and running quickly. Using the template allows you to automatically provision your AWS resources and launch a Vertica cluster and Management Console, with minimal configuration required.

If you prefer to deploy a VPC, instances, and related resources manually, see Manually deploy Vertica on AWS .

For details about creating an Eon Mode or Enterprise Mode database after you create a cluster with CFTs, see Amazon Web Services in MC .

In this section

- CloudFormation template (CFT) overview

- Creating a Vritual Private Cloud
- Deploy MC and AWS resources with a CloudFormation template
- Access Management Console

CloudFormation template (CFT) overview

With Vertica on AWS, use CloudFormation Templates (CFTs) to easily manage provisioning the AWS resources with a running Vertica system. After you provide a few parameters to the template, you can create a stack to automatically provision the AWS resources for your Vertica system.

To access Vertica CFTs, go to the AWS Marketplace .

CFT licensing models

Licensing models for CFTs are:

- **Bring Your Own License (BYOL)** : By default, free CE license is installed with 3 nodes and 1 TB. To extend nodes or size, you can purchase the Vertica BYOL license.
  Outside of the BYOL license on CFTs, you can also access the Community Edition without a license file:
  - If you are using Management Console, simply leave the license field blank.
  - If you are using a command line (see Install Vertica with the installation script), specify CE in the --license parameter during installation.
- **By the Hour** : A pay-as-you-go model where you pay for only the number of hours you use for each node. One advantage of using the Paid Listing is that all charges appear on your Amazon AWS bill. This offers an alternative to purchasing a full Vertica license. This eliminates the need to compute potential storage needs in advance.

CFT prerequisites

Before you can deploy Vertica on AWS using CloudFormation Templates (CFTs), verify that you have:

- AWS account with permissions to create a VPC, subnet, security group, EC2 instances, and IAM roles (For more information about AWS accounts, see the AWS documentation)
- Amazon key pair for SSH access to an EC2 instance. (See the AWS documentation for key pairs .)

Supported CFTs and Vertica offerings

Available Vertica CFTs are:

- **Management Console with 3 Vertica nodes** : The easiest way to deploy Vertica. This CFT deploys an Eon Mode database by default. However, this environment can also be used to create an Enterprise Mode database. For more information, see Creating a database .
- **Deploy Management Console into new VPC** : This CFT deploys all required AWS resources and installs the Vertica Management Console (MC). After stack creation completes, log in to the MC to provision a Vertica database cluster.
- **Deploy Management Console into existing VPC** : This CFT deploys the Vertica Management Console (MC) in an already-existing VPC and subnet. After stack creation completes, the MC is available. Log in to MC to provision either a Vertica database cluster or an Eon Mode database cluster. For this CFT, you must first set up the VPC, subnet, and related network resources. For more information about the correct configuration of these resources for Vertica, see the following topics in the AWS documentation: * Creating a Vritual Private Cloud * Configure the network

Using the license models and supported CFTs, you can deploy the following Vertica products:

- Vertica BYOL, Amazon Linux 2.0
- Vertica by the Hour, Amazon Linux 2.0
- Vertica BYOL, Red Hat
- Vertica by the Hour, Red Hat

See Deploy MC and AWS resources with a CloudFormation template for information on deploying these products.

Creating a Vritual Private Cloud

A Vertica cluster on AWS must be logically located in the same network. This is similar to placing the nodes of an on-premises cluster within the same network. Create a virtual private cloud (VPC) to ensure the nodes in your cluster will be able to communicate with each other within AWS.

Create a single public subnet VPC with the following configurations:

- Assign a Network Access Control List (ACL) that is appropriate to your situation. The default ACL does not provide a high level of security.
- Enable DNS resolution and enable DNS hostname support for instances launched in this VPC.
- Add the required network inbound and outbound rules to the Network ACL associated to the VPC .

> **Note**
> A Vertica cluster must be operated within a single availability zone.

For more information about VPCs, including how to create one, see the AWS documentation .

Deploy MC and AWS resources with a CloudFormation template

You can deploy Management Console (MC) and its associated AWS resources using CloudFormation templates (CFTs) that are available through the AWS Marketplace. For a list of available CFTs, see CloudFormation template (CFT) overview .

Complete the following to deploy the Vertica MC and related resources in AWS:

1. Log in to the AWS Marketplace with an AWS account (see the **Prerequisites** section above).
2. Search for "Vertica" in the AWS Marketplace.
3. Select a Vertica CFT. Each CFT leads you to a product overview page, with pricing estimates. (Also see CloudFormation template (CFT) overview for an overview of available templates and products).
4. Click Continue to Subscribe.
5. On the next page, select your launch settings based on your requirements for deployment.
6. If you have not agreed to Vertica EULA terms on the AWS Marketplace before, click Accept Software Terms to subscribe.
7. Click **Launch with CloudFormation Console** . The CloudFormation Console opens.
8. The CloudFormation Console automatically supplies the URL in the **Specify an Amazon S3 template URL field** . Click **Next** .
9. Follow the CloudFormation workflow and enter the parameters (collectively called a stack).

> **Note**
> **Important:** Take note of the username and password you set for Management Console during this step. You cannot recover or reset these credentials after you create the stack.

10. After confirming the details you have provided for your new stack, click **Create** . The AWS console brings you to the Stacks page, where you can view the progress of the creation process. The process takes several minutes.
11. The **Outputs** tab displays information about accessing your environment after the process completes.

Next, access the Management Console (MC) to deploy your cluster instances and create a database, as described in Access Management Console .

Access Management Console

Complete the following steps to access Management Console on your deployed AWS resources:

1. On the AWS CloudFormation Stacks page, select your new stack and view the **Outputs** tab. This tab provides information about accessing your environment, as well as documentation and licensing resources.
2. In the **ManagementConsole** row, select the URL in the Value column to open the MC login page.
3. To log in, enter the MC username and password that you created using the CloudFormation Console.
   After login, MC displays the home page, with options to provision a new cluster or database or import existing ones. If you chose a CFT that also creates a database, your new database is also displayed on the home page.
   This page also provides a Resources section with links to online training, blogs, community, and help resources.

You have successfully launched and connected to Management Console on AWS resources.

If you have not yet provisioned a Vertica cluster and database, complete the steps in one of the following:

- Creating an Eon Mode database in AWS with MC
- Creating an Enterprise Mode database in AWS with MC

Manually deploy Vertica on AWS

Vertica provides tested and pre-configured Amazon Machine Images (AMIs) to deploy cluster hosts or MC hosts on AWS. When you create an EC2 instance on AWS using a Vertica AMI, the instance includes the Vertica software and the recommended configuration. The Vertica AMI acts as a template, requiring fewer configuration steps.

This section will guide you through configuring your network settings on AWS, launching and preparing EC2 instances using the Vertica AMI, and creating a Vertica cluster on those EC2 instances.

Choose this method of installation if you are familiar with configuring AWS and have many specific AWS configuration needs. To automatically deploy AWS resources and a Vertica cluster instead, see Deploy Vertica using CloudFormation templates .

In this section

- Configure your network
- Deploy AWS instances for your Vertica database cluster

Configure your network

Before you deploy your cluster, you must configure the network on which Vertica will run. Vertica requires a number of specific network configurations to operate on AWS. You may also have specific network configuration needs beyond the default Vertica settings.

Important
You can create a Vertica database that uses IPv6 for internal communications running on AWS. However, if you do so, you must identify the hosts in your cluster using IP addresses rather than host names. The AWS DNS resolution service is incompatible with IPv6.

The following sections explain which Amazon EC2 features you need to configure for instance creation.

In this section

- Create a placement group, key pair, and VPC
- Network ACL settings
- Configure TCP keepalive with AWS network load balancer
- Create and assign an internet gateway
- Assign an elastic IP address
- Create a security group

Create a placement group, key pair, and VPC

Part of configuring your network for AWS is to create the following:

- Placement Group
- Key Pair
- Virtual Private Cloud (VPC)

Create a placement group

A placement group is a logical grouping of instances in a single Availability Zone . Placement Groups are required for clusters and all Vertica nodes must be in the same Placement Group.

Vertica recommends placement groups for applications that benefit from low network latency, high network throughput, or both. To provide the lowest latency, and the highest packet-per-second network performance for your Placement Group, choose an instance type that supports enhanced networking.

For information on creating placement groups, see Placement Groups in the AWS documentation.

Create a key pair

You need a key pair to access your instances using SSH. Create the key pair using the AWS interface and store a copy of your key (*.pem) file on your local machine. When you access an instance, you need to know the local path of your key.

Use a key pair to:

- Authenticate your connection as dbadmin to your instances from outside your cluster.
- Install and configure Vertica on your AWS instances.

for information on creating a key pair, see Amazon EC2 Key Pairs in the AWS documentation.

Create a virtual private cloud (VPC)

You create a Virtual Private Cloud (VPC) on Amazon so that you can create a network of your EC2 instances. Your instances in the VPC all share the same network and security settings.

A Vertica cluster on AWS must be logically located in the same network. Create a VPC to ensure the nodes in you cluster can communicate with each other in AWS.

Create a single public subnet VPC with the following configurations:

- Assign a Network Access Control List (ACL) that is appropriate to your situation.
- Enable DNS resolution and enable DNS hostname support for instances launched in this VPC.
- Add the required network inbound and outbound rules to the Network ACL associated to the VPC.

> **Note**
> A Vertica cluster must be operated in a single availability zone.

For information on creating a VPC, see Create a Virtual Private Cloud (VPC) in the AWS documentation.

Network ACL settings

Vertica requires the following basic network access control list (ACL) settings on an AWS instance running the Vertica AMI. Vertica recommends that you secure your network with additional ACL settings that are appropriate to your situation.

Inbound Rules

Type	Protocol	Port Range	Use	Source	Allow/Deny
SSH	TCP (6)	22	SSH (Optional—for access to your cluster from outside your VPC)	User Specific	Allow
Custom TCP Rule	TCP (6)	5450	MC (Optional—for MC running outside of your VPC)	User Specific	Allow
Custom TCP Rule	TCP (6)	5433	SQL Clients (Optional—for access to your cluster from SQL clients)	User Specific	Allow
Custom TCP Rule	TCP (6)	50000	Rsync (Optional—for backup outside of your VPC)	User Specific	Allow
Custom TCP Rule	TCP (6)	1024-65535	Ephemeral Ports (Needed if you use any of the above)	User Specific	Allow
ALL Traffic	ALL	ALL	N/A	0.0.0.0/0	Deny

Outbound Rules

Type	Protocol	Port Range	Use	Source	Allow/Deny
Custom TCP Rule	TCP (6)	0–65535	Ephemeral Ports	0.0.0.0/0	Allow

You can use the entire port range specified in the previous table, or find your specific ephemeral ports by entering the following command:

```
$ cat /proc/sys/net/ipv4/ip_local_port_range
```

More information

For detailed information on network ACLs within AWS, refer to Network ACLs in the Amazon documentation.

For detailed information on ephemeral ports within AWS, refer to Ephemeral Ports in the Amazon documentation.

Configure TCP keepalive with AWS network load balancer

AWS supports three types of elastic load balancers (ELBs):

- Classic Load Balancers
- Application Load Balancers
- Network Load Balancers

Vertica strongly recommends the AWS Network Load Balancer (NLB), which provides the best performance with your Vertica database. The Network Load Balancer acts as a proxy between clients (such as JDBC) and Vertica servers. The Classic and Application Load Balancers do not work with Vertica, in Enterprise Mode or Eon Mode.

To avoid timeouts and hangs when connecting to Vertica through the NLB, it is important to understand how AWS NLB handles idle timeouts for connections. For the NLB, AWS sets the idle timeout value to 350 seconds and you cannot change this value. The timeout applies to both connection points.

For a long-running query, if either the client or the server fails to send a timely keepalive, that side of the connection is terminated. This can lead to situations where a JDBC client hangs waiting for results that would never be returned because the server fails to send a keepalive within 350 seconds.

To identify an idle timeout/keepalive issue, run a query like this via a client such as JDBC:

```
=> SELECT SLEEP(355);
```

If there's a problem, one of the following situations occurs:

- The client connection terminates before 355 seconds. In this case, lower the JDBC keepalive setting so that keepalives are sent less than 350 seconds apart.
- The client connection doesn't return a result after 355 seconds. In this case, you need to adjust the server keepalive settings (tcp_keepalive_time and tcp_keepalive_intvl) so that keepalives are sent less than 350 seconds apart.
  You can adjust the keepalive settings on the server, or you can adjust them in Vertica .

For detailed information about AWS Network Load Balancers, see the AWS documentation .

Create and assign an internet gateway

When you create a VPC, an Internet gateway is automatically assigned to it. You can use that gateway, or you can assign your own. If you are using the default Internet gateway, continue with the procedure described in Create a security group .

Otherwise, create an Internet gateway specific to your needs. Associate that internet gateway with your VPC and subnet.

For information about how to create an Internet Gateway, see Internet Gateways in the AWS documentation.

Assign an elastic IP address

An elastic IP address is an unchanging IP address that you can use to connect to your cluster externally. Vertica recommends you assign a single elastic IP to a node in your cluster. You can then connect to other nodes in your cluster from your primary node using their internal IP addresses dictated by your VPC settings.

Create an elastic IP address. For information, see Elastic IP Addresses in the AWS documentation.

Create a security group

The Vertica AMI has specific security group requirements. When you create a Virtual Private Cloud (VPC), AWS automatically creates a default security group and assigns it to the VPC. You can use the default security group, or you can name and assign your own.

Create and name your own security group using the following basic security group settings. You may make additional modifications based on your specific needs.

Inbound

Type	Use	Protocol	Port Range	IP
SSH		TCP	22	The CIDR address range of administrative systems that require SSH access to the Vertica nodes. Make this range as restrictive as possible. You can add multiple rules for separate network ranges, if necessary.
DNS (UDP)		UDP	53	Your private subnet address range (for example, 10.0.0.0/24).
Custom UDP	Spread	UDP	4803 and 4804	Your private subnet address range (for example, 10.0.0.0/24).
Custom TCP	Spread	TCP	4803	Your private subnet address range (for example, 10.0.0.0/24).
Custom TCP	VSQL/SQL	TCP	5433	The CIDR address range of client systems that require access to the Vertica nodes. This range should be as restrictive as possible. You can add multiple rules for separate network ranges, if necessary.
Custom TCP	Inter-node Communication	TCP	5434	Your private subnet address range (for example, 10.0.0.0/24).

Custom TCP		TCP	5444	Your private subnet address range (for example, 10.0.0.0/24).
Custom TCP	MC	TCP	5450	The CIDR address of client systems that require access to the management console. This range should be as restrictive as possible. You can add multiple rules for separate network ranges, if necessary.
Custom TCP	Rsync	TCP	50000	Your private subnet address range (for example, 10.0.0.0/24).
ICMP	Installer	Echo Reply	N/A	Your private subnet address range (for example, 10.0.0.0/24).
ICMP	Installer	Traceroute	N/A	Your private subnet address range (for example, 10.0.0.0/24).

Note
In Management Console (MC), the Java IANA discovery process uses port 7 once to detect if an IP address is reachable before the database import operation. Vertica tries port 7 first. If port 7 is blocked, Vertica switches to port 22.

Outbound

Type	Protocol	Port Range	Destination	IP
All TCP	TCP	0-65535	Anywhere	0.0.0.0/0
All ICMP	ICMP	0-65535	Anywhere	0.0.0.0/0
All UDP	UDP	0-65535	Anywhere	0.0.0.0/0

For information about what a security group is, as well as how to create one, see Amazon EC2 Security Groups for Linux Instances in the AWS documentation.

Deploy AWS instances for your Vertica database cluster

After you Configure your network, you can create AWS instances and deploy Vertica. Follow these procedures to deploy and run Vertica on AWS.

In this section

- Configure and launch an instance
- Connect to an instance
- Prepare instances for cluster formation
- Change instances on AWS
- Configure storage
- Create a cluster
- Management Console on AWS

Configure and launch an instance

After you configure your network settings on AWS, configure and launch the instances where you will install Vertica. An Elastic Compute Cloud (EC2) instance without a Vertica AMI is similar to a traditional host. Just like with an on-premises cluster, you must prepare and configure your cluster and network at the hardware level before you can install Vertica.

When you create an EC2 instance on AWS using a Vertica AMI, the instance includes the Vertica software and the recommended configuration. Vertica recommends that you use the Vertica AMI unmodified. The Vertica AMI acts as a template, requiring fewer configuration steps:

1. Choose a Vertica AMI Operating Systems
2. Configure EC2 instances.
3. Add storage to instances.
4. Optionally, configure EBS volumes as a RAID array.
5. Set the security group and S3 access.
6. Launch instances and verify they are running.

Choose a Vertica AMI operating systems

Vertica provides Vertica and Management Console AMIs in the following operating systems.

- Red Hat 7.4 and later
- Amazon Linux 2.0 and later

You can use the AMI to deploy MC hosts or cluster hosts. For more information, see the AWS Marketplace .

Important
When using Amazon Linux 2.0, you must compile C++ UDX libraries with a supported gcc version. For more information see Setting up a development environment .

Configure EC2 instances in AWS

1. Select a Vertica AMI from the AWS marketplace.For instance type recommendations for Eon Mode databases, see Choosing AWS Eon Mode Instance Types .
2. Select the desired fulfillment method.
3. Configure the following:
   - Supported instance type
   - Number of instances to launch. A Vertica cluster usually uses identically configured instances of the same type.
   - VPC placement group

Add storage to instances

Consider the following issues when you add storage to your instances:

- Add a number of drives equal to the number of physical cores in your instance—for example, for a c3.8xlarge instance, 16 drives; for an r3.4xlarge, 8 drives.
- Do not store your information on the root volume.
- Amazon EBS provides durable, block-level storage volumes that you can attach to running instances. For guidance on selecting and configuring an Amazon EBS volume type, see Amazon EBS Volume Types .

Configure EBS volumes as a RAID array

You can configure your EBS volumes into a RAID 0 array to improve disk performance. Before doing so, use the vioperf utility to determine whether the performance of the EBS volumes is fast enough without using them in a RAID array. Pass vioperf the path to a mount point for an EBS volume. In this example, an EBS volume is mounted on a directory named /vertica/data:

```
[dbadmin@ip-10-11-12-13 ~]$ /opt/vertica/bin/vioperf /vertica/data

The minimum required I/O is 20 MB/s read and write per physical processor core on
each node, in full duplex i.e. reading and writing at this rate simultaneously,
concurrently on all nodes of the cluster. The recommended I/O is 40 MB/s per
physical core on each node. For example, the I/O rate for a server node with 2
hyper-threaded six-core CPUs is 240 MB/s required minimum, 480 MB/s recommended.

Using direct io (buffer size=1048576, alignment=512) for directory "/vertica/data"

test      | directory     | counter name         | counter | counter   | counter       | counter       | thread | %CPU  | %IO Wait  | elapsed | remaining
          |               |                      | value   | value (10 | value/core    | value/core    | count  |       |           | time (s)| time (s)
          |               |                      |         | sec avg)  |               | (10 sec avg)  |        |       |           |         |
----------------------------------------------------------------------------------------------------------------------------------------------------------
Write     | /vertica/data | MB/s                 | 259     | 259       | 32.375        | 32.375        | 8      | 4     | 11        | 10      | 65
Write     | /vertica/data | MB/s                 | 248     | 232       | 31            | 29            | 8      | 4     | 11        | 20      | 55
Write     | /vertica/data | MB/s                 | 240     | 234       | 30            | 29.25         | 8      | 4     | 11        | 30      | 45
Write     | /vertica/data | MB/s                 | 240     | 233       | 30            | 29.125        | 8      | 4     | 13        | 40      | 35
Write     | /vertica/data | MB/s                 | 240     | 233       | 30            | 29.125        | 8      | 4     | 13        | 50      | 25
Write     | /vertica/data | MB/s                 | 240     | 232       | 30            | 29            | 8      | 4     | 12        | 60      | 15
Write     | /vertica/data | MB/s                 | 240     | 238       | 30            | 29.75         | 8      | 4     | 12        | 70      | 5
Write     | /vertica/data | MB/s                 | 240     | 235       | 30            | 29.375        | 8      | 4     | 12        | 75      | 0
ReWrite   | /vertica/data | (MB-read+MB-write)/s| 237+237 | 237+237   | 29.625+29.625 | 29.625+29.625 | 8      | 4     | 22        | 10      | 65
ReWrite   | /vertica/data | (MB-read+MB-write)/s| 235+235 | 234+234   | 29.375+29.375 | 29.25+29.25   | 8      | 4     | 20        | 20      | 55
ReWrite   | /vertica/data | (MB-read+MB-write)/s| 234+234 | 235+235   | 29.25+29.25   | 29.375+29.375 | 8      | 4     | 20        | 30      | 45
ReWrite   | /vertica/data | (MB-read+MB-write)/s| 233+233 | 234+234   | 29.125+29.125 | 29.25+29.25   | 8      | 4     | 18        | 40      | 35
ReWrite   | /vertica/data | (MB-read+MB-write)/s| 233+233 | 234+234   | 29.125+29.125 | 29.25+29.25   | 8      | 4     | 20        | 50      | 25
ReWrite   | /vertica/data | (MB-read+MB-write)/s| 234+234 | 235+235   | 29.25+29.25   | 29.375+29.375 | 8      | 3     | 19        | 60      | 15
ReWrite   | /vertica/data | (MB-read+MB-write)/s| 233+233 | 236+236   | 29.125+29.125 | 29.5+29.5     | 8      | 4     | 21        | 70      | 5
ReWrite   | /vertica/data | (MB-read+MB-write)/s| 232+232 | 236+236   | 29+29         | 29.5+29.5     | 8      | 4     | 21        | 75      | 0
Read      | /vertica/data | MB/s                 | 248     | 248       | 31            | 31            | 8      | 4     | 12        | 10      | 65
Read      | /vertica/data | MB/s                 | 241     | 236       | 30.125        | 29.5          | 8      | 4     | 15        | 20      | 55
Read      | /vertica/data | MB/s                 | 240     | 232       | 30            | 29            | 8      | 4     | 10        | 30      | 45
Read      | /vertica/data | MB/s                 | 240     | 232       | 30            | 29            | 8      | 4     | 12        | 40      | 35
Read      | /vertica/data | MB/s                 | 240     | 234       | 30            | 29.25         | 8      | 4     | 12        | 50      | 25
Read      | /vertica/data | MB/s                 | 238     | 235       | 29.75         | 29.375        | 8      | 4     | 15        | 60      | 15
Read      | /vertica/data | MB/s                 | 238     | 232       | 29.75         | 29            | 8      | 4     | 13        | 70      | 5
Read      | /vertica/data | MB/s                 | 238     | 238       | 29.75         | 29.75         | 8      | 3     | 9         | 75      | 0
SkipRead  | /vertica/data | seeks/s              | 22909   | 22909     | 2863.62       | 2863.62       | 8      | 0     | 6         | 10      | 65
SkipRead  | /vertica/data | seeks/s              | 21989   | 21068     | 2748.62       | 2633.5        | 8      | 0     | 6         | 20      | 55
SkipRead  | /vertica/data | seeks/s              | 21639   | 20936     | 2704.88       | 2617          | 8      | 0     | 7         | 30      | 45
SkipRead  | /vertica/data | seeks/s              | 21478   | 20999     | 2684.75       | 2624.88       | 8      | 0     | 6         | 40      | 35
SkipRead  | /vertica/data | seeks/s              | 21381   | 20995     | 2672.62       | 2624.38       | 8      | 0     | 5         | 50      | 25
SkipRead  | /vertica/data | seeks/s              | 21310   | 20953     | 2663.75       | 2619.12       | 8      | 0     | 5         | 60      | 15
SkipRead  | /vertica/data | seeks/s              | 21280   | 21103     | 2660          | 2637.88       | 8      | 0     | 8         | 70      | 5
SkipRead  | /vertica/data | seeks/s              | 21272   | 21142     | 2659          | 2642.75       | 8      | 0     | 6         | 75      | 0
```

If the EBS volume read and write performance (the entries with Read and Write in column 1 of the output) is greater than 20MB/s per physical processor core (columns 6 and 7), you do not need to configure the EBS volumes as a RAID array to meet the minimum requirements to run Vertica. You may still consider configuring your EBS volumes as a RAID array if the performance is less than the optimal 40MB/s per physical core (as is the case in this example).

Note

If your EC2 instance has hyper-threading enabled, vioperf may incorrectly count the number of cores in your system. The 20MB/s throughput per core requirement only applies to physical cores, rather than virtual cores. If your EC2 instance has hyper-threading enabled, divide the counter value (column 4 in the output) by the number of physical cores. See CPU Cores and Threads Per CPU Core Per Instance Type section in the AWS documentation topic Optimizing CPU Options for a list of physical cores in each instance type.

If you determine you need to configure your EBS volumes as a RAID 0 array, see the AWS documentation topic RAID Configuration on Linux the steps you need to take.

Security group and access

1. Choose between your previously configured security group or the default security group.
2. Configure S3 access for your nodes by creating and assigning an IAM role to your EC2 instance. See AWS authentication for more information.

Connect to an instance

Using your private key, take these steps to connect to your cluster through the instance to which you attached an elastic IPaddress:

1. As the dbadmin user, type the following command, substituting your ssh key:
   ```
   $ ssh --ssh-identity <ssh key> dbadmin@elasticipaddress
   ```
2. Select **Instances** from the Navigation panel.
3. Select the instance that is attached to the Elastic IP.
4. Click **Connect** .
5. On **Connect to Your Instance** , choose one of the following options:
   - **A Java SSH Client directly from my browser** —Add the path to your private key in the field **Private key path** , andclick **Launch SSH Client** .
   - **Connect with a standalone SSH client** **—**Follow the steps required by your standalone SSH client.

Connect to an instance from windows using putty

If you connect to the instance from the Windows operating system, and plan to use Putty:

1. Convert your key file using PuTTYgen.
2. Connect with Putty or WinSCP (connect via the elastic IP), using your converted key (i.e., the *ppk file).
3. Move your key file (the *pem file) to the root dir using Putty or WinSCP.

Prepare instances for cluster formation

After you create your instances, you need to prepare them for cluster formation. Prepare your instances by adding your AWS .pem key and your Vertica license.

By default, each AMI includes a Community Edition license. Once Vertica is installed, you can find the license at this location:

```
/opt/vertica/config/licensing/vertica_community_edition.license.key
```

1. As the dbadmin user, copy your *pem file (from where you saved it locally) onto your primary instance.
   Depending upon the procedure you use to copy the file, the permissions on the file may change. If permissions change, the install_vertica script fails with a message similar to the following:
   ```
   FATAL (19): Failed Login Validation 10.0.3.158, cannot resolve or connect to host as root.
   ```
   If you receive a failure message, enter the following command to correct permissions on your *pem file:
   ```
   $ chmod 600 /<name-of-pem>.pem
   ```
2. Copy your Vertica license over to your primary instance, placing it in your home directory or other known location.

Change instances on AWS

You can change instance types on AWS. For example, you can downgrade a c3.8xlarge instance to c3.4xlarge. See Supported AWS instance types for a list of valid AWS instances.

When you change AWS instances you may need to:

- Reconfigure memory settings
- Reset memory size in a resource pool
- Reset number of CPUs in a resource pool

Reconfigure memory settings

If you change to an AWS instance type that requires a different amount of memory, you may need to recompute the following and then reset the values:

- min_free_kbytes setting
- max_map_count

Note
You may need root user permissions to reset these values.

Reset memory size in a resource pool

If you used absolute memory in a resource pool, you may need to reconfigure the memory using the MEMORYSIZE parameter in ALTER RESOURCE POOL .

Note
If you set memory size as a percentage when you created the original resource pool, you do not need to change it here.

Reset number of CPUs in a resource pool

If your new instance requires a different number of CPUs, you may need to reset the CPUAFFINITYSET parameter in ALTER RESOURCE POOL .

Configure storage

Vertica recommends that you store information — especially your data and catalog directories — on dedicated Amazon EBS volumes formatted with a supported file system. The /opt/vertica/sbin/configure_software_raid.sh script automates the storage configuration process.

Caution
Do not store information on the root volume because it might result in data loss.

Vertica performance tests Eon Mode with a per-node EBS volume of up to 2TB. For best performance, combine multiple EBS volumes into a RAID 0 array.

For more information about RAID 0 arrays and EBS volumes, see RAID configuration on Linux .

Determining volume names

Because the storage configuration script requires the volume names that you want to configure, you must identify the volumes on your machine. The following command lists the contents of the /dev directory. Search for the volumes that begin with xvd :

```
$ ls /dev
```

Important
Ignore the root volume. Do not include any of your root volumes in the RAID creation process.

Combining volumes for storage

The configure_software_raid.sh shell script combines your EBS volumes into a RAID 0 array.

Caution
Run configure_software_raid.sh in the default setting only if you have a fresh configuration with no existing RAID settings.

If you have existing RAID settings, open the script in a text editor and manually edit the raid_dev value to reflect your current RAID settings. If you have existing RAID settings and you do not edit the script, the script deletes important operating system device files.

Alternately, use the Management Console (MC) console to add storage nodes without unwanted changes to operating system device files. For more information, see Managing database clusters .

The following steps combine your EBS volumes into RAID 0 with the configure_software_raid.sh script:

1. Edit the /opt/vertica/sbin/configure_software_raid.sh shell file as follows:
    1. Comment out the safety exit command at the beginning .
    2. Change the sample volume names to your own volume names, which you noted previously. Add more volumes, if necessary.
2. Run the /opt/vertica/sbin/configure_software_raid.sh shell file. Running this file creates a RAID 0 volume and mounts it to /vertica/data .
3. Change the owner of the newly created volume to dbadmin with chown .

4. Repeat steps 1-3 for each node on your cluster.

Create a cluster

On AWS, use the install_vertica script to combine instances and create a cluster. Check your **My Instances** page on AWS for a list of current instances and their associated IP addresses. You need these IP addresses when you run install_vertica .

Create a cluster as follows:

1. While connected to your primary instance, enter the following command to combine your instances into a cluster. Substitute the IP addresses for your instances and include your root *.pem file name.

```
$ sudo /opt/vertica/sbin/install_vertica --hosts 10.0.11.164,10.0.11.165,10.0.11.166 \
 --dba-user-password-disabled --point-to-point --data-dir /vertica/data \
 --ssh-identity ~/name-of-pem.pem --license license.file
```

> **Note**
> - If you are using Vertica Community Edition, which limits you to three instances, you can specify -L CE with no license file.
> - When you issue install_vertica or update_vertica on a Vertica AMI script, --point-to-point is the default. This parameter configures Spread to use direct point-to-point communication between all Vertica nodes, which is a requirement for clusters on AWS.
> - If you are using IPv6 network addresses to identify the hosts in your cluster, use the --ipv6 flag in your install_vertica command. You must also use IP addresses instead of host names, as the AWS DNS server cannot resolve host names to IPv6 addresses.

2. After combining your instances, Vertica recommends deleting your *.pem key from your cluster to reduce security risks. The example below uses the shred command to delete the file:

```
$ shred name-of-pem.pem
```

3. After creating one or more clusters, create your database or connect to Management Console on AWS .

For complete information on the install_vertica script and its parameters, see Install Vertica with the installation script .

Important

Stopping or rebooting an instance or cluster without first shutting down the database down, may result in disk or database corruption. To safely shut down and restart your cluster, see Operating the database .

Check open ports manually using the netcat utility

Once your cluster is up and running, you can check ports manually through the command line using the netcat (nc) utility. What follows is an example using the utility to check ports.

Before performing the procedure, choose the private IP addresses of two nodes in your cluster.

The examples given below use nodes with the private IPs:

```
10.0.11.60 10.0.11.61
```

Install the nc utility on your nodes. Once installed, you can issue commands to check the ports on one node from another node.

To check a TCP port:

1. Put one node in listen mode and specify the port. The following sample shows how to put IP 10.0.11.60 into listen mode for port 480

```
[root@ip-10-0-11-60 ~]# nc -l 4804
```

2. From the other node, run nc specifying the IP address of the node you just put in listen mode, and the same port number.

```
[root@ip-10-0-11-61 ~]# nc 10.0.11.60 4804
```

3. Enter sample text from either node and it should show up on the other node. To cancel after you have checked a port, enter **Ctrl+C** .

> **Note**
> To check a UDP port, use the same nc commands with the –u option.

```
[root@ip-10-0-11-60 ~]# nc -u -l 4804
[root@ip-10-0-11-61 ~]# nc -u 10.0.11.60 4804
```

Management Console on AWS

Management Console (MC) is a database management tool that allows you to view and manage aspects of your cluster. Vertica provides an MC AMI, which you can use with AWS. The MC AMI allows you to create an instance, dedicated to running MC, that you can attach to a new or existing Vertica cluster on AWS. You can create and attach an MC instance to your Vertica on AWS cluster at any time.

After you launch your MC instance and configure your security group settings, you can log in to your database. To do so, use the elastic IP you specified during instance creation.

From this elastic IP, you can manage your Vertica database on AWS using standard MC procedures.

Considerations when using MC on AWS

- Because MC is already installed on the MC AMI, the MC installation process does not apply.
- To uninstall MC on AWS, follow the procedures provided in Uninstalling Management Console before terminating the MC Instance.

Related topics

- Using Management Console
- Managing Database Clusters
- Network ACL settings

Vertica on Microsoft Azure

Vertica supports automatic deployment on Azure through the Microsoft Azure Marketplace, or manual installation and deployment on Azure VMs.

You can deploy a Vertica database on the Microsoft Azure Cloud running in either Enterprise Mode or Eon Mode . In Eon Mode, Vertica stores its data communally using Azure block blob storage.

This section explains how to deploy a Vertica database to Microsoft Azure.

For more information about Azure, see the Azure documentation .

In this section

- Recommended Azure VM types and operating systems
- Eon Mode on Azure prerequisites
- Deploy Vertica from the Azure Marketplace
- Manually deploy Vertica on Microsoft Azure

Recommended Azure VM types and operating systems

Recommended Azure VM types

Vertica supports a range of Microsoft Azure virtual machine (VM) types, each optimized for different purposes. Choose the VM type that best matches your performance and price needs as a user.

> **Note**
> The GS VMs are not available in all regions, or from the Azure Marketplace.
>
> You can use them by following the manual deployment steps described in Manually deploy Vertica on Microsoft Azure .

For the best performance in most common scenarios, use one of the following VMs:

Virtual Machine Types	Virtual Machine Size

Memory optimized	DS13_v2 DS14_v2 DS15_v2 D8s_v3 D16s_v3 D32s_v3
High memory and I/O throughput	GS3 GS4 GS5 E8s_v3 E16s_v3 E32s_v3 L8s L16s L32s

Recommended Azure operating systems

For best performance, use one of the following operating systems when deploying Vertica on Azure:

- Red Hat 7.3 or later
- CentOS 7.3 or later. The Azure Marketplace solution as of this writing (June 2017) is based on CentOS 7.3.1611.

For more information, see Supported platforms .

Eon Mode on Azure prerequisites

Before you can create an Eon Mode database on Azure, you must have a database cluster and an Azure blob storage container to store your database's data.

You can create an Eon Mode database on a cluster that is hosted on Azure. In this configuration, your database stores its data communally in Azure Blob storage. See Eon Mode to learn more about this database mode.

> **Note**
> If you have an existing Enterprise Mode database, you can migrate it to an to Eon Mode database running on Azure. See Migrating an enterprise database to Eon Mode .

Cluster requirements

Before you can create an Eon Mode database on Azure, you must provision a cluster to host it. See Configuring your Vertica cluster for Eon Mode for suggestions on choosing VM configurations and the number of nodes your cluster should start with.

Storage requirements

An Eon Mode database on Azure stores its data communally in Azure blob storage. Vertica only supports block blob storage for communal data storage, not append or page blob storage.

You must create a storage path for Vertica to use exclusively. This path can be a blob container or a folder within a blob container. This path must not contain any files. If you attempt to create an Eon Mode database with a container or folder that contains files, admintools returns an error.

You pass Vertica a URI for the storage path using the azb:// schema. See Azure Blob Storage object store for the format of this URI.

You must also configure the storage container so Vertica is authorized to access it. Depending on authentication method you use, you may need to supply Vertica the with credentials to access the container. Vertica can use one of following methods to authenticate with the blob storage container:

- Using Azure managed identities. This authentication method is transparent—you do not need to add any authentication configuration information to Vertica. Vertica automatically uses the managed identity bound to the VMs it runs on to authenticate with the blob storage container. See the Azure AD-managed identities for Azure resources documentation page in the Azure documentation for more information.
  If you provide credentials for either of the other two supported authentication methods, Vertica uses them instead of authenticating using a managed identity bound to your VM.

  > **Note**
  > If your Azure VMs have more than one managed identity bound to them, you must tell Vertica which identity to use when authenticating with the blob storage container. Vertica gets the identity to use from a tag set on the VMs that it is running on.
  >
  > On your VMs, create a tag with its key named VerticaManagedIdentityClientId and its value to the name of a managed identity bound to your VMs. See the Use tags to organize your Azure resources and management hierarchy page in the Azure documentation for more information.

- Using an account name and access key credentials for a service account that has full access to the blob storage container. In this case, you provide Vertica with the credentials when you create the Eon Mode database. See Creating an Authentication File for details.
- Using a shared access signature (SAS) that grants Vertica access to the storage container. See Grant limited access to Azure Storage resources using shared access signatures (SAS) in the Azure documentation. See Creating an Authentication File for details.

For details on how Vertica accesses Azure blob storage, see Azure Blob Storage object store .

Azure Blob Storage encryption

Eon Mode databases on Azure support some of the encryption features built into Azure Storage. You can use its encryption at rest feature transparently—you do not need to configure Vertica to take advantage of it. You can use Microsoft-managed or customer-managed keys for storage encryption. Vertica does not support Azure Storage's client-side encryption and encryption using customer-provided keys. See the Azure Data Encryption at rest page in the Azure documentation for more information about the encryption at rest features in Azure Storage.

Deploy Vertica from the Azure Marketplace

Deploy Vertica in the Microsoft Azure Cloud using the Vertica Data Warehouse entry in the Azure Marketplace. Vertica provides the following deployment options:

- Eon Mode : Deploy a Management Console (MC) instance, and then provision and create an Eon Mode database from the MC. For cluster and storage requirements, see Eon Mode on Azure prerequisites .
- Enterprise Mode : Deploy a four-node Enterprise Mode database comprised of one MC instance and three database nodes. This requires an Azure subscription with a minimum of 12 cores for the Vertica Marketplace solution.
  The Enterprise Mode deployment uses the MC primarily as a monitoring tool. For example, you cannot provision and create a database with an Enterprise Mode MC. For information about creating and managing an Enterprise Mode database, see Create a database using administration tools .

Create a deployment

Eon Mode and Enterprise Mode require much of the same information for deployment. Any information that is not required for both deployment types is clearly marked.

1. Select the deployment type

1. Sign in to your Microsoft Azure account. From the **Home** screen, select **Create a resource** under **Azure services** .
2. Search for **Vertica Data Warehouse** and select it from the search results.
3. On the **Vertica Data Warehouse** page, select one of the following:
   - To deploy an MC instance that can manage an Eon Mode database, select Vertica **Data Warehouse, Eon BYOL** .
   - To deploy an Enterprise Mode database, select Vertica **Data Warehouse, Enterprise BYOL** .
4. On the next screen, select **Create** .

After you select your deployment type, the **Basics** tab on the **Create** Vertica **Data Warehouse** page displays.

2. Add project and instance details on the basics tab

Provide the following information in the **Project details** and **Instance details** sections:

1. **Subscription** : Azure bills this subscription for the cluster resources.
2. **Resource group** : The location to save all of the Azure resources. Create a new resource group or choose an existing one from the dropdown list.

3. **Region** : The location where the virtual machine running your MC instance is deployed.
4. VerticaManagement Console **User** : Eon Mode only. The administrator username for the MC.
5. **SSH public key for OS Access** : Provide the SSH public key associated with the Vertica **User** , for command line access to the virtual machine.
6. **Password for MC Access** : Enter a password to log in to Management Console. Note that Management Console requires that you change your password after the initial login.
7. **Confirm password** : Reenter the value you entered in **Password for MC Access** .
8. Select **Next: Virtual Machine Settings >** .

3. Select virtual machine settings

Provide the following information on the **Virtual Machine Settings** tab:

1. Management Console **VM size** : Select Change size to customize the VM settings or select the default. For a list of VM types recommended by use case, see Recommended Azure VM types and operating systems .
2. **Storage account of Eon DB** : Eon Mode only. The storage account associated with the database deployment.
3. **Number of** Vertica **Cluster nodes** : Enterprise Mode only. The number of nodes to deploy in the cluster, in addition to the MC instance.
   The Community Edition (CE) license is automatically applied to the cluster. This license is limited to 1 TB of RAW data 3 Vertica nodes. If you select more than 3 nodes with a CE license, the initial database is created on the first 3 nodes. For information about upgrading your license, see Managing licenses .
4. Vertica **Node VM size** : Enterprise Mode only. Select the VM type to deploy in your cluster. Use the default or select **Change size** to customize the VM settings. For a list of VM types recommended by use case, see Recommended Azure VM types and operating systems .
5. **Total RAW storage per node** : Enterprise Mode only. Select the amount of storage per node from the dropdown list. Each VM has a set of premium data disks that are configured and presented as a single storage location.
6. Select **Next: Network Settings >** .

4. Select network settings

Provide the following information on the **Network Settings** tab:

1. **Virtual Network** : The virtual network that hosts the Vertica cluster. Create a new virtual network or select an existing one from the dropdown list.
   If you select an existing virtual network, Vertica recommends that you already created a subnet to use for the deployment.
2. **First subnet** : The subnet for the associated **Virtual Network** . Create a new subnet or select an existing one from the dropdown list.
3. **Public IP Address Resource Name** : Each VM is configured with a publicly accessible IP address. This field allows you to specify the resource name for those IP addresses, and whether they are static or dynamic. The first public IP address resource is created exactly as entered, and associated with the VerticaManagement Console. Azure appends a number from 1 to 16 to the resource name for each additional Vertica cluster node created. This number associates each VM with a resource.
4. **Domain Name Label for** Management Console: Because each VM has a public IP address, each node requires a DNS name. Enter a prefix for the name. The first DNS name is created exactly as entered, and associated with the VerticaManagement Console. Azure appends a number from 1 to 16 to the DNS name for each Vertica cluster node created. That number associates each VM with a resource. Azure adds the remaining part of the fully qualified domain name based on the location where you created the cluster.
5. Select **Next: Review + create >** .

5. Verify on review + create

As the **Review + create** page loads, Azure validates your settings. After it passes validation, review your settings. When you are satisfied with your selections, select **Create** .

Access the MC after deployment

After your resources are successfully deployed, you are brought to the **Overview** page on **Home** > *resources-name* > **Deployments** . You must retrieve your Management Console IP address and username to log in.

1. From the **Overview** page, select **Outputs** in the left navigation.
2. Copy the **Vertica Management Console URL** and *Vertica Management Console user name* *.
3. Paste the **Vertica Management Console URL** in the browser address bar and press **Enter** .
4. Depending on your browser, you might receive a warning of a security risk. If you receive the warning, select the **Advanced** button and follow the browsers instructions to proceed to the Management Console.
5. On the VerticaManagement Console log in page, paste the **Vertica Management Console user name** , and enter the **Password for MC Access** that you entered on **Basics** > **Project details** when you were deploying your MC instance.

Delete a resource group

For details about the Azure Resource Manager and deleting a resource group, see the Azure documentation .

Manually deploy Vertica on Microsoft Azure

Manually creating a database cluster for your Vertica deployment lets you customize your VMs to meet your specific needs. You often want to manually configure your VMs when deploying a Vertica cluster to host an Eon Mode database.

To start creating your Vertica cluster in Azure using manual steps, you first need to create a VM. During the VM creation process, you create and configure the other resources required for your cluster, which are then available for any additional VMs that you create.

In this section

- Configure and launch a new instance
- Connect to a virtual machine
- Prepare the virtual machines
- Configure storage
- Form a cluster and install Vertica

Configure and launch a new instance

An Azure VM is similar to a traditional host. Just as with an on-premises cluster, you must prepare and configure the hardware settings for your cluster and network before you install Vertica.

The first steps are:

1. From the Azure marketplace, select an operating system that Vertica supports.
2. Select a VM type.See Recommended Azure VM types and operating systems .
3. Choose a deployment model. For best results, choose the resource manager deployment model.

Configure network security group

Vertica has specific network security group requirements, as described in Create a security group .

Create and name your own network security group, following these guidelines.

You must configure SSH as:

- Protocol: TCP
- Source port range: Any
- Destination port range: 22
- Source: Any
- Destination: Any

You can make additional modifications, based on your specific requirements.

Add disk containers

Create an Azure storage account, which later contains your cluster storage disk containers.

For optimal throughput, select Premium storage and align the storage to your chosen VM type.

For more information about what a storage account is, and how to create one, refer to About Azure storage accounts .

For an Enterprise Mode database deployment, provision enough space

Configure credentials

Create a password or assign an SSH key pair to use with Vertica.

For information about how to use key pairs in Azure, see How to create and use an SSH public and private key pair for Linux VMs in Azure .

Assign a public IP address

A public IP is an IP address that you can use to connect to your cluster externally. For best results, assign a single static public IP to a node in your cluster. You can then connect to other nodes in your cluster from your primary node using the internal IP addresses that Azure generated when you specified your virtual network settings.

By default, a public IP address is dynamic; it changes every time you shut down the server. You can choose a static IP address, but doing so can add cost to your deployment.

During a VM installation, you cannot set a DNS name. If you use dynamic public IPs, set the DNS name in the public IP resource for each VM after deployment.

For information about public IP addresses, refer to IP address types and allocation methods in Azure .

Create additional VMs

If needed, to create additional VMs, repeat the previous instructions in this document.

Connect to a virtual machine

Before you can connect to any of the VMs you created, you must first make your virtual network externally accessible. To do so, you must attach the public IP address you created during network configuration to one of your VMs.

Connect to your VM

To connect to your VM, complete the following tasks:

1. Connect to your VM using SSH with the public IP address you created in the configuration steps.
2. Authenticate using the credentials and authentication method you specified during the VM creation process.

Connect to other VMs

Connect to other virtual machines in your virtual network by first using SSH to connect to your publicly connected VM. Then, use SSH again from that VM to connect through the private IP addresses of your other VMs.

If you are using private key authentication, you may need to move your key file to the root directory of your publicly connected VM. Then, use PuTTY or WinSCP to connect to other VMs in your virtual network.

Prepare the virtual machines

After you create your VMs, you need to prepare them for cluster formation.

Add the Vertica license and private key

Prepare your nodes by adding your private key (if you are using one) to each node and to your Vertica license. These steps assume that the initial user you configured is the DBADMIN user.

1. As the dbadmin user, copy your private key file from where you saved it locally onto your primary node.
   Depending upon the procedure you use to copy the file, the permissions on the file may change. If permissions change, the install_vertica script fails with a message similar to the following:
   ```
   Failed Login Validation 10.0.2.158, cannot resolve or connect to host as root.
   ```
   If you receive a failure message, enter the following command to correct permissions on your private key file:
   ```
   $ chmod 600 /<name-of-key>.pem
   ```
2. Copy your Vertica license to your primary VM. Save it in your home directory or other known location.

Install software dependencies for Vertica on Azure

In addition to the Vertica standard Package dependencies , as the root user, you must install the following packages before you install Vertica on Azure:

- pstack
- mcelog
- sysstat
- dialog

Configure storage

Use a dedicated Azure storage account for node storage.

> **Caution**
> **Caution:** Do *not* store your information on the root volume, especially your data and catalog directories. Storing information on the root volume may result in data loss.

When configuring your storage, make sure to use a supported file system. For details, see File system .

Attach disk containers to virtual machines (VMs)

Using your previously created storage account, attach disk containers to your VMs that are appropriate to your needs.

For best performance, combine multiple storage volumes into RAID-0. For most RAID-0 implementations, attach 6 storage disk containers per VM.

Combine disk containers for storage

If you are using RAID, follow these steps to create a RAID-0 drive on your VMs. The following example shows how you can create a RAID-0 volume named md10 from 6 individual volumes named:

- sdc
- sdd
- sde
- sdf
- sdg
- sdh

1. Form a RAID-0 volume using the mdadm utility:
```
$ mdadm --create /dev/md10 --level 0 --raid-devices=6 \
  /dev/sdc /dev/sdd /dev/sde /dev/sdf /dev/sdg /dev/sdh
```
2. Format the file system to be one that Vertica supports:
```
$ mkfs.ext4 /dev/md10
```
3. Find the UUID on the newly-formed RAID volume using the blkid command. In the output, look for the device you assigned to the RAID volume:
```
$ blkid
. . .
/dev/md10 : UUID="e7510a6f-2922-4413-b5fa-9dcd725967fd" TYPE="ext4" PARTUUID="fb9b7449-08c3-4231-9ee5-086f7b0c9001"
. . .
```
4. The RAID device can be renamed after a reboot. To ensure the filesystem is mounted in a predictable location on your VM, create a directory to use as the mount point to mount the filesystem. For example, you can choose to create a mount point named /data that you will use to store your database's catalog and data (or depot, if you are running Vertica in Eon Mode).
```
$ mkdir /data
```
5. Using a text editor, add an entry to the /etc/fstab file for the UUID of the filesystem and your mount point so it is mounted when the system boots:
```
UUID=RAID_UUID mountpoint      ext4    defaults,nofail,nobarrier    0   2
```
   For example, if you have the UUID shown in the previous example and the mount point /data , add the following line to the /etc/fstab file:
```
UUID=e7510a6f-2922-4413-b5fa-9dcd725967fd  /data      ext4    defaults,nofail,nobarrier    0   2
```
6. Mount the RAID filesystem you added to the fstab file. For example, to mount a mount point named /data use the command:
```
$ mount /data
```
7. Create folders for your Vertica data and catalog under your mount point.
```
$ mkdir /data/vertica
$ mkdir /data/vertica/data
```
   If you are planning to run Vertica in Eon Mode, create a directory for the depot instead of data:
```
$ mkdir /data/vertica/depot
```

Create a swap file

In addition to storage volumes to store your data, Vertica requires a swap volume or swap file to operate.

Create a swap file or swap volume of at least 2 GB. The following steps show how to create a swap file within Vertica on Azure:

1. Install devnull and swapfile:
```
$ install -o root -g root -m 0600 /dev/null /swapfile
```
2. Create the swap file:
```
$ dd if=/dev/zero of=/swapfile bs=1024 count=2048k
```
3. Prepare the swap file using mkswap :
```
$ mkswap /swapfile
```
4. Use swapon to instruct Linux to swap on the swap file:
```
$ swapon /swapfile
```
5. Persist the swapfile in FSTAB:
```
$ echo "/swapfile       swap    swap    auto      0       0" >> /etc/fstab
```

Repeat the volume attachment, combination, and swap file creation procedures for each VM in your cluster.

For more information

- About Azure storage accounts
- Prepare disk storage locations
- Storage Requirements

Form a cluster and install Vertica

Use the install_vertica script to combine two or more individual VMs to form a cluster and install the Vertica database.

Download Vertica

To download the Vertica server appropriate for your operating system and license type, go to www.vertica.com/download/vertica .

Run the rpm to extract the files.

After you complete the download and extraction, the next section describes how to use the install_vertica script to form a cluster and install the Vertica database software.

Before you start

Before you run the install_vertica script:

- Check the **Virtual Network** page for a list of current VMs and their associated private IP addresses.
- Identify your storage location. The installer assumes that you have mounted your storage to /vertica/data . To specify another location, use the --data-dir argument.
- Identify your storage location. To create your database's data directory on mounted RAID drive, when you run the install_vertica script, provide /vertica/data as the value of the --data-dir option .

> **Caution**
> **Caution:** Do *not* store your data on the root drive.

Combine virtual machines (VMs)

The following example shows how to combine VMs using the install_vertica script.

1. While connected to your primary node, construct the following command to combine your nodes into a cluster.
   ```
   $ sudo /opt/vertica/sbin/install_vertica --hosts 10.2.0.164,10.2.0.165,10.2.0.166 --dba-user-password-disabled --point-to-point --data-dir /vertica/data --ssh-identity ~/<name-of-private-key>.pem --license <license.file>
   ```
2. Substitute the IP addresses for your VMs and include your root key file name, if applicable.
3. Include the --point-to-point parameter to configure spread to use direct point-to-point communication between all Vertica nodes, as required for clusters on Azure when installing or updating Vertica.
4. If you are using Vertica Community Edition, which limits you to three nodes, specify -L CE with no license file.
5. After you combine your nodes, to reduce security risks, keep your key file in a secure place—separate from your cluster—and delete your on-cluster key with the **shred** command:
   ```
   $ shred examplekey.pem
   ```
   Important
   You need your key file to perform future Vertica updates.
6. Reboot your cluster to complete the cluster formation and Vertica installation.

For complete information on the install_vertica script and its parameters, see Install Vertica with the installation script .

After your cluster is up and running

Now that your cluster is configured and running, take these steps:

1. Log into one of the database nodes using the database administrator account (named dbadmin by default).
2. Create and start a database:
   - To create an Enterprise Mode database, see Create a database using administration tools .
   - To create an Eon Mode database, see Manually create an Eon Mode database on Azure .
3. Configure your database. See Configuring the database .

Vertica on Google Cloud Platform

Vertica supports automatic deployment on Google Cloud Platform (GCP) through the Google Cloud Launcher , or manual installation and deployment on GCP machines.

You can deploy a Vertica database on GCP running in either Enterprise Mode or Eon Mode . In Eon Mode, Vertica stores its data communally using Google Cloud Storage (GCS).

This section explains how to deploy a Vertica database to GCP.

For more information about GCP, see the Google Cloud documentation .

In this section

- Supported GCP machine types
- Deploy Vertica from the Google cloud marketplace
- Manually deploy an Enterprise Mode database on GCP

Supported GCP machine types

Vertica Analytic Database supports a range of machine types, each optimized for different workloads. When you deploy your Vertica Analytic Database cluster to the Google Cloud Platform (GCP), different machine types are available depending on how you provision your database.

> **Note**
> Some machine types are not available across all regions.

The sections below list the GCP machine types that Vertica supports for Vertica cluster hosts, and for use in Management Console. For details on the configuration of the machine type options, see the Google Cloud documentation's Machine types page.

Machine types available for MC hosts

Vertica supports all N1, N2, E2, M1, M2, and C2 machine types to deploy an instance for running the Vertica Management Console.

> **Tip**
> In most cases, 8 vCPUs are sufficient when selecting a machine type for running the Management Console.

Machine types available for Vertica database cluster hosts

Vertica supports all N1, N2, E2, M1, M2, and C2 machine types to deploy cluster hosts.

Machine types for Vertica database cluster hosts provisioned from MC

The table below lists the GCP machine types that Vertica supports when you provision your cluster from Management Console.

Machine Type	Machine Name
N1 standard	n1-standard-16 n1-standard-32 n1-standard-64
N1 high-memory	n1-highmem-16 n1-highmem-32 n1-highmem-64

N2 standard	n2-standard-16 n2-standard-32 n2-standard-48 n2-standard-64
N2 high-memory	n2-highmem-16 n2-highmem-32 n2-highmem-48 n2-highmem-64

Deploy Vertica from the Google cloud marketplace

The Vertica entries in the Google Cloud Launcher Marketplace let you quickly deploy a Vertica cluster in the Google Cloud Platform (GCP). Currently, three entries let you select the database mode and the license you want to use:

- The Enterprise Mode launcher deploys a Vertica database with 3 or more nodes, plus an additional VM running the Management Console (MC). See Deploy an Enterprise Mode database in GCP from the marketplace for more information.
- The Eon Mode BYOL (bring your own license) launcher deploys a single instance running the MC. You use this MC instance to deploy a Vertica database running on Eon Mode. This database has a community license applied to it initially. You can later upgrade it to a license you have obtained from Vertica. See Deploy an MC instance in GCP for Eon Mode for more information.
- The Eon Mode BTH (by the hour) launcher also deploys a single instance running the MC that you use to deploy a database. This database has a by-the-hour license applied to it. Instead of paying for a license up front, you pay an hourly fee that covers both Vertica and running your instances. The BTH license is automatically applied to all clusters you create using a BTH MC instance. See Deploy an MC instance in GCP for Eon Mode for more information. If you choose, you can upgrade this hourly license to a longer-term license you purchase from Vertica. To move a BTH cluster to a BYOL license, follow the instructions in Moving a cloud installation from by the hour (BTH) to bring your own license (BYOL) .

> **Note**
> Vertica clusters that use IPv6 to identify hosts have not been tested on GCP. Vertica recommends you use IPv4 addresses to identify the hosts in your cluster on GCP.

In this section

- Eon Mode on GCP prerequisites
- Deploy an Enterprise Mode database in GCP from the marketplace
- Deploy an MC instance in GCP for Eon Mode

Eon Mode on GCP prerequisites

Before deploying an Eon Mode database on GCP, you must take several steps:

- Review the default service account's permissions for your GCP project.
- Create an HMAC key to use when creating your cluster.
- Create a communal storage location.

Service account permissions

Service accounts allow automated processes to authenticate with GCP. The Eon Mode database deployment process uses the project's service account for your GCP project to deploy instances. When you create a new project, GCP automatically creates a default service account (identified by *project_number* -compute@developer.gserviceaccount.com) for the project and grants it the IAM role Editor. See the Google Cloud documentation's Understanding roles for details about this and other IAM roles.

The Editor role lets the service account create resources from the Marketplace. When you create an instance of the Management Console (MC), the MC uses the account to deploy further resources, such as provisioning instances for an database.

For details, see the Google Cloud documentation's Understanding service accounts page.

Permissions and roles

To deploy Vertica on GCP, your user account must have the:

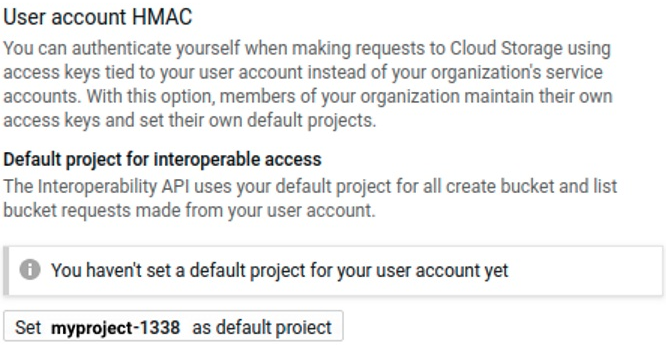
- **Editor** role.
- runtimeconfig.waiters.getIamPolicy permission.

Creating an HMAC key

Vertica uses a hash-based message authentication code (HMAC) key to authenticate requests to access the communal storage location. This key has two parts: an access ID and a secret. When you create an Eon Mode database in GCP, you provide both parts of an HMAC key for the nodes to use to access communal storage.

To create an HMAC key:

1. Log in to your Google Cloud account.
2. If the name of the project you will use to create your database does not appear in the top banner, click the dropdown and select the correct project.
3. In the navigation menu in the upper-left corner, under the Storage heading, click **Storage** and select **Settings** .
4. In the Settings page, click **Interoperability** .
5. Scroll to the bottom of the page and find the User account HMAC heading.
6. Unless you have already set a default project, you will see the message stating you haven't set a default project for your user account yet. Click the **Set *project-id* as default project** button to choose the current project as your default for interoperability.



> **Note**
> The project ID appears in the button label, not the project name.

7. Under Access keys for your user account, click **Create a key** .
8. Your new access key and secret appear in the HMAC key list. You will need them when you create your Eon Mode database. You can copy them to a handy location (such as a text editor) or leave a browser tab open to this page while you use another tab or window to create your database. These keys remain available on this page, so you do not need to worry about saving them elsewhere.

> **Caution**
> It is vital that you protect the security of your HMAC key. It can grant others access to your Eon Mode database's communal storage location. This means they could access all of the data in your database. Do not write the HMAC key anyplace where it may be exposed, such as email, shared folders, or similar insecure locations.

Creating a communal storage location

Your Eon Mode database needs a storage location for its communal storage. Eon Mode databases running on GCP use Google Cloud Storage (GCS) for their communal storage location. When you create your new Eon Mode database, you will supply the MC's wizard with a GCS URL for the storage location.

This location needs to meet the following criteria:

- The URL must include at least a bucket name. You can use one or more levels of folders, as well. For example, the following GCS URLs are valid:
  - gs://verticabucket/mydatabase
  - gs://verticabucket/databases/mydatabase
  - gs://verticabucket

  Multiple databases can share the same bucket, as long as each has its own folder.

- If provided, the lowest-level folder in the URL must not already exist. For example, in the GCS URL gs://verticabucket/databases/mydatabase , the bucket named verticabucket and the directory named databases must exist. The subdirectory named mydatabase must not exist. The Vertica install process expects to create the final folder itself. If the folder already exists, the installation process fails.

  > **Note**
  > If you have a communal storage location that already contains data from a previous Eon Mode database that you want to access, use the revive process, rather than installing a new database. See Stopping, starting, terminating, and reviving Eon Mode database clusters for details.

- The permissions on the bucket must be set to allow the service account read, write, and delete privileges on the bucket. The best role to assign to the user to gain these permissions is Storage Object Admin .
- To prevent performance issues, the bucket must be in the same region as all of the nodes running the Eon Mode database.
- If you create the database through the admintools UI, you must set gcsauth as a bootstrap parameter in admintools.conf . For more information on this and other GCP parameters, see Google Cloud Storage parameters .

```
[BootstrapParameters]
gcsauth = ID:secret
```

Deploy an Enterprise Mode database in GCP from the marketplace

The Vertica Cloud Launcher solution creates a Vertica Enterprise Mode database. The solution includes the Vertica Management Console (MC) as the primary UI for you to get started.

The launcher automatically creates a database named vdb using the Community Edition (CE) license. The CE license is limited to a maximum of 3 nodes. You can tell the launcher to add more than 3 nodes to your deployment. In this case, it uses the first three nodes in the cluster to create the database. The remaining nodes are not part of the database, but are added to your cluster. To add these nodes to your database, you must replace the Community Edition license with a license key you receive from the Software Entitlement support site. See Managing licenses for more information.

After the launcher creates the initial database, it configures the MC to attach to that database automatically.

Configure the Vertica cloud launcher solution

To get started with a deployment of Vertica from the Google Cloud Launcher , search for the Vertica Data Warehouse, Enterprise Mode entry.

Follow these steps:

1. Verify that your user account has the **Editor** role and the runtimeconfig.waiters.getIamPolicy permission.
2. From the listing page, click **LAUNCH** .
3. On the New Vertica Analytics Platform deployment page, enter the following information:
   - **Deployment name** : Each deployment must have a unique name. That name is used as the prefix for the names of all VMs created during the deployment. The deployment name can only contain lowercase characters, numbers, and dashes. The name must start with a lowercase letter and cannot end with a dash.
   - **Zone** : GCP breaks its cloud data centers into regions and zones. *Regions* are a collection of zones in the same geographical location. *Zones* are collections of compute resources, which vary from zone to zone.
     For best results, pick the zone in your designated region that supports the latest Intel CPUs. For a complete listing of regions and zones, including supported processors, see Regions and Zones .
   - **Service Account** : Service accounts allow automated processes to authenticate with GCP. Select the default service account, identified by *project_number* -compute@developer.gserviceaccount.com .
   - Under **Vertica Management Console** , choose the configuration for the virtual machine that will run the Management Console. The Vertica Analytics Platform in Cloud Launcher always deploys the Vertica Management Console (MC) as part of the solution.
     The default machine type for MC is sufficient for most deployments. You can choose another machine type that better suits any additional purposes, such serving as a target node for backups, data transformation, or additional management tools.
   - **Node count for Vertica Cluster** : The total number of VMs you want to deploy in the Vertica Cluster. The default is 3.

     > **Note**
     > As mentioned above, the Cloud Launcher automatically deploys the Vertica Community Edition license, which limits the database to 3 nodes and up to 1 TB in raw data. Any additional nodes will be part of your database cluster, but will not be part of your database.
     >
     > If you intend to use the Community Edition license for your database, leave the setting at 3. Otherwise, you would add nodes that will sit idle and cost you money without being part of your database.

- **Machine type for Vertica Cluster nodes** : The Cloud Launcher builds each node in the cluster using the same machine type. Modify the machine type for your nodes based on the workloads you expect your database to handle. See Supported GCP machine types for more information.
- **Data disk type** : GCP offers two types of persistent disk storage: Standard and SSD. The costs associated with Standard are less, but the performance of SSD storage is much better. Vertica recommends you use SSD storage. For more information on Standard and SSD persistent disks, see Storage Options .
- **Disk size in GB** : Disk performance is directly tied to the disk size in GCP. The default value of 2000 GBs (2 TB) is the minimum disk size for SSD persistent disks that allows maximum throughput.
  If you select a smaller disk size, the throughput performance decreases. If you select a large disk size, the performance remains the same as the 2 TB option.
- **Network** : VMs in GCP must exist on a virtual private cloud (VPC). When you created your GCP account, a default VPC was created. Create additional VPCs to isolate solutions or projects from one another. The Vertica Analytics Plaform creates all the nodes in the same VPC.
- **Subnetwork** : Just as a GCP account may have multiple VPCs, each VPC may also have multiple subnets. Use additional subnets to group or isolate solutions within the same VPC.
- **Firewall** : If you want your MC to be accessible via the internet, check the Allow access to the Management Console from the Internet box. Vertica recommends you protect your MC using a firewall that restricts access to just the IP addresses of users that need to access it. You can enter one or more comma-separated CIDR address ranges.

After you have entered all the required information, click **Deploy** to begin the deployment process.

Monitor the deployment

After the deployment begins, Google Cloud Launcher automatically opens the Deployment Manager page that displays the status of the deployment. Items that are still being processed have a spinning circle to the left of them and the text is a light gray color. Items that have been created are dark gray in color, with an icon designating that resource type on the left.

After the deployment completes, a green check mark appears next to the deployment name in the upper left-hand section of the screen.

Accessing the cluster after deployment

After the deployment completes, the right-hand section of the screen displays the following information:

- dbadmin password: A randomly generated password for the dbadmin account on the nodes. For security reasons, change the dbadmin password when you first log in to one of the Vertica cluster nodes.
- mcadmin password: A randomly generated password for the mcadmin account for accessing the Management Console. For security reasons, change the mcadmin password after you first log in to the MC.
- Vertica Node 1 IP address: The external IP address for the first node in the Vertica cluster is exposed here so that you can connect to the VM using a standard SSH client.To access the MC, press the **Access Vertica MC** button in the **Get Started** section of the dialog box. Copy the mcadmin password and paste it when asked.

For more information on using the MC, see Management Console .

Access the cluster nodes

There are two ways to access the cluster nodes directly:

- Use GCP's integrated SSH shell by selecting the SSH button in the **Get Started** section. This shell opens a pop-up in your browser that runs GCP's web-based SSH client. You are automatically logged on as the user you authenticated as in the GCP environment.
  After you have access to the first Vertica cluster node, execute the su dbadmin command, and authenticate using the dbadmin password.
- In addition, use other standard SSH clients to connect directly to the first Vertica cluster node. Use the Vertica Node 1 IP address listed on the screen as the dbadmin user, and authenticate with the dbadmin password.
  Follow the on-screen directions to log in using the mcadmin account and accept the EULA. After you've been authenticated, access the initial database by clicking the vdb icon (looks like a green cylinder) in the **Recent Databases** section.

Using a custom service account

In general, you should use the default service account created by the GCP deployment (*project_number* -compute@developer.gserviceaccount.com), but if you want to use a custom service account:

- The custom service account must have the **Editor** role.
- Individual user accounts must have the **Service Account User** role on the custom service account.

Deploy an MC instance in GCP for Eon Mode

To deploy an Eon Mode database to GCP using Google Cloud Platform Launcher, you must deploy a Management Console (MC) instance. You then use the MC instance to provision and deploy an Eon Mode database.

Once you have taken the steps listed in Eon Mode on GCP prerequisites , you are ready to deploy an Eon Mode database in GCP. To deploy an MC instance that is able to deploy Eon Mode databases to GCP:

1. Log into your GCP account, if you are not currently logged in.
2. Verify that your user account has the **Editor** role and the runtimeconfig.waiters.getIamPolicy permission.
3. Verify that the name of the GCP project you want to use for the deployment appears in the top banner. If it does not, click the down arrow next to the project name and select the correct project.
4. Click the navigation menu icon in the top left of the page and select **Marketplace** .
5. In the **Search for solutions** box, type Vertica Eon Mode and press enter.
6. Click the search result for Vertica **Data Warehouse,** Eon Mode. There are two license options: by the hour (BTH) and bring your own license (BYOL). See Deploy Vertica from the Google cloud marketplace for more information on this license choice.
7. Click **Launch** on the license option you prefer.
8. On the following page, fill in the fields to configure your MC instance:
   - **Deployment name** identifies your MC deployment in the GCP Deployments page.
   - **Zone** is the location where the virtual machine running your MC instance will be deployed. Make this the same location where your communal storage bucket is located.
   - **Service Account** : Service accounts allow automated processes to authenticate with GCP. Select the default service account, identified by *project_number* -compute@developer.gserviceaccount.com .
   - **Machine Type** is the virtual hardware configuration of the instance that will run the MC. The default values here are "middle of the road" settings which are sufficient for most use cases. If you are doing a small proof-of-concept deployment, you can choose a less powerful instance to save some money. If you are planning on deploying multiple large databases, consider increasing the count of virtual CPUs and RAM.
     For details about Vertica's default volume configurations, see Eon Mode volume configuration defaults for GCP .
   - **User Name for Access to MC** is the administrator username for the MC. You can customize this if you want.
   - **Network** and **Subnetwork** are the virtual private cloud (VPC) network and subnet within that network you want your MC instance and your Vertica nodes to use. This setting does not affect your MC's external network address. If you want to isolate your Vertica cluster from other GCP instances in your project, create a custom VPC network and optionally a subnet in your GCP project and select them in these fields. See the Google Cloud documentation's VPC network overview page for more information.
   - **Firewall** enables access to the MC from the internet by opening port 5450 in the firewall. You can choose to not open this port by clearing the **I accept opening a port in the firewall (5450) for Vertica** box. However, if you do not open the port in the firewall, your MC instance will only be accessible from within the VPC network. Not opening the port will make accessing your MC instance much harder.
   - **Source IP ranges for MC traffic** : If you choose to open the MC for external access, add one or more or more CIDR address ranges to this box for network addresses that you want to be able to access to the MC.

     > **Caution**
     > Make the address ranges as limited as possible to reduce the chances of unauthorized access to your MC instance.

9. Click the **Deploy** button to start the deployment of your MC instance.

The deployment process will take several minutes.

Using a custom service account

In general, you should use the default service account created by the GCP deployment (*project_number* -compute@developer.gserviceaccount.com), but if you want to use a custom service account:

- The custom service account must have the **Editor** role.
- Individual user accounts must have the **Service Account User** role on the custom service account.

Connect and log into the MC instance

After the deployment process is finished, the Deployment Manager page for your MC instance contains links to connect to the MC via your browser or ssh.

To connect to the MC instance:

1. The MC administrator user has a randomly-generated password that you need to log into the MC. Copy the password in the **MC Admin Password** field to the clipboard.
2. Click **Access Management Console** .
3. A new browser tab or window opens, showing you a page titled Redirection Notice. Click the link for the MC URL to continue to the MC login page.
4. Your browser will likely show you a security warning. The MC instance uses a self-signed security certificate. Most browsers treat these certificates as a security hazard because they cannot verify their origin. You can safely ignore this warning and continue. In most browsers, click the **Advanced** button on the warning page, and select the option to proceed. In Chrome, this is a link titled **Proceed to** *xxx.xxx.xxx.xxx* **(unsafe)** . In

Firefox, it is a button labeled **Accept the Risk and Continue** .
5. At the login screen, enter the MC administrator user name into the **Username** box. This user name is mcadmin, unless you changed the user name in the MC deployment form.
6. Paste the automatically-generated password you copied from the MC Admin Password field earlier into the **Password** box.
7. Click **Log In** .

Once you have logged into the MC, change the MC administrator account's password.

> **Caution**
> The automatically-generated password appears on the MC instance's deployment page and can be revealed in several locations in the deployment logs. Failure to change this password can lead to unauthorized access to your MC instance.

To change the password:

1. On the home page of the MC, under the MC Tools section, click **MC Settings.**
2. In the left-hand menu, click **User Management** .
3. Select the entry for the MC administrator account and click **Edit** .
4. Click either the **Generate new** or **Edit password** button to change the password. If you click the **Generate new** button, be sure to save the automatically-generated password in a safe location. If you click **Edit password** , you are prompted to enter a new password twice.
5. Click **Save** to update the password.

Now that you have created your MC instance, you are ready to deploy a Vertica Eon Mode cluster. See Provision an Eon Mode cluster and database on GCP in MC .

Manually deploy an Enterprise Mode database on GCP

Before you create your Vertica cluster in Google Cloud Platform (GCP) using manual steps, you must create a virtual machine (VM) instance from the Compute Engine section of GCP.

Configure and launch a new instance

All VM instances that you create should be launched in the same virtual public cloud (VPC).

To configure and launch a new VM instance, follow these instructions:

1. From within the Compute Engine section of GCP, from the menu on the left-hand site of the screen, select **VM Instances** .
   GCP displays all the VM instances that you have created so far.
2. Click **CREATE INSTANCE** .
3. Enter a name for the new instance.
4. Select the zone where you plan to deploy the instance.
   GCP breaks its cloud data centers down by regions and zones. *Regions* are a collection of zones that are all in the same geographical location. Zones are collections of compute resources, which vary from zone to zone. Always pick the zone in your designated region that supports the latest Intel CPUs.
   For a complete listing of regions and zones, including supported processors, see Regions and Zones .
5. Select a machine type.
   GCE offers many different types of VM instances. For best results, only deploy Vertica on VM instances with 8 vCPus or more and at least 30 GB of RAM.
6. Select the boot disk (image).
   You create VM instances from a public or custom image. If you are starting with Vertica in GCP for the first time, select either the CentOS 7 or RHEL 7 public image. Those images have been tested thoroughly with Vertica.
   For more information about deploying a VM instance, see Creating and Starting an Instance .

After you have configured the VM instance to be used as a Vertica cluster node, GCP allows you to convert that instance into a custom image. Doing so allows you to deploy multiple versions of that VM instance; each VM instance is identical except for the node name and IP address.

For more information about creating a custom image, see Creating, Deleting, and Deprecating Custom Images .

Connect to a virtual machine

Before you can connect to any of the VMs you created, you must first identify the external IP address. The VM instance section of GCP contains a list of all currently deployed VMs and their associated external IP addresses.

Connect to your VM

To connect to your VM, complete the following tasks:

1. Connect to your VM using SSH with the external IP address you created in the configuration steps.
2. Authenticate using the credentials and SSH key that you provided to your GCP account upon creation.

Connect to other VMs

To connect to other virtual machines in your virtual network:

1. Use SSH to connect to your publicly connected VM.
2. Use SSH again from that VM to connect through the private IP addresses of your other VMs.

Because GCP forces the use of private key authentication, you may need to move your key file to the root directory of your publicly connected VM. Then, use SSH to connect to other VMs in your virtual network.

Prepare the virtual machines

After you create your VMs, you need to prepare them for cluster formation.

Add the Vertica license and private key

Prepare your nodes by adding your private key (if you are using one) to each node and to your Vertica license. The following steps assume that the initial user you configured is the DBADMIN user:

1. As the DBADMIN user, copy your private key file from where you saved it locally onto your primary node.
   Depending upon the procedure you use to copy the file, the permissions on the file may change. If permissions change, the install_vertica script fails with a message similar to the following:
   ```
   Failed Login Validation 10.0.2.158, cannot resolve or connect to host as root.
   ```
   If you see the previous failure message, enter the following command to correct permissions on your private key file:
   ```
   $ chmod 600 /<name-of-key>.pem
   ```
2. Copy your Vertica license to your primary VM. Save it in your home directory or other known location.

Install software dependencies for Vertica on GCP

In addition to the Vertica standard Package dependencies , as the root user, you must install the following packages before you install Vertica:

- pstack
- mcelog
- sysstat
- dialog

Configure storage

For best disk performance in GCP, Vertica recommends customers use SSD persistent storage, configured to at least 2TB (2000 GB) in size. Disk performance is directly tied to the disk size in GCP. 2000 GBs (2TB) is the minimum disk size for SSD persistent disks that allows maximum throughput.

> **Caution**
> Do not store your information on the root volume, especially in your data and catalog directories. Storing information on the root volume may result in data loss.

When configuring your storage, make sure to use a supported file system. See for details.

Create a swap file

In addition to storage volumes to store your data, Vertica requires a swap volume or swap file for the setup script to complete.

Create a swap file or swap volume of at least 2 GB. The following steps show how to create a swap file within Vertica on GCP:

1. Install the devnull and swapfile files:
   ```
   $ install -o root -g root -m 0600 /dev/null /swapfile
   ```
2. Create the swap file:
   ```
   $ dd if=/dev/zero of=/swapfile bs=1024 count=2048k
   ```
3. Prepare the swap file using mkswap :
   ```
   $ mkswap /swapfile
   ```
4. Use swapon to instruct Linux to swap on the swap file:

```
$ swapon /swapfile
```

5. Persist the swapfile in FSTAB:
```
$ echo "/swapfile swap swap auto 0 0" >> /etc/fstab
```

6. Repeat the volume attachment, combination, and swap file creation procedures for each VM in your cluster.

Download Vertica

To download the Vertica server appropriate for your operating system and license type, follow the steps in described in Download and install the Vertica server package .

After you complete the download and extraction, use the install_vertica script to form a cluster and install the Vertica database software, as described in the next section.

Form a cluster and install Vertica

Use the install_vertica script to combine two or more individual VMs to form a cluster and install your Vertica database.

Before you run the install_vertica script, follow these steps:

1. Check the **VM Instances** page of the Compute Engine section on GCP to locate a list of current VMs and their associated internal IP addresses.
2. Identify your storage location on your VMs. The installer assumes that you have mounted your storage to /home/dbadmin . To specify another location, use the --data-dir argument.

> **Caution**
> Do not store your data on the root drive.

The following steps show how to combine virtual machines (VMs) into a cluster using the install_vertica script:

1. While connected to your primary node, construct the following command to combine your nodes into a cluster.
```
$ sudo /opt/vertica/sbin/install_vertica --hosts 10.2.0.164,10.2.0.165,10.2.0.166 --dba-user-password-disabled --point-to-point --data-dir /vertica/data --ssh-identity ~/.pem --license
```

2. Substitute the IP addresses for your VMs, and include your root key file name, if applicable.
3. Include the --point-to-point parameter to configure spread to use direct point-to-point communication among all Vertica nodes, as required for clusters on GCP when installing or updating Vertica.
4. If you are using Vertica Community Edition, which limits you to three nodes, specify -L CE with no license file.
5. After you combine your nodes, to reduce security risks, keep your key file in a secure place—separate from your cluster—and delete your on-cluster key with the shred command:
```
$ shred examplekey.pem
```

Important
You need your key file to perform future Vertica updates.

For complete information about the install_vertica script and its parameters, see Install Vertica with the installation script .

After your cluster is up and running

Now that your cluster is configured and running, and Vertica is running, take these steps:

1. Create a database. See Creating a database for details.
2. When you installed Vertica, a database administrator user was created with the DBADMIN role (usually named dbadmin). Use this account to create and start a database.
3. See Configuring the database for important database configuration steps.

Set up Vertica on-premises

This section discusses the procedure for installing Vertica manually in an on-premises environment.

In this section

- Installation overview and checklist
- Before you install Vertica
- Install Vertica using the command line
- After you install Vertica

Installation overview and checklist

Carefully review and complete the installation tasks in all sections of this topic.

Important notes

- Vertica supports only one running database per cluster.
- Vertica supports installation on one, two, or multiple nodes. The steps for Installing Vertica are the same, no matter how many nodes are in the cluster.
- Prerequisites listed in Before You Install Vertica are required for all Vertica configurations.
- Only one instance of Vertica can be running on a host at any time.
- To run the install_vertica script, as well as adding, updating, or deleting nodes, you must be logged in as root, or sudo as a user with all privileges. You must run the script for all installations, including upgrades and single-node installations.

Installation scenarios

The three main scenarios for installing Vertica on hosts are:

- A single node install, where Vertica is installed on a single host as a *localhost* process. This form of install cannot be expanded to more hosts later on and is typically used for development or evaluation purposes.
- Installing to a cluster of physical host hardware. This is the most common scenario when deploying Vertica in a testing or production environment.
- Installing to a local cluster of virtual host hardware. This is similar to installing on physical hosts, but with network configuration differences.

Before you install

Before You Install Vertica describes how to construct a hardware platform and prepare Linux for Vertica installation.

These preliminary steps are broken into two categories:

- Configuring Hardware and Installing Linux
- Configuring the Network

Install or upgrade Vertica

Once you have completed the steps in the Before You Install Vertica section, you are ready to run the install script.

Installing Vertica describes how to:

- Back up any existing databases.
- Download and install the Vertica RPM package.
- Install a cluster using the install_vertica script.
- [Optional] Create a properties file that lets you install Vertica silently.

Post-installation tasks

After You Install Vertica describes subsequent steps to take after you've run the installation script. Some of the steps can be skipped based on your needs:

- Install the license key.
- Verify that kernel and user parameters are correctly set.
- Install the vsql client application on non-cluster hosts.
- Resolve any SLES 11.3 issues during spread configuration.
- Use the Vertica documentation online, or download and install Vertica documentation. Find the online documentation and documentation packages to download at https://docs.vertica.com/latest .
- Install client drivers.
- Extend your installation with Vertica packages.
- Install or upgrade the Management Console.

Before you install Vertica

Complete all of the tasks in this section before you install Vertica. When you have completed this section, proceed to Install Vertica using the command line .

In this section

- Platform and hardware requirements and recommendations
- Communal storage for on-premises Eon Mode databases
- Configure the network
- Operating system configuration overview
- Automatically configured operating system settings

- Manually configured operating system settings
- System user configuration

Platform and hardware requirements and recommendations

Hardware recommendations

The Vertica Analytics Platform is based on a massively parallel processing (MPP), shared-nothing architecture, in which the query processing workload is divided among all nodes of the Vertica database. OpenText highly recommends using a homogeneous hardware configuration for your Vertica cluster; that is, each node of the cluster should be similar in CPU, clock speed, number of cores, memory, and operating system version.

Note that OpenText has not tested Vertica on clusters made up of nodes with disparate hardware specifications. While it is expected that a Vertica database would functionally work in a mixed hardware configuration, performance will be limited to that of the slowest node in the cluster.

Vertica performs best on processors with higher clock frequency. When possible, choose a faster processor with fewer cores as opposed to a slower processor with more cores.

Tests performed both internally and by customers have shown performance differences between processor architectures even when accounting for differences in core count and clock frequency. When possible, compare platforms by installing Vertica and running experiments using your data and workloads. Consider testing on cloud platforms that offer VMs running on different processor architectures, even if you intend to deploy your Vertica database on premises.

Detailed hardware recommendations are available in Recommendations for Sizing Vertica Nodes and Clusters .

Platform requirements and recommnedations

You must verify that your servers meet the platform requirements described in Supported Platforms . The Supported Platforms topics detail supported versions for the following:

- OS for Server and Management Console (MC)
- Supported Browsers for MC
- Supported File Systems

Important

Deploy Vertica as the only active process on each host—other than Linux processes or software explicitly approved by Vertica. Vertica cannot be co-located with other software. Remove or disable all non-essential applications from cluster hosts.

Install the latest vendor-specific system software

Install the latest vendor drivers for your hardware.

Data storage recommendations

- All internal drives connect to a single RAID controller.
- The RAID array should form one hardware RAID device as a contiguous /data volume.

Install Perl

Before you perform the cluster installation, install Perl 5 on all the target hosts. Perl is available for download from www.perl.org .

Validation utilities

Vertica provides several validation utilities that validate the performance on prospective hosts. The utilities are installed when you install the Vertica RPM, but you can use them before you run the install_vertica script. See Validation scripts for more details on running the utilities and verifying that your hosts meet the recommended requirements.

Verify sudo

Vertica uses the sudo command during installation and some administrative tasks. Ensure that sudo is available on all hosts with the following command:

```
# which sudo
/usr/bin/sudo
```

If sudo is not installed, on all hosts, follow the instructions in How to Enable sudo on Red Hat Enterprise Linux .

When you use sudo to install Vertica, the user that performs the installation must have privileges on all nodes in the cluster.

Configuring sudo with privileges for the individual commands can be a tedious and error-prone process; thus, the Vertica documentation does not include every possible sudo command that you can include in the sudoers file. Instead, Vertica recommends that you temporarily elevate the sudo user to have all privileges for the duration of the install.

Note
See the sudoers and visudo man pages for the details on how to write/modify a sudoers file.

To allow root sudo access on all commands as any user on any machine, use visudo as root to edit the /etc/sudoers file and add this line:

```
## Allow root to run any commands anywhere
root    ALL=(ALL) ALL
```

After the installation completes, remove (or reset) sudo privileges to the pre-installation settings.

BASH shell requirements

All shell scripts included in Vertica must run under the BASH shell. If you are on a Debian system, then the default shell can be DASH. DASH is not supported. Change the shell for root and for the dbadmin user to BASH with the chsh command.

For example:

```
# getent passwd | grep root
root:x:0:0:root:/root:/bin/dash

# chsh
Changing shell for root.
New shell [/bin/dash]: /bin/bash
Shell changed.
```

Then, as root, change the symbolic link for /bin/sh from /bin/dash to /bin/bash :

```
# rm /bin/sh
# ln -s /bin/bash /bin/sh
```

Log out and back in for the change to take effect.

Communal storage for on-premises Eon Mode databases

If you create an Eon Mode database, you must plan for your use of communal storage to store your database's data. Communal storage is based on a shared storage, such as AWS S3 or Pure Storage FlashBlade servers.

Whatever communal storage platform you use, you must ensure that it is durable (protected against data loss). The data in your Eon Mode database is only as safe as the communal storage that contains it. Most cloud provider's object stores come with a guaranteed redundancy to prevent data loss. When you install an Eon Mode database on-premises, you may have to take additional steps to prevent data loss.

Planning communal storage capacity for on-premises databases

Most cloud providers do not limit the amount of data you can store in their object stores. The only real limit is your budget; storing more data costs more money.

When you create an Eon Mode database on-premises, your storage is limited to the size of your communal storage. Unlike the cloud, you must plan ahead for the amount of storage you will need. For example, if you have a Pure Admin FlashBlade installation with three 8TB blades, then in theory, your database can grow up to 24TB. In practice, you need to account other uses of your object store, as well as factors such as data compression, and space consumed by unreaped ROS containers (storage containers no longer used by Vertica but not yet deleted by the object store).

The following calculator helps you determine the size for your communal storage needs, based on your estimated data size and additional uses of your communal storage. The values with white backgrounds in the Value column are editable. Change them to reflect your environment.

Note
The calculator currently does not work in mobile browsers. Please use a desktop browser to view the calculator.

Configure the network

This group of steps involve configuring the network. These steps differ depending on your installation scenario. A single node installation requires little network configuration, because the single instance of the Vertica server does not need to communication with other nodes in a cluster. For cluster install scenarios, you must make several decisions regarding your configuration.

Vertica supports server configuration with multiple network interfaces. For example, you might want to use one as a private network interface for internal communication among cluster hosts (the ones supplied via the --hosts option to install_vertica) and a separate one for client connections.

Important
Vertica performs best when all nodes are on the same subnet and have the same broadcast address for one or more interfaces. A cluster that has nodes on more than one subnet can experience lower performance due to the network latency associated with a multi-subnet system at high network utilization levels.

Important notes

- Network configuration is exactly the same for single nodes as for multi-node clusters, with one special exception. If you install Vertica on a single host machine that is to remain a permanent single-node configuration (such as for development or Proof of Concept), you can install Vertica using localhost or the loopback IP (typically 127.0.0.1) as the value for --hosts . Do not use the hostname localhost in a node definition if you are likely to add nodes to the configuration later.
- If you are using a host with multiple network interfaces, configure Vertica to use the address which is assigned to the NIC that is connected to the other cluster hosts.
- Use a dedicated gigabit switch. If you do not performance could be severely affected.
- Do not use DHCP dynamically-assigned IP addresses for the private network. Use only static addresses or permanently-leased DHCP addresses.

Choose IPv4 or IPv6 addresses for host identification and communications

Vertica supports using either IPv4 or IPv6 IP addresses for identifying the hosts in a database cluster. Vertica uses a single address to identify a host in the database cluster. All the IP addresses used to identify hosts in the cluster must use the same IP family.

The hosts in your database cluster can have both IPv4 and IPv6 network addresses assigned to them. Only one of these addresses is used to identify the node within the cluster. You can use the other addresses to handle client connections or connections to other systems.

You tell Vertica which address family to use when you install it. By default, Vertica uses IPv4 addresses for hosts. If you want the nodes in your database to use IPv6 addresses, add the --ipv6 option to the arguments you pass to the install_vertica script.

> **Note**
> You cannot change the address family a database cluster uses after you create it. For example, suppose you created a Vertica database using IPv4 addresses to identify the hosts in your cluster. Then you cannot later change the hosts to use an IPv6 address for internal communications.

In most cases, the address family you select does not impact how your database functions. However, there are a few exceptions:

- Use IPv4 addresses to identify the nodes in your cluster if you want to use the Management Console to manage your database. Currently, the MC does not support databases that use IPv6 addresses.
- If you select IPv6 addressing for your cluster, it automatically uses point-to-point networking mode.
- Currently, AWS is the only cloud platform on which Vertica supports IPv6 addressing. To use IPv6 on AWS, you must identify cluster hosts using IP addresses instead of host names. The AWS DNS does not support resolving host names to IPv6.
- If you only assign IPv6 addresses to the hosts in your database cluster, you may have problems interfacing to other systems that do not support IPv6.

Part of the information you pass to the install script is the list of hosts it will use to form the Vertica cluster. If you use host names in this list instead of IP addresses, ensure that the host names resolve to the IP address family you want to use for your cluster. For example, if you want your cluster to use IPv6 addresses, ensure your DNS or /etc/hosts file resolves the host names to IPv6 addresses.

You can configure DNS to return both IPv4 and IPv6 addresses for a host name. In this case, the installer uses the IPv4 address unless you supply the --ipv6 argument. If you use /etc/hosts for host name resolution (which is the best practice), host names cannot resolve to both IPv4 and IPv6 addresses.

Optionally run spread on a separate control network

If your query workloads are network intensive, you can use the --control-network parameter with the install_vertica script (see Install Vertica with the installation script) to allow spread communications to be configured on a subnet that is different from other Vertica data communications.

The --control-network parameter accepts either the default value or a broadcast network IP address (for example, 192.168.10.255).

Configure SSH

- Verify that root can use Secure Shell (SSH) to log in (ssh) to all hosts that are included in the cluster. SSH (SSH client) is a program for logging into a remote machine and for running commands on a remote machine.
- If you do not already have SSH installed on all hosts, log in as root on each host and install it before installing Vertica. You can download a free version of the SSH connectivity tools from OpenSSH .

- Make sure that /dev/pts is mounted. Installing Vertica on a host that is missing the mount point /dev/pts could result in the following error when you create a database:

```
TIMEOUT ERROR: Could not login with SSH. Here is what SSH said:Last login: Sat Dec 15 18:05:35 2007 from v_vmart_node0001
```

Allow passwordless SSH access for the dbadmin user

The dbadmin user must be authorized for passwordless ssh. In typical installs, you won't need to change anything; however, if you set up your system to disallow passwordless login, you'll need to enable it for the dbadmin user. See Enable secure shell (SSH) logins .

In this section

- Reserved ports
- Firewall considerations

Reserved ports

The install_vertica script checks that required ports are open and available to Vertica. The installer reports any issues with identifier N0020 .

You can also verify that ports required by Vertica are not in use by running the following command as the root user and comparing it with the ports required, as shown below:

```
$ netstat -atupn
```

If you are using a Red Hat 7/CentOS 7 system, use the following command instead:

```
$ ss -atupn
```

Firewall requirements

Vertica requires several ports to be open on the local network. Vertica does not recommend placing a firewall between nodes (all nodes should be behind a firewall), but if you must use a firewall between nodes, ensure the following ports are available:

Port	Protocol	Service	Notes
22	TCP	sshd	Required by Administration tools and the Management Console Cluster Installation wizard.
5433	TCP	Vertica	Vertica client (vsql, ODBC, JDBC, etc) port.
5434	TCP	Vertica	Intra- and inter-cluster communication. Vertica opens the Vertica client port +1 (5434 by default) for intra-cluster communication, such as during a plan. If the port +1 from the default client port is not available, then Vertica opens a random port for intra-cluster communication.
5433	UDP	Vertica	Vertica spread monitoring and MC cluster import.
5444	TCP	Vertica Management Console	MC-to-node and node-to-node (agent) communications port. See Changing MC or agent ports .
5450	TCP	Vertica Management Console	Port used to connect to MC from a web browser and allows communication from nodes to the MC application/web server. See Connecting to Management Console .
4803	TCP	Spread	Client connections.
8443	TCP	HTTPS	HTTPS service. To change the port, use HTTPServerPortOffset .
4803	UDP	Spread	Daemon to daemon connections.
4804	UDP	Spread	Daemon to daemon connections.
6543	UDP	Spread	Monitor to daemon connection.

Firewall considerations

Vertica requires multiple ports be open between nodes. You may use a firewall (IP Tables) on Redhat/CentOS and Ubuntu/Debian based systems. Note that firewall use is not supported on SuSE systems and that SuSE systems must disable the firewall. The installer reports issues found with your IP tables configuration with the identifiers **N0010** for (systems that use IP Tables) and **N011** (for SuSE systems).

The installer checks the IP tables configuration and issues a warning if there are any configured rules or chains. The installer does not detect if the configuration may conflict with Vertica. It is your responsibility to verify that your firewall allows traffic for Vertica as described in Reserved ports .

> **Note**
> The installer does not check NAT entries in iptables.

You can modify your firewall to allow for Vertica network traffic, or you can disable the firewall if your network is secure. Note that firewalls are not supported for Vertica systems running on SuSE.

Important
You may encounter the **N0010** issue even when the firewall is disabled. If this occurs, you can workaround this issue and install Vertica by ignoring installer WARN messages. To do this, install (or update) with a failure threshold of FAIL. For example, /opt/vertica/sbin/install_vertica --failure-threshold FAIL <other install options...> .

Red hat 6 and CentOS 6 systems

For details on how to configure iptables and allow specific ports to be open, see the platform-specific documentation for your platform:

- RedHat: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/sect-Security_Guide-IPTables.html
- CentOS: http://wiki.centos.org/HowTos/Network/IPTables

To disable iptables, run the following command as root or sudo:

```
# service iptables save
# service iptables stop
# chkconfig iptables off
```

To disable iptables if you are using the ipv6 versions of iptables, run the following command as root or sudo:

```
# service ip6tables save
# service ip6tables stop
# chkconfig ip6tables off
```

Red hat 7 and CentOS 7 systems:

To disable the system firewall, run the following command as root or sudo:

```
# systemctl mask firewalld
# systemctl disable firewalld
# systemctl stop firewalld
```

Ubuntu and debian systems

For details on how to configure iptables and allow specific ports to be open, see the platform-specific documentation for your platform:

- Debian: https://wiki.debian.org/iptables
- Ubuntu: https://help.ubuntu.com/12.04/serverguide/firewall.html .

> **Note**
> Ubuntu uses the ufw program to manage iptables.

To disable iptables on Debian, run the following command as root or sudo:

```
$ /etc/init.d/iptables stop

$ update-rc.d -f iptables remove
```

To disable iptables on Ubuntu, run the following command:

```
$ sudo ufw disable
```

SuSE systems

The firewall must be disabled on SUSE systems. To disable the firewall on SuSE systems, run the following command:

```
# /sbin/SuSEfirewall2 off
```

Operating system configuration overview

This topic provides a high-level overview of the OS settings required for Vertica. Each item provides a link to additional details about the setting and detailed steps on making the configuration change. The installer tests for all of these settings and provides hints, warnings, and failures if the current configuration does not meet Vertica requirements.

Before you install the operating system

The below sections detail system settings that must be configured when you install the operating system. These settings cannot be easily changed after the operating system is installed.

Configuration	Description
Supported Platforms	Verify that your servers meet the platform requirements described in Supported Platforms . Unsupported operating systems are detected by the installer. The installer generates one of the following issue identifiers if it detects an unsupported operating system: • [**S0320**] - Fedora OS is not supported. • [**S0321**] - The version of Red Hat/CentOS is not supported. • [**S0322**] - The version of Ubuntu/Debian is not supported. • [**S0323**] - The operating system could not be determined. The unknown operating system is not supported because it does not match the list of supported operating systems. • [**S0324**] - The version of Red Hat is not supported.
LVM	Vertica Analytic Database supports Linux Volume Manager (LVM) on all supported operating systems. For information on LVM requirements and restrictions, see the section, Vertica Support for LVM.
File system	Choose the storage format type based on deployment requirements. Vertica recommends the following storage format types where applicable: • ext3 • ext4 • NFS for backup • XFS • Amazon S3 Standard, Azure Blob Storage, or Google Cloud Storage for communal storage and related backup tasks when running in Eon Mode **Note** For the Vertica I/O profile, the ext4 file system is considerably faster than ext3. The storage format type at your backup and temporary directory locations must support fcntl lockf (POSIX) file locking.

Swap Space	A 2GB swap partition is required, regardless of the amount of RAM installed on your system. Larger swap space is acceptable, but unnecessary. Partition the remaining disk space in a single partition under "/". If you do not have the required 2GB swap partition, the installer reports this issue with identifier **S0180** . You typically define the swap partition when you install Linux. See your platform's documentation for details on configuring the swap partition. **Note** Do not place a swap file on a disk containing the Vertica data files. If a host has only two disks (boot and data), put the swap file on the boot disk.
Disk Block Size	The disk block size for the Vertica data and catalog directories should be 4096 bytes, the default on ext4 and XFS file systems. You set the disk block size when you format your file system. If you change the block size, you will need to reformat the disk.
Memory	Vertica requires that your hosts have a minimum of 1GB of RAM per logical processor. If your hosts do not meet this requirement, the installer reports this issue with the identifier **S0190** . For performance reasons, you typically require more RAM than the minimum. In addition to the individual host RAM requirement, the installer also reports a hint if the hosts in your cluster do not have identical amounts of RAM. Ensuring your host have the same amount of RAM helps prevent performance issues if one or more nodes has less RAM than the other nodes in your database. **Note** In an Eon Mode database, after you create the initial cluster, you can configure subclusters that have different hardware specifications (including RAM) than the initial primary subcluster the installer creates. For more information on sizing your hardware, see the Vertica Knowledge Base Hardware documents .

Automatically configured operating system settings

These general OS settings are automatically made by the installer if they do not meet Vertica requirements. You can prevent the installer from automatically making these configuration changes by using the --no-system-configuration parameter for the install_vertica script.

For more information on each configuration setting, see Automatically configured operating system settings .

Configuration	Description
Nice Limits	The database administration user must be able to *nice* processes back to the default level of 0.
min_free_kbytes	The vm.min_free_kbytes setting in /etc/sysctl.conf must be configured sufficiently high. The specific value depends on your hardware configuration.
User Open Files Limit	The open file limit for the dbadmin user should be at least 1 file open per MB of RAM, 65536, or the amount of RAM in MB; whichever is greater.
System Open File Limits	The maximum number of files open on the system must not be less than at least the amount of memory in MB, but not less than 65536.
Pam Limits	/etc/pam.d/su must contain the line: session required pam_limits.so This allows for the conveying of limits to commands run with the su - command.
Address Space Limits	The address space limits (as setting) defined in /etc/security/limits.conf must be **unlimited** for the database administrator.

File Size Limits	The file sizelimits (fsize setting) defined in /etc/security/limits.conf must be **unlimited** for the database administrator.
User Process Limits	The nproc setting defined in /etc/security/limits.conf must be 1024 or the amount of memory in MB, whichever is greater.
Maximum Memory Maps	The vm.max_map_count in /etc/sysctl.conf must be **65536** or the amount of memory in KB / 16, whichever is greater.

Manually configured operating system settings

For more information on each configuration setting, see Manually configured operating system settings .

Configuration	Description
Disk Readahead	This disk readahead must be at least 2048, with a high of 8192. Set this high limit only with the help of Vertica support. The specific value depends on your hardware configuration.
NTP Services	The NTP daemon must be enabled and running, with the exception of Red Hat 7 and CentOS 7 systems.
chrony	For Red Hat 7 and CentOS 7 systems, chrony must be enabled and running.
SELinux	SElinux must be disabled or run in permissive mode.
CPU Frequency Scaling	Vertica recommends that you disable CPU Frequency Scaling. Important Your systems may use significantly more energy when CPU frequency scaling is disabled.
Transparent Hugepages	For Red Hat 7, CentOS 7 and Amazon Linux 2.0, Transparent Hugepages must be set to *always* . For all other operating systems, Transparent Hugepages must be disabled or set to *madvise* .
I/O Scheduler	The I/O Scheduler for disks used by Vertica must be set to *deadline* or *noop* .
Support Tools	Several optional packages can be installed to assist Vertica support when troubleshooting your system.

System user requirements

The following tasks pertain to the configuration of the system user required by Vertica.

For more information on each configuration setting, see System user configuration .

Configuration	Required Setting(s)
System User Requirements	The installer automatically creates a user with the correct settings. If you specify a user with --dba-use r, then the user must conform to the requirements for the Vertica system user.
LANG Environment Settings	The LANG environment variable must be set and valid for the database administration user.
TZ Environment Settings	The TZ environment variable must be set and valid for the database administration user.

Automatically configured operating system settings

These general Operating System settings are automatically made by the installer. You can prevent the installer from automatically making these configuration changes by using the --no-system-configuration parameter for the install_vertica script.

In this section

- Sysctl
- Nice limits configuration
- min_free_kbytes setting
- User max open files limit
- System max open files limit
- Pam limits
- pid_max setting
- User address space limits
- User file size limit
- User process limit
- Maximum memory maps configuration

Sysctl

During installation, Vertica attempts to automatically change various OS level settings. The installer may not change values on your system if they exceed the threshold required by the installer. You can prevent the installer from automatically making these configuration changes by using the --no-system-configuration parameter for the install_vertica script.

To permanently edit certain settings and prevent them from reverting on reboot, use sysctl.

The sysctl settings relevant to the installation of Vertica include:

- min_free_kbytes
- fs.file_max
- vm.max_map_count

Permanently changing settings with sysctl:

1. As the root user, open the /etc/sysctl.conf file:
   ```
   # vi /etc/sysctl.conf
   ```
2. Enter a parameter and value:
   ```
   parameter = value
   ```
   For example, to set the parameter and value for fs.file-max to meet Vertica requirements, enter:
   ```
   fs.file-max = 65536
   ```
3. Save your changes, and close the /etc/sysctl.conf file.
4. As the root user, reload the config file:
   ```
   # sysctl -p
   ```

Identifying settings added by the installer

You can see whether the installer has added a setting by opening the /etc/sysctl.conf file:

```
# vi /etc/sysctl.conf
```

If the installer has added a setting, the following line appears:

```
# The following 1 line added by Vertica tools. 2015-02-23 13:20:29
parameter = value
```

Nice limits configuration

The Vertica system user (dbadmin by default) must be able to raise and lower the priority of Vertica processes. To do this, the nice option in the /etc/security/limits.conf file must include an entry for the dbadmin user. The installer reports this issue with the identifier: **S0010** .

The installer automatically configures the correct setting if the default value does not meet system requirements. If an issue occurs when setting this value, or you use the --no-system-configuration argument to the installer and the current setting is incorrect, then the installer reports this as an issue.

> **Note**
> Vertica never raises priority above the default level of 0. However, Vertica does lower the priority of certain Vertica threads and needs to able to raise the priority of these threads back up to the default level. This setting allows Vertica to raise the priorities back to the default level.

All systems

To set the Nice Limit configuration for the dbadmin user, edit /etc/security/limits.conf and add the following line. Replace *dbadmin* with the name of your system user.

```
dbadmin -       nice    0
```

min_free_kbytes setting

This topic details how to update the min_free_kbytes setting so that it is within the range supported by Vertica. The installer reports this issue with the identifier: **S0050** if the setting is too low, or **S0051** if the setting is too high.

The vm.min_free_kbytes setting configures the page reclaim thresholds. When this number is increased the system starts reclaiming memory earlier, when its lowered it starts reclaiming memory later. The default min_free_kbytes is calculated at boot time based on the number of pages of physical RAM available on the system.

The setting must be whichever value is the greatest from the following options:

- The default value configured by the system
- 4096
- The result of running the commands:

```
$ memtot=`grep MemTotal /proc/meminfo | awk '{printf "%.0f",$2}'`
$ echo "scale=0;sqrt ($memtot*16)" | bc
```

The installer automatically configures the correct setting if the default value does not meet system requirements. If an issue occurs when setting this value, or you use the --no-system-configuration argument to the installer and the current setting is incorrect, then the installer reports this as an issue.

All systems

To manually set min_free_kbytes:

1. Determine the current/default setting with the following command:

```
$ sysctl vm.min_free_kbytes
```

2. If the result of the previous command is No such file or directory or the default value is less than 4096, then run these commands to determine the correct value:

```
$ memtot=`grep MemTotal /proc/meminfo | awk '{printf "%.0f",$2}'`
$ echo "scale=0;sqrt ($memtot*16)" | bc
```

3. Edit or add the current value of vm.min_free_kbytes in /etc/sysctl.conf with the value from the output of the previous command.

```
# The min_free_kbytes setting
vm.min_free_kbytes=16132
```

4. Run sysctl -p to apply the changes in sysctl.conf immediately.

> **Note**
> These steps must be repeated for each node in the cluster.

User max open files limit

This topic details how to change the user max open-files limit setting to meet Vertica requirements. The installer reports this issue with the identifier S0060.

The installer automatically configures the correct setting if the default value does not meet system requirements. If an issue occurs when setting this value, or you use the --no-system-configuration argument to the installer and the current setting is incorrect, then the installer reports this as an issue.

Vertica requires that the dbadmin user not be limited when opening files. The open file limit per user is calculated as follows:

user max open files = greater of { ≥ 65536 | ≤ *RAM-MBs* }

As a dbadmin user, you can determine the open file limit by running ulimit -n . For example:

```
$ ulimit -n
65536
```

To manually set the limit, edit /etc/security/limits.conf and edit/add the nofile setting for the user who is configured as the database administrator—by default, dbadmin . For example:

```
dbadmin -       nofile  65536
```

The setting must be no less than 65536 MB, but not greater than the system value of fs.nr_open . For example, the default value of fs.nr_open value on Red Hat Enterprise Linux 9 is 1048576 MB.

> **Note**
> The system also has a limit on open files .

System max open files limit

This topic details how to modify the limit for the number of open files on your system so that it meets Vertica requirements. The installer reports this issue with the identifier: **S0120** .

Vertica opens many files. Some platforms have global limits on the number of open files. The open file limit must be set sufficiently high so as not to interfere with database operations.

The recommended value is at least the amount of memory in MB, but not less than 65536.

The installer automatically configures the correct setting if the default value does not meet system requirements. If an issue occurs when setting this value, or you use the --no-system-configuration argument to the installer and the current setting is incorrect, then the installer reports this as an issue.

All systems

To manually set the open file limit:

1. Run /sbin/sysctl fs.file-max to determine the current limit.
2. If the limit is not **65536** or the amount of system memory in MB (whichever is higher), then edit or add fs.file-max= *max number of files* to /etc/sysctl.conf .

   ```
   # Controls the maximum number of open files
   fs.file-max=65536
   ```

3. Run sysctl -p to apply the changes in sysctl.conf immediately.

> **Note**
> These steps will need to be replicated for each node in the cluster.

Pam limits

This topic details how to enable the "su" pam_limits.so module required by Vertica. The installer reports issues with the setting with the identifier: **S0070** .

On some systems the pam module called pam_limits.so is not set in the file /etc/pam.d/su . When it is not set, it prevents the conveying of limits (such as open file descriptors) to any command started with su - .

In particular, the Vertica init script would fail to start Vertica because it calls the Administration Tools to start a database with the su - command. This problem was first noticed on Debian systems, but the configuration could be missing on other Linux distributions. See the pam_limits man page for more details.

The installer automatically configures the correct setting if the default value does not meet system requirements. If an issue occurs when setting this value, or you use the --no-system-configuration argument to the installer and the current setting is incorrect, then the installer reports this as an issue.

All systems

To manually configure this setting, append the following line to the /etc/pam.d/su file:

```
session required pam_limits.so
```

See the pam_limits man page for more details: man pam_limits .

pid_max setting

This topic explains how to change pid_max to a supported value. The value of pid_max should be

```
pid_max = num-user-proc + 2**15 = num-user-proc + 32768
```

where *num-user-proc* is the size of memory in megabytes.

The minimum value for pid_max is 524288.

If your pid_max value is too low, the installer reports this problem and indicates the minimum value.

The installer automatically configures the correct setting if the default value does not meet system requirements. If an issue occurs when setting this value, or you use the --no-system-configuration argument to the installer and the current setting is incorrect, then the installer reports this as an issue.

All systems

To change the pid_max value:

```
# sysctl -w kernel.pid_max=524288
```

User address space limits

This topic details how to modify the Linux address space limit for the dbadmin user so that it meets Vertica requirements. The address space setting controls the maximum number of threads and processes for each user. If this setting does not meet the requirements then the installer reports this issue with the identifier: **S0090** .

The installer automatically configures the correct setting if the default value does not meet system requirements. If an issue occurs when setting this value, or you use the --no-system-configuration argument to the installer and the current setting is incorrect, then the installer reports this as an issue.

The address space available to the dbadmin user must not be reduced via user limits and must be set to **unlimited** .

All systems

To manually set the address space limit:

1. Run ulimit -v as the dbadmin user to determine the current limit.
2. If the limit is not **unlimited** , then add the following line to /etc/security/limits.conf . Replace *dbadmin* with your database admin user

```
dbadmin -       as      unlimited
```

User file size limit

This topic details how to modify the file size limit for files on your system so that it meets Vertica requirements. The installer reports this issue with the identifier: **S0100** .

The installer automatically configures the correct setting if the default value does not meet system requirements. If an issue occurs when setting this value, or you use the --no-system-configuration argument to the installer and the current setting is incorrect, then the installer reports this as an issue.

The file size limit for the dbadmin user must not be reduced via user limits and must be set to **unlimited** .

All systems

To manually set the file size limit:

1. Run ulimit -f as the dbadmin user to determine the current limit.
2. If the limit is not **unlimited** , then edit/add the following line to /etc/security/limits.conf . Replace *dbadmin* with your database admin user.

```
dbadmin -       fsize      unlimited
```

User process limit

This topic details how to change the user process limit so that it meets Vertica requirements.The installer reports this issue with the identifier: **S0110** .

The installer automatically configures the correct setting if the default value does not meet system requirements. If an issue occurs when setting this value, or you use the --no-system-configuration argument to the installer and the current setting is incorrect, then the installer reports this as an issue.

The user process limit must be high enough to allow for the many threads opened by Vertica. The recommended limit is the amount of RAM in MB and must be at least 1024.

All systems

To manually set the user process limit:

1. Run ulimit -u as the dbadmin user to determine the current limit.
2. If the limit is not the amount of memory in MB on the server, then edit/add the following line to /etc/security/limits.conf . Replace *4096* with the amount of system memory, in MB, on the server.

```
dbadmin -       nproc     4096
```

Maximum memory maps configuration

This topic details how to modify the limit for the number memory maps a process can have on your system so that it meets Vertica requirements. The installer reports this issue with the identifier: **S0130** .

The installer automatically configures the correct setting if the default value does not meet system requirements. If an issue occurs when setting this value, or you use the --no-system-configuration argument to the installer and the current setting is incorrect, then the installer reports this as an issue.

Vertica uses a lot of memory while processing and can approach the default limit for memory maps per process.

The recommended value is at least the amount of memory on the system in KB / 16, but not less than 65536.

All systems

To manually set the memory map limit:

1. Run /sbin/sysctl vm.max_map_count to determine the current limit.
2. If the limit is not **65536** or the amount of system memory in KB / 16 (whichever is higher), then edit/add the following line to /etc/sysctl.conf . Replace *65536* with the value for your system.

```
# The following 1 line added by Vertica tools. 2014-03-07 13:20:31


vm.max_map_count=65536
```

3. Run sysctl -p to apply the changes in sysctl.conf immediately.

> **Note**
> These steps will need to be replicated for each node in the cluster.

Manually configured operating system settings

The topics in this section detail general Operating System settings that must be set manually.

Persisting operating system settings

To prevent manually set Operating System settings from reverting on reboot, you should configure some of these settings in the /etc/rc.local script. This script contains commands and scripts that run each time the system is booted.

Important

On reboot, SUSE systems use the /etc/init.d/after.local file rather than /etc/rc.local .

Vertica uses settings in /etc/rc.local to set the following functionality:

- Disk readahead
- I/O scheduling
- Enabling or disabling transparent hugepages

Editing /etc/rc.local

1. As the root user, open /etc/rc.local :

```
# vi /etc/rc.local
```

2. Enter a script or command. For example, to configure transparent hugepages to meet Vertica requirements, enter the following:

```
echo never > /sys/kernel/mm/redhat_transparent_hugepage/enabled
```

Important
On some Ubuntu/Debian systems, the last line in /etc/rc.local must be exit 0 . All additions to /etc/rc.local must precede this line.

3. Save your changes, and close /etc/rc.local .
4. If you use Red Hat 7.0 or CentOS 7.0 or higher, run the following command as root or sudo:

```
$ chmod +x /etc/rc.d/rc.local
```

On reboot, the command runs during startup. You can also run the command manually as the root user, if you want it to take effect immediately.

Disabling tuning system service

If you use Red Hat 7.0 or CentOS 7.0 or higher, make sure the tuning system service does not start on when Vertica reboots. Turning off tuning prevents monitoring of your OS and any tuning of your OS based on this monitoring. Tuning also enables THP silently, which can cause issues in other areas such as read ahead.

Run the following command as sudo or root:

```
$ chkconfig tuned off
```

In this section

- SUSE control groups configuration
- Cron required for scheduled jobs
- Disk readahead
- I/O scheduling
- Enabling or disabling transparent hugepages
- Check for swappiness
- Enabling network time protocol (NTP)
- Enabling chrony or ntpd for red hat 7/CentOS 7 systems
- SELinux configuration
- CPU frequency scaling
- Enabling or disabling defrag
- Support tools

SUSE control groups configuration

On SuSE 12, the installer checks the control group (cgroup) setting for the cgroups that Vertica may run under:

- verticad
- vertica_agent
- sshd

The installer verifies that the pid.max resource is large enough for all the threads that Vertica creates. We check the contents of:

- /sys/fs/cgroup/pids/system.slice/verticad.service/pids.max
- /sys/fs/cgroup/pids/system.slice/vertica_agent.service/pids.max
- /sys/fs/cgroup/pids/system.slice/sshd.service/pids.max

If these files exist and they fail to include the value max , the installation stops and the installer returns a failure message (code S0340).

If these files do not exist, they are created automatically when the systemd runs the verticad and vertica_agent startup scripts. However, the site's cgroup configuration process managed their default values. Vertica does not change the defaults.

Pre-installation configuration

Before installing Vertica, configure your system as follows:

```
# Create the following directories:
sudo mkdir /sys/fs/cgroup/pids/system.slice/verticad.service/
sudo mkdir /sys/fs/cgroup/pids/system.slice/vertica_agent.service/
# sshd service dir should already exist, so don't need to create it

# Set pids.max values:
sudo sh -c 'echo "max" > /sys/fs/cgroup/pids/system.slice/verticad.service/pids.max'
sudo sh -c 'echo "max" > /sys/fs/cgroup/pids/system.slice/vertica_agent.service/pids.max'
sudo sh -c 'echo "max" > /sys/fs/cgroup/pids/system.slice/sshd.service/pids.max'
```

Persisting configuration for restart

After installation, you can configure control groups for subsequent reboots of the Vertica database. You do so by editing configuration file /etc/init.d/after.local and adding the commands shown earlier.

> **Note**
> Because after.local is executed as root, it can omit sudo commands.

Cron required for scheduled jobs

Admintools uses the Linux cron package to schedule jobs that regularly rotate the database logs. Without this package installed, the database logs will never be rotated. The lack of rotation can lead to a significant consumption of storage for logs. On busy clusters, Vertica can produce hundreds of gigabytes of logs per day.

cron is installed by default on most Linux distributions, but it may not be present on some SUSE 12 systems.

To install cron , run this command:

```
$ sudo zypper install cron
```

Disk readahead

Vertica requires that Disk Readahead be set to at least 2048. The installer reports this issue with the identifier: **S0020** .

> **Note**
> - These commands must be executed with root privileges and assumes the blockdev program is in /sbin .
> - The blockdev program operates on whole devices, and not individual partitions. You cannot set the readahead value to different settings on the same device. If you run blockdev against a partition, for example: /dev/sda1, then the setting is still applied to the entire /dev/sda device. For instance, running /sbin/blockdev --setra 2048 /dev/sda1 also causes /dev/sda2 *through* /dev/sda *N* to use a readahead value of 2048.

RedHat/CentOS and SuSE based systems

For each drive in the Vertica system, Vertica recommends that you set the readahead value to at least 2048 for most deployments. The command immediately changes the readahead value for the specified disk. The second line adds the command to /etc/rc.local so that the setting is applied each time the system is booted. Note that some deployments may require a higher value and the setting can be set as high as 8192, under guidance of support.

> **Note**
> For systems that do not support /etc/rc.local , use the equivalent startup script that is run after the destination runlevel has been reached. For example SUSE uses /etc/init.d/after.local .

The following example sets the readahead value of the drive sda to 2048:

```
$ /sbin/blockdev --setra 2048 /dev/sda
$ echo '/sbin/blockdev --setra 2048 /dev/sda' >> /etc/rc.local
```

If you are using Red Hat 7.0 or CentOS 7.0 or higher, run the following command as root or sudo:

```
$ chmod +x /etc/rc.d/rc.local
```

Ubuntu and debian systems

For each drive in the Vertica system, set the readahead value to 2048. Run the command once in your shell, then add the command to /etc/rc.local so that the setting is applied each time the system is booted. Note that on Ubuntu systems, the last line in rc.local must be " exit 0 ". So you must manually add the following line to etc/rc.local before the last line with exit 0 .

> **Note**
> For systems that do not support /etc/rc.local , use the equivalent startup script that is run after the destination runlevel has been reached. For

example SuSE uses /etc/init.d/after.local .

```
/sbin/blockdev --setra 2048 /dev/sda
```

I/O scheduling

Vertica requires that I/O Scheduling be set to deadline or noop . The installer checks what scheduler the system is using, reporting an unsupported scheduler issue with identifier: **S0150** . If the installer cannot detect the type of scheduler in use (typically if your system is using a RAID array), it reports that issue with identifier: **S0151** .

If your system is not using a RAID array, then complete the following steps to change your system to a supported I/O Scheduler. If you are using a RAID array, then consult your RAID vendor documentation for the best performing scheduler for your hardware.

Configure the I/O scheduler

The Linux kernel can use several different I/O schedulers to prioritize disk input and output. Most Linux distributions use the Completely Fair Queuing (CFQ) scheme by default, which gives input and output requests equal priority. This scheduler is efficient on systems running multiple tasks that need equal access to I/O resources. However, it can create a bottleneck when used on Vertica drives containing the catalog and data directories, because it gives write requests equal priority to read requests, and its per-process I/O queues can penalize processes making more requests than other processes.

Instead of the CFQ scheduler, configure your hosts to use either the Deadline or NOOP I/O scheduler for the drives containing the catalog and data directories:

- The Deadline scheduler gives priority to read requests over write requests. It also imposes a deadline on all requests. After reaching the deadline, such requests gain priority over all other requests. This scheduling method helps prevent processes from becoming starved for I/O access. The Deadline scheduler is best used on physical media drives (disks using spinning platters), since it attempts to group requests for adjacent sectors on a disk, lowering the time the drive spends seeking.
- The NOOP scheduler uses a simple FIFO approach, placing all input and output requests into a single queue. This scheduler is best used on solid state drives (SSDs). Because SSDs do not have a physical read head, no performance penalty exists when accessing non-adjacent sectors.

Failure to use one of these schedulers for the Vertica drives containing the catalog and data directories can result in slower database performance. Other drives on the system (such as the drive containing swap space, log files, or the Linux system files) can still use the default CFQ scheduler (although you should always use the NOOP scheduler for SSDs).

You can set your disk device scheduler by writing the name of the scheduler to a file in the /sys directory or using a kernel boot parameter.

Changing the scheduler through the /sys directory

You can view and change the scheduler Linux uses for I/O requests to a single drive using a virtual file under the /sys directory. The name of the file that controls the scheduler a block device uses is:

```
/sys/block/deviceName/queue/scheduler
```

Where *deviceName* is the name of the disk device, such as sda or cciss\!c0d1 (the first disk on an OpenText RAID array). Viewing the contents of this file shows you all of the possible settings for the scheduler. The currently-selected scheduler is surrounded by square brackets:

```
# cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
```

To change the scheduler, write the name of the scheduler you want the device to use to its scheduler file. You must have root privileges to write to this file. For example, to set the sda drive to use the deadline scheduler, run the following command as root:

```
# echo deadline > /sys/block/sda/queue/scheduler
# cat /sys/block/sda/queue/scheduler
noop [deadline] cfq
```

Changing the scheduler immediately affects the I/O requests for the device. The Linux kernel starts using the new scheduler for all of the drive's input and output requests.

Note

While tests show that changing the scheduler settings while Vertica is running does not cause problems, Vertica recommends shutting down. Before changing the I/O schedule, or making any other changes to the system configuration, consider shutting down any running database.

Changes to the I/O scheduler made through the /sys directory only last until the system is rebooted, so you need to add the commands that change the I/O scheduler to a startup script (such as those stored in /etc/init.d , or though a command in /etc/rc.local). You also need to use a separate command for each drive on the system whose scheduler you want to change.

For example, to make the configuration take effect immediately and add it to rc.local so it is used on subsequent reboots.

> **Note**
> For systems that do not support /etc/rc.local , use the equivalent startup script that is run after the destination runlevel has been reached. For example SuSE uses /etc/init.d/after.local .

```
echo deadline > /sys/block/sda/queue/scheduler
echo 'echo deadline > /sys/block/sda/queue/scheduler' >> /etc/rc.local
```

> **Note**
> On some Ubuntu/Debian systems, the last line in rc.local must be " exit 0 ". So you must manually add the following line to etc/rc.local before the last line with exit 0 .

You may prefer to use this method of setting the I/O scheduler over using a boot parameter if your system has a mix of solid-state and physical media drives, or has many drives that do not store Vertica catalog and data directories.

If you are using Red Hat 7.0 or CentOS 7.0 or higher, run the following command as root or sudo:

```
$ chmod +x /etc/rc.d/rc.local
```

Changing the scheduler with a boot parameter

Use the elevator kernel boot parameter to change the default scheduler used by all disks on your system. This is the best method to use if most or all of the drives on your hosts are of the same type (physical media or SSD) and will contain catalog or data files. You can also use the boot parameter to change the default to the scheduler the majority of the drives on the system need, then use the /sys files to change individual drives to another I/O scheduler. The format of the elevator boot parameter is:

```
elevator=schedulerName
```

Where *schedulerName* is deadline , noop , or cfq . You set the boot parameter using your bootloader (grub or grub2 on most recent Linux distributions). See your distribution's documentation for details on how to add a kernel boot parameter.

Enabling or disabling transparent hugepages

You can modify transparent hugepages to meet Vertica configuration requirements:

- For Red Hat 7/CentOS 7 and Amazon Linux 2.0, you must enable transparent hugepages. The installer reports this issue with the identifier: **S0312** .
- For Red Hat 8/CentOS 8 and SUSE 15.1, Vertica provides recommended settings to optimize your system performance by workload.
- For all other systems, you must disable transparent hugepages or set them to madvise . The installer reports this issue with the identifier: **S0310** .

Recommended settings by workload for red hat 8/CentOS 8 and SUSE 15.1

Vertica recommends transparent hugepages settings to optimize performance by workload. The following table contains recommendations for systems that primarily run concurrent queries (such as short-running dashboard queries), or sequential SELECT or load (COPY) queries:

Operating System	Concurrent	Sequential	Important Notes
Red Hat 8.0/CentOS 8.0	Disable	Enable	

SUSE 15.1	Disable	Enable	Additionally, Vertica recommends the following khugepaged settings to optimize for each workload: **Concurrent Workloads:** Disable khugepaged with the following command: echo 0 > /sys/kernel/mm/transparent_hugepage/khugepaged/defrag **Sequential Workloads:** Enable khugepaged with the following command: echo 1 > /sys/kernel/mm/transparent_hugepage/khugepaged/defrag

See Enabling or disabling defrag for additional settings that optimize your system performance by workload.

Enabling transparent hugepages on red hat 7/8, CentOS 7/8, SUSE 15.1, and Amazon Linux 2.0

Determine if transparent hugepages is enabled. To do so, run the following command.

```
cat /sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
```

The setting returned in brackets is your current setting.

For systems that do not support /etc/rc.local , use the equivalent startup script that is run after the destination runlevel has been reached. For example SuSE uses /etc/init.d/after.local .

You can enable transparent hugepages by editing /etc/rc.local and adding the following script:

```
if test -f /sys/kernel/mm/transparent_hugepage/enabled; then
   echo always > /sys/kernel/mm/transparent_hugepage/enabled
fi
```

You must reboot your system for the setting to take effect, or, as root, run the following echo line to proceed with the install without rebooting:

```
# echo always > /sys/kernel/mm/transparent_hugepage/enabled
```

If you are using Red Hat 7.0 or CentOS 7.0 or higher, run the following command as root or sudo:

```
$ chmod +x /etc/rc.d/rc.local
```

Disabling transparent hugepages on other systems

> **Note**
> SUSE did not offer transparent hugepage support in its initial 11.0 release. However, subsequent SUSE service packs do include support for transparent hugepages.

To determine if transparent hugepages is enabled, run the following command.

```
cat /sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
```

The setting returned in brackets is your current setting. Depending on your platform OS, the madvise setting may not be displayed.

You can disable transparent hugepages one of two ways:

- Edit your boot loader (for example /etc/grub.conf). Typically, you add the following to the end of the kernel line. However, consult the documentation for your system before editing your bootloader configuration.
  ```
  transparent_hugepage=never
  ```
- Edit /etc/rc.local (on systems that support rc.local) and add the following script.
  ```
  if test -f /sys/kernel/mm/transparent_hugepage/enabled; then
     echo never > /sys/kernel/mm/transparent_hugepage/enabled
  fi
  ```

For systems that do not support /etc/rc.local , use the equivalent startup script that is run after the destination runlevel has been reached. For example SuSE uses /etc/init.d/after.local .

Regardless of which approach you choose, you must reboot your system for the setting to take effect, or run the following two echo lines to proceed with the install without rebooting:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

Check for swappiness

The swappiness kernel parameter defines the amount, and how often, the kernel copies RAM contents to a swap space. Vertica recommends a value of 0. The installer reports any swappiness issues with identifier **S0112** .

You can check the swappiness value by running the following command:

```
$ cat /proc/sys/vm/swappiness
```

To set the swappiness value add or update the following line in /etc/sysctl.conf :

```
vm.swappiness = 0
```

This also ensures that the value persists after a reboot.

If necessary, you change the swappiness value at runtime by logging in as root and running the following:

```
$ echo 0 > /proc/sys/vm/swappiness
```

Enabling network time protocol (NTP)

Important

Data damage and performance issues might occur if you change host NTP settings while the database is running. Before you change the NPT settings, stop the database. If you cannot stop the database, stop the Vertica process of each host and change the NTP settings one host at a time.

For details, see Stopping Vertica on host .

The network time protocol (NTP) daemon must be running on all of the hosts in the cluster so that their clocks are synchronized. The spread daemon relies on all of the nodes to have their clocks synchronized for timing purposes. If your nodes do not have NTP running, the installation can fail with a spread configuration error or other errors.

> **Note**
> Different Linux distributions refer to the NTP daemon in different ways. For example, SUSE and Debian/Ubuntu refer to it as ntp , while CentOS and Red Hat refer to it as ntpd . If the following commands produce errors, try using the other NTP daemon reference name.

Verify that NTP is running

To verify that your hosts are configured to run the NTP daemon on startup, enter the following command:

```
$ chkconfig --list ntpd
```

Debian and Ubuntu do not support chkconfig , but they do offer an optional package. You can install this package with the command sudo apt-get install sysv-rc-conf . To verify that your hosts are configured to run the NTP daemon on startup with the sysv-rc-conf utility, enter the following command:

```
$ sysv-rc-conf --list ntpd
```

The chkconfig command can produce an error similar to ntpd: unknown service . If you get this error, verify that your Linux distribution refers to the NTP daemon as ntpd rather than ntp . If it does not, you need to install the NTP daemon package before you can configure it. Consult your Linux documentation for instructions on how to locate and install packages.

If the NTP daemon is installed, your output should resemble the following:

```
ntp 0:off 1:off 2:on 3:on 4:off 5:on 6:off
```

The output indicates the runlevels where the daemon runs. Verify that the current runlevel of the system (usually 3 or 5) has the NTP daemon set to on . If you do not know the current runlevel, you can find it using the runlevel command:

```
$ runlevel
N 3
```

Configure NTP for red hat 6/CentOS 6 and SLES

If your system is based on Red Hat 6/CentOS 6 or SUSE Linux Enterprise Server, use the service and chkconfig utilities to start NTP and have it start at startup.

```
$ /sbin/service ntpd restart
$ /sbin/chkconfig ntpd on
```

- **Red Hat 6/CentOS 6** —NTP uses the default time servers at ntp.org. You can change the default NTP servers by editing /etc/ntpd.conf .
- **SLES** —By default, no time servers are configured. You must edit /etc/ntpd.conf after the install completes and add time servers.

Configure NTP for ubuntu and debian

By default, the NTP daemon is not installed on some Ubuntu and Debian systems. First, install NTP, and then start the NTP process. You can change the default NTP servers by editing /etc/ntpd.conf as shown:

```
$ sudo apt-get install ntp
$ sudo /etc/init.d/ntp reload
```

Verify that NTP is operating correctly

To verify that the Network Time Protocol Daemon (NTPD) is operating correctly, issue the following command on all nodes in the cluster.

For Red Hat 6/CentOS 6 and SLES:

```
$ /usr/sbin/ntpq -c rv | grep stratum
```

For Ubuntu and Debian:

```
$ ntpq -c rv | grep stratum
```

A stratum level of 16 indicates that NTP is not synchronizing correctly.

If a stratum level of 16 is detected, wait 15 minutes and issue the command again. It may take this long for the NTP server to stabilize.

If NTP continues to detect a stratum level of 16, verify that the NTP port (UDP Port 123) is open on all firewalls between the cluster and the remote machine to which you are attempting to synchronize.

Red hat documentation related to NTP

The preceding links were current as of the last publication of the Vertica documentation and could change between releases.

- http://kbase.redhat.com/faq/docs/DOC-6731
- http://kbase.redhat.com/faq/docs/DOC-6902
- http://kbase.redhat.com/faq/docs/DOC-6991

Enabling chrony or ntpd for red hat 7/CentOS 7 systems

Before you can install Vertica, you must enable one of the following on your system for clock synchronization:

- chrony
- NTPD

You must enable and activate the Network Time Protocol (NTP) before installation. Otherwise, the installer reports this issue with the identifier **S0030** .

For information on installing and using chrony, see the information below. For information on NTPD see Enabling network time protocol (NTP) . For more information about chrony, see Using chrony in the Red Hat documentation.

Install chrony

The chrony suite consists of:

- chronyd - the daemon for clock synchronization.
- chronyc - the command-line utility for configuring chronyd .

chrony is installed by default on some versions of Red Hat/CentOS 7. However, if chrony is not installed on your system, you must download it. To download chrony, run the following command as sudo or root:

```
# yum install chrony
```

Verify that chrony is running

To view the status of the chronyd daemon, run the following command:

```
$ systemctl status chronyd
```

If chrony is running, an output similar to the following appears:

```
chronyd.service - NTP client/server
   Loaded: loaded (/usr/lib/systemd/system/chronyd.service; enabled)
   Active: active (running) since Mon 2015-07-06 16:29:54 EDT; 15s ago
Main PID: 2530 (chronyd)
   CGroup: /system.slice/chronyd.service
           ââ2530 /usr/sbin/chronyd -u chrony
```

If chrony is not running, execute the following command as sudo or root. This command also causes chrony to run at boot time:

```
# systemctl enable chronyd
```

Verify that chrony is operating correctly

To verify that the chrony daemon is operating correctly, issue the following command on all nodes in the cluster:

```
$ chronyc tracking
```

An output similar to the following appears:

```
Reference ID    : 198.247.63.98 (time01.website.org)
Stratum         : 3
Ref time (UTC)  : Thu Jul  9 14:58:01 2015
System time     : 0.000035685 seconds slow of NTP time
Last offset     : -0.000151098 seconds
RMS offset      : 0.000279871 seconds
Frequency       : 2.085 ppm slow
Residual freq   : -0.013 ppm
Skew            : 0.185 ppm
Root delay      : 0.042370 seconds
Root dispersion : 0.022658 seconds
Update interval : 1031.0 seconds
Leap status     : Normal
```

A stratum level of 16 indicates that chrony is not synchronizing correctly. If chrony continues to detect a stratum level of 16, verify that the UDP port 323 is open. This port must be open on all firewalls between the cluster and the remote machine to which you are attempting to synchronize.

SELinux configuration

Vertica does not support SELinux except when SELinux is running in permissive mode. If it detects that SELinux is installed and the mode cannot be determined the installer reports this issue with the identifier: **S0080** . If the mode can be determined, and the mode is not permissive, then the issue is reported with the identifier: **S0081** .

Red hat and SUSE systems

You can either disable SELinux or change it to use permissive mode.

To disable SELinux:

1. Edit /etc/selinux/config and change setting for SELinux to disabled (SELINUX=disabled). This disables SELinux at boot time.
2. As root/sudo, type setenforce 0 to disable SELinux immediately.

To change SELinux to use permissive mode:

1. Edit /etc/selinux/config and change setting for SELINUX to permissive (SELINUX=Permissive).
2. As root/sudo, type setenforce Permissive to switch to permissive mode immediately.

Ubuntu and debian systems

You can either disable SELinux or change it to use permissive mode.

To disable SELinux:

1. Edit /selinux/config and change setting for SELinux to disabled (SELINUX=disabled). This disables SELinux at boot time.
2. As root/sudo, type setenforce 0 to disable SELinux immediately.

To change SELinux to use permissive mode:

1. Edit /selinux/config and change setting for SELinux to permissive (SELINUX=Permissive).
2. As root/sudo, type setenforce Permissive to switch to permissive mode immediately.

CPU frequency scaling

This topic details the various CPU frequency scaling methods supported by Vertica. In general, if you do not require CPU frequency scaling, then disable it so as not to impact system performance.

Important

Your systems may use significantly more energy when frequency scaling is disabled.

The installer allows CPU frequency scaling to be enabled when the cpufreq scaling governor is set to performance . If the cpu scaling governor is set to *ondemand* , and ignore_nice_load is 1 (true), then the installer **fails** with the error **S0140** . If the cpu scaling governor is set to *ondemand* and ignore_nice_load is 0 (false), then the installer **warns** with the identifier **S0141** .

CPU frequency scaling is a hardware and software feature that helps computers conserve energy by slowing the processor when the system load is low, and speeding it up again when the system load increases. This feature can impact system performance, since raising the CPU frequency in response to higher system load does not occur instantly. Always disable this feature on the Vertica database hosts to prevent it from interfering with performance.

You disable CPU scaling in your host's system BIOS. There may be multiple settings in your host's BIOS that you need to adjust in order to completely disable CPU frequency scaling. Consult your host hardware's documentation for details on entering the system BIOS and disabling CPU frequency scaling.

If you cannot disable CPU scaling through the system BIOS, you can limit the impact of CPU scaling by disabling the scaling through the Linux kernel or setting the CPU frequency governor to always run the CPU at full speed.

> **Caution**
> This method is not reliable, as some hardware platforms may ignore the kernel settings. For more information, see Vertica Hardware Guide .

The method you use to disable frequency depends on the CPU scaling method being used in the Linux kernel. See your Linux distribution's documentation for instructions on disabling scaling in the kernel or changing the CPU governor.

Enabling or disabling defrag

You can modify the defrag utility to meet Vertica configuration requirements, or to optimize your system performance by workload.

On all Red Hat/CentOS systems, you must disable the defrag utility to meet Vertica configuration requirements.

> **Note**
> The steps to disable defrag on Red Hat 6/CentOS 6 systems differ from those used to disable defrag on Red Hat 7/CentOS 7 and Red Hat 8/CentOS 8.

For SUSE 15.1, Vertica recommends that you enable defrag for optimized performance.

Recommended settings by workload for red hat 8/CentOS 8 and SUSE 15.1

Vertica recommends defrag settings to optimize performance by workload. The following table contains recommendations for systems that primarily run concurrent queries (such as short-running dashboard queries), or sequential SELECT or load (COPY) queries:

Operating System	Concurrent	Sequential
Red Hat 8.0/CentOS 8.0	Disable	Disable
SUSE 15.1	Enable	Enable

See Enabling or disabling transparent hugepages for additional settings that optimize your system performance by workload.

Disabling defrag on red hat 6/CentOS 6 systems

1. Determine if defrag is enabled by running the following command:

```
cat /sys/kernel/mm/redhat_transparent_hugepage/defrag
[always] madvise never
```

   The setting returned in brackets is your current setting. If you are not using madvise or never as your defrag setting, then you must disable defrag.
2. Edit /etc/rc.local, and add the following script:

```
if test -f /sys/kernel/mm/redhat_transparent_hugepage/enabled; then
   echo never > /sys/kernel/mm/redhat_transparent_hugepage/defrag
fi
```

   You must reboot your system for the setting to take effect, or run the following echo line to proceed with the install without rebooting:

```
# echo never > /sys/kernel/mm/redhat_transparent_hugepage/defrag
```

Disabling defrag on red hat 7/CentOS 7, red hat 8/CentOS 8, and SUSE 15.1

1. Determine if defrag is enabled by running the following command:

```
cat /sys/kernel/mm/transparent_hugepage/defrag
[always] madvise never
```

   The setting returned in brackets is your current setting. If you are not using madvise or never as your defrag setting, then you must disable defrag.
2. Edit /etc/rc.local, and add the following script:

```
if test -f /sys/kernel/mm/transparent_hugepage/enabled; then
   echo never > /sys/kernel/mm/transparent_hugepage/defrag
fi
```

   You must reboot your system for the setting to take effect, or run the following echo line to proceed with the install without rebooting:

```
# echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

3. If you are using Red Hat 7.0/CentOS 7.0 or Red Hat 8.0/CentOS 8.0, run the following command as root or sudo:

```
$ chmod +x /etc/rc.d/rc.local
```

Enabling defrag on red hat 7/8, CentOS 7/8, and SUSE 15.1

1. Determine if defrag is enabled by running the following command:

```
cat /sys/kernel/mm/transparent_hugepage/defrag
[never] madvise never
```

   The setting returned in brackets is your current setting. If you are not using madvise or always as your defrag setting, then you must enable defrag.
2. Edit /etc/rc.local, and add the following script:

```
if test -f /sys/kernel/mm/transparent_hugepage/enabled; then
   echo always > /sys/kernel/mm/transparent_hugepage/defrag
fi
```

   You must reboot your system for the setting to take effect, or run the following echo line to proceed with the install without rebooting:

```
# echo always > /sys/kernel/mm/transparent_hugepage/defrag
```

3. If you are using Red Hat 7.0/CentOS 7.0 or Red Hat 8.0/CentOS 8.0, run the following command as root or sudo:

```
$ chmod +x /etc/rc.d/rc.local
```

Support tools

Vertica suggests that the following tools are installed so support can assist in troubleshooting your system if any issues arise:

- pstack (or gstack) package. Identified by issue **S0040** when not installed.
  - On Red Hat 7 and CentOS 7 systems, the pstack package is installed as part of the gdb package.
- mcelog package. Identified by issue **S0041** when not installed.

- sysstat package. Identified by issue **S0045** when not installed.

Red hat 6 and CentOS 6 systems

To install the required tools on Red Hat 6 and CentOS 6 systems, run the following commands as sudo or root:

```
# yum install pstack
# yum install mcelog
# yum install sysstat
```

Red hat 7 and CentOS 7 systems

To install the required tools on Red Hat 7/CentOS 7 systems, run the following commands as sudo or root:

```
# yum install gdb
# yum install mcelog
# yum install sysstat
```

Ubuntu and debian systems

To install the required tools on Ubuntu and Debian systems, run the following commands as sudo or root:

```
$ apt-get install pstack
$ apt-get install mcelog
$ apt-get install sysstat
```

Important

For Ubuntu versions 18.04 and higher, run apt-get install rasdaemon instead of apt-get install mcelog .

SuSE systems

To install the required tools on SuSE systems, run the following commands as sudo or root.

```
# zypper install sysstat
# zypper install mcelog
```

There is no individual SuSE package for pstack/gstack. However, the gdb package contains gstack, so you could optionally install gdb instead, or build pstack/gstack from source. To install the gdb package:

```
# zypper install gdb
```

System user configuration

The following tasks pertain to the configuration of the system user required by Vertica.

In this section

- System user requirements
- TZ environment variable
- LANG environment variable settings
- Package dependencies

System user requirements

Vertica has specific requirements for the system user that runs and manages Vertica. If you specify a user during install, but the user does not exist, then the installer reports this issue with the identifier: **S0200** .

System user requirement details

Vertica requires a system user to own database files and run database processes and administration scripts. By default, the install script automatically configures and creates this user for you with the username *dbadmin* . See Linux users created by Vertica for details on the default user created by the install script. If you decide to manually create your own system user, then you must create the user **before** you run the install script. If you manually create the user:

> **Note**
> Instances of dbadmin and verticadba are placeholders for the names you choose if you do not use the default values.

- the user must have the same username and password on all nodes

- the user must use the BASH shell as the user's default shell. If not, then the installer reports this issue with identifier **[S0240]** .
- the user must be in the *verticadba* group (for example: usermod -a -G verticadba *userNameHere*). If not, the installer reports this issue with identifier **[S0220]** .

> **Note**
> You must create a verticadba group on all nodes. If you do not, then the installer reports the issue with identifier **[S0210]** .

- the user's login group must be either *verticadba* or a group with the same name as the user (for example, the home group for dbadmin is dbadmin). You can check the groups for a user with the id command. For example: id dbadmin . The "gid" group is the user's primary group. If this is not configured correctly then the installer reports this issue with the identifier **[S0230]** . Vertica recommends that you use *verticadba* as the user's primary login group. For example: usermod -g verticadba *userNameHere* . If the user's primary group is not *verticadba* as suggested, then the installer reports this with HINT **[S0231]** .
- the user must have a home directory. If not, then the installer reports this issue with identifier **[S0260]** .
- the user's home directory must be owned by the user. If not, then the installer reports the issue with identifier **[S0270]** .
- the system must be aware of the user's home directory (you can set it with the usermod command: usermod -m -d /path/to/new/home/dir *userNameHere*). If this is not configured correctly then the installer reports the issue with **[S0250]** .
- the user's home directory must be owned by the dbadmin's primary group (use the chown and chgrp commands if necessary). If this is not configured correctly, then the installer reports the issue with identifier **[S0280]** .
- the user's home directory *should* have secure permissions. Specifically, it should not be writable by anyone or by the group. Ideally the permissions should be, when viewing with *ls* , " --- " (nothing), or " r-x " (read and execute). If this is not configured as suggested then the installer reports this with HINT **[S0290]** .

TZ environment variable

This topic details how to set or change the TZ environment variable and update your tzdata package. If this variable is not set, then the installer reports this issue with the identifier: **S0305** .

Before installing Vertica, update the tzdata package for your system and set the default time zone for your database administrator account by specifying the TZ environmental variable. If your database administrator is being created by the install_vertica script, then set the TZ variable after you have installed Vertica.

Update tzdata package

The tzdata package is a public-domain time zone database that is pre-installed on most Linux systems. The tzdata package is updated periodically for time-zone changes across the world. You should update to the latest tzdata package before installing or updating Vertica.

Update your tzdata package with the following command:

- RedHat based systems: yum update tzdata
- Debian and Ubuntu systems: apt-get install tzdata

Setting the default time zone

When a client receives the result set of a SQL query, all rows contain data adjusted, if necessary, to the same time zone. That time zone is the default time zone of the initiator node unless the client explicitly overrides it using the SQL SET TIME ZONE command described in the SQL Reference Manual. The default time zone of any node is controlled by the TZ environment variable. If TZ is undefined, the operating system time zone.

Important
The TZ variable must be set to the same value on all nodes in the cluster.

If your operating system timezone is not set to the desired timezone of the database then make sure that the Linux environment variable TZ is set to the desired value on all cluster hosts.

The installer returns a warning if the TZ variable is not set. If your operating system timezone is appropriate for your database, then the operating system timezone is used and the warning can be safely ignored.

Setting the time zone on a host

Important
If you explicitly set the TZ environment variable at a command line before you start the Administration tools , the current setting will not take effect. The Administration Tools uses SSH to start copies on the other nodes, so each time SSH is used, the TZ variable for the startup command is reset. TZ must be set in the .profile or .bashrc files on all nodes in the cluster to take affect properly.

You can set the time zone several different ways, depending on the Linux distribution or the system administrator's preferences.

- To set the system time zone on Red Hat and SUSE Linux systems, edit:
  ```
  /etc/sysconfig/clock
  ```
- To set the TZ variable, edit, /etc/profile , or /home/dbadmin/.bashrc or /home/dbadmin/.bash_profile and add the following line (for example, for the US Eastern Time Zone):
  ```
  export TZ="America/New_York"
  ```
  For details on which timezone names are recognized by Vertica, see the appendix: Using time zones with Vertica .

LANG environment variable settings

This topic details how to set or change the LANG environment variable. The LANG environment variable controls the locale of the host. If this variable is not set, then the installer reports this issue with the identifier: **S0300** . If this variable is not set to a valid value, then the installer reports this issue with the identifier: **S0301** .

Set the host locale

Each host has a system setting for the Linux environment variable LANG . LANG determines the locale category for native language, local customs, and coded character set in the absence of the LC_ALL and other LC_ environment variables. LANG can be used by applications to determine which language to use for error messages and instructions, collating sequences, date formats, and so forth.

To change the LANG setting for the database administrator, edit, /etc/profile , or /dbadmin/.bashrc or /home/dbadmin/.bash_profile on all cluster hosts and set the environment variable; for example:

```
export LANG=en_US.UTF-8
```

The LANG setting controls the following in Vertica:

- OS-level errors and warnings, for example, "file not found" during COPY operations.
- Some formatting functions, such as TO_CHAR and TO_NUMBER . See also Template patterns for numeric formatting .

The LANG setting does not control the following:

- Vertica specific error and warning messages. These are always in English at this time.
- Collation of results returned by SQL issued to Vertica. This must be done using a database parameter instead. See Implement locales for international data sets section for details.

> **Note**
> If the LC_ALL environment variable is set, it supersedes the setting of LANG .

Package dependencies

For successful Vertica installation, you must first install three packages on all nodes in your cluster before installing the database platform.

The required packages are:

- openssh—Required for Administration tools connectivity between nodes.
- which—Required for Vertica operating system integration and for validating installations.
- dialog—Required for interactivity with Administration Tools.

Installing the required packages

The procedure you follow to install the required packages depends on the operating system on which your node or cluster is running. See your operating system's documentation for detailed information on installing packages.

- **For CentOS/Red Hat Systems** —Typically, you manage packages on Red Hat and CentOS systems using the yum utility.
  Run the following yum commands to install each of the package dependencies. The yum utility guides you through the installation:
  ```
  # yum install openssh
  # yum install which
  # yum install dialog
  ```
- **For Debian/Ubuntu Systems** —Typically, you use the apt-get utility to manage packages on Debian and Ubuntu systems.

Run the following apt-get commands to install each of the package dependencies. The apt-get utility guides you through the installation:

```
# apt-get install openssh
# apt-get install which
# apt-get install dialog
```

Install Vertica using the command line

This section describes how to install the Vertica software on a cluster of nodes. It assumes that you have already performed the tasks in Before You Install Vertica , and that you have a Vertica license key.

To install Vertica, complete the following tasks:

1. Download and install the Vertica server package
2. Install Vertica with the installation script

Special notes

- Downgrade installations are not supported.
- Be sure that you download the RPM for the correct operating system and architecture.
- Vertica supports two-node clusters with zero fault tolerance (K=0 safety). This means that you can add a node to a single-node cluster, as long as the installation node (the node upon which you build) is not the loopback node (localhost/127.0.0.1).
- The installer performs platform verification tests that prevent the install from continuing if the platform requirements are not met. These tests ensure that your platform meets the hardware and software requirements for Vertica. You can simply run the installer and view a list of the failures and warnings to determine which configuration changes you must make.

In this section

- Download and install the Vertica server package
- Linux users created by Vertica
- Validation scripts
- Install Vertica with the installation script
- Install Vertica silently
- Enable secure shell (SSH) logins

Download and install the Vertica server package

To download and install the Vertica server package:

1. Use a Web browser to go to the Vertica website .
2. Click the **Support** tab and select **Customer Downloads** .
3. Log into the portal to download the install package.
   Be sure the package you download matches the operating system and the machine architecture on which you intend to install it.
4. Transfer the installation package to the Administration host .
5. If you installed a previous version of Vertica on any of the hosts in the cluster, use the Administration tools to shut down any running database.
   The database must stop normally; you cannot upgrade a database that requires recovery.
6. If you are using sudo, skip to the next step. If you are root, log in to the Administration Host as root (or log in as another user and switch to root).

   ```
   $ su - root
   password: root-password
   #
   ```

   Caution
   When installing Vertica using an existing user as the dba, you must exit all UNIX terminal sessions for that user after setup completes and log in again to ensure that group privileges are applied correctly.

   After Vertica is installed, you no longer need root privileges. To verify sudo, see Platform and hardware requirements and recommendations .
7. Use one of the following commands to run the RPM package installer:
   - If you are root and installing an RPM:

     ```
     # rpm -Uvh pathname
     ```

     Note

When installing a Vertica RPM, you might see an unexpected warning about a SHA256 signature. This warning indicates that you need to import a GPG key. Only necessary for versions after 10.0, keys can be downloaded under the Security section of your chosen release from the Vertica Client Drivers page . After downloading the key, you can import it with the following command:

rpm --import RPM-GPG-KEY-VERTICA

- If you are using sudo and installing an RPM:

```
$ sudo rpm -Uvh pathname
```

- If you are using Debian:

```
$ sudo dpkg -i pathname
```

where *pathname* is the Vertica package file you downloaded.

Note
If the package installer reports multiple dependency problems, or you receive the error *"ERROR: You're attempting to install the wrong RPM for this operating system"* , then you are trying to install the wrong Vertica server package.

After you install the Vertica RPM, you can use several Validation scripts to help determine if your hosts and network can properly handle the processing and network traffic required by Vertica.

Linux users created by Vertica

This topic describes the Linux accounts that the installer creates and configures so Vertica can run. When you install Vertica, the installation script optionally creates the following Linux user and group:

- dbadmin—Administrative user
- verticadba—Group for DBA users

dbadmin and verticadba are the default names. If you want to change what these Linux accounts are called, you can do so using the installation script. See Install Vertica with the installation script for details.

Dbadmin privileges

The Linux dbadmin user owns the database catalog and data storage on disk. When you run the install script, Vertica creates this user on each node in the database cluster. It also adds dbadmin to the Linux dbadmin and verticadba groups, and configures the account as follows:

- Configures and authorizes dbadmin for passwordless SSH between all cluster nodes. SSH must be installed and configured to allow passwordless logins. See Enable secure shell (SSH) logins .
- Sets the dbadmin user's BASH shell to /bin/bash , required to run scripts, such as install_vertica and the Administration tools .
- Provides read-write-execute permissions on the following directories:
  - /opt/vertica/*
  - /home/dbadmin —the default directory for database data and catalog files (configurable through the install script)

Note
The Vertica installation script also creates a Vertica database superuser named dbadmin. They share the same name, but they are not the same; one is a Linux user and the other is a Vertica user. See Database administration user for information about the database superuser.

After you install Vertica

Root or sudo privileges are not required to start or run Vertica after the installation process completes.

The dbadmin user can log in and perform Vertica tasks, such as creating a database, installing/changing the license key, or installing drivers. If dbadmin wants database directories in a location that differs from the default, the root user (or a user with sudo privileges) must create the requested directories and change ownership to the dbadmin user.

Vertica prevents administration from users other than the dbadmin user (or the user name you specified during the installation process if not dbadmin). Only this user can run Administration Tools.

See also

- Installation overview and checklist
- Before you install Vertica
- Platform and hardware requirements and recommendations
- Enable secure shell (SSH) logins

Validation scripts

Vertica provides several validation utilities that can be used prior to deploying Vertica to help determine if your hosts and network can properly handle the processing and network traffic required by Vertica. These utilities can also be used if you are encountering performance issues and need to troubleshoot the issue.

After you install the Vertica RPM, you have access to the following scripts in /opt/vertica/bin :

- Vcpuperf - a CPU performance test used to verify your CPU performance.
- Vioperf - an Input/Output test used to verify the speed and consistency of your hard drives.
- Vnetperf - a Network test used to test the latency and throughput of your network between hosts.

These utilities can be run at any time, but are well suited to use before running the install_vertica script.

In this section

- Vcpuperf
- Vioperf
- Vnetperf

Vcpuperf

The vcpuperf utility measures your server's CPU processing speed and compares it against benchmarks for common server CPUs. The utility performs a CPU test and measures the time it takes to complete the test. The lower the number scored on the test, the better the performance of the CPU.

The vcpuperf utility also checks the high and low load times to determine if CPU throttling is enabled. If a server's low-load computation time is significantly longer than the high-load computation time, CPU throttling may be enabled. CPU throttling is a power-saving feature. However, CPU throttling can reduce the performance of your server. Vertica recommends disabling CPU throttling to enhance server performance.

Syntax

```
vcpuperf [-q]
```

Option

-q

Run in quiet mode. Quiet mode displays only the CPU Time, Real Time, and high and low load times.

Returns

- CPU Time: the amount of time it took the CPU to run the test.
- Real Time: the total time for the test to execute.
- High load time: The amount of time to run the load test while simulating a high CPU load.
- Low load time: The amount of time to run the load test while simulating a low CPU load.

Example

The following example shows a CPU that is running slightly slower than the expected time on a Xeon 5670 CPU that has CPU throttling enabled.

```
[root@node1 bin]# /opt/vertica/bin/vcpuperf
Compiled with: 4.1.2 20080704 (Red Hat 4.1.2-52) Expected time on Core 2, 2.53GHz: ~9.5s
Expected time on Nehalem, 2.67GHz: ~9.0s
Expected time on Xeon 5670, 2.93GHz: ~8.0s

This machine's time:
  CPU Time: 8.540000s
  Real Time:8.710000s

Some machines automatically throttle the CPU to save power.
  This test can be done in <100 microseconds (60-70 on Xeon 5670, 2.93GHz).
  Low load times much larger than 100-200us or much larger than the corresponding high load time
    indicate low-load throttling, which can adversely affect small query / concurrent performance.

This machine's high load time: 67 microseconds.
This machine's low load time: 208 microseconds.
```

Vioperf

The vioperf utility quickly tests the performance of your host's input and output subsystem. The utility performs the following tests:

- sequential write
- sequential rewrite
- sequential read
- skip read (read non-contiguous data blocks)

The utility verifies that the host reads the same bytes that it wrote and prints its output to STDOUT. The utility also logs the output to a JSON formatted file.

For data in HDFS, the utility tests reads but not writes.

Syntax

```
vioperf [--help] [--duration=<INTERVAL>] [--log-interval=<INTERVAL>]
  [--log-file=<FILE>] [--condense-log] [--thread-count=<N>] [--max-buffer-size=<SIZE>]
  [--preserve-files] [--disable-crc] [--disable-direct-io] [--debug]
  [<DIR>*]
```

Minimum and recommended I/O performance

- The minimum required I/O is 20 MB/s read/write per physical processor core on each node, in full duplex (reading and writing) simultaneously, concurrently on all nodes of the cluster.

  > **Note**
  > Vertica supports some AWS instance types that do not meet these minimum I/O requirements. However, all supported AWS instances types, regardless of vioperf performance, can be used as Vertica cluster hosts. See Supported AWS instance types for a list of all supported AWS instance types.

- The recommended I/O is 40 MB/s per physical core on each node.
- The minimum required I/O rate for a node with 2 hyper-threaded six-core CPUs (12 physical cores) is 240 MB/s. Vertica recommends 480 MB/s.

For example, the I/O rate for a node with 2 hyper-threaded six-core CPUs (12 physical cores) is 240 MB/s required minimum, 480 MB/s recommended.

Disk space vioperf needs

vioperf requires about 4.5 GB to run.

Options

--help

Prints a help message and exits.

--duration

The length of time vioprobe runs performance tests. The default is 5 minutes. Specify the interval in seconds, minutes, or hours with any of these suffixes:

- Seconds: s , sec , secs , second , seconds . Example: --duration=60sec
- Minutes: m , min , mins , minute , minutes . Example: --duration=10min
- Hours: h , hr , hrs , hour , hours . Example: --duration=1hrs

--log-interval

The interval at which the log file reports summary information. The default interval is 10 seconds. This option uses the same interval notation as --duration .

--log-file

The path and name where log file contents are written, in JSON. If not specified, then vioperf creates a file named results *date-time* .JSON in the current directory.

--condense-log

Directs vioperf to write the log file contents in condensed format, one JSON entry per line, rather than as indented JSON syntax.

--thread-count=<N>

The number of execution threads to use. By default, vioperf uses all threads available on the host machine.

--max-buffer-size=<SIZE>

The maximum size of the in-memory buffer to use for reads or writes. Specify the units with any of these suffixes:

- Bytes: b , byte , bytes .
- Kilobytes: k , kb , kilobyte , kilobytes .
- Megabytes: m , mb , megabyte , megabytes .
- Gigabytes: g , gb , gigabyte , gigabytes .

--preserve-files

Directs vioperf to keep the files it writes. This parameter is ignored for HDFS tests, which are read-only. Inspecting the files can help diagnose write-related failures.

--disable-crc

Directs vioperf to ignore CRC checksums when validating writes. Verifying checksums can add overhead, particularly when running vioperf on slower processors. This parameter is ignored for HDFS tests.

--disable-direct-io

When reading from or writing to a local file system, vioperf goes directly to disk by default, bypassing the operating system's page cache. Using direct I/O allows vioperf to measure performance quickly without having to fill the cache.

Disabling this behavior can produce more realistic performance results but slows down the operation of vioperf .

--debug

Directs vioperf to report verbose error messages.

<DIR>

Zero or more directories to test. If you do not specify a directory, vioperf tests the current directory. To test the performance of each disk, specify different directories mounted on different disks.

To test reads from a directory on HDFS:

- Use a URL in the hdfs scheme that points to a single directory (not a path) containing files at least 10MB in size. For best results, use 10GB files and verify that there is at least one file per vioperf thread.
- If you do not specify a host and port, set the HADOOP_CONF_DIR environment variable to a path including the Hadoop configuration files. This value is the same value that you use for the HadoopConfDir configuration parameter in Vertica. For more information see Configuring HDFS access .
- If the HDFS cluster uses Kerberos, set the HADOOP_USER_NAME environment variable to a Kerberos principal.

Returns

The utility returns the following information:

test

The test being run (Write, ReWrite, Read, or Skip Read)

directory

The directory in which the test is being run.

counter name

The counter type of the test being run. Can be either MB/s or Seeks per second.

counter value

The value of the counter in MB/s or Seeks per second across all threads. This measurement represents the bandwidth at the exact time of measurement. Contrast with counter value (avg).

counter value (10 sec avg)

The average amount of data in MB/s, or the average number of Seeks per second, for the test being run in the duration specified with --log-interval . The default interval is 10 seconds. The counter value (avg) is the average bandwidth since the last log message, across all threads.

counter value/core

The counter value divided by the number of cores.

counter value/core (10 sec avg)

The counter value (10 sec avg) divided by the number of cores.

thread count

The number of threads used to run the test.

%CPU

The available CPU percentage used during this test.

%IO Wait

The CPU percentage in I/O Wait state during this test. I/O wait state is the time working processes are blocked while waiting for I/O operations to complete.

elapsed time

The amount of time taken for a particular test. If you run the test multiple times, elapsed time increases the next time the test is run.

remaining time

The time remaining until the next test. Based on the --duration option, each of the tests is run at least once. If the test set is run multiple times, then remaining time is how much longer the test will run. The remaining time value is cumulative. Its total is added to elapsed time each time the same test is run again.

Example

Invoking vioperf from a terminal outputs the following message and sample results:

```
[dbadmin@v_vmart_node0001 ~]$ /opt/vertica/bin/vioperf --duration=60s
The minimum required I/O is 20 MB/s read and write per physical processor core on each node, in full duplex
i.e. reading and writing at this rate simultaneously, concurrently on all nodes of the cluster.
The recommended I/O is 40 MB/s per physical core on each node.
For example, the I/O rate for a server node with 2 hyper-threaded six-core CPUs is 240 MB/s required minimum, 480 MB/s recommended.

Using direct io (buffer size=1048576, alignment=512) for directory "/home/dbadmin"

test     | directory     | counter name         | counter value | counter value (10 sec avg) | counter value/core  | counter value/core (10 sec avg) | thread count  |
%CPU  | %IO Wait  | elapsed time (s)| remaining time (s)
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
Write    | /home/dbadmin | MB/s                | 420          | 420                        | 210              | 210                    | 2            | 89    | 10        | 10            | 5
Write    | /home/dbadmin | MB/s                | 412          | 396                        | 206              | 198                    | 2            | 89    | 9         | 15            | 0
ReWrite  | /home/dbadmin | (MB-read+MB-write)/s | 150+150      | 150+150                    | 75+75            | 75+75                  | 2            | 58    | 40        |
10            | 5
ReWrite  | /home/dbadmin | (MB-read+MB-write)/s | 158+158      | 172+172                    | 79+79            | 86+86                  | 2            | 64    | 33        | 15
| 0
Read     | /home/dbadmin | MB/s                | 194          | 194                        | 97               | 97                     | 2            | 69    | 26        | 10            | 5
Read     | /home/dbadmin | MB/s                | 192          | 190                        | 96               | 95                     | 2            | 71    | 27        | 15            | 0
SkipRead | /home/dbadmin | seeks/s             | 659          | 659                        | 329.5            | 329.5                  | 2            | 2     | 85        | 10            | 5
SkipRead | /home/dbadmin | seeks/s             | 677          | 714                        | 338.5            | 357                    | 2            | 2     | 59        | 15            | 0
```

> **Note**
> When evaluating performance for minimum and recommended I/O, include the Write and Read values in your evaluation. ReWrite and SkipRead values are not relevant to determining minimum and recommended I/O.

Vnetperf

The vnetperf utility measures network performance of database hosts, as well as network latency and throughput for TCP and UDP protocols.

> **Caution**
> This utility incurs high network load, which degrades database performance. Do not use this utility on a Vertica production database.

This utility helps identify the following issues:

- Low throughput for all hosts or one
- High latency for all hosts or one
- Bottlenecks between one or more hosts or subnets
- Too-low limit on the number of TCP connections that can be established simultaneously
- High rates of network packet loss

Syntax

```
vnetperf [[options](#Options)] [[tests](#Tests)]
```

Options

--condense

Condenses the log into one JSON entry per line, instead of indented JSON syntax.

--collect-logs

Collects test log files from each host.

--datarate ***rate***

Limits throughput to this rate in MB/s. A rate of 0 loops the tests through several different rates.

Default: 0

--duration ***seconds***

Time limit for each test to run in seconds.

Default: 1

--hosts ***host-name*** **[,...]**

Comma-separated list of host names or IP addresses on which to run the tests. The list must not contain embedded spaces.

--hosts ***file***

File that specifies the hosts on which to run the tests. If you omit this option, then the vnetperf tries to access admintools to identify cluster hosts.

--identity-file ***file***

If using passwordless SSH/SCP access between hosts, then specify the key file used to gain access to the hosts.

--ignore-bad-hosts

If set, runs tests on reachable hosts even if some hosts are not reachable. If you omit this option and a host is unreachable, then no tests are run on any hosts.

--log-dir ***directory***

If --collect-logs is set, specifies the directory in which to place the collected logs.

Default: logs.netperf. *<timestamp>*

--log-level ***level***

Log level to use, one of the following:

- INFO
- ERROR
- DEBUG
- WARN

Default: WARN

--list-tests

Lists the tests that vnetperf can run.

--output-file ***file***

The file to which JSON results are written.

Default: results. *<timestamp>* .json

--ports port#[,...]

Comma-delimited list of port numbers to use. If only one port number is specified, then the next two numbers in sequence are also used.

Default: 14159,14160,14161

--scp-options ' *scp-args* '

Specifies one or more standard SCP command line arguments. SCP is used to copy test binaries over to the target hosts.

--ssh-options ' *ssh-args* '

Specifies one or more standard SSH command line arguments. SSH is used to issue test commands on the target hosts.

--tmp-dir *directory*

Specifies the temporary directory for vnetperf, where *directory* must have execute permission on all hosts, and does not include the unsupported characters " , ```, or ' .

Default: /tmp (execute permission required)

--vertica-install *directory*

Indicates that Vertica is installed on each of the hosts, so vnetperf uses test binaries on the target system rather than copying them over with SCP.

Tests

vnetperf can specify one or more of the following tests. If no test is specified, vnetperf runs all tests. Test results are printed for each host.

Test	Description	Results
latency	Measures latency from the host that is running the script to other hosts. Hosts with unusually high latency should be investigated further.	• Round trip time latency for each host in milliseconds. • Clock skew—the difference in time shown by the clock on the target host relative to the host running the utility.
tcp-throughput	Tests TCP throughput among hosts.	• Date/time and test name • Rrate limit in MB/s • Tested node • Sent and received data in MB/s and bytes • Duration of the test in seconds
udp-throughput	Tests UDP throughput among hosts	

Recommended network performance

- Maximum recommended RTT (round-trip time) latency is 1000 microseconds. Ideal RTT latency is 200 microseconds or less. Vertica recommends that clock skew be less than 1 second.
- Minimum recommended throughput is 100 MB/s. Ideal throughput is 800 MB/s or more.

> **Note**
> UDP throughput can be lower; multiple network switches can adversely affect performance.

Example

```
$ vnetperf latency tcp-throughput

The maximum recommended rtt latency is 2 milliseconds. The ideal rtt latency is 200 microseconds or less. It is recommended that clock skew be kept to
under 1 second.
test          | date                    | node           | index | rtt latency (us)  | clock skew (us)
-------------------------------------------------------------------------------------------------------------
latency       | 2022-03-29_10:23:55,739 | 10.20.100.247  | 0     | 49                | 3
latency       | 2022-03-29_10:23:55,739 | 10.20.100.248  | 1     | 272               | -702
latency       | 2022-03-29_10:23:55,739 | 10.20.100.249  | 2     | 245               | 1037

The minimum recommended throughput is 100 MB/s. Ideal throughput is 800 MB/s or more. Note: UDP numbers may be lower, multiple network switches
may reduce performance results.
date                    | test           | rate limit (MB/s) | node           | MB/s (sent) | MB/s (rec)  | bytes (sent)       | bytes (rec)        | duration (s)
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
2022-03-29_10:23:55,742 | tcp-throughput | 32                | 10.20.100.247  | 30.579      | 30.579      | 32112640           | 32112640           | 1.00151
2022-03-29_10:23:55,742 | tcp-throughput | 32                | 10.20.100.248  | 30.5791     | 30.5791     | 32112640           | 32112640           | 1.0015
2022-03-29_10:23:55,742 | tcp-throughput | 32                | 10.20.100.249  | 30.5791     | 30.5791     | 32112640           | 32112640           | 1.0015
2022-03-29_10:23:55,742 | tcp-throughput | 32                | average        | 30.579      | 30.579      | 32112640           | 32112640           | 1.0015
2022-03-29_10:23:57,749 | tcp-throughput | 64                | 10.20.100.247  | 61.0952     | 61.0952     | 64094208           | 64094208           | 1.00049
2022-03-29_10:23:57,749 | tcp-throughput | 64                | 10.20.100.248  | 61.096      | 61.096      | 64094208           | 64094208           | 1.00048
2022-03-29_10:23:57,749 | tcp-throughput | 64                | 10.20.100.249  | 61.0952     | 61.0952     | 64094208           | 64094208           | 1.00049
2022-03-29_10:23:57,749 | tcp-throughput | 64                | average        | 61.0955     | 61.0955     | 64094208           | 64094208           | 1.00048
2022-03-29_10:23:59,753 | tcp-throughput | 128               | 10.20.100.247  | 122.131     | 122.131     | 128122880          | 128122880          | 1.00046
2022-03-29_10:23:59,753 | tcp-throughput | 128               | 10.20.100.248  | 122.132     | 122.132     | 128122880          | 128122880          | 1.00046
2022-03-29_10:23:59,753 | tcp-throughput | 128               | 10.20.100.249  | 122.132     | 122.132     | 128122880          | 128122880          | 1.00046
2022-03-29_10:23:59,753 | tcp-throughput | 128               | average        | 122.132     | 122.132     | 128122880          | 128122880          | 1.00046
2022-03-29_10:24:01,757 | tcp-throughput | 256               | 10.20.100.247  | 243.819     | 244.132     | 255754240          | 256081920          | 1.00036
2022-03-29_10:24:01,757 | tcp-throughput | 256               | 10.20.100.248  | 244.125     | 243.282     | 256049152          | 255164416          | 1.00025
2022-03-29_10:24:01,757 | tcp-throughput | 256               | 10.20.100.249  | 244.172     | 243.391     | 256114688          | 255295488          | 1.00032
2022-03-29_10:24:01,757 | tcp-throughput | 256               | average        | 244.039     | 243.601     | 255972693          | 255513941          | 1.00031
2022-03-29_10:24:03,761 | tcp-throughput | 512               | 10.20.100.247  | 337.232     | 485.247     | 355893248          | 512098304          | 1.00645
2022-03-29_10:24:03,761 | tcp-throughput | 512               | 10.20.100.248  | 446.16      | 231.001     | 467894272          | 242253824          | 1.00013
2022-03-29_10:24:03,761 | tcp-throughput | 512               | 10.20.100.249  | 349.667     | 409.961     | 368476160          | 432013312          | 1.00497
2022-03-29_10:24:03,761 | tcp-throughput | 512               | average        | 377.686     | 375.403     | 397421226          | 395455146          | 1.00385
2022-03-29_10:24:05,772 | tcp-throughput | 640               | 10.20.100.247  | 328.279     | 509.256     | 383975424          | 595656704          | 1.11548
2022-03-29_10:24:05,772 | tcp-throughput | 640               | 10.20.100.248  | 505.626     | 217.217     | 532250624          | 228655104          | 1.00389
2022-03-29_10:24:05,772 | tcp-throughput | 640               | 10.20.100.249  | 390.355     | 474.89      | 410812416          | 499777536          | 1.00365
2022-03-29_10:24:05,772 | tcp-throughput | 640               | average        | 408.087     | 400.454     | 442346154          | 441363114          | 1.04101
2022-03-29_10:24:07,892 | tcp-throughput | 768               | 10.20.100.247  | 300.5       | 426.762     | 318734336          | 452657152          | 1.01154
2022-03-29_10:24:07,892 | tcp-throughput | 768               | 10.20.100.248  | 268.252     | 402.891     | 283017216          | 425066496          | 1.00616
2022-03-29_10:24:07,892 | tcp-throughput | 768               | 10.20.100.249  | 510.569     | 243.649     | 535592960          | 255590400          | 1.00042
2022-03-29_10:24:07,892 | tcp-throughput | 768               | average        | 359.774     | 357.767     | 379114837          | 377771349          | 1.00604
2022-03-29_10:24:09,911 | tcp-throughput | 1024              | 10.20.100.247  | 304.545     | 444.261     | 334987264          | 488669184          | 1.049
2022-03-29_10:24:09,911 | tcp-throughput | 1024              | 10.20.100.248  | 422.246     | 192.773     | 474284032          | 216530944          | 1.07121
2022-03-29_10:24:09,911 | tcp-throughput | 1024              | 10.20.100.249  | 353.206     | 446.809     | 378732544          | 479100928          | 1.0226
2022-03-29_10:24:09,911 | tcp-throughput | 1024              | average        | 359.999     | 361.281     | 396001280          | 394767018          | 1.0476
2022-03-29_10:24:11,988 | tcp-throughput | 2048              | 10.20.100.247  | 343.324     | 414.559     | 387710976          | 468156416          | 1.07697
2022-03-29_10:24:11,988 | tcp-throughput | 2048              | 10.20.100.248  | 292.44      | 246.254     | 308314112          | 259620864          | 1.00544
2022-03-29_10:24:11,988 | tcp-throughput | 2048              | 10.20.100.249  | 437.559     | 405.02      | 459145216          | 425000960          | 1.00072
2022-03-29_10:24:11,988 | tcp-throughput | 2048              | average        | 357.774     | 355.278     | 385056768          | 384259413          | 1.02771

JSON results available at: ./results.2022-03-29_10:23:51,548.json
```

Install Vertica with the installation script

You can run the installation script after you install the Vertica package. The installation script runs on a single node, using a Bash shell. The script copies the Vertica package to all other hosts (identified by the --hosts argument) in your planned cluster.

The installation script runs several tests on each of the target hosts to verify that the hosts meet system and performance requirements for a Vertica node. The installation script modifies some operating system configuration settings to meet these requirements. Other settings cannot be modified by the installation script and must be manually reconfigured. For details on operating system configuration settings, see Manually configured

operating system settings and Automatically configured operating system settings .

> **Note**
> The installation script sets up passwordless ssh for the admin user across all hosts. If passwordless ssh is already set up, the installation script verifies that it functions correctly.

In this section

- Install on a FIPS 140-2 enabled machine
- Specifying disk storage location during installation
- Perform a basic install
- install_vertica options

Install on a FIPS 140-2 enabled machine

Vertica supports the implementation of the Federal Information Processing Standard 140-2 (FIPS). You enable FIPS mode in the operating system.

> **Note**
> Enabling FIPS on the operating system occurs outside of Vertica.

During installation, the install_vertica script detects whether the host is operating in FIPS mode. The installer searches for the file /proc/sys/crypto/fips_enabled and examines its content. If the file exists and contains a '1' in the filename, the host is operating in FIPS mode and the following message appears:

```
/proc/sys/crypto/fips_enabled exists and contains '1', this is a FIPS system
```

Important
On certain systems where the libssl and libcrypto libraries do not have versioning information, when starting Vertica, you may see the message

```
No version information available
```

This message is benign and you can ignore it.

To implement FIPS 140-2 on your Vertica Analytic Database, you need to configure both the server and the client you are using. To see the detailed configuration steps, go to Implementing FIPS 140-2 .

Symbolic links for OpenSSL

On some non-FIPS systems, versioning anomalies can occur when you install a new version of OpenSSL. Sometimes, the default OpenSSL build procedure produces libraries with versions named 1.0.0. For Vertica to recognize that a library has a higher version number, the library name with a higher version number must be provided. As part of the Vertica installation, symbolic links are created to the appropriate OpenSSL files. The steps are as follows:

1. The RPM installer places two OpenSSL library files in /opt/vertica/lib:
   - libssl.so.1.1
   - libcrypto.so.1.1
2. The install_vertica script creates two symbolic links in /opt/vertica/lib:
   - libssl.so
   - libcrypto.so
3. The symbolic links point to libssl.so.1.1 and libcrypto.so.1.1, which the RPM installer placed in /opt/vertica/lib.

Specifying disk storage location during installation

You can specify the disk storage location when you:

- Install Vertica (see below).
- Create a database using the Administration Tools .
- Installing Management Console

Specifying disk storage location when you install

When you install Vertica, the --data-dir parameter in the install_vertica script lets you specify a directory to contain database data and catalog files. The script defaults to the database administrator's default home directory /home/dbadmin .

Important
Replace *this default* with a directory that has adequate space to hold your data and catalog files.

Requirements

- The data and catalog directory must exist on each node in the cluster.
- The directory on each node must be owned by the database administrator
- Catalog and data path names must contain only alphanumeric characters and cannot have leading space characters. Failure to comply with these restrictions will result in database creation failure.
- Vertica refuses to overwrite a directory if it appears to be in use by another database. Therefore, if you created a database for evaluation purposes, dropped the database, and want to reuse the database name, make sure that the disk storage location previously used has been completely cleaned up. See Managing storage locations for details.

Perform a basic install

For all installation options, see install_vertica options .

1. As root (or sudo) run the install script. The script must be run by a BASH shell as root or as a user with sudo privileges. You can configure many options when running the install script. See Basic Installation Parameters for the required options.
   If the installer fails due to any requirements not being met, you can correct the issue and then rerun the installer with the same command line options.
   To perform a basic installation:
   - As root:
     ```
     # /opt/vertica/sbin/install_vertica --hosts host_list --rpm package_name\
         --dba-user dba_username --parallel-no-prompts
     ```
   - Using sudo:
     ```
     $ sudo /opt/vertica/sbin/install_vertica --hosts host_list --rpm package_name \
         --dba-user dba_username --parallel-no-prompts
     ```

   Important
   If you place install_vertica in a location other than /opt/vertica , create a symlink from that location to /opt/vertica . Create this symlink on all cluster nodes, otherwise the database will not start.

2. When prompted for a password to log into the other nodes, provide the requested password. Doing so allows the installation of the package and system configuration on the other cluster nodes.
   - If you are root, this is the root password.
   - If you are using sudo, this is the sudo user password.

   The password does not echo on the command line. For example:
   ```
   Vertica Database 12.0.x Installation Tool
   Please enter password for root@host01:password
   ```

3. If the dbadmin user, or the user specified in the argument --dba-user , does not exist, then the install script prompts for the password for the user. Provide the password. For example:
   ```
   Enter password for new UNIX user dbadmin:password
   Retype new UNIX password for user dbadmin:password
   ```

4. Carefully examine any warnings or failures returned by install_vertica and correct the problems.
   For example, insufficient RAM, insufficient network throughput, and too high readahead settings on the file system could cause performance problems later on. Additionally, LANG warnings, if not resolved, can cause database startup to fail and issues with VSQL. The system LANG attributes must be UTF-8 compatible. After you fix the problems, rerun the install script.
5. When installation is successful, disconnect from the Administration host , as instructed by the script. Then, complete the required post-installation steps.
   At this point, root privileges are no longer needed and the database administrator can perform any remaining steps.

install_vertica options

The following tables describe install_vertica script options. Most options have long and short forms—for example, --hosts and -s .

Required

install_vertica requires the following options:

- --hosts / -s
- --rpm / -r | --deb
- --dba-user *username* | -u *username*
  Required only if installing using root or upgrading versions.

For example:

```
# /opt/vertica/sbin/install_vertica --hosts node0001,node0002,node0003 --rpm /tmp/vertica-10.1.1-0.x86_64.RHEL6.rpm
```

--hosts *hostlist* -s *hostlist*

Comma-separated list of host names or IP addresses to include in the cluster. The list must not include embedded spaces. For example:

- --hosts node01,node02,node03
- --hosts 192.168.233.101,192.168.233.102,192.168.233.103
- --hosts fd95:ff5d:5549:bdb0::1,fd95:ff5d:5549:bdb0::2,fd95:ff5d:5549:bdb0::3

The following requirements apply:

- If upgrading an existing installation of Vertica, use the same host names used previously.
- IP addresses or hostnames must be for unique hosts. Do not list the same host using multiple IP addresses/hostnames.

> **Note**
> Vertica stores only IP addresses in its configuration files. If you provide host names, they are converted to IP addresses when the script runs.

--rpm *package-name* -r *package-name* --deb *package-name*

Path and name of the Vertica RPM or Debian package. For example:

```
--rpm /tmp/vertica-10.1.1-0.x86_64.RHEL6.rpm
```

For Debian and Ubuntu installs, provide the name of the Debian package:

```
--deb /tmp/vertica_10.1_amd64.deb
```

The install package must be provided if installing or upgrading multiple nodes and the nodes do not have the latest server package installed, or if you are adding a new node. The install_vertica and update_vertica scripts serially copy the server package to the other nodes and install the package.

> **Tip**
> If installing or upgrading a large number of nodes, consider manually installing the package on all nodes before running the upgrade script. The script runs faster if it does not need to serially upload and install the package on each node.

--dba-user *username* -u *username*

Name of the database superuser account to create. Only this account can run the Administration Tools. If you omit this parameter, then the default administrator account name is dbadmin .

This parameter is optional for new installations done as root; they must be specified when upgrading or when installing using sudo. If upgrading, use this parameter to specify the same account name that you used previously. If installing using sudo, *username* must already exist. If you manually create the user, modify the user's .bashrc file to include the line: PATH=/opt/vertica/bin:$PATH so Vertica tools such as vsql and admintools can be easily started by this user.

For details on a minimal installation procedure, see Perform a basic install .

Optional

The following install_vertica options are not required. Many of them enable greater control over the installation process.

--help

Display help for this script.

--accept-eula -Y

Silently accepts the EULA agreement. On multi-node installations, this option is propagated across the cluster at the end of the installation, at the same time as the Administration Tools metadata.

Combine this option with --license (-L) to activate your license.

--add-hosts *hostlist* -A *hostlist*

Comma-separated list of hosts to add to an existing Vertica cluster.

--add-hosts modifies an existing installation of Vertica by adding a host to the database cluster and then reconfiguring spread . This is useful for improving system performance, or making the database K-safe.
If spread is configured in your installation to use point-to-point communication within the existing cluster, you must also use it when you add a new host; otherwise, the new host automatically uses UDP broadcast traffic, resulting in cluster communication problems that prevent Vertica from running properly. For example:

```
--add-hosts host01
--add-hosts 192.168.233.101
```

You can also use this option with the update_vertica script. For details, see Adding nodes .

--broadcast -U

Configures spread to use UDP broadcast traffic between nodes on the subnet. This is the default setting. Up to 80 spread daemons are supported by broadcast traffic. You can exceed the 80-node limit by using large cluster mode, which does not install a spread daemon on each node.

Do not combine this option with -- point-to-point .
Important
When changing the configuration from --point-to-point to --broadcast , you must also specify --control-network .

--clean

Forcibly cleans previously stored configuration files. Use this option if you need to change the hosts that are included in your cluster. Only use this option when no database is defined.

This option is not supported by the update_vertica script.

--config-file *file* -z *file*

Use the properties file created earlier with [--record-config](#record-config) . This properties file contains key/value settings that map to install_vertica option.

--control-network { *IPaddress* | default} -S { *IPaddess* | default }

Set to one of the following arguments:

- *IPaddress* : A broadcast network IP address that enables configuration of spread communications on a subnet different from other Vertica data communications.
- default
  Important
  IPaddress must match the subnet for at least some database nodes. If the address does not match the subnet of any database node, then the installer displays an error and stops. If the provided address matches some, but not all of the node's subnets, the installer displays a warning, but installation continues.

  Optimally, the value for --control-network matches all node subnets.

You can also use this option to force a cluster-wide spread reconfiguration when changing spread-related options.

--data-dir *directory* -d *directory*

Directory for database data and catalog files. For details, see Specifying disk storage location during installation and Managing storage locations .

> **Caution**
> Do not use a shared directory over more than one host for this setting. Data and catalog directories must be distinct for each node. Multiple nodes must not be allowed to write to the same data or catalog directory.

Default: /home/dbadmin

--dba-group *group* -g *group*

UNIX group for DBA users.

Default: verticadba

--dba-user-home *directory* -l *directory*

Home directory for the database administrator.

Default: /home/dbadmin

--dba-user-password *password* -p *password*

Password for the database administrator account. If omitted, the script prompts for a password and does not echo the input.

--dba-use-password-disabled

Disables the password for --dba-user . This argument stops the installer from prompting for a password for --dba-user . You can assign a password later using standard user management tools such as passwd .

--failure-threshold [*threshold-arg*]

Stops the installation when the specified failure threshold is encountered, where *threshold-arg* is one of the following:

- HINT : Stop the install if a HINT or greater issue is encountered during the installation tests. HINT configurations are settings you should make, but the database runs with no significant negative consequences if you omit the setting.
- WARN : Stop the installation if a WARN or greater issue is encountered. WARN issues might affect database performance. However, for environments where high-level performance is not a priority—for example, testing—WARN issues can be ignored.
- FAIL : Stop the installation if a FAIL or greater issue is encountered. FAIL issues can have severely negative performance consequences and possible later processing issues if not addressed. However, Vertica can start even if FAIL issues are ignored.
- HALT : Stop the installation if a HALT or greater issue is encountered. The database might be unable to start if you choose his option. This option is not supported in production environments.
- NONE : Do not stop the installation. The database might be unable to start if you choose this option. This option is not supported in production environments.

Default: WARN

--ipv4

Hosts in the cluster are identified by IPv4 network addresses. This is the default behavior.

--ipv6

Hosts in the cluster are identified by IPv6 network addresses, required if the --hosts list specifies Pv6 addresses. This option automatically enables the -- point-to-point option.

--large-cluster [*num-control-nodes* | default]

Enables the large cluster feature, where a subset of nodes called control nodes connect to spread to send and receive broadcast messages. Consider using this option for a cluster with more than 50 nodes in Enterprise Mode. Vertica automatically enables this feature if you install onto 120 or more nodes in Enterprise Mode, or 16 or more nodes in Eon Mode.

Supply this option with one of the following arguments:

- *num-control-nodes* : Sets the number of control nodes in the new database to the smaller of this value or the value of --hosts . This value is applied differently in Enterprise Mode and Eon Mode:
  - Enterprise Mode: Sets the number of control nodes in the entire cluster.
  - Eon Mode: Sets the number of control nodes in the initial default subcluster. This value must be between 1 to 120, inclusive.
- default : Vertica sets the number of control nodes to the square root of the total number of cluster nodes listed in --hosts (-s).

For details, see Enable Large Cluster When Installing Vertica .

Default: default

--license { *licensefile* | CE} -L { *licensefile* | CE }

Silently and automatically deploys the license key to /opt/vertica/config/share . On multi-node installations, the —-license option also applies the license to all nodes declared by --hosts . To activate your license, combined this option with --accept-eula option. If you do not use the --accept-eula option, you are asked to accept the EULA when you connect to your database. After you accept the EULA, your license is activated.

If specified with CE , this option automatically deploys the Community Edition license key, which is included in your download.

---no-system-configuration

Installer makes no changes to system properties. By default, the installer makes system configuration changes that meet server requirements.

If you use this option, the installer posts warnings or failures for configuration settings that do not meet requirements that it otherwise configures automatically.
This option has no effect on creating or updating user accounts.

--parallel-no-prompts

Installs the server binary package (.rpm or .deb) on the hosts in parallel without prompting for confirmation. This option reduces the installation time, especially on large clusters. If omitted, the install script installs the package on one host at a time. .

This option requires that the installer use passwordless ssh to connect to the hosts. It has no effect if the installer is not using passwordless ssh.

--point-to-point -T

Configures spread to use direct point-to-point communication between all Vertica nodes. Use this option if nodes are not located on the same subnet. Also use this option for all virtual environment installations, whether or not virtual servers are on the same subnet.

Up to 80 spread daemons are supported by point-to-point communication. You can exceed the 80-node limit by using large cluster mode, which does not install a spread daemon on each node.
Do not combine this option with -- broadcast .

This option is automatically enabled by the --ipv6 option.

Important
When changing the configuration from --broadcast to --point-to-point , you must also specify --control-network .

--record-config *filename* -B *filename*

File name used with command line options to create a properties file that can be used with [--config-file](#record-config) . This option creates the properties file and exits; it does not affect installation.

--remove-hosts *hostlist* -R *hostlist*

Comma-separated list of hosts to remove from an existing Vertica cluster. After removing the specified hosts, spread is reconfigured on the cluster.

This option is useful for removing an obsolete or over-provisioned system.
If you use --point-to-point (-T) to configure spread to use direct point-to-point communication within the existing cluster, you must also use it when you remove a host; otherwise, the hosts automatically use UDP broadcast traffic, resulting in cluster communication problems that prevents Vertica from running properly.

The update_vertica script (see Removing hosts from a cluster) calls install_vertica to update the installation. You can use either script with this option.

--spread-logging -w

Configures spread to output logging to /opt/vertica/log/spread_ *hostname* .log . This option does not apply to upgrades.

> **Note**
> Enable spread logging only if requested by Vertica technical support.

--ssh-identity *file* -i *file*

The root private-key *file* to use if passwordless ssh was already configured between the hosts. Before using this option, verify that normal SSH works without a password . The file can be private key file—for example, id_rsa —or PEM file. Do not use with the --ssh-password (-P) option.

Vertica accepts the following:

- By providing an SSH private key which is not password protected. You cannot run the install_vertica script with the sudo command when using this method.
- By providing a password-protected private key and using an SSH-Agent. Note that sudo typically resets environment variables when it is invoked. Specifically, the SSH_AUTHSOCK variable required by the SSH-Agent may be reset. Therefore, configure your system to maintain SSH_AUTHSOCK or invoke install_vertica using a method similar to the following:

```
sudo SSH_AUTHSOCK=$SSH_AUTHSOCK /opt/vertica/sbin/install_vertica ...
```

--ssh-password *password* -P *password*

The password to use by default for each cluster host. If you omit this option and also omit --ssh-identity (-i), then the script prompts for the password as necessary and does not echo input.

Do not use this option together with --ssh-identity (-i).
Important
If you run the install_vertica script as root, specify the root password:

```
# /opt/vertica/sbin/install_vertica -P root-passwd
```

If you run the install_vertica script with the sudo command, specify the password of the user who runs install_vertica , not the root password. For example if the dbadmin user runs install_vertica with sudo and has the password dbapasswd , then specify the password as dbapasswd :

```
$ sudo /opt/vertica/sbin/install_vertica -P dbapasswd
```

--temp-dir *directory*

Temporary directory used for administrative purposes. If it is a directory within /opt/vertica , then it is created by the installer. Otherwise, the directory should already exist on all nodes in the cluster. The location should allow dbadmin write privileges.

> **Note**
> This is not a temporary data location for the database.

Default: /tmp

Install Vertica silently

This section describes how to create a properties file that lets you install and deploy Vertica-based applications quickly and without much manual intervention.

> **Note**
> The procedure assumes that you have already performed the tasks in Before you install Vertica .

Install the properties file:

1. Download and install the Vertica install package, as described in Download and install the Vertica server package .
2. Create the properties file that enables non-interactive setup by supplying the parameters you want Vertica to use. For example:
   The following command assumes a multi-node setup:

```
# /opt/vertica/sbin/install_vertica --record-config file_name --license /tmp/license.txt --accept-eula \
# --dba-user-password password --ssh-password password --hosts host_list --rpm package_name
```

   The following command assumes a single-node setup:

```
# /opt/vertica/sbin/install_vertica --record-config file_name --license /tmp/license.txt --accept-eula \
# --dba-user-password password
```

Option	Description
--record-file *file_name*	[Required] Accepts a file name, which when used in conjunction with command line options, creates a properties file that can be used with the --config-file option during setup. This flag creates the properties file and exits; it has no impact on installation.
--license { *license_file* | CE }	Silently and automatically deploys the license key to /opt/vertica/config/share. On multi-node installations, the —license option also applies the license to all nodes declared in the --hosts *host_list* . If specified with CE, automatically deploys the Community Edition license key, which is included in your download. You do not need to specify a license file.
--accept-eula	Silently accepts the EULA agreement during setup.
--dba-user-password *password*	The password for the Database Superuser account; if not supplied, the script prompts for the password and does not echo the input.

--ssh-password *password*	The root password to use by default for each cluster host; if not supplied, the script prompts for the password if and when necessary and does not echo the input.
--hosts *host_list*	A comma-separated list of hostnames or IP addresses to include in the cluster; do not include space characters in the list. Examples: --hosts host01,host02,host03 --hosts 192.168.233.101,192.168.233.102,192.168.233.103
--rpm *package_name* --deb *package_name*	The name of the RPM or Debian package that contained this script. Example: --rpm vertica-12.0.x.x86_64.RHEL6.rpm This parameter is required on multi-node installations if the RPM or DEB package is not already installed on the other hosts.

See install_vertica options for the complete set of installation parameters.

Tip
Supply the parameters to the properties file once only. You can then install Vertica using just the --config-file parameter, as described below.

1. Use one of the following commands to run the installation script.

- If you are root:
```
/opt/vertica/sbin/install_vertica --config-file file_name
```
- If you are using sudo:
```
$ sudo /opt/vertica/sbin/install_vertica --config-file file_name
```

--config-file *file_name* accepts an existing properties file created by --record-config *file_name* . This properties file contains key/value parameters that map to values in the install_vertica script, many with boolean arguments that default to false
The command for a single-node install might look like this:
```
# /opt/vertica/sbin/install_vertica --config-file /tmp/vertica-inst.prp
```

1. If you did not supply a --ssh-password password parameter to the properties file, you are prompted to provide the requested password to allow installation of the RPM/DEB and system configuration of the other cluster nodes. If you are root, this is the root password. If you are using sudo, this is the sudo user password. The password does not echo on the command line.

Note
If you are root on a single-node installation, you are not prompted for a password.

1. If you did not supply a --dba-user-password password parameter to the properties file, you are prompted to provide the database administrator account password.

The installation script creates a new Linux user account (dbadmin by default) with the password that you provide.

1. Carefully examine any warnings produced by install_vertica and correct the problems if possible. For example, insufficient RAM, insufficient Network throughput and too high readahead settings on file system could cause performance problems later on.

Note
You can redirect any warning outputs to a separate file, instead of having them display on the system. Use your platforms standard redirected mechanisms. For example: install_vertica *[options]* > /tmp/file 1>&2 .

1. **Optionally** perform the following steps:

- Install the ODBC and JDBC driver .
- Install vsql Client Application on Non-Cluster Hosts .

1. Disconnect from the Administration Host as instructed by the script. This is required to:

- Set certain system parameters correctly.
- Function as the Vertica database administrator.

At this point, Linux root privileges are no longer needed. The database administrator can perform the remaining steps.

> **Note**
> When creating a new database, the database administrator might want to use different data or catalog locations than those created by the installation script. In that case, a Linux administrator might need to create those directories and change their ownership to the database administrator.

If you supplied the --license and --accept-eula parameters to the properties file, then proceed to Getting started and then see Configuring the database .

Otherwise:

1. Log in to the Database Superuser account on the administration host.
2. Accept the End User License Agreement and install the license key you downloaded previously as described in Install the License Key .
3. Proceed to Getting started and then see Configuring the database .

Notes

- Downgrade installations are not supported.
- The following is an example of the contents of the configuration properties file:

```
accept_eula = True
license_file = /tmp/license.txt
record_to = file_name
root_password = password
vertica_dba_group = verticadba
vertica_dba_user = dbadmin
vertica_dba_user_password = password
```

Enable secure shell (SSH) logins

The administrative account must be able to use Secure Shell (SSH) to log in (ssh) to all hosts without specifying a password. The shell script install_vertica does this automatically. This section describes how to do it manually if necessary.

1. If you do not already have SSH installed on all hosts, log in as root on each host and install it now. You can download a free version of the SSH connectivity tools from OpenSSH .
2. Log in to the Vertica administrator account (dbadmin in this example).
3. Make your home directory (~) writable only by yourself. Choose one of:

   ```
   $ chmod 700 ~
   ```

   or

   ```
   $ chmod 755 ~
   ```

   where:

700 includes	755 includes

400 read by owner 200 write by owner 100 execute by owner	400 read by owner 200 write by owner 100 execute by owner 040 read by group 010 execute by group 004 read by anybody (other) 001 execute by anybody

4. Change to your home directory:

```
$ cd ~
```

1. Generate a private key/ public key pair:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/dbadmin/.ssh/id_rsa):
Created directory '/home/dbadmin/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/dbadmin/.ssh/id_rsa.
Your public key has been saved in /home/dbadmin/.ssh/id_rsa.pub.
```

1. Make your .ssh directory readable and writable only by yourself:

```
$ chmod 700 ~/.ssh
```

1. Change to the .ssh directory:

```
$ cd ~/.ssh
```

1. Copy the file id_rsa.pub onto the file authorized_keys2 .

```
$ cp id_rsa.pub authorized_keys2
```

1. Make the files in your .ssh directory readable and writable only by yourself:

```
$ chmod 600 ~/.ssh/*
```

1. For each cluster host:

```
$ scp -r ~/.ssh <host>:.
```

1. Connect to each cluster host. The first time you ssh to a new remote machine, you could get a message similar to the following:

```
$ ssh dev0 Warning: Permanently added 'dev0,192.168.1.92' (RSA) to the list of known hosts.
```

This message appears only the first time you ssh to a particular remote host.

See also

- OpenSSH

After you install Vertica

The tasks described in this section are optional and are provided for your convenience. When you have completed this section, proceed to one of the following:

- Getting started
- Configuring the database

Install client drivers

After you install Vertica, install drivers on the client systems from which you plan to access your databases. Vertica supplies drivers for ADO.NET, JDBC, ODBC, OLE DB, Perl, and Python. For instructions on installing these drivers, see Client drivers .

Install the license key

If you did not supply the -L parameter during setup, or if you did not bypass the -L parameter for a silent install , the first time you log in as the Database Superuser and run the Vertica Administration tools or Management Console, Vertica requires you to install a license key.

Follow the instructions in Managing licenses in the Administrator's Guide.

Create a database

To get started using Vertica immediately after installation, create a database. You can use either the Administration Tools or the Management Console. To create a database using MC, refer to Creating a database using MC . For instructions on creating a database with admintools, see Creating a database .

Install vsql client application on non-cluster hosts

You can use the Vertica vsql executable image on a non-cluster Linux host to connect to a Vertica database.

- On Red Hat, CentOS, and SUSE systems, you can install the client driver RPM, which includes the vsql executable. See Installing the vsql client for details.
- If the non-cluster host is running the same version of Linux as the cluster, copy the image file to the remote system. For example:
  ```
  $ scp host01:/opt/vertica/bin/vsql .$ ./vsql
  ```
- If the non-cluster host is running a different distribution or version of Linux than your cluster hosts, you must install the Vertica server RPM in order to get vsql:
  1. Download the appropriate RPM package by browsing to Vertica website. On the **Support** tab, select **Customer Downloads** .
  2. If the system you used to download the RPM is not the non-cluster host, transfer the file to the non-cluster host.
  3. Log into the non-cluster host as root and install the RPM package using the command:
     ```
     # rpm -Uvh filename
     ```
     Where *filename* is the package you downloaded. Note that you do not have to run the install_vertica script on the non-cluster host to use vsql.

Notes

- Use the same Command-line options that you would on a cluster host.
- You cannot run vsql on a Cygwin bash shell (Windows). Use ssh to connect to a cluster host, then run vsql.

vsql is also available for additional platforms. See Installing the vsql client .

Upgrading Vertica

Upgrading your database with a new Vertica version includes the following steps:

1. Complete upgrade prerequisites
2. Upgrade Vertica
3. Perform post-upgrade tasks

You can upgrade your database from its current Vertica version to any higher version. Before upgrading, make sure that you have performed a full database backup and have tested the new version in an environment that closely resembles your production database.

> **Tip**
> If you want to test out a new version of Vertica for an Eon Mode database without spinning up a new cluster, you can sandbox a secondary subcluster and then upgrade the subcluster within the sandbox. By sandboxing the subcluster, you can try out the new version of Vertica stress-free and continue to use your main cluster as usual. After confirming that the upgrade works and performs as expected, you can downgrade the sandboxed subcluster, remove the sandbox , and proceed to upgrade the main cluster.

Be sure to read the Release Notes and New Features for the Vertica version to which you intend to upgrade. Documentation and release notes for the current Vertica version are available in the RPM and at https://docs.vertica.com/latest , which also provides access to documentation for earlier versions.

For guidance on upgrading from unsupported versions, contact Vertica Technical Support .

In this section

- Before you upgrade
- Upgrade Vertica
- After you upgrade

Before you upgrade

Before you upgrade the Vertica database, perform the following steps:

- Verify that you have enough RAM available to run the upgrade. The upgrade requires approximately three times the amount of memory your database catalog uses.
  You can calculate catalog memory usage on all nodes by querying system table RESOURCE_POOL_STATUS :
  ```
  => SELECT node_name, pool_name, memory_size_kb FROM resource_pool_status WHERE pool_name = 'metadata';
  ```
- Perform a full database backup. This precautionary measure allows you to restore the current version if the upgrade is unsuccessful.
- Perform a backup of your grants .
- Verify platform requirements for the new version.
- Determine whether you are using any third-party user-defined extension libraries (UDxs). UDx libraries that are compiled (such as those developed using C++ or Java) may need to be recompiled with a new version of the Vertica SDK libraries to be compatible with the new version of Vertica. See UDx library compatibility with new server versions .
- Check catalog storage space .
- Note that any user or role with the same name as a predefined role is renamed to OLD_ *n*_*name* , where *n* is an integer that increments from zero until the resulting name is unique and *name* is the previous name of the user or role.
- If you're upgrading from Vertica 9.2.x and have set the PasswordMinCharChange or PasswordMinLifeTime system-level security parameters , take note of their current values. You will have to set these parameters again, this time at the PROFILE-level, to reproduce your configuration. To view the current values for these parameters, run the following query:
  ```
  => SELECT parameter_name,current_value from CONFIGURATION_PARAMETERS
    WHERE parameter_name IN ('PasswordMinCharChange', 'PasswordMinLifeTime');
  ```

After you complete these tasks, shut down the database gracefully .

In this section

- Verifying platform requirements
- Checking catalog storage space
- Verify license compliance for ORC and Parquet data
- Backing up and restoring grants
- Nonsequential FIPS database upgrades

Verifying platform requirements

The Vertica installer checks the target platform as it runs, and stops whenever it determines the platform fails to meet an installation requirement. Before you update the server package on your systems, manually verify that your platform meets all hardware and software requirements (see Platform and hardware requirements and recommendations).

By default, the installer stops on all warnings. You can configure the level where the installer stops installation, through the installation parameter --failure-threshold . If you set the failure threshold to FAIL , the installer ignores warnings and stops only on failures.

> **Caution**
> Changing the failure threshold lets you immediately upgrade and bring up the Vertica database. However, Vertica cannot fully optimize performance until you correct all warnings.

Checking catalog storage space

Use the commands documented here to determine how much catalog space is available before upgrading. This helps you determine how much space the updated catalog may take up.

Compare how much space the catalog currently uses against space that is available in the same directory:

1. Use the du command to determine how much space the catalog directory currently uses:
   ```
   $ du -s -BG v_vmart_node0001_catalog
   2G      v_vmart_node0001_catalog
   ```

2. Determine how much space is available in the same directory:

```
$ df -BG v_vmart_node0001_catalog
Filesystem     1G-blocks  Used Available Use% Mounted on
/dev/sda2            48G   19G       26G  43% /
```

Verify license compliance for ORC and Parquet data

If you are upgrading from a version before 9.1.0 and:

- Your database has external tables based on ORC or Parquet files (whether stored locally on the Vertica cluster or on a Hadoop cluster)
- Your Vertica license has a raw data allowance

follow the steps in this topic before upgrading.

Background

Vertica licenses can include a raw data allowance. Since 2016, Vertica licenses have allowed you to use ORC and Parquet data in external tables. This data has always counted against any raw data allowance in your license. Previously, the audit of data in ORC and Parquet format was handled manually. Because this audit was not automated, the total amount of data in your native tables and external tables could exceed your licensed allowance for some time before being spotted.

Starting in version 9.1.0, Vertica automatically audits ORC and Parquet data in external tables. This auditing begins soon after you install or upgrade to version 9.1.0. If your Vertica license includes a raw data allowance and you have data in external tables based on Parquet or ORC files, review your license compliance before upgrading to Vertica 9.1.x. Verifying your database is compliant with your license terms avoids having your database become non-compliant soon after you upgrade.

Verifying your ORC and Parquet usage complies with your license terms

To verify your data usage is compliant with your license, run the following query as the database administrator:

```
SELECT (database_size_bytes + file_size_bytes) <= license_size_bytes
    "license_compliant?"
    FROM (SELECT database_size_bytes,
            license_size_bytes FROM license_audits
            WHERE audited_data='Total'
            ORDER BY audit_end_timestamp DESC LIMIT 1) dbs,
        (SELECT sum(total_file_size_bytes) file_size_bytes
            FROM external_table_details
            WHERE source_format IN ('ORC', 'PARQUET')) ets;
```

This query returns one of three values:

- If you do not have any external data in ORC or Parquet format, the query returns 0 rows:

  ```
  license_compliant?
  --------------------
  (0 rows)
  ```

  In this case, you can proceed with your upgrade.
- If you have data in external tables based on ORC or Parquet format, and that data does not cause your database to exceed your raw data allowance, the query returns t:

  ```
  license_compliant?
  --------------------
  t
  (1 row)
  ```

  In this case, you can proceed with your upgrade.
- If the data in your external tables based on ORC and Parquet causes your database to exceed your raw data allowance, the query returns f:

  ```
  license_compliant?
  --------------------
  f
  (1 row)
  ```

  In this case, resolve the compliance issue before you upgrade. See below for more information.

Resolving non-compliance

If query in the previous section indicates that your database is not in compliance with your license, you should resolve this issue before upgrading. There are two ways you can bring your database into compliance:

- Contact Vertica to upgrade your license to a larger data size allowance. See Obtaining a license key file .
- Delete data (either from ORC and Parquet-based external tables or Vertica native tables) to bring your data size into compliance with your license. You should always backup any data you are about to delete from Vertica. Dropping external tables is a less disruptive way to reduce the size of your database, as the data is not lost—it is still in the files that your external table is based on.

> **Note**
> You can still choose to upgrade your database if it is not compliant. However, soon after you upgrade, you will begin getting warnings that your database is out of compliance. See Managing license warnings and limits for more information.

Backing up and restoring grants

After an upgrade, if the prototypes of UDx libraries change, Vertica will drop the grants on those libraries since they aren't technically the same function anymore. To resolve these types of issues, it's best practice to back up the grants on these libraries so you can restore them after the upgrade.

1. Save the following SQL to a file named user_ddl.sql . It creates a view named user_ddl which contains the grants on all objects in the database.

```
CREATE OR REPLACE VIEW user_ddl AS
(
SELECT 0 as grant_order,
      name principal_name,
      'CREATE ROLE "' || name || '"' ||    ';' AS sql,
      'NONE' AS object_type,
      'NONE' AS object_name
  FROM v_internal.vs_roles vr
 WHERE NOT vr.predefined_role -- Exclude system roles
   AND ldapdn = ''            -- Limit to NON-LDAP created roles
)
UNION ALL
(
 SELECT 1, -- CREATE USERs
        user_name,
        'CREATE USER "' || user_name || '"' ||
        DECODE(is_locked, TRUE, ' ACCOUNT LOCK', '') ||
        DECODE(grace_period, 'undefined', '', ' GRACEPERIOD  "' || grace_period || '"') ||
        DECODE(idle_session_timeout, 'unlimited', '', ' IDLESESSIONTIMEOUT "' || idle_session_timeout || '"') ||
        DECODE(max_connections, 'unlimited', '', ' MAXCONNECTIONS ' || max_connections || ' ON ' || connection_limit_mode) ||
        DECODE(memory_cap_kb, 'unlimited', '', ' MEMORYCAP "' || memory_cap_kb || 'K"') ||
        DECODE(profile_name, 'default', '', ' PROFILE ' || profile_name) ||
        DECODE(resource_pool, 'general', '', ' RESOURCE POOL ' || resource_pool) ||
        DECODE(run_time_cap, 'unlimited', '', ' RUNTIMECAP "' || run_time_cap || '"') ||
        DECODE(search_path, '', '', ' SEARCH_PATH ' || search_path) ||
        DECODE(temp_space_cap_kb, 'unlimited', '', ' TEMPSPACECAP "' || temp_space_cap_kb || 'K"') || ';' AS sql,
        'NONE' AS object_type,
        'NONE' AS object_name
   FROM v_catalog.users
  WHERE NOT is_super_user -- Exclude database superuser
    AND ldap_dn = ''      -- Limit to NON-LDAP created users
)
UNION ALL
(
 SELECT 2, -- GRANTs
        grantee,
        'GRANT ' || REPLACE(TRIM(BOTH ' ' FROM words), '*', '') ||
        CASE
          WHEN object_type = 'RESOURCEPOOL' THEN ' ON RESOURCE POOL '
          WHEN object_type = 'STORAGELOCATION' THEN ' ON LOCATION '
          WHEN object_type = 'CLIENTAUTHENTICATION' THEN 'AUTHENTICATION '
```

```
        WHEN object_type IN ('DATABASE', 'LIBRARY', 'MODEL', 'SEQUENCE', 'SCHEMA') THEN ' ON ' || object_type || ' '
        WHEN object_type = 'PROCEDURE' THEN (SELECT ' ON ' || CASE REPLACE(procedure_type, 'User Defined ', '')
                                               WHEN 'Transform' THEN 'TRANSFORM FUNCTION '
                                               WHEN 'Aggregate' THEN 'AGGREGATE FUNCTION '
                                               WHEN 'Analytic' THEN 'ANALYTIC FUNCTION '
                                               ELSE UPPER(REPLACE(procedure_type, 'User Defined ', '')) || ' '
                                              END
                                         FROM vs_procedures
                                        WHERE proc_oid = object_id)
        WHEN object_type = 'ROLE' THEN ''
        ELSE ' ON '
       END ||
       NVL2(object_schema, object_schema || '.', '') || CASE WHEN object_type = 'STORAGELOCATION' THEN (SELECT '''' || location_path || ''' ON ' ||
node_name FROM storage_locations WHERE location_id = object_id) ELSE object_name END ||
       CASE
        WHEN object_type = 'PROCEDURE' THEN (SELECT CASE WHEN procedure_argument_types = '' OR procedure_argument_types = 'Any' THEN
'()' ELSE '(' || procedure_argument_types || ')' END
                                         FROM vs_procedures
                                        WHERE proc_oid = object_id)
        ELSE ''
       END ||
       ' TO ' || grantee ||
       CASE WHEN INSTR(words, '*') > 0 THEN ' WITH GRANT OPTION' ELSE '' END
       || ';',
       object_type,
       object_name
  FROM (SELECT grantee, object_type, object_schema, object_name, object_id,
               v_txtindex.StringTokenizerDelim(DECODE(privileges_description, '', ',' , privileges_description), ',')
                OVER (PARTITION BY grantee, object_type, object_schema, object_name, object_id)
        FROM v_catalog.grants) foo
 ORDER BY CASE REPLACE(TRIM(BOTH ' ' FROM words), '*', '') WHEN 'USAGE' THEN 1 ELSE 2 END
)
UNION ALL
(
  SELECT 3, -- Default ROLEs
       user_name,
       'ALTER USER ''' || user_name || '''' ||
        DECODE(default_roles, '', '', ' DEFAULT ROLE ' || REPLACE(default_roles, '*', ''))  || ';' ,
       'NONE' AS object_type,
       'NONE' AS object_name
   FROM v_catalog.users
  WHERE default_roles <> ''
)
UNION ALL -- GRANTs WITH ADMIN OPTION
(
  SELECT 4, user_name, 'GRANT ' || REPLACE(TRIM(BOTH ' ' FROM words), '*', '') || ' TO ' || user_name || ' WITH ADMIN OPTION;',
       'NONE' AS object_type ,
       'NONE' AS object_name
   FROM (SELECT user_name, v_txtindex.StringTokenizerDelim(DECODE(all_roles, '', ',', all_roles), ',') OVER (PARTITION BY user_name)
          FROM v_catalog.users
         WHERE all_roles <> '') foo
  WHERE INSTR(words, '*') > 0
)
UNION ALL
(
  SELECT 5, 'public', 'ALTER SCHEMA ' || name || ' DEFAULT ' || CASE WHEN defaultinheritprivileges THEN 'INCLUDE PRIVILEGES;' ELSE 'EXCLUDE
PRIVILEGES;' END, 'SCHEMA', name
   FROM v_internal.vs_schemata
  WHERE NOT issys -- Exclude system schemas
)
UNION ALL
```

```
(
 SELECT 6, 'public', 'ALTER DATABASE ' || database_name ||  ' SET disableinheritedprivileges = ' || current_value || ';',
       'DATABASE', database_name
  FROM v_internal.vs_configuration_parameters
 CROSS JOIN v_catalog.databases
 WHERE parameter_name = 'DisableInheritedPrivileges'
)
UNION ALL -- TABLE PRIV INHERITENCE
(
 SELECT 7, 'public' , 'ALTER TABLE ' || table_schema || '.' || table_name ||
       CASE WHEN inheritprivileges THEN ' INCLUDE PRIVILEGES;' ELSE ' EXCLUDE PRIVILEGES;' END,
       'TABLE' AS object_type,
       table_schema || '.' || table_name AS object_name
  FROM v_internal.vs_tables
  JOIN v_catalog.tables ON (table_id = oid)
)
UNION ALL -- VIEW PRIV INHERITENCE
(
 SELECT 8, 'public', 'ALTER VIEW ' || table_schema || '.' || table_name || CASE WHEN inherit_privileges THEN ' INCLUDE PRIVILEGES;' ELSE '
EXCLUDE PRIVILEGES; ' END,
       'TABLE' AS object_type, table_schema || '.' || table_name AS object_name
  FROM v_catalog.views
)
UNION ALL
(
 SELECT 9, owner_name, 'ALTER TABLE ' || table_schema || '.' || table_name || ' OWNER TO ' || owner_name || ';',
       'TABLE', table_schema || '.' || table_name
  FROM v_catalog.tables
)
UNION ALL
(
 SELECT 10, owner_name, 'ALTER VIEW ' || table_schema || '.' || table_name || ' OWNER TO ' || owner_name || ';', 'TABLE',
       table_schema || '.' || table_name
  FROM v_catalog.views
);
```

2. From the Linux command line, run the script in the user_ddl.sql file:

```
$ vsql -f user_ddl.sql
CREATE VIEW
```

3. Connect to Vertica using vsql.
4. Export the content of the user_ddl's sql column ordered on the grant_order column to a file:

```
=> \o pre-upgrade.txt
=> SELECT sql FROM user_ddl ORDER BY grant_order ASC;
=> \o
```

5. Upgrade Vertica .
6. Select and save to a different file the view's SQL column with the same command.

```
=> \o post-upgrade.txt
=> SELECT sql FROM user_ddl ORDER BY grant_order ASC;
=> \o
```

7. Create a diff between pre-upgrade.txt and post-upgrade.txt . This collects the missing grants into grants-list.txt .

```
$ diff pre-upgrade.txt post-upgrade.txt > grants-list.txt
```

8. To restore any missing grants, run the remaining grants in grants-list.txt , if any:

```
=> \i 'grants-list.txt'
```

Note

Attempting to restore grants to users with the ANY keyword triggers the following error:

```
ERROR 4856: Syntax error at or near "Any" at character
```

To avoid this error, use () instead of (ANY) as shown in the following example:

```
=> GRANT EXECUTE ON FUNCTION public.MapLookup() TO public;
GRANT PRIVILEGE
```

Nonsequential FIPS database upgrades

As of Vertica 10.1.1, FIPS support has been reinstated. Prior to this, the last version to support FIPS was Vertica 9.2.x. If you are upgrading from 9.2.x and want to maintain your FIPS certification, you must first perform a direct upgrade from 9.2.x to 10.1.1 before performing further upgrades.

The following procedure performs a direct upgrade from Vertica 9.2.x running on RHEL 6.x to Vertica 10.1.1 on RHEL 8.1.

Important
If you have any questions or want additional guidance for performing this upgrade, contact Vertica Support .

1. Create a full backup of your Vertica 9.2.x database. This example uses the configuration file fullRestore.ini .

```
$ vbr --config-file=/tmp/fullRestore.ini -t init
$ vbr --config-file=/tmp/fullRestore.ini -t backup
```

```
[Transmission]
concurrency_backup = 1
port_rsync = 50000
encrypt = False
serviceAccessPass = rsyncpw
hardLinkLocal = False
checksum = False
total_bwlimit_restore = 0
serviceAccessUser = rsyncuser
total_bwlimit_backup = 0
concurrency_restore = 1

[Misc]
snapshotName = full_restore
restorePointLimit = 1
retryDelay = 1
objects =
retryCount = 0
tempDir = /tmp/vbr

[Mapping]
v_fips_db_node0001 = 198.51.100.0:/home/release/backup/
v_fips_db_node0002 = 198.51.100.1:/home/release/backup/
v_fips_db_node0003 = 198.51.100.2:/home/release/backup/

[Database]
dbPort = 5433
dbPromptForPassword = False
dbUser =
dbPassword =
dbName = fips_db
```

2. Shut down the database gracefully . Do not start the database until instructed.
3. Acquire a RHEL 8.1 cluster with one of the following methods:
   1. Upgrade in place
   2. Reimage your machines
   3. Use a completely different RHEL 8.1 cluster
4. Enable FIPS on your RHEL 8.1 machines and reboot.

```
$ fips-mode-setup --enable
```

5. Install Vertica 10.1.1 on the RHEL 8.1 cluster.
```
$ install_vertica --hosts node0001, node0002, node0003 \
  --rpm /tmp/vertica-10.1.1-0/x86_64.RHEL8.rpm
```

6. If you acquired your RHEL 8.1 cluster by reimaging or using a different cluster, you must restore your database .
```
$ vbr -c /tmp/fullRestore.ini -t restore
```

   If you encounter the following warning, you can safely ignore it.
```
Warning: Vertica versions do not match: v9.2.1-xx -> v10.1.1-xxxxxxxx. This operation may not be supported.
```

7. Start the Vertica 10.1.1 database to trigger the upgrade. This should be the first time you've started your database since shutting it down in step 2.
```
$ admintools -t start_db -d fips_db
```

Upgrade Vertica

Important
Before running the upgrade script, be sure to review the tasks described in Before you upgrade .

To upgrade your database to a new Vertica version, complete the following steps:

1. Perform a full backup of your existing database. This precautionary measure lets you restore from the backup, if the upgrade is unsuccessful. If the upgrade fails, you can reinstall the previous version of Vertica and restore your database to that version.
2. Use admintools to stop the database .
3. On each host where an additional package is installed, such as the R language pack , uninstall it. For example:
```
rpm -e vertica-R-lang
```

   Important
   If you omit this step and do not uninstall additional packages, the Vertica server package fails to install in the next step.

4. Make sure you are logged in as root or sudo and use one of the following commands to run the RPM package installer:
   - If you are root and installing an RPM:
```
# rpm -Uvh pathname
```
   - If you are using sudo and installing an RPM:
```
$ sudo rpm -Uvh pathname
```
   - If you are using Debian:
```
$ sudo dpkg -i pathname
```

5. On the same node on which you just installed the RPM, run update_vertica as root or sudo. This installs the RPM on all the hosts in the cluster. For example:
   - Red Hat or CentOS:
```
# /opt/vertica/sbin/update_vertica --rpm /home/dbadmin/vertica-12.0.x.x86_64.RHEL6.rpm --dba-user mydba
```
   - Debian:
```
# /opt/vertica/sbin/update_vertica --deb /home/dbadmin/vertica-amd64.deb --dba-user mydba
```

> **Note**
> You can upgrade the Vertica server running on AWS instances created from a Vertica AMI. To upgrade the Vertica server on these AWS instances, you need to add the --dba-user-password-disabled and --point-to-point arguments to the upgrade script.

The following requirements and restrictions apply:
- The DBADMIN user must be able to read the RPM or DEB file when upgrading. Some upgrade scripts are run as the DBADMIN user, and that user must be able to read the RPM or DEB file.
- Use the same options that you used when you last installed or upgraded the database. You can find these options in /opt/vertica/config/admintools.conf , on the install_opts line. For details on all options, see Install Vertica with the installation script .

> **Caution**
> If you omit any previous options, their default settings are restored. If you do so, or if you change any options, the upgrade script uses

the new settings to reconfigure the cluster. This can cause issues with the upgraded database.

   - Omit the --hosts/-s *host-list* parameter. The upgrade script automatically identifies cluster hosts.
   - If the root user is not in /etc/sudoers, an error appears. The installer reports this issue with **S0311** . See the Sudoers Manual for more information.
6. Start the database . The start-up scripts analyze the database and perform necessary data and catalog updates for the new version.
   If Vertica issues a warning stating that one or more packages cannot be installed, run the admintools --force-reinstall option to force reinstallation of the packages. For details, see Reinstalling packages .
   When the upgrade completes, the database automatically restarts.

   Note
   Any user or role with the same name as a predefined role is renamed to OLD_ *n_name* , where *n* is an integer that increments from zero until the resulting name is unique and *name* is the previous name of the user or role.

7. Manually restart any nodes that fail to start.
8. Perform another database backup.

Upgrade duration

Duration depends on average in-memory size of catalogs across all cluster nodes. For every 20GB, you can expect the upgrade to last between one and two hours.

You can calculate catalog memory usage on all nodes by querying system table RESOURCE_POOL_STATUS :

```
=> SELECT node_name, pool_name, memory_size_kb FROM resource_pool_status WHERE pool_name = 'metadata';
```

Post-upgrade tasks

After you complete the upgrade, review post-upgrade tasks in After you upgrade .

After you upgrade

After you finish upgrading the Vertica server package on your cluster, a number of tasks remain.

Required tasks

- If you created projections in earlier releases with pre-aggregated data (for example, LAPs and TopK projections) and the projections were partitioned with a GROUP BY clause, you must rebuild these projections .
- Verify on each node that the upgrade reduced database catalog memory usage .
- Verify your database retained the grants from before you upgraded. See Backing up and restoring grants for more information.
- Reinstall packages such as the R language pack that you uninstalled before upgrading. For each package, see its install/upgrade instructions.

  Note
  Vertica Place is automatically reinstalled with the Vertica server package.

- If the upgrade was unable to install one or more packages, reinstall them with admintools .
- Upgrade the Management Console .
- If your Vertica installation is integrated with Hadoop, upgrade the HCatalog connector .

Optional tasks

- Import directed queries that you exported from the previous version. For details, see Batch query plan export and Exporting directed queries from the catalog .
- If you're upgrading from Vertica 9.2.x and have set the PasswordMinCharChange or PasswordMinLifeTime system-level security parameters , set them again at the PROFILE-level .

In this section

- Rebuilding partitioned projections with pre-aggregated data
- Verifying catalog memory consumption
- Reinstalling packages
- Writing bundle metadata to the catalog

- Upgrading the streaming data scheduler utility

Rebuilding partitioned projections with pre-aggregated data

If you created projections in earlier (pre-10.0.x) releases with pre-aggregated data (for example, LAPs and TopK projections) and the anchor tables were partitioned with a GROUP BY clause, their ROS containers are liable to be corrupted from various DML and ILM operations. In this case, you must rebuild the projections:

1. Run the meta-function REFRESH on the database. If REFRESH detects problematic projections, it returns with failure messages. For example:

```
=> SELECT REFRESH();
                                    REFRESH
-----------------------------------------------------------------------------------------------------
Refresh completed with the following outcomes:
Projection Name: [Anchor Table] [Status] [ Refresh Method] [Error Count]
"public"."store_sales_udt_sum": [store_sales] [failed: Drop and recreate projection] [] [1]
"public"."product_sales_largest": [store_sales] [failed: Drop and recreate projection] [] [1]
"public"."store_sales_recent": [store_sales] [failed: Drop and recreate projection] [] [1]

(1 row)
```

   Vertica also logs messages to vertica.log :

```
2020-07-07 11:28:41.618 Init Session:ox7fabbbfff700-aoo000000oosbs [Txnl <INFO> Be in Txn: aooooooooo5b5 'Refresh: Evaluating which projection to refresh'
2020-07-07 11:28:41.640 Init Session:ex7fabbbfff7oe-aoooeeeeoosbs [Refresh] <INFO> Storage issues detected, unable to refresh projection 'store_sales_recent'. Drop and recreate this projection, then refresh.
2020-07-07 11:28:41.641 Init Session:Ox7fabbbfff700-aoooeoooooosbs [Refresh] <INFO> Storage issues detected, unable to refresh projection 'product_sales_largest'. Drop and recreate this projection, then refresh.
2020-07-07 11:28:41.641 Init Session:Ox7fabbbfff700-aeoeeeaeeeosbs [Refresh] <INFO> Storage issues detected, unable to refresh projection 'store_sales_udt_sum'. Drop and recreate this projection, then refresh.
```

2. Export the DDL of these projections with EXPORT_OBJECTS or EXPORT_TABLES .
3. Drop the projections, then recreate them as defined in the exported DDL.
4. Run REFRESH. Vertica rebuilds the projections with new storage containers.

Verifying catalog memory consumption

Vertica versions ≥ 9.2 significantly reduce how much memory database catalogs consume. After you upgrade, check catalog memory consumption on each node to verify that the upgrade refactored catalogs correctly. If memory consumption for a given catalog is as large as or larger than it was in the earlier database, restart the host node.

Known issues

Certain operations might significantly inflate catalog memory consumption. For example:

- You created a backup on a 9.1.1 database and restored objects from the backup to a new database of version ≥ 9.2.
- You replicated objects from a 9.1.1 database to a database of version ≥ 9.2.

To refactor database catalogs and reduce their memory footprint, restart the database.

Reinstalling packages

In most cases, Vertica automatically reinstalls all default packages when you restart your database for the first time after running the upgrade script. Occasionally, however, one or more packages might fail to reinstall correctly.

To verify that Vertica succeeded in reinstalling all packages:

1. Restart the database after upgrading.
2. Enter a correct password.

If any packages failed to reinstall, Vertica issues a message that specifies the uninstalled packages. In this case, run the admintools command install_package with the option --force-reinstall :

```
$ admintools -t install_package -d db-name -p password -P pkg-spec --force-reinstall
```

Options

Option	Function
-d *db-name* --dbname= * db-name	Database name
-p *password* --password= *pword*	Database administrator password
-P *pkg* --package= *pkg-spec*	Specifies which packages to install, where *pkg* is one of the following: • The name of a package—for example, flextable • all : All available packages • default : All default packages that are currently installed
--force-reinstall	Force installation of a package even if it is already installed.

Examples

Force reinstallation of default packages:

```
$ admintools -t install_package -d VMart -p 'password' -P default --force-reinstall
```

Force reinstallation of one package, flextable :

```
$ admintools -t install_package -d VMart -p 'password' -P flextable --force-reinstall
```

Writing bundle metadata to the catalog

Vertica internally stores physical table data in bundles together with metadata on the bundle contents. The query optimizer uses bundle metadata to look up and fetch the data it needs for a given query.

Vertica stores bundle metadata in the database catalog. This is especially beneficial in Eon mode: instead of fetching this metadata from remote (S3) storage, the optimizer can find it in the local catalog. This minimizes S3 reads, and facilitates faster query planning and overall execution.

Vertica writes bundle metadata to the catalog on two events:

- Any DML operation that changes table content, such as INSERT , UPDATE , or COPY . Vertica writes bundle metadata to the catalog on the new or changed table data. DML operations have no effect on bundle metadata for existing table data.
- Invocations of function UPDATE_STORAGE_CATALOG , as an argument to Vertica meta-function DO_TM_TASK , on existing data. You can narrow the scope of the catalog update operation to a specific projection or table. If no scope is specified, the operation is applied to the entire database.
  Important
  After upgrading to any Vertica version ≥ 9.2.1, you only need to call UPDATE_STORAGE_CATALOG once on existing data. Bundle metadata on all new or updated data is always written automatically to the catalog.

For example, the following DO_TM_TASK call writes bundle metadata on all projections in table store.store_sales_fact :

```
=> SELECT DO_TM_TASK ('update_storage_catalog', 'store.store_sales_fact');
                              do_tm_task
-------------------------------------------------------------------------------
 Task: update_storage_catalog
(Table: store.store_sales_fact) (Projection: store.store_sales_fact_b0)
(Table: store.store_sales_fact) (Projection: store.store_sales_fact_b1)
(1 row)
```

Validating bundle metadata

You can query system table STORAGE_BUNDLE_INFO_STATISTICS to determine which projections have invalid bundle metadata in the database catalog. For example, results from the following query show that the database catalog has invalid metadata for projections inventory_fact_b0 and inventory_fact_b1 :

```
=> SELECT node_name, projection_name, total_ros_count, ros_without_bundle_info_count
   FROM v_monitor.storage_bundle_info_statistics where ros_without_bundle_info_count > 0
   ORDER BY projection_name, node_name;
   node_name     |  projection_name  | total_ros_count | ros_without_bundle_info_count
-----------------+-------------------+-----------------+-------------------------------
 v_vmart_node0001 | inventory_fact_b0 |               1 |                             1
 v_vmart_node0002 | inventory_fact_b0 |               1 |                             1
 v_vmart_node0003 | inventory_fact_b0 |               1 |                             1
 v_vmart_node0001 | inventory_fact_b1 |               1 |                             1
 v_vmart_node0002 | inventory_fact_b1 |               1 |                             1
 v_vmart_node0003 | inventory_fact_b1 |               1 |                             1
(6 rows)
```

Best practices

Updating the database catalog with UPDATE_STORAGE_CATALOG is recommended only for Eon users. Enterprise users are unlikely to see measurable performance improvements from this update.

Calls to UPDATE_STORAGE_CATALOG can incur considerable overhead, as the update process typically requires numerous and expensive S3 reads. Vertica advises against running this operation on the entire database. Instead, consider an incremental approach:

- Call UPDATE_STORAGE_CATALOG on a single large fact table. You can use performance metrics to estimate how much time updating other files will require.
- Identify which tables are subject to frequent queries and prioritize catalog updates accordingly.

Upgrading the streaming data scheduler utility

If you have integrated Vertica with a streaming data application, such as Apache Kafka, you must update the streaming data scheduler utility after you update Vertica.

From a command prompt, enter the following command:

```
/opt/vertica/packages/kafka/bin/vkconfig scheduler --upgrade --upgrade-to-schema schema_name
```

Running the upgrade task more than once has no effect.

For more information on the Scheduler utility, refer to Scheduler tool options .

Uninstall Vertica

To uninstall Vertica, perform the following steps for each host in the cluster:

1. Choose a host machine and log in as root (or log in as another user and switch to root).
   ```
   $ su - root
   password: root-password
   ```
2. Find the name of the package that is installed:
   RPM:
   ```
   # rpm -qa | grep vertica
   ```
   DEB:
   ```
   # dpkg -l | grep vertica
   ```
3. Remove the package:
   RPM:
   ```
   # rpm -e package
   ```
   DEB:
   ```
   # dpkg -r package
   ```

> **Note**
> If you want to delete the configuration file used with your installation, you can choose to delete the /opt/vertica/ directory and all subdirectories using this command: # rm -rf /opt/vertica/

Perform the following steps for each client system:

1. Delete the JDBC driver jar file.
2. Delete ODBC driver data source names.
3. Delete the ODBC driver software:
    1. In Windows, go to **Start > Control Panel > Add or Remove Programs** .
    2. Locate Vertica.
    3. Click **Remove** .

Eon Mode

You can operate your Vertica database in Eon Mode instead of in Enterprise Mode. The two modes differ primarily in how they store data:

- Eon Mode databases use communal storage for their data.
- Enterprise Mode databases store data locally in the file system of nodes that make up the database.

These different storage methods lead to a number of important differences between the two modes. In Enterprise Mode, each database node stores a portion of the data and performs a portion of the computation. In Eon Mode, computational processes are separated from a communal (shared) storage layer, which enables rapid scaling of computational resources as demand changes.

For more on how these two modes compare, see Architecture .

In this section

- Create a database in Eon Mode
- Configuring your Vertica cluster for Eon Mode
- Migrating an enterprise database to Eon Mode
- Managing subclusters
- Depot management
- Scaling your Eon Mode database
- Subcluster sandboxing
- Local caching of storage containers
- Managing an Eon Mode database in MC
- Stopping and starting an Eon Mode cluster
- Terminating an Eon Mode database cluster
- Reviving an Eon Mode database cluster
- Synchronizing metadata

Create a database in Eon Mode

Create an Eon Mode database in a cloud environment

The easiest way to create an Eon Mode database in the cloud is to use the MC. The MC can create your database and provision the nodes to run the database at the same time. For specific instructions for your cloud environment, see:

- Amazon Web Services (AWS): Amazon Web Services in MC
- Google Cloud Platform (GCP): Google Cloud Platform in MC
- Microsoft Azure: Microsoft Azure in MC

On AWS and Azure, you can also create an Eon Mode database using admintools. For specific instructions for your cloud environment, see:

- AWS: Create an Eon Mode Database
- Azure: Manually create an Eon Mode database on Azure

Create an on-premises Eon Mode database

If you have an on-premises install, you can create an Eon Mode database using admintools. See Eon on-premises storage for a list of object stores that Vertica supports for communal storage. The following topics detail installation instructions for each on-premises communal storage option:

- Create an Eon Mode database on-premises with FlashBlade
- Create an Eon Mode database on-premises with MinIO
- Create an Eon Mode database on-premises with HDFS

In this section

- Create an Eon Mode database on-premises with FlashBlade
- Create an Eon Mode database on-premises with HDFS

- Create an Eon Mode database on-premises with MinIO
- Manually create an Eon Mode database on Azure

Create an Eon Mode database on-premises with FlashBlade

You have two options on how to create an Eon Mode database on premises with Pure Storage FlashBlade as your S3-compatible communal storage:

- **Using the admintools Command Line:** Use the Vertica admintools command line, and complete all the steps in this topic.
- **Using Management Console** : Execute the steps in Creating an Eon Mode database on premises with FlashBlade in MC .

Step 1: create a bucket and credentials on the Pure Storage FlashBlade

To use a Pure Storage FlashBlade appliance as a communal storage location for an Eon Mode database you must have:

- The IP address of the FlashBlade appliance. You must also have the connection port number if your FlashBlade is not using the standard port 80 or 443 to access the bucket. All of the nodes in your Vertica cluster must be able to access this IP address. Make sure any firewalls between the FlashBlade appliance and the nodes are configured to allow access.
- The name of the bucket on the FlashBlade to use for communal storage.
- An access key and secret key for a user account that has read and write access to the bucket.

See the Pure Storage support site for instructions on how to create the bucket and the access keys needed for a communal storage location.

Step 2: install Vertica on your cluster

To install Vertica:

1. Ensure your nodes are configured properly by reviewing all of the content in the Before you install Vertica section.
2. Use the install_vertica script to verify that your nodes are correctly configured and to install the Vertica binaries on all of your nodes. Follow the steps under Install Vertica using the command line to install Vertica.

> **Note**
> These installation steps are the same ones you follow to install Vertica in Enterprise Mode. The difference between Eon Mode and Enterprise Mode on-premises databases is how you create the database, not how you install the Vertica software.

Step 3: create an authorization file

Before you create your Eon Mode on-premises database, you must create an authorization file that admintools will use to authenticate with the FlashBlade storage.

1. On the Vertica node where you will run admintools to create your database, use a text editor to create a file. You can name this file anything you wish. In these steps, it is named auth_params.conf . The location of this file isn't important, as long as it is readable by the Linux user you use to create the database (usually, dbadmin).
   Important
   The auth_params.conf file contains the secret key to access the bucket containing your Eon Mode database's data. This information is sensitive, and can be used to access the raw data in your database. Be sure this file is not readable by unauthorized users. After you have created your database, you can delete this file.

2. Add the following lines to the file:
   ```
   awsauth = FlasbBlade_Access_Key:FlashBlade_Secret_Key
   awsendpoint = FlashBladeIp:FlashBladePort
   ```

   > **Note**
   > You do not need to supply a port number in the awsendpoint setting if you are using the default port for the connection between Vertica and the FlashBlade (80 for an unencrypted connection or 443 for an encrypted connection).

3. If you are not using TLS encryption for the connection between Vertica and the FlashBlade, add the following line to the file:
   ```
   awsenablehttps = 0
   ```

4. Save the file and exit the editor.

This example auth_params.conf file is for an unencrypted connection between the Vertica cluster and a FlashBlade appliance at IP address 10.10.20.30 using the standard port 80.

```
awsauth = PIWHSNDGSHVRPIQ:339068001+e904816E02E5fe9103f8MQOEAEHFFVPKBAAL
awsendpoint = 10.10.20.30
awsenablehttps = 0
```

Step 4: choose a depot path on all nodes

Choose or create a directory on each node for the depot storage path. The directory you supply for the depot storage path parameter must:

- Have the same path on all nodes in the cluster (i.e. /home/dbadmin/depot).
- Be readable and writable by the dbadmin user.
- Have sufficient storage. By default, Vertica uses 60% of the filesystem space containing the directory for depot storage. You can limit the size of the depot by using the --depot-size argument in the create_db command. See Configuring your Vertica cluster for Eon Mode for guidelines on choosing a size for your depot.

The admintools create_db tool will attempt to create the depot path for you if it doesn't exist.

Step 5: create the Eon on-premises database

Use the admintools create_db tool to create the database. You must pass this tool the following arguments:

Argument	Description
-x	The path to the auth_params.conf file.
--communal-storage-location	The S3 URL for the bucket on the FlashBlade appliance (usually, this is s3:// *bucketname*).
--depot-path	The absolute path to store the depot on the nodes in the cluster.
--shard-count	The number of shards for the database. This is an integer number that is usually either a multiple of the number of nodes in your cluster, or an even divider. See Planning for Scaling Your Cluster for more information.
-s	A comma-separated list of the nodes in your database.
-d	The name for your database.

Some common optional arguments include:

Argument	Description
-l	The absolute path to the Vertica license file to apply to the new database.
-p	The password for the new database.
--depot-size	The maximum size for the depot. Defaults to 60% of the filesystem containing the depot path. You can specify the size in two ways: • *integer* % : Percentage of filesystem's disk space to allocate. • *integer* {K|M|G|T} : Amount of disk space to allocate for the depot in kilobytes, megabytes, gigabytes, or terabytes. However you specify this value, the depot size cannot be more than 80 percent of disk space of the file system where the depot is stored.

To view all arguments for the create_db tool, run the command:

```
admintools -t create_db --help
```

The following example demonstrates creating a three-node database named verticadb, specifying the depot will be stored in the home directory of the dbadmin user.

```
$ admintools -t create_db -x auth_params.conf \
  --communal-storage-location=s3://verticadbbucket \
  --depot-path=/home/dbadmin/depot --shard-count=6 \
  -s vnode01,vnode02,vnode03 -d verticadb -p 'YourPasswordHere'
```

Step 6: disable streaming limitations

After creating the database, disable the AWSStreamingConnectionPercentage configuration parameter. This setting is unnecessary for an Eon Mode on-premises install with communal storage on FlashBlade or MinIO. This configuration parameter controls the number of connections to the object store that Vertica uses for streaming reads. In a cloud environment, this setting helps avoid having streaming data from the object store use up all of the available file handles. It leaves some file handles available for other object store operations. Due to the low latency of on-premises object stores, this option is unnecessary. Set it to 0 to disable it.

The following example shows how to disable this parameter using ALTER DATABASE...SET PARAMETER :

```
=> ALTER DATABASE DEFAULT SET PARAMETER AWSStreamingConnectionPercentage = 0;
ALTER DATABASE
```

Deciding whether to disable the depot

The FlashBlade object store's performance is fast enough that you may consider disabling the depot in your Vertica database. If you disable the depot, you can get by with less local storage on your nodes. However, there is always a performance impact of disabling the depot. The exact impact depends mainly on the types of workloads you run on your database. The performance impact can range from a 30% to 4000% decrease in query performance. Only consider disabling the depot if you will see a significant benefit from reducing the storage requirements of your nodes. Before disabling the depot on a production database, always run a proof of concept test that executes the same workloads as your production database.

To disable the depot, set the UseDepotForReads configuration parameter to 0. The following example demonstrates disabling this parameter using ALTER DATABASE...SET PARAMETER :

```
=> ALTER DATABASE DEFAULT SET PARAMETER UseDepotForReads = 0;
ALTER DATABASE
```

Create an Eon Mode database on-premises with HDFS

Step 1: satisfy HDFS environment prerequisites

To use HDFS as a communal storage location for an Eon Mode database you must:

- Run the WebHDFS service.
- If using Kerberos, create a Kerberos principal for the Vertica (system) user as described in Kerberos authentication , and grant it read and write access to the location in HDFS where you will place your communal storage. Vertica always uses this system principal to access communal storage.
- If using High Availability Name Node or swebhdfs, distribute the HDFS configuration files to all Vertica nodes as described in Configuring HDFS access . This step is necessary even though you do not use the hdfs scheme for communal storage.
- If using swebhdfs (wire encryption) instead of webhdfs, configure the HDFS cluster with certificates trusted by the Vertica hosts and set dfs.encrypt.data.transfer in hdfs-site.xml.
- Vertica has no additional requirements for encryption at rest. Consult the documentation for your Hadoop distribution for information on how to configure encryption at rest for WebHDFS.

> **Note**
> Hadoop currently does not support IPv6 network addresses. Your cluster must use IPv4 addresses to access HDFS. If you choose to use IPv6 network addresses for the hosts in your database cluster, make sure they can access IPv4 addresses. One way to enable this access is to assign your Vertica hosts an IPv4 address in addition to an IPv6 address.

Step 2: install Vertica on your cluster

To install Vertica:

1. Ensure your nodes are configured properly by reviewing all of the content in the Before you install Vertica section.
2. Use the install_vertica script to verify that your nodes are correctly configured and to install the Vertica binaries on all of your nodes. Follow the steps under Install Vertica using the command line to install Vertica.

> **Note**
> These installation steps are the same ones you follow to install Vertica in Enterprise Mode. The difference between Eon Mode and Enterprise Mode on-premises databases is how you create the database, not how you install the Vertica software.

Step 3: create a bootstrapping file

Before you create your Eon Mode on-premises database, you must create a bootstrapping file to specify parameters that are required for database creation. This step applies if you are using Kerberos, High Availability Name Node, or TLS (wire encryption).

1. On the Vertica node where you will run admintools to create your database, use a text editor to create a file. You can name this file anything you wish. In these steps, it is named bootstrap_params.conf . The location of this file isn't important, as long as it is readable by the Linux user you use to create the database (usually, dbadmin).
2. Add the following lines to the file. HadoopConfDir is typically set to /etc/hadoop/conf ; KerberosServiceName is usually set to vertica .

   ```
   HadoopConfDir = config-path
   KerberosServiceName = principal-name
   KerberosRealm = realm-name
   KerberosKeytabFile = keytab-path
   ```

   If you are not using HA Name Node, for example in a test environment, you can omit HadoopConfDir and use an explicit Name Node host and port when specifying the location of the communal storage.
3. Save the file and exit the editor.

Step 4: choose a depot path on all nodes

Choose or create a directory on each node for the depot storage path. The directory you supply for the depot storage path parameter must:

- Have the same path on all nodes in the cluster (i.e. /home/dbadmin/depot).
- Be readable and writable by the dbadmin user.
- Have sufficient storage. By default, Vertica uses 60% of the filesystem space containing the directory for depot storage. You can limit the size of the depot by using the --depot-size argument in the create_db command. See Configuring your Vertica cluster for Eon Mode for guidelines on choosing a size for your depot.

The admintools create_db tool will attempt to create the depot path for you if it doesn't exist.

Step 5: create the Eon on-premises database

Use the admintools create_db tool to create the database. You must pass this tool the following arguments:

Argument	Description
-x	The path to the bootstrap configuration file (bootstrap_params.conf in the examples in this section).
--communal-storage-location	The webhdfs or swebhdfs URL for the HDFS location. You cannot use the hdfs scheme.
--depot-path	The absolute path to store the depot on the nodes in the cluster.
--shard-count	The number of shards for the database. This is an integer number that is usually either a multiple of the number of nodes in your cluster, or an even divider. See Planning for Scaling Your Cluster for more information.
-s	A comma-separated list of the nodes in your database.
-d	The name for your database.

Some common optional arguments include:

Argument	Description
-l	The absolute path to the Vertica license file to apply to the new database.
-p	The password for the new database.

--depot-size	The maximum size for the depot. Defaults to 60% of the filesystem containing the depot path. You can specify the size in two ways: • *integer* % : Percentage of filesystem's disk space to allocate. • *integer* {K|M|G|T} : Amount of disk space to allocate for the depot in kilobytes, megabytes, gigabytes, or terabytes. However you specify this value, the depot size cannot be more than 80 percent of disk space of the file system where the depot is stored.

To view all arguments for the create_db tool, run the command:

```
admintools -t create_db --help
```

The following example demonstrates creating a three-node database named verticadb, specifying the depot will be stored in the home directory of the dbadmin user.

```
$ admintools -t create_db -x bootstrap_params.conf \
 --communal-storage-location=webhdfs://mycluster/verticadb \
 --depot-path=/home/dbadmin/depot  --shard-count=6 \
 -s vnode01,vnode02,vnode03 -d verticadb -p 'YourPasswordHere'
```

If you are not using HA Name Node, for example in a test environment, you can use an explicit Name Node host and port for --communal-storage-location as in the following example.

```
$ admintools -t create_db -x bootstrap_params.conf \
 --communal-storage-location=webhdfs://namenode.hadoop.example.com:50070/verticadb \
 --depot-path=/home/dbadmin/depot  --shard-count=6 \
 -s vnode01,vnode02,vnode03 -d verticadb -p 'YourPasswordHere'
```

Create an Eon Mode database on-premises with MinIO

Step 1: create a bucket and credentials on MinIO

To use MinIO as a communal storage location for an Eon Mode database, you must have:

- The IP address and port number of the MinIO cluster. MinIO's default port number is 9000. A Vertica database running in Eon Mode defaults to using port 80 for unencrypted connections and port 443 for TLS encrypted connection. All of the nodes in your Vertica cluster must be able to access the MinIO cluster's IP address. Make sure any firewalls between the MinIO cluster and the nodes are configured to allow access.
- The name of the bucket on the MinIO cluster to use for communal storage.
- An access key and secret key for a user account that has read and write access to the bucket.

See the MinIO documentation for instructions on how to create the bucket and the access keys needed for a communal storage location.

Step 2: install Vertica on your cluster

To install Vertica:

1. Ensure your nodes are configured properly by reviewing all of the content in the Before you install Vertica section.
2. Use the install_vertica script to verify that your nodes are correctly configured and to install the Vertica binaries on all of your nodes. Follow the steps under Install Vertica using the command line to install Vertica.

> **Note**
> These installation steps are the same ones you follow to install Vertica in Enterprise Mode. The difference between Eon Mode and Enterprise Mode on-premises databases is how you create the database, not how you install the Vertica software.

Step 3: create an authorization file

Before you create your Eon Mode on-premises database, you must create an authorization file that admintools will use to authenticate with the MinIO storage cluster.

1. On the Vertica node where you will run admintools to create your database, use a text editor to create a file. You can name this file anything you wish. In these steps, it is named auth_params.conf . The location of this file isn't important, as long as it is readable by the Linux user you use to create the database (usually, dbadmin).
   Important
   The auth_params.conf file contains the secret key to access the bucket containing your Eon Mode database's data. This information is sensitive, and can be used to access the raw data in your database. Be sure this file is not readable by unauthorized users. After you have created your database, you can delete this file.

2. Add the following lines to the file:
   ```
   awsauth = MinIO_Access_Key:MinIO_Secret_Key
   awsendpoint = MinIOIp:MinIOPort
   ```

   Note
   You do not need to supply a port number in the awsendpoint setting if you configured your MinIO cluster to use the default HTTP ports (80 for an unencrypted connection or 443 for an encrypted connection). MinIO uses port 9000 by default.

3. If you are not using TLS encryption for the connection between Vertica and MinIO, add the following line to the file:
   ```
   awsenablehttps = 0
   ```

4. Save the file and exit the editor.

This example auth_params.conf file is for an unencrypted connection between the Vertica cluster and a MinIO cluster at IP address 10.20.30.40 using port 9000 (which is the default for MinIO).

```
awsauth = PIWHSNDGSHVRPIQ:339068001+e904816E02E5fe9103f8MQOEAEHFFVPKBAAL
awsendpoint = 10.20.30.40:9000
awsenablehttps = 0
```

Step 4: choose a depot path on all nodes

Choose or create a directory on each node for the depot storage path. The directory you supply for the depot storage path parameter must:

- Have the same path on all nodes in the cluster (i.e. /home/dbadmin/depot).
- Be readable and writable by the dbadmin user.
- Have sufficient storage. By default, Vertica uses 60% of the filesystem space containing the directory for depot storage. You can limit the size of the depot by using the --depot-size argument in the create_db command. See Configuring your Vertica cluster for Eon Mode for guidelines on choosing a size for your depot.

The admintools create_db tool will attempt to create the depot path for you if it doesn't exist.

Step 5: create the Eon on-premises database

Use the admintools create_db tool to create the database. You must pass this tool the following arguments:

Argument	Description
-x	The path to the auth_params.conf file.
--communal-storage-location	The S3 URL for the bucket on the MinIO cluster (usually, this is s3:// *bucketname*).
--depot-path	The absolute path to store the depot on the nodes in the cluster.
--shard-count	The number of shards for the database. This is an integer number that is usually either a multiple of the number of nodes in your cluster, or an even divider. See Planning for Scaling Your Cluster for more information.
-s	A comma-separated list of the nodes in your database.
-d	The name for your database.

Some common optional arguments include:

Argument	Description
-l	The absolute path to the Vertica license file to apply to the new database.
-p	The password for the new database.
--depot-size	The maximum size for the depot. Defaults to 60% of the filesystem containing the depot path. You can specify the size in two ways: • *integer* % : Percentage of filesystem's disk space to allocate. • *integer* {K|M|G|T} : Amount of disk space to allocate for the depot in kilobytes, megabytes, gigabytes, or terabytes. However you specify this value, the depot size cannot be more than 80 percent of disk space of the file system where the depot is stored.

To view all arguments for the create_db tool, run the command:

```
admintools -t create_db --help
```

The following example demonstrates creating a three-node database named verticadb, specifying the depot will be stored in the home directory of the dbadmin user.

```
$ admintools -t create_db -x auth_params.conf \
 --communal-storage-location=s3://verticadbbucket \
 --depot-path=/home/dbadmin/depot  --shard-count=6 \
 -s vnode01,vnode02,vnode03 -d verticadb -p 'YourPasswordHere'
```

Step 6: disable streaming limitations

After creating the database, disable the AWSStreamingConnectionPercentage configuration parameter. This setting is unnecessary for an Eon Mode on-premises install with communal storage on FlashBlade or MinIO. This configuration parameter controls the number of connections to the object store that Vertica uses for streaming reads. In a cloud environment, this setting helps avoid having streaming data from the object store use up all of the available file handles. It leaves some file handles available for other object store operations. Due to the low latency of on-premises object stores, this option is unnecessary. Set it to 0 to disable it.

The following example shows how to disable this parameter using ALTER DATABASE...SET PARAMETER :

```
=> ALTER DATABASE DEFAULT SET PARAMETER AWSStreamingConnectionPercentage = 0;
ALTER DATABASE
```

Manually create an Eon Mode database on Azure

Once you have met the cluster and storage requirements for using an Eon Mode database on Azure, you are ready to create an Eon Mode database. Use the admintools create_db tool to create your Eon Mode database.

Creating an authentication file

If your database will use a managed identity to authenticate with the Azure storage container, you do not need to supply any additional configuration information to the create_db tool.

If your database will not use a managed identity, you must supply create_db with authentication information in a configuration file. It must contain at least the AzureStorageCredentials parameter that defines one or more account names and keys Vertica will use to access blob storage. It can also contain an AzureStorageEnpointConfig parameter that defines an alternate endpoint to use instead of the the default Azure host name. This option is useful if you are creating a test environment using an Azure storage emulator such as Azurite.

Important
Vertica does not officially support Azure storage emulators as a communal storage location.

The following table defines the values that can be set in these two parameters.

AzureStorageCredentials

Collection of JSON objects, each of which specifies connection credentials for one endpoint. This parameter takes precedence over Azure

managed identities.

The collection must contain at least one object and may contain more. Each object must specify at least one of accountName or blobEndpoint , and at least one of accountKey or sharedAccessSignature .

- accountName : If not specified, uses the label of blobEndpoint .
- blobEndpoint : Host name with optional port (host:port). If not specified, uses *account* .blob.core.windows.net .
- accountKey : Access key for the account or endpoint.
- sharedAccessSignature : Access token for finer-grained access control, if being used by the Azure endpoint.

AzureStorageEndpointConfig

Collection of JSON objects, each of which specifies configuration elements for one endpoint. Each object must specify at least one of accountName or blobEndpoint .

- accountName : If not specified, uses the label of blobEndpoint .
- blobEndpoint : Host name with optional port (host:port). If not specified, uses *account* .blob.core.windows.net .
- protocol : HTTPS (default) or HTTP.
- isMultiAccountEndpoint : true if the endpoint supports multiple accounts, false otherwise (default is false). To use multiple-account access, you must include the account name in the URI. If a URI path contains an account, this value is assumed to be true unless explicitly set to false.

The authentication configuration file is a text file containing the configuration parameter names and their values. The values are in a JSON format. The name of this file is not important. The following examples use the file name auth_params.conf .

The following example is a configuration file for a storage account hosted on Azure. The storage account name is mystore, and the key value is a placeholder. In your own configuration file, you must provide the storage account's access key. You can find this value by right-clicking the storage account in the Azure Storage Explorer and selecting **Copy Primary Key** .

```
AzureStorageCredentials=[{"accountName": "mystore", "accountKey": "access-key"}]
```

The following example shows a configuration file that defines an account for a storage container hosted on the local system using the Azurite storage system. The user account and key are the "well-known" account provided by Azurite by default. Because this configuration uses an alternate storage endpoint, it also defines the AzureStorageEndpointConfig parameter. In addition to reiterating the account name and endpoint definition, this example sets the protocol to the non-encrypted HTTP.

Important
This example wraps the contents of the JSON values for clarity. In an actual configuration file, you **cannot wrap these values** . They must be on a single line.

```
AzureStorageCredentials=[{"accountName": "devstoreaccount1", "blobEndpoint": "127.0.0.1:10000 ",
                          "accountKey":
"Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw=="
                         }]

AzureStorageEndpointConfig=[{"accountName": "devstoreaccount1",
                             "blobEndpoint": "127.0.0.1:10000", "protocol": "http"}]
```

Creating the Eon Mode database

Use the admintools create_db tool to create your Eon Mode database. The required arguments you pass to this tool are:

Argument	Description
--communal-storage-location	The URI for the storage container Vertica will use for communal storage. This URI must use the azb:// schema. See Azure Blob Storage object store for the format of this URI.
-x	The path to the file containing the authentication parameters Vertica needs to access the communal storage location. This argument is only required if your database will use a storage account name and key to authenticate with the storage container. If it is using a managed identity, you do not need to specify this argument.

--depot-path	The absolute path to store the depot on the nodes in the cluster.
--shard-count	The number of shards for the database. This is an integer number that is usually either a multiple of the number of nodes in your cluster, or an even divisor. See Planning for Scaling Your Cluster for more information.
-s	A comma-separated list of the nodes in your database.
-d	The name for your database.

Some other common optional arguments for create_db are:

Argument	Description
-l	The absolute path to the Vertica license file to apply to the new database.
-p	The password for the new database.
--depot-size	The maximum size for the depot. Defaults to 60% of the filesystem containing the depot path. You can specify the size in two ways: • *integer* % : Percentage of filesystem's disk space to allocate. • *integer* {K|M|G|T} : Amount of disk space to allocate for the depot in kilobytes, megabytes, gigabytes, or terabytes. However you specify this value, the depot size cannot be more than 80 percent of disk space of the file system where the depot is stored.

To view all arguments for the create_db tool, run the command:

```
admintools -t create_db --help
```

The following example demonstrates creating an Eon Mode database with the following settings:

- Vertica will use a storage account named mystore.
- The communal data will be stored in a directory named verticadb located in a storage container named db_blobs .
- The authentication information Vertica needs to access the storage container is in the file named auth_params.conf in the current directory. The contents of this file are shown in the first example under Creating an Authentication File .
- The hostnames of the nodes in the cluster are node01 through node03.

```
$ admintools -t create_db \
        --communal-storage-location=azb://mystore/db_blobs/verticadb \
        -x auth_params.conf -s node01,node02,node03  \
        -d verticadb --depot-path /vertica/depot --shard-count 3 \
        -p 'mypassword'
```

Configuring your Vertica cluster for Eon Mode

Running Vertica in Eon Mode decouples the cluster size from the data volume and lets you configure for your compute needs independently from your storage needs. There are a number of factors you must consider when deciding what sorts of instances to use and how large your Eon Mode cluster will be.

Before you begin

Vertica Eon Mode works both in the cloud and on-premises. As a Vertica administrator setting up your production cluster running in Eon Mode, you must make decisions about the virtual or physical hardware you will use to meet your needs. This topic provides guidelines and best practices for selecting server types and cluster sizes for a Vertica database running in Eon Mode.It assumes that you have a basic understanding of the Eon Mode architecture and concepts such as communal storage , depot , and shards . If you need to refresh your understanding, see Eon Mode architecture .

Cluster size overview

Because Eon Mode separates data storage from the computing power of your nodes, choosing a cluster size is more complex than for an Enterprise Mode database. Usually, you choose a base number of nodes that will form one or more primary subclusters . These subclusters contain nodes that are always running in your database. You usually use them for workloads such as data loading and executing DDL statements. You rarely alter the size of these subclusters dynamically. As you need additional compute resources to execute queries, you add one or more subclusters (usually secondary subclusters) of nodes to your database.

When choosing your instances and sizing your cluster for Eon Mode, consider the working data size that your database will be dealing with. This is the amount of data that most of your queries operate on. For example, suppose your database stores sales data. If most of the queries running on your database analyze the last month or two of sales to create reports on sales trends, then your working data size is the amount of data you typically load over a few months.

Choosing instances or physical hardware

Depending on the complexity of your workload and expected concurrency, choose physical or virtual hardware that has sufficient CPU and memory. For production clusters Vertica recommends the following minimum configuration for either virtual or physical nodes in an Eon Mode database:

- 16 cores
- 128 GB RAM
- 2 TB of local storage

> **Note**
> The above specifications are just a minimum recommendation. You should consider increasing these specifications based on your workloads and the amount of data you are processing.

You must have a minimum of 3 nodes in the an Eon Mode database cluster.

For specific recommendations of instances for cloud-based Eon Mode database, see:

- Choosing AWS Eon Mode Instance Types
- GCP Eon Mode instance recommendations

Determining local storage requirements

For both virtual and physical hardware, you must decide how much local storage your nodes need. In Eon Mode, the definitive copy of your database's data resides in the communal storage. This storage is provided by either a cloud-based object store such as AWS S3, or by an on-premises object store, such as a Pure Storage FlashBlade appliance.

Even though your database's data is stored in communal storage, your nodes still need some local storage. A node in an Eon Mode database uses local storage for three purposes:

- **Depot storage** : To get the fastest response time for frequently executed queries, provision a depot large enough to hold your working data set after data compression. Divide the working data size by the number of nodes you will have in your subcluster to estimate the size of the depot you need for each node in a subcluster. See Choosing the Number of Shards and the Initial Node Count below to get an idea of how many nodes you want in your initial database subcluster. In cases where you expect to dynamically scale your cluster up, estimate the depot size based on the minimum number of nodes you anticipate having in the subcluster.
  Also consider how much data you will load at once when sizing your depot. When loading data, Vertica defaults to writing uncommitted ROS files into the depot before uploading the files to communal storage. If the free space in the depot is not sufficient, Vertica evicts files from the depot to make space for new files.
  Your data load fails if the amount of data you try to load in a single transaction is larger the total sizes of all the depots in the subcluster. To load more data than there is space in the subcluster's combined depots, set UseDepotForWrites to 0. This configuration parameter tells Vertica to load the data directly into communal storage.
- **Data storage** : The data storage location holds files that belong to temporary tables and temporary data from sort operators that spill to disk. When loading data into Vertica, the sort operator may spill to disk. Depending on the size of the load, Vertica may perform the sort in multiple merge phases. The amount of data concurrently loaded into Vertica cannot be larger than the sum of temporary storage location sizes across all nodes divided by 2.
- **Catalog storage** . The catalog size depends on the number of database objects per shard and the number of shard subscriptions per node.

Vertica recommends a minimum local storage capacity of 2 TB per node, out of which 60% is reserved for the depot and the other 40% is shared between the catalog and data location. If you determine that you need a depot larger than 1.2TB per node (which is 60% of 2TB) then add more storage than this minimum recommendation.You can calculate the space you need using this equation:

$$\text{Disk Space Per Node} = \frac{\text{Compressed Working Data Size}}{\text{\# of Nodes in Subcluster}} \times 1.67$$

For example, suppose you have a compressed working data size of 24TB, and you want to have a initial primary subcluster of 3 nodes. Using these values in the equation results in 13.33TB:

$$\frac{24\text{TB}}{3\text{ Nodes}} \times 1.67 = 13.33\text{TB}$$

Choosing the number of shards and the initial node count

Shards are how Vertica divides the responsibility for the data in communal storage among nodes. Each node in a subcluster subscribes to at least one shard in communal storage. During queries, data loads, and other database tasks, each node is responsible for the data in the shards it subscribes to. See Shards and subscriptions for more information.

The relation between shards and nodes means that when selecting the number of shards for your database, you must consider the number of nodes you will have in your database.

You set the initial number of shards when you create your database. Should your cluster size or usage patterns change in the future, you can call RESHARD_DATABASE to change the number of shards. For details, see Change the number of shards in the database .

The initial node count is the number of nodes you will have in your core primary subcluster. The number of shards should always be a multiple or divisor of the node count. This ensures that the shards are evenly divided between the nodes. For example, in a six-shard database, you should have subclusters that contain two, three, six, or a multiple of six nodes. If the number of shards is not a divisor or multiple of the node count, the shard subscriptions are not spread evenly across the nodes. This leads to some nodes being more heavily loaded than others.

> **Note**
> In a database where the number of nodes is a multiple of the number of shards, the subcluster uses Elastic Crunch Scaling (ECS) to evenly divide shard coverage among two or more nodes. For more information, see Using elastic crunch scaling to improve query performance .

When choosing the number of shards, consider how you might expand or contract the number of nodes in your subclusters. Certain number of shards allow for greater flexibility. For example, if you have seven shards in your database, the shards can be equally divided only among subclusters with a multiple of seven number of nodes. With a database containing 12 shards, the shards can be equally distributed in subclusters that have 2, 3, 4, 6, 12, or a multiple of 12 nodes.

The following table shows the recommended shard count and initial node count based on the working data size:

Cluster Type	Working Data Size	Number of Shards	Initial Node Count
Small	Up to 24 TB	6	3
Medium	Up to 48 TB	12	6
Large	Up to 96 TB	24	12
Extra large	Up to 192 TB	48	24

Important
A 2:1 ratio for shards to nodes is a performance recommendation, rather than a hard limit. If you attempt to go higher than 3:1, MC offers a warning to make sure you have taken all aspects of shard count into consideration.

How shard count affects scaling your cluster

The number of shards you choose for your database impacts your ability to scale your database in the future. If you have too few shards, your ability to efficiently scale your database can be limited. If you have too many shards, your database performance can suffer. A larger number of shards increases inter-node communication and catalog complexity.

One key factor in deciding your shard count is determining how you want to scale your database. There are two strategies you can use when adding nodes to your database. Each of these strategies let you improve different types of database performance:

- To increase the performance of complex, long-running queries, add nodes to an existing subcluster. These additional nodes improve the overall performance of these complex queries by splitting the load across more compute nodes. You can add more nodes to a subcluster than you have shards in the database. In this case, nodes in the subcluster that subscribe to the same shard will split up the data in the shard when performing a query. See Elasticity for more information.
- To increase the throughput of multiple short-term queries (often called "dashboard queries"), improve your cluster's parallelism by adding additional subclusters . Subclusters work independently and in parallel on these shorter queries. See Subclusters for more information.

These two approaches have an impact on the number of shards you choose to start your database with. Complex analytic queries perform better on subclusters with more nodes, which means that 6 nodes with 6 shards perform better than 3 nodes and 6 shards. Having more nodes than shards can increase performance further, but the performance gain is not linear. For example, a subcluster containing 12 nodes in a 6-shard database is not as efficient as a 12-node subcluster in a 12-shard database. Dashboard-type queries operating on smaller data sets may not see much difference between a 3-node subcluster in a 6-shard database and 6-node subcluster in a 6-shard database.

In general, choose a shard count that matches your expected working data size 6–12 months in the future. For more information about scaling your database, see Elasticity .

Use cases

Let's look at some use cases to learn how to size your Eon Mode cluster to meet your own particular requirements.

Use case 1: save compute by provisioning when needed, rather than for peak times

This use case highlights increasing query throughput in Eon Mode by scaling a cluster from 6 to 18 nodes with 3 subclusters of 6 nodes each. In this use case, you need to support a high concurrent, short query workload on a 24 TB or less working data set. You create an initial database with 6 nodes and 6 shards. You scale your database for concurrent throughput on demand by adding one or more subclusters during certain days of the week or for specific date ranges when you are expecting a peak load. You can then shut down or terminate the additional subclusters when your database experiences lower demand. With Vertica in Eon Mode, you save money by provisioning on demand, rather than provisioning for the peak times.

Use case 2: complex analytic workload requires more compute nodes

This use case showcases the idea that complex analytic workloads on large working data sets benefit from high shard count and node count. You create an initial subcluster with 24 nodes and 24 shards. As needed, you can add an additional 24 nodes to your initial subcluster. These additional nodes enable the subcluster to use elastic crunch scaling to reduce the time it takes to complete complex analytic queries.

Use case 3: workload isolation

This use case showcases the idea of having separate subclusters to isolate ETL and report workloads. You create an initial primary subcluster with 6 nodes and 6 shards for servicing ETL workloads. Then add another 6-node secondary subcluster for executing query workloads. To separate the two workloads, you can configure a network load balancer or create connection load balancing policies in Vertica to direct clients to the correct subcluster based on the type of workloads they need to execute.

Migrating an enterprise database to Eon Mode

The MIGRATE_ENTERPRISE_TO_EON function migrates an Enterprise database to Eon Mode. The migration process includes the following stages:

1. Check migration prerequisites
2. Verify compliance
3. Execute the migration
4. Check migration results
5. Activate the Eon database

> **Tip**
> When planning how to provision the target Eon database, consider that MIGRATE_ENTERPRISE_TO_EON creates the same number of nodes as on the original Enterprise database , and an equal number of segmented shards. Choose the appropriate instance type accordingly.

Migration prerequisites

The following conditions must be true; otherwise, MIGRATE_ENTERPRISE_TO_EON returns with an error:

- The source Enterprise database version must be ≥10.0.
- All nodes in the source database must be in an UP state and of type PERMANENT or EPHEMERAL. Verify by querying the NODES system table:

```
=> SELECT node_name, node_type, node_state FROM nodes;
   node_name     | node_type | node_state
-----------------+-----------+------------
 v_vmart_node0001 | PERMANENT | UP
 v_vmart_node0002 | PERMANENT | UP
 v_vmart_node0003 | PERMANENT | UP
(3 rows)
```

- The source database must be configured as an elastic cluster . By default, any database created since Vertica release 9.2.1 is configured as an elastic cluster. To verify whether an Enterprise database is configured as an elastic cluster, query the ELASTIC_CLUSTER system table:

```
=> SELECT is_enabled FROM elastic_cluster;
 is_enabled
------------
 t
(1 row)
```

If the query returns false, call the ENABLE_ELASTIC_CLUSTER function on the Enterprise database.

- The source Enterprise database must configure Eon parameters as required by the target Eon object store (see Configuration Requirements below).
- The database must not contain projections that are unsupported by Eon.

Unsupported projections

Eon databases do not support four types of projections, as described below. If MIGRATE_ENTERPRISE_TO_EON finds any of these projection types in the Enterprise database, it rolls back the migration and reports the offending projections or their anchor tables in the migration error log. For example:

```
The following projections are inconsistent with cluster segmentation. Rebalance them with REBALANCE_CLUSTER() or REBALANCE_TABLE():
Projection(Anchor Table): public.incon1_p1_b0(public.incon1)
```

Why projection is invalid	Notes	Resolution
Inconsistent with cluster segmentation.	For example, nodes were added to the cluster, so current distribution of projection data is inconsistent with new cluster segmentation requirements.	Rebalance cluster or table. The error log file lists the names of all tables with problematic projections. You can use these names as arguments to meta-function REBALANCE_TABLE. You can also rebalance all projections by calling REBALANCE_CLUSTER.
Does not support elastic segmentation.	Projection was created with the NODES option, or in a database where elastic segmentation was disabled.	Drop projection, recreate with ALL NODES.
Defined with a GROUPED clause.	Consolidates multiple columns in a single ROS container.	Drop projection, recreate without GROUPED clause.
Data stored in unbundled storage containers.	Found only in Vertica databases that were created before storage container bundling was introduced in version 7.2.	Bundle storage containers in database with meta-function COMPACT_STORAGE. The error log names all tables with projections that store data in unbundled storage containers. You can use these names as arguments to meta-function COMPACT_STORAGE.

Configuration requirements

Before migration, you must set certain configuration parameters in the source database. The specific parameters depend on the environment of the Eon database.

Important
All parameters must be set at the database level.

S3: AWS, Pure Storage, MinIO

The following requirements apply to all supported cloud and non-cloud (on-premises) S3 environments: AWS, Pure Storage, and MinIO. One exception applies: migration from an Enterprise Mode database on AWS.

- AWSEndpoint (Pure Storage, MinIO only)
- AWSRegion (AWS only)
- AWSAuth / IAM role

- AWSEnableHttps

Important
If migrating to on-premises communal storage with Pure Storage and MinIO, set AWSEnableHttps to be compatible with the database TLS encryption setup: AWSEnableHttps=1 if using TLS, otherwise 0. If settings are incompatible, the migration returns with an error.

Azure

- You must use an azb:// schema URI to set the Azure Blob Storage location for communal data storage. See Azure Blob Storage object store for the format of this URI.
- Select one of the following authentication methods to grant Vertica access to the storage location:
  - Configure managed identities to grant your Azure VMs access to the storage location. This option does not require any configuration within Vertica.
  - Set the AzureStorageCredentials and AzureStorageEndpointConfig configuration parameters at the database level to have Vertica authenticate with the storage location.

See Azure Blob Storage object store for more about the two authentication methods.

GCP

- GCSEndpoint
- GCSAuth
- GCSEnableHttp

HDFS

- The source database must be configured to access HDFS, including (as applicable) high-availability (HA) and Kerberos authentication settings.
- Set HadoopConfDir at the database level (not required for non-HA environments).

> **Tip**
> If using Kerberos authentication, record settings for the following Kerberos configuration parameters. You will need to apply these settings to the migrated Eon database:
>
> - KerberosServiceName
> - KerberosRealm
> - KerberosKeytabFile

Important
The following restrictions apply to Kerberos authentication:

- Migration only supports Vertica authentication. User authentication through delegation tokens or proxy users is not supported.
- Migration does not support authentication of multiple Kerberos realms.

Compliance verification

Before running migration, check whether the Enterprise source database complies with all migration requirements. You do so by setting the last Boolean argument of MIGRATE_ENTERPRISE_TO_EON to true to indicate that this is a dry run and not an actual migration:

```
=> SELECT migrate_enterprise_to_eon('s3://dbbucket', '/vertica/depot', true);
```

If the function encounters any compliance issues, it writes these to the migration error log, migrate_enterprise_to_eon_error.log, in the database directory.

Migration execution

MIGRATE_ENTERPRISE_TO_EON migrates an Enterprise database to an Eon Mode database. For example:

```
=> SELECT migrate_enterprise_to_eon('s3://dbbucket', '/vertica/depot', false);
```

If the last argument is omitted or false, the function executes the migration. MIGRATE_ENTERPRISE_TO_EON runs in the foreground, and until it returns—either with success or an error—it blocks all operations in the same session on the source Enterprise database. If successful, MIGRATE_ENTERPRISE_TO_EON returns with a list of nodes in the migrated database. You can then proceed to revive the migrated Eon database.

Handling interrupted migration

If migration is interrupted before the function returns—for example, the client disconnects, or a network outage occurs—the migration errors out. In this case, call MIGRATE_ENTERPRISE_TO_EON to restart migration.

Communal storage of the target database retains data that was already copied before the error occurred. When you call MIGRATE_ENTERPRISE_TO_EON to resume migration, the function first checks the data on communal storage and only copies unprocessed data from the source database.

Important
When migrating to an Eon database with HDFS communal storage, migration interruptions can leave files in an incomplete state that is visible to users. When you call MIGRATE_ENTERPRISE_TO_EON to resume migration, the function first compares source and target file sizes. If it finds inconsistent sizes for a given file, it truncates the target cluster file and repeats the entire transfer.

Repeating migration

You can repeat migration multiple times to the same communal storage location. This can be useful for backfilling changes that occurred in the source database during the previous migration.

The following constraints apply:

- You can migrate from only one database to the same communal storage location.
- After reviving the newly migrated Eon database , you cannot migrate again to its communal storage, unless you first drop the database and then clean up storage.

Monitoring migration

The DATABASE_MIGRATION_STATUS system table displays the progress of a migration in real time, and also stores data from previous migrations. The following example shows data of a migration that is in progress:

```
=> SELECT node_name, phase, status, bytes_to_transfer, bytes_transferred, communal_storage_location FROM database_migration_status ORDER BY node_name, start_time;
    node_name    |       phase        |  status   | bytes_to_transfer | bytes_transferred | communal_storage_location
-----------------+--------------------+-----------+-------------------+-------------------+--------------------------
 v_vmart_node0001 | Catalog Conversion | COMPLETED |                 0 |                 0 | s3://verticadbbucket/
 v_vmart_node0001 | Data Transfer      | COMPLETED |              1134 |              1134 | s3://verticadbbucket/
 v_vmart_node0001 | Catalog Transfer   | COMPLETED |              3765 |              3765 | s3://verticadbbucket/
 v_vmart_node0002 | Catalog Conversion | COMPLETED |                 0 |                 0 | s3://verticadbbucket/
 v_vmart_node0002 | Data Transfer      | COMPLETED |              1140 |              1140 | s3://verticadbbucket/
 v_vmart_node0002 | Catalog Transfer   | COMPLETED |              3766 |              3766 | s3://verticadbbucket/
 v_vmart_node0003 | Catalog Conversion | COMPLETED |                 0 |                 0 | s3://verticadbbucket/
 v_vmart_node0003 | Data Transfer      | RUNNING   |           5272616 |            183955 | s3://verticadbbucket/
```

Error logging

MIGRATE_ENTERPRISE_TO_EON logs migration-related warnings, errors, and hints in migrate_enterprise_to_eon_error.log in the database directory. During execution, the function also prints messages to standard output, together with the error log's pathname.

Conversion results

Visible objects in the source database are handled as follows:

- Global catalog objects: synced to communal storage.
- Multiple segmented projections in identical buddy projection group: one projection in the group is migrated.
- Unsegmented projection replicated on only one node: distributed across all nodes.
- Number of nodes: same number of nodes, and an equal number of segmented shards. You might want to alter the number of shards to better align with the number of nodes in your subclusters. For details, see RESHARD_DATABASE .
- USER and TEMP storage locations: migrated. Consider evaluating all migrated storage locations for their relevance in an Eon Mode database. For details, see S3 Storage of Temporary Data .
- DATA and TEMP,DATA storage locations: not migrated. New default DATA and TEMP,DATA locations are on the same path as the depot.
- Fault groups and storage policies: not migrated.
- External procedures: not migrated.
- Catalog objects related to network settings (load balance groups, network addresses, routing rules, subnets, etc.): not migrated.

The depot location is specified in MIGRATE_ENTERPRISE_TO_EON. Default depot size is set to 80% of local file system after revive.

Eon database activation

HDFS prerequisites

If migrating to an Eon database with HDFS communal storage, create a bootstrapping file to use when you revive the new Eon database. The bootstrapping file must be on the same node where the revive operation is launched, and readable by the user who launches the revive operation.

A bootstrapping file is required only if the new Eon database uses one or both of the following:

- High Availability (HA) NameNodes: Set HadoopConfDir to the location of the hdfs-site.xml configuration file—typically, /etc/hadoop/conf . This file defines the hdfs.nameservices parameter and individual NameNodes, and must be distributed across all cluster nodes. For details, see Configuring HDFS access .
- Kerberos authentication: Set the following Kerberos configuration parameters :
  - KerberosServiceName
  - KerberosRealm
  - KerberosKeytabFile

For example, the bootstrapping file for an Eon database with HA and Kerberos authentication must have the following settings:

```
HadoopConfDir = config-path
KerberosServiceName = principal-name
KerberosRealm = realm-name
KerberosKeytabFile = keytab-path
```

All migrations

After migration is complete and the Eon database is ready for use, perform these steps:

1. Revive the database using one of the following methods:
   - From communal storage on S3 or GCP with Management Console.
   - From communal storage on Azure, S3, GCP, or HDFS with admintools .

   In the following example, the admintools revive_db command revives a three-node database that uses S3 communal storage:

   ```
   admintools -t revive_db
     -x auth_params.conf \
     --communal-storage-location=s3://verticadbbucket \
     -d VMart \
     -s 172.16.116.27,172.16.116.28,172.16.116.29 \
     --force
   ```

   In the next example, revive_db revives a three-node database that uses HDFS communal storage:

   ```
   admintools -t revive_db
     -x bootstrap_params.conf \
    --communal-storage-location=webhdfs://mycluster/verticadb \
    -d verticadb \
    -s vnode01,vnode02,vnode03
   ```

2. Check the controlmode setting in /opt/vertica/config/admintools.conf . This setting must be compatible with the network messaging requirements of your Eon implementation. For example, S3/AWS (Amazon Cloud) relies on unicast messaging, which is compatible with a controlmode setting of point-to-point (pt2pt). If the source database controlmode setting was broacast and you migrate to S3/AWS communal storage, you must change controlmode with admintools:

   ```
   $ admintools -t re_ip -d dbname -T
   ```

   Important

   If controlmode is set incorrectly, attempts to start the migrated Eon database will fail.

3. Start the Eon Mode database.
4. Call CLEAN_COMMUNAL_STORAGE to remove unneeded data files that might be left over from the migration.
5. If migrating to S3 on-premises communal storage—Pure Storage or MinIO—set the AWSStreamingConnectionPercentage configuration parameter to 0 with ALTER DATABASE...SET PARAMETER .
6. Review the depot storage location size and adjust as needed .
7. Consider re-sharding the Eon Mode database if the number of shards is not optimal. See Choosing the Number of Shards and the Initial Node Count for more information. If needed, use RESHARD_DATABASE to change the number of shards.

Managing subclusters

Subclusters help you organize the nodes in your clusters to isolate workloads and make elastic scaling easier. See Subclusters for an overview of how subclusters can help you.

See also

- Eon Mode architecture
- Subclusters
- Adding and removing nodes from subclusters
- Altering subcluster settings
- Change the number of shards in the database

In this section

- Creating subclusters
- Duplicating a subcluster
- Adding and removing nodes from subclusters
- Managing workloads with subclusters
- Starting and stopping subclusters
- Altering subcluster settings
- Removing subclusters

Creating subclusters

By default, new Eon Mode databases contain a single primary subcluster named default_subcluster. This subcluster contains all of the nodes that are part of your database when you create it. You will often want to create subclusters to separate and manage workloads. You have three options to create new subclusters in your database:

- Use the admintools command line to add a new subcluster from nodes in your database cluster
- Use admintools to create a duplicate of an existing subcluster
- Use the Management Console to provision and create a new subcluster

Create a subcluster using admintools

To create a new subcluster, use the admintools db_add_subcluster tool:

```
$ admintools -t db_add_subcluster --help
Usage: db_add_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be modified
  -s HOSTS, --hosts=HOSTS
                        Comma separated list of hosts to add to the subcluster
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  -c SCNAME, --subcluster=SCNAME
                        Name of the new subcluster for the new node
  --is-primary          Create primary subcluster
  --is-secondary        Create secondary subcluster
  --control-set-size=CONTROLSETSIZE
                        Set the number of nodes that will run spread within
                        the subcluster
  --like=CLONESUBCLUSTER
                        Name of an existing subcluster from which to clone
                        properties for the new subcluster
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
```

The simplest command adds an empty subcluster. It requires the database name, password, and name for the new subcluster. This example adds a subcluster named analytics_cluster to the database named verticadb:

```
$ adminTools -t db_add_subcluster -d verticadb -p 'password' -c analytics_cluster
Creating new subcluster 'analytics_cluster'
Subcluster added to verticadb successfully.
```

By default, admintools creates the new subcluster as a secondary subcluster . You can have it create a primary subcluster instead by supplying the --is-primary argument.

Adding nodes while creating a subcluster

You can also specify one or more hosts for admintools to add to the subcluster as new nodes. These hosts must be part of the cluster but not already part of the database. For example, you can use hosts that you added to the cluster using the MC or admintools, or hosts that remain part of the cluster after you dropped nodes from the database. This example creates a subcluster named analytics_cluster and uses the -s option to specify the available hosts in the cluster:

```
$ adminTools -t db_add_subcluster -c analytics_cluster -d verticadb -p 'password' -s 10.0.33.77,10.0.33.181,10.0.33.85
```

View the subscription status of all nodes in your database with the following query that joins the V_CATALOG.NODES and V_CATALOG.NODE_SUBSCRIPTIONS system tables:

```
=> SELECT subcluster_name, n.node_name, shard_name, subscription_state FROM
  v_catalog.nodes n LEFT JOIN v_catalog.node_subscriptions ns ON (n.node_name
  = ns.node_name) ORDER BY 1,2,3;

subcluster_name   |      node_name       | shard_name  | subscription_state
------------------+----------------------+-------------+--------------------
analytics_cluster  | v_verticadb_node0004 | replica     | ACTIVE
analytics_cluster  | v_verticadb_node0004 | segment0001 | ACTIVE
analytics_cluster  | v_verticadb_node0004 | segment0003 | ACTIVE
analytics_cluster  | v_verticadb_node0005 | replica     | ACTIVE
analytics_cluster  | v_verticadb_node0005 | segment0001 | ACTIVE
analytics_cluster  | v_verticadb_node0005 | segment0002 | ACTIVE
analytics_cluster  | v_verticadb_node0006 | replica     | ACTIVE
analytics_cluster  | v_verticadb_node0006 | segment0002 | ACTIVE
analytics_cluster  | v_verticadb_node0006 | segment0003 | ACTIVE
default_subcluster | v_verticadb_node0001 | replica     | ACTIVE
default_subcluster | v_verticadb_node0001 | segment0001 | ACTIVE
default_subcluster | v_verticadb_node0001 | segment0003 | ACTIVE
default_subcluster | v_verticadb_node0002 | replica     | ACTIVE
default_subcluster | v_verticadb_node0002 | segment0001 | ACTIVE
default_subcluster | v_verticadb_node0002 | segment0002 | ACTIVE
default_subcluster | v_verticadb_node0003 | replica     | ACTIVE
default_subcluster | v_verticadb_node0003 | segment0002 | ACTIVE
default_subcluster | v_verticadb_node0003 | segment0003 | ACTIVE
(18 rows)
```

If you do not include hosts when you create the subcluster, you must manually rebalance the shards in the subcluster when you add nodes at a later time. For more information, see Updating Shard Subscriptions After Adding Nodes .

Subclusters and large cluster

Vertica has a feature named large cluster that helps manage broadcast messages as the database cluster grows. It has several impacts on adding new subclusters:

- If you create a new subcluster with 16 or more nodes, Vertica automatically enables the large cluster feature. It sets the number of control nodes to the square root of the number of nodes in your subcluster. See Planning a large cluster .
- You can set the number of control nodes in a subcluster by using the --control-set-size option in the admintools command line.
- If your database cluster has 120 control nodes, Vertica returns an error if you try to add a new subcluster. Every subcluster must have at least one control node. Your database cannot have more than 120 control nodes. When your database reaches this limit, you must reduce the number of control nodes in other subclusters before you can add a new subcluster. See Changing the number of control nodes and realigning for more information.
- If you attempt to create a subcluster with a number of control nodes that would exceed the 120 control node limit, Vertica warns you and creates the subcluster with fewer control nodes. It adds as many control nodes as it can to the subcluster, which is 120 minus the current count of control nodes in the cluster. For example, suppose you create a 16-node subcluster in a database cluster that already has 118 control nodes. In this case, Vertica warns you and creates your subcluster with just 2 control nodes rather than the default 4.

See Large cluster for more information about the large cluster feature.

Duplicating a subcluster

Subclusters have many settings you can tune to get them to work just the way you want. After you have tuned a subcluster, you may want additional subclusters that are configured the same way. For example, suppose you have a subcluster that you have tuned to perform analytics workloads. To improve query throughput, you can create several more subclusters configured exactly like it. Instead of creating the new subclusters and then manually configuring them from scratch, you can duplicate the existing subcluster (called the source subcluster) to a new subcluster (the target subcluster).

When you create a new subcluster based on another subcluster, Vertica copies most of the source subcluster's settings. See below for a list of the settings that Vertica copies. These settings are both on the node level and the subcluster level.

> **Note**
> After you duplicate a subcluster, the target is not connected to the source in any way. Any changes you make to the source subcluster's settings after duplication are not copied to the target. The subclusters are completely independent after duplication.

Requirements for the target subcluster

You must have a set of hosts in your database cluster that you will use as the target of the subcluster duplication. Vertica forms these hosts into a target subcluster that receives most of the settings of the source subcluster. The hosts for the target subcluster must meet the following requirements:

- They must be part of your database cluster but not part of your database. For example, you can use hosts you have dropped from a subcluster or whose subcluster you have removed. Vertica returns an error if you attempt to duplicate a subcluster onto one or more nodes that are currently participating in the database.

  > **Tip**
  > If you want to duplicate the settings of a subcluster to another subcluster, remove the target subcluster (see Removing subclusters). Then duplicate the source subcluster onto the hosts of the now-removed target subcluster.

- The number of nodes you supply for the target subcluster must equal the number of nodes in the source subcluster. When duplicating the subcluster, Vertica performs a 1:1 copy of some node-level settings from each node in the source subcluster to a corresponding node in the target.
- The RAM and disk allocation for the hosts in the target subcluster should be at least the same as the source nodes. Technically, your target nodes can have less RAM or disk space than the source nodes. However, you will usually see performance issues in the new subcluster because the settings of the original subcluster will not be tuned for the resources of the target subcluster.

You can duplicate a subcluster even if some of the nodes in the source subcluster or hosts in the target are down. If nodes in the target are down, they use the catalog Vertica copied from the source node when they recover.

Duplication of subcluster-level settings

The following table lists the subcluster-level settings that Vertica copies from the source subcluster to the target.

Setting Type	Setting Details
Basic subcluster settings	Whether the subcluster is a primary or secondary subcluster .
Large cluster settings	The number of control nodes in the subcluster.

Resource pool settings	Vertica creates a new resource pool for every subcluster-specific resource pool in the source subcluster. **Note** Duplicating a subcluster can fail due to subcluster-specific resource pools. If creating the subcluster-specific resource pools leave less than 25% of the total memory free for the general pool, Vertica stops the duplication and reports an error. Subcluster-specific resource pool cascade settings are copied from the source subcluster and are applied to the newly-created resource pool for the target subcluster. Subcluster-level overrides on global resource pools settings such as MEMORYSIZE. See Managing workload resources in an Eon Mode database for more information. Grants on resource pools are copied from the source subcluster.
Connection load balancing settings	If the source subcluster is part of a subcluster-based load balancing group (you created the load balancing group using CREATE LOAD BALANCE GROUP...WITH SUBCLUSTER) the new subcluster is added to the group. See Creating Connection Load Balance Groups . Important Vertica adds the new subcluster to the subcluster-based load balancing group. However, it does not create network addresses for the nodes in the target subcluster. Load balancing policies cannot direct connections to the new subcluster until you create network addresses for the nodes in the target subcluster. See Creating network addresses for the steps you must take.
Storage policy settings	Table and table partition pinning policies are copied from the source to the target subcluster. See Pinning Depot Objects for more information. Any existing storage policies on the target subcluster are dropped before the policies are copied from the source.

Vertica **does not** copy the following subcluster settings:

Setting Type	Setting Details
Basic subcluster settings	• Subcluster name (you must provide a new name for the target subcluster). • If the source is the default subcluster , the setting is not copied to the target. Your Vertica database has a single default subcluster. If Vertica copied this value, the source subcluster could no longer be the default.
Connection load balancing settings	Address-based load balancing groups are not duplicated for the target subcluster. For example, suppose you created a load balancing group for the source subcluster by adding the network addresses of all subcluster's nodes . In this case, Vertica does not create a load balancing group for the target subcluster because it does not duplicate the network addresses of the source nodes (see the next section). Because it does not copy the addresses, it cannot not create an address-based group.

Duplication of node-level settings

When Vertica duplicates a subcluster, it maps each node in the source subcluster to a node in the destination subcluster. Then it copies relevant node-level settings from each individual source node to the corresponding target node.

For example, suppose you have a three-node subcluster consisting of nodes named node01, node02, and node03. The target subcluster has nodes named node04, node05, and node06. In this case, Vertica copies the settings from node01 to node04, from node02 to node05, and from node03 to node06.

The node-level settings that Vertica copies from the source nodes to the target nodes are:

Setting Type	Setting Details

Configuration parameters	Vertica copies the value of configuration parameters that you have set at the node level in the source node to the target node. For example, suppose you set CompressCatalogOnDisk on the source node using the statement: `ALTER NODE node01 SET CompressCatalogOnDisk = 0;` If you then duplicated the subcluster containing node01, the setting is copied to the target node.
Eon Mode settings	• Shard subscriptions are copied from the source node to the target. • Whether the node is the participating primary node for the shard.
Storage location settings	The DATA, TEMP, DEPOT, and USER storage location paths on the source node are duplicated on the target node. When duplicating node-specific paths (such as DATA or DEPOT) the path names are adjusted for the new node name. For example, suppose node 1 has a depot path of `/vertica/depot/vmart/v_vmart_node0001_depot` . If Vertica duplicates node 1 to node 4, it adjusts the path to `/vertica/depot/vmart/v_vmart_node0004_depot` . Important The directories for these storage locations on the target node must be empty. They must also have the correct file permissions to allow Vertica to read and write to them. Vertica does not duplicate a storage location if it cannot access its directory on the target node or if the directory is not empty. In this case, the target node will not have the location defined after the duplication process finishes. Admintools does not warn you if any locations were not duplicated. If you find that storage locations have not been duplicated on one or more target nodes, you must fix the issues with the directories on the target nodes. Then re-run the duplication command.
Large cluster settings	Control node assignments are copied from the source node to the target node: • If the source node is a control node, then the target node is made into a control node. • If the source node depends on a control node, then the target node becomes a dependent of the corresponding control node in the new subcluster.

Vertica **does not** copy the following node-level settings:

Setting Type	Setting Details
Connection load balancing settings	Network Addresses are not copied. The destination node's network addresses do not depend on the settings of the source node. Therefore, Vertica cannot determine what the target node's addresses should be.
Depot settings	Depot-related configuration parameters that can be set on a node level (such as FileDeletionServiceInterval) are not copied from the source node to the target node.

Using admintools to duplicate a subcluster

To duplicate a subcluster, you use the same admintools `db_add_subcluster` tool that you use to create a new subcluster (see Creating subclusters). In addition to the required options to create a subcluster (the list of hosts, name for the new subcluster, database name, and so on), you also pass the `--like` option with the name of the source subcluster you want to duplicate.

Important

When you use the `--like` option, you cannot use the `--is-secondary` or `--control-set-size` options. Vertica determines whether the new subcluster is secondary and the number of control nodes it contains based on the source subcluster. If you supply these options along with the `--like` option, admintools returns an error.

The following examples demonstrate duplicating a three-node subcluster named analytics_1. The first example examines some of the settings in the analytics_1 subcluster:

- An override of the global TM resource pool's memory size.
- Its own resource pool named analytics
- Its membership in a subcluster-based load balancing group named analytics

```
=> SELECT name, subcluster_name, memorysize FROM SUBCLUSTER_RESOURCE_POOL_OVERRIDES;
 name | subcluster_name | memorysize
------+-----------------+------------
 tm   | analytics_1     | 0%
(1 row)

=> SELECT name, subcluster_name, memorysize, plannedconcurrency
     FROM resource_pools WHERE subcluster_name IS NOT NULL;
     name      | subcluster_name | memorysize | plannedconcurrency
---------------+-----------------+------------+--------------------
 analytics_pool | analytics_1     | 70%        | 8
(1 row)

=> SELECT * FROM LOAD_BALANCE_GROUPS;
   name    |   policy   |  filter   |    type    | object_name
-----------+------------+-----------+------------+-------------
 analytics | ROUNDROBIN | 0.0.0.0/0 | Subcluster | analytics_1
(1 row)
```

The following example calls admintool's db_add_subcluster tool to duplicate the analytics_1 subcluster onto a set of three hosts to create a subcluster named analytics_2.

```
$ admintools -t db_add_subcluster -d verticadb \
           -s 10.11.12.13,10.11.12.14,10.11.12.15 \
         -p mypassword --like=analytics_1 -c analytics_2

Creating new subcluster 'analytics_2'
Adding new hosts to 'analytics_2'
Eon database detected, creating new depot locations for newly added nodes
Creating depot locations for 1 nodes
 Warning when creating depot location for node: v_verticadb_node0007
 WARNING: Target node v_verticadb_node0007 is down, so depot size has been
         estimated from depot location on initiator. As soon as the node comes
         up, its depot size might be altered depending on its disk size
Eon database detected, creating new depot locations for newly added nodes
Creating depot locations for 1 nodes
 Warning when creating depot location for node: v_verticadb_node0008
 WARNING: Target node v_verticadb_node0008 is down, so depot size has been
         estimated from depot location on initiator. As soon as the node comes
         up, its depot size might be altered depending on its disk size
Eon database detected, creating new depot locations for newly added nodes
Creating depot locations for 1 nodes
 Warning when creating depot location for node: v_verticadb_node0009
 WARNING: Target node v_verticadb_node0009 is down, so depot size has been
         estimated from depot location on initiator. As soon as the node comes
         up, its depot size might be altered depending on its disk size
Cloning subcluster properties
NOTICE: Nodes in subcluster analytics_1 have network addresses, you
might need to configure network addresses for nodes in subcluster
analytics_2 in order to get load balance groups to work correctly.

    Replicating configuration to all nodes
    Generating new configuration information and reloading spread
    Starting nodes:
        v_verticadb_node0007 (10.11.12.81)
        v_verticadb_node0008 (10.11.12.209)
        v_verticadb_node0009 (10.11.12.186)
    Starting Vertica on all nodes. Please wait, databases with a large catalog
        may take a while to initialize.
    Checking database state for newly added nodes
    Node Status: v_verticadb_node0007: (DOWN) v_verticadb_node0008:
                 (DOWN) v_verticadb_node0009: (DOWN)
    Node Status: v_verticadb_node0007: (INITIALIZING) v_verticadb_node0008:
                 (INITIALIZING) v_verticadb_node0009: (INITIALIZING)
    Node Status: v_verticadb_node0007: (UP) v_verticadb_node0008:
                 (UP) v_verticadb_node0009: (UP)
Syncing catalog on verticadb with 2000 attempts.
    Multi-node DB add completed
Nodes added to subcluster analytics_2 successfully.
Subcluster added to verticadb successfully.
```

Re-running the queries in the first part of the example shows that the settings from analytics_1 have been duplicated in analytics_2:

```
=> SELECT name, subcluster_name, memorysize FROM SUBCLUSTER_RESOURCE_POOL_OVERRIDES;
 name | subcluster_name | memorysize
------+-----------------+------------
 tm   | analytics_1     | 0%
 tm   | analytics_2     | 0%
(2 rows)

=> SELECT name, subcluster_name, memorysize, plannedconcurrency
      FROM resource_pools WHERE subcluster_name IS NOT NULL;
     name      | subcluster_name | memorysize |  plannedconcurrency
---------------+-----------------+------------+---------------------
 analytics_pool | analytics_1     | 70%        | 8
 analytics_pool | analytics_2     | 70%        | 8
(2 rows)

=> SELECT * FROM LOAD_BALANCE_GROUPS;
   name    |   policy   |  filter   |    type    | object_name
-----------+------------+-----------+------------+-------------
 analytics | ROUNDROBIN | 0.0.0.0/0 | Subcluster | analytics_2
 analytics | ROUNDROBIN | 0.0.0.0/0 | Subcluster | analytics_1
(2 rows)
```

As noted earlier, even though analytics_2 subcluster is part of the analytics load balancing group, its nodes do not have network addresses defined for them. Until you define network addresses for the nodes, Vertica cannot redirect client connections to them.

Adding and removing nodes from subclusters

You will often want to add new nodes to and remove existing nodes from a subcluster. This ability lets you scale your database to respond to changing analytic needs. For more information on how adding nodes to a subcluster affects your database's performance, see Scaling your Eon Mode database .

Adding new nodes to a subcluster

You can add nodes to a subcluster to meet additional workloads. The nodes that you add to the subcluster must already be part of your cluster. These can be:

- Nodes that you removed from other subclusters.
- Nodes you added following the steps in Add nodes to a cluster in AWS using Management Console .
- Nodes you created using your cloud provider's interface, such as the AWS EC2 "Launch more like this" feature.

To add new nodes to a subcluster, use the db_add_node command of admintools:

```
$ adminTools -t db_add_node -h
Usage: db_add_node [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of the database
  -s HOSTS, --hosts=HOSTS
                        Comma separated list of hosts to add to database
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  -a AHOSTS, --add=AHOSTS
                        Comma separated list of hosts to add to database
  -c SCNAME, --subcluster=SCNAME
                        Name of subcluster for the new node
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
  --compat21            (deprecated) Use Vertica 2.1 method using node names
                        instead of hostnames
```

If you do not use the -c option, Vertica adds new nodes to the default subcluster (set to default_subcluster in new databases). This example adds a new node without specifying the subcluster:

```
$ adminTools -t db_add_node -p 'password' -d verticadb -s 10.11.12.117
Subcluster not specified, validating default subcluster
Nodes will be added to subcluster 'default_subcluster'
        Verifying database connectivity...10.11.12.10
Eon database detected, creating new depot locations for newly added nodes
Creating depots for each node
    Generating new configuration information and reloading spread
    Replicating configuration to all nodes
    Starting nodes
    Starting nodes:
        v_verticadb_node0004 (10.11.12.117)
    Starting Vertica on all nodes. Please wait, databases with a
      large catalog may take a while to initialize.
    Checking database state
    Node Status: v_verticadb_node0004: (DOWN)
    Node Status: v_verticadb_node0004: (DOWN)
    Node Status: v_verticadb_node0004: (DOWN)
    Node Status: v_verticadb_node0004: (DOWN)
    Node Status: v_verticadb_node0004: (UP)
Communal storage detected: syncing catalog

    Multi-node DB add completed
Nodes added to verticadb successfully.
You will need to redesign your schema to take advantage of the new nodes.
```

To add nodes to a specific existing subcluster, use the db_add_node tool's -c option:

```
$ adminTools -t db_add_node -s 10.11.12.178 -d verticadb -p 'password' \
          -c analytics_subcluster
Subcluster 'analytics_subcluster' specified, validating
Nodes will be added to subcluster 'analytics_subcluster'
             Verifying database connectivity...10.11.12.10
Eon database detected, creating new depot locations for newly added nodes
Creating depots for each node
      Generating new configuration information and reloading spread
      Replicating configuration to all nodes
      Starting nodes
      Starting nodes:
           v_verticadb_node0007 (10.11.12.178)
      Starting Vertica on all nodes. Please wait, databases with a
          large catalog may take a while to initialize.
      Checking database state
      Node Status: v_verticadb_node0007: (DOWN)
      Node Status: v_verticadb_node0007: (DOWN)
      Node Status: v_verticadb_node0007: (DOWN)
      Node Status: v_verticadb_node0007: (DOWN)
      Node Status: v_verticadb_node0007: (UP)
Communal storage detected: syncing catalog

      Multi-node DB add completed
Nodes added to verticadb successfully.
You will need to redesign your schema to take advantage of the new nodes.
```

Updating shard subscriptions after adding nodes

After you add nodes to a subcluster they do not yet subscribe to shards. You can view the subscription status of all nodes in your database using the following query that joins the V_CATALOG.NODES and V_CATALOG.NODE_SUBSCRIPTIONS system tables:

```
=> SELECT subcluster_name, n.node_name, shard_name, subscription_state FROM
  v_catalog.nodes n LEFT JOIN v_catalog.node_subscriptions ns ON (n.node_name
  = ns.node_name) ORDER BY 1,2,3;

  subcluster_name    |      node_name       | shard_name  | subscription_state
---------------------+----------------------+-------------+--------------------
 analytics_subcluster | v_verticadb_node0004 |             |
 analytics_subcluster | v_verticadb_node0005 |             |
 analytics_subcluster | v_verticadb_node0006 |             |
 default_subcluster   | v_verticadb_node0001 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0001 | segment0001 | ACTIVE
 default_subcluster   | v_verticadb_node0001 | segment0003 | ACTIVE
 default_subcluster   | v_verticadb_node0002 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0002 | segment0001 | ACTIVE
 default_subcluster   | v_verticadb_node0002 | segment0002 | ACTIVE
 default_subcluster   | v_verticadb_node0003 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0003 | segment0002 | ACTIVE
 default_subcluster   | v_verticadb_node0003 | segment0003 | ACTIVE
(12 rows)
```

You can see that none of the nodes in the newly-added analytics_subcluster have subscriptions.

To update the subscriptions for new nodes, call the REBALANCE_SHARDS function. You can limit the rebalance to the subcluster containing the new nodes by passing its name to the REBALANCE_SHARDS function call. The following example runs rebalance shards to update the analytics_subcluster's subscriptions:

```
=> SELECT REBALANCE_SHARDS('analytics_subcluster');
 REBALANCE_SHARDS
-------------------
 REBALANCED SHARDS
(1 row)

=> SELECT subcluster_name, n.node_name, shard_name, subscription_state FROM
   v_catalog.nodes n LEFT JOIN v_catalog.node_subscriptions ns ON (n.node_name
   = ns.node_name) ORDER BY 1,2,3;

   subcluster_name    |      node_name       | shard_name  | subscription_state
----------------------+----------------------+-------------+--------------------
 analytics_subcluster | v_verticadb_node0004 | replica     | ACTIVE
 analytics_subcluster | v_verticadb_node0004 | segment0001 | ACTIVE
 analytics_subcluster | v_verticadb_node0004 | segment0003 | ACTIVE
 analytics_subcluster | v_verticadb_node0005 | replica     | ACTIVE
 analytics_subcluster | v_verticadb_node0005 | segment0001 | ACTIVE
 analytics_subcluster | v_verticadb_node0005 | segment0002 | ACTIVE
 analytics_subcluster | v_verticadb_node0006 | replica     | ACTIVE
 analytics_subcluster | v_verticadb_node0006 | segment0002 | ACTIVE
 analytics_subcluster | v_verticadb_node0006 | segment0003 | ACTIVE
 default_subcluster   | v_verticadb_node0001 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0001 | segment0001 | ACTIVE
 default_subcluster   | v_verticadb_node0001 | segment0003 | ACTIVE
 default_subcluster   | v_verticadb_node0002 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0002 | segment0001 | ACTIVE
 default_subcluster   | v_verticadb_node0002 | segment0002 | ACTIVE
 default_subcluster   | v_verticadb_node0003 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0003 | segment0002 | ACTIVE
 default_subcluster   | v_verticadb_node0003 | segment0003 | ACTIVE
(18 rows)
```

Removing nodes

Your database must meet these requirements before you can remove a node from a subcluster:

- To remove a node from a primary subcluster , all of the primary nodes in the subcluster must be up, and the database must be able to maintain quorum after the primary node is removed (see Data integrity and high availability in an Eon Mode database). These requirements are necessary because Vertica calls REBALANCE_SHARDS to redistribute shard subscriptions among the remaining nodes in the subcluster. If you attempt to remove a primary node when the database does not meet the requirements, the rebalance shards process waits until either the down nodes recover or a timeout elapses. While it waits, you periodically see a message "Rebalance shards polling iteration number [*nn*]" indicating that the rebalance process is waiting to complete.
  You can remove nodes from a secondary subcluster even when nodes in the subcluster are down.
- If your database has the large cluster feature enabled, you cannot remove a node if it is the subcluster's last control node and there are nodes that depend on it. See Large cluster for more information.
  If there are other control nodes in the subcluster, you can drop a control node. Vertica reassigns the nodes that depend on the node being dropped to other control nodes.

To remove one or more nodes, use admintools's db_remove_node tool:

```
$ adminTools -t db_remove_node -p 'password' -d verticadb -s 10.11.12.117
connecting to 10.11.12.10
Waiting for rebalance shards. We will wait for at most 36000 seconds.
Rebalance shards polling iteration number [0], started at [14:56:41], time out at [00:56:41]
Attempting to drop node v_verticadb_node0004 ( 10.11.12.117 )
      Shutting down node v_verticadb_node0004
      Sending node shutdown command to ['v_verticadb_node0004', '10.11.12.117', '/vertica/data', '/vertica/data']'
      Deleting catalog and data directories
      Update admintools metadata for v_verticadb_node0004
      Eon mode detected. The node v_verticadb_node0004 has been removed from host 10.11.12.117. To remove the
      node metadata completely, please clean up the files corresponding to this node, at the communal
      location: s3://eonbucket/metadata/verticadb/nodes/v_verticadb_node0004
      Reload spread configuration
      Replicating configuration to all nodes
      Checking database state
      Node Status: v_verticadb_node0001: (UP) v_verticadb_node0002: (UP) v_verticadb_node0003: (UP)
Communal storage detected: syncing catalog
```

When you remove one or more nodes from a subcluster, Vertica automatically rebalances shards in the subcluster. You do not need to manually rebalance shards after removing nodes.

Moving nodes between subclusters

To move a node from one subcluster to another:

1. Remove the node or nodes from the subcluster it is currently a part of.
2. Add the node to the subcluster you want to move it to.

Managing workloads with subclusters

By default, queries are limited to executing on the nodes in the subcluster that contains the initiator node (the node the client is connected to). This example demonstrates executing an explain plan for a query when connected to node 4 of a cluster. Node 4 is part of a subcluster containing nodes 4 through 6. You can see that only the nodes in the subcluster will participate in a query:

```
=> EXPLAIN SELECT customer_name, customer_state FROM customer_dimension LIMIT 10;

                            QUERY PLAN
------------------------------------------------------------------------------


 QUERY PLAN DESCRIPTION:
 ------------------------------

 EXPLAIN SELECT customer_name, customer_state FROM customer_dimension LIMIT 10;

 Access Path:
 +-SELECT  LIMIT 10 [Cost: 442, Rows: 10 (NO STATISTICS)] (PATH ID: 0)
 |  Output Only: 10 tuples
 |  Execute on: Query Initiator
 | +---> STORAGE ACCESS for customer_dimension [Cost: 442, Rows: 10K (NO
         STATISTICS)] (PATH ID: 1)
 | |      Projection: public.customer_dimension_b0
 | |      Materialize: customer_dimension.customer_name,
           customer_dimension.customer_state
 | |      Output Only: 10 tuples
 | |      Execute on: v_verticadb_node0004, v_verticadb_node0005,
                    v_verticadb_node0006
    .  .  .
```

In Eon Mode, you can override the MEMORYSIZE , MAXMEMORYSIZE , and MAXQUERYMEMORYSIZE settings for built-in global resource pools to fine-tune workloads within a subcluster. See Managing workload resources in an Eon Mode database for more information.

What happens when a subcluster cannot run a query

In order to process queries, each subcluster's nodes must have full coverage of all shards in the database. If the nodes do not have full coverage (which can happen if nodes are down), the subcluster can no longer process queries. This state does not cause the subcluster to shut down. Instead, if you attempt to run a query on a subcluster in this state, you receive error messages telling you that not enough nodes are available to complete the query.

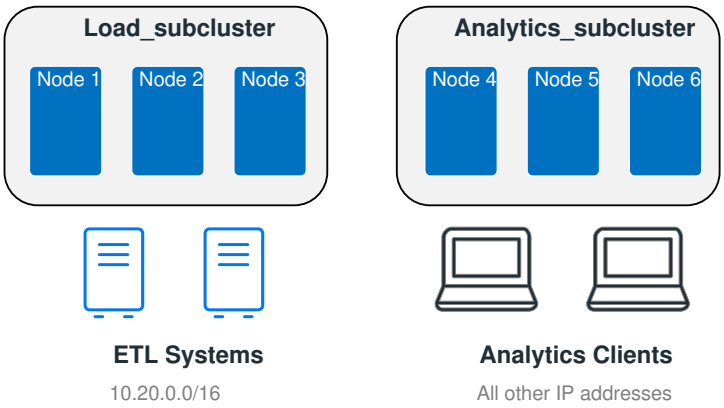
```
=> SELECT node_name, node_state FROM nodes
   WHERE subcluster_name = 'analytics_cluster';
     node_name       | node_state
---------------------+------------
 v_verticadb_node0004 | DOWN
 v_verticadb_node0005 | UP
 v_verticadb_node0006 | DOWN
(3 rows)

=> SELECT * FROM online_sales.online_sales_fact;
ERROR 9099:  Cannot find participating nodes to run the query
```

Once the down nodes have recovered and the subcluster has full shard coverage, it will be able to process queries.

Controlling where a query runs

You can control where specific types of queries run by controlling which subcluster the clients connect to. The best way to enforce restrictions is to create a set of connection load balancing policies to steer clients from specific IP address ranges to clients in the correct subcluster.

For example, suppose you have the following database with two subclusters: one for performing data loading, and one for performing analytics.



The data load tasks come from a set of ETL systems in the IP 10.20.0.0/16 address range. Analytics tasks can come from any other IP address. In this case, you can create set of connection load balance policies that ensure that the ETL systems connect to the data load subcluster, and all other connections go to the analytics subcluster.

```
=> SELECT node_name,node_address,node_address_family,subcluster_name
   FROM v_catalog.nodes;
     node_name       | node_address | node_address_family |  subcluster_name
---------------------+--------------+---------------------+--------------------
 v_verticadb_node0001 | 10.11.12.10  | ipv4                | load_subcluster
 v_verticadb_node0002 | 10.11.12.20  | ipv4                | load_subcluster
 v_verticadb_node0003 | 10.11.12.30  | ipv4                | load_subcluster
 v_verticadb_node0004 | 10.11.12.40  | ipv4                | analytics_subcluster
 v_verticadb_node0005 | 10.11.12.50  | ipv4                | analytics_subcluster
 v_verticadb_node0006 | 10.11.12.60  | ipv4                | analytics_subcluster
(6 rows)

=> CREATE NETWORK ADDRESS node01 ON v_verticadb_node0001 WITH '10.11.12.10';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node02 ON v_verticadb_node0002 WITH '10.11.12.20';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node03 ON v_verticadb_node0003 WITH '10.11.12.30';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node04 ON v_verticadb_node0004 WITH '10.11.12.40';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node05 ON v_verticadb_node0005 WITH '10.11.12.50';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node06 ON v_verticadb_node0006 WITH '10.11.12.60';
CREATE NETWORK ADDRESS

=> CREATE LOAD BALANCE GROUP load_subcluster WITH SUBCLUSTER load_subcluster
   FILTER '0.0.0.0/0';
CREATE LOAD BALANCE GROUP
=> CREATE LOAD BALANCE GROUP analytics_subcluster WITH SUBCLUSTER
   analytics_subcluster FILTER '0.0.0.0/0';
CREATE LOAD BALANCE GROUP
```

```
=> CREATE ROUTING RULE etl_systems ROUTE '10.20.0.0/16' TO load_subcluster;
CREATE ROUTING RULE
=> CREATE ROUTING RULE analytic_clients ROUTE '0.0.0.0/0' TO analytics_subcluster;
CREATE ROUTING RULE
```

Once you have created the load balance policies, you can test them using the DESCRIBE_LOAD_BALANCE_DECISION function.

```
=> SELECT describe_load_balance_decision('192.168.1.1');

              describe_load_balance_decision
          --------------------------------
 Describing load balance decision for address [192.168.1.1]
Load balance cache internal version id (node-local): [1]
Considered rule [etl_systems] source ip filter [10.20.0.0/16]...
   input address does not match source ip filter for this rule.
Considered rule [analytic_clients] source ip filter [0.0.0.0/0]...
   input address matches this rule
Matched to load balance group [analytics_cluster] the group has
   policy [ROUNDROBIN] number of addresses [3]
(0) LB Address: [10.11.12.181]:5433
(1) LB Address: [10.11.12.205]:5433
(2) LB Address: [10.11.12.192]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [10.11.12.205]
    port [5433]

(1 row)

=> SELECT describe_load_balance_decision('10.20.1.1');

       describe_load_balance_decision
    ---------------------------------
 Describing load balance decision for address [10.20.1.1]
Load balance cache internal version id (node-local): [1]
Considered rule [etl_systems] source ip filter [10.20.0.0/16]...
  input address matches this rule
Matched to load balance group [default_cluster] the group has policy
  [ROUNDROBIN] number of addresses [3]
(0) LB Address: [10.11.12.10]:5433
(1) LB Address: [10.11.12.20]:5433
(2) LB Address: [10.11.12.30]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [10.11.12.20]
  port [5433]

(1 row)
```

Normally, with these policies, all queries run by the ETL system will run on the load subcluster. All other queries will run on the analytics subcluster. There are some cases (especially if a subcluster is down or draining) where a client may connect to a node in another subcluster. For this reason, clients should always verify they are connected to the correct subcluster. See Connection load balancing policies for more information about load balancing policies.

Starting and stopping subclusters

Subclusters make it convenient to start and stop a group of nodes as needed. You start and stop them with admintools commands or Vertica functions. You can also start and stop subclusters with Management Console .

Starting a subcluster

To start a subcluster, use the admintools command restart_subcluster :

```
$ adminTools -t restart_subcluster -h
Usage: restart_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database whose subcluster is to be restarted
  -c SCNAME, --subcluster=SCNAME
                        Name of subcluster to be restarted
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
  -F, --force           Force the nodes in the subcluster to start and auto
                        recover if necessary
```

This example starts the subcluster analytics_cluster :

```
$ adminTools -t restart_subcluster -c analytics_cluster \
         -d verticadb -p password
*** Restarting subcluster for database verticadb ***
        Restarting host [10.11.12.192] with catalog [v_verticadb_node0006_catalog]
        Restarting host [10.11.12.181] with catalog [v_verticadb_node0004_catalog]
        Restarting host [10.11.12.205] with catalog [v_verticadb_node0005_catalog]
        Issuing multi-node restart
        Starting nodes:
                v_verticadb_node0004 (10.11.12.181)
                v_verticadb_node0005 (10.11.12.205)
                v_verticadb_node0006 (10.11.12.192)
        Starting Vertica on all nodes. Please wait, databases with a large
           catalog may take a while to initialize.
        Node Status: v_verticadb_node0002: (UP) v_verticadb_node0004: (DOWN)
                     v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
        Node Status: v_verticadb_node0002: (UP) v_verticadb_node0004: (DOWN)
                     v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
        Node Status: v_verticadb_node0002: (UP) v_verticadb_node0004: (DOWN)
                     v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
        Node Status: v_verticadb_node0002: (UP) v_verticadb_node0004: (DOWN)
                     v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
        Node Status: v_verticadb_node0002: (UP) v_verticadb_node0004: (UP)
                     v_verticadb_node0005: (UP) v_verticadb_node0006: (UP)
Communal storage detected: syncing catalog

Restart Subcluster result:  1
```

Stopping a subcluster

You can stop a subcluster gracefully with the function SHUTDOWN_WITH_DRAIN , or immediately with SHUTDOWN_SUBCLUSTER . You can also shut down subclusters with the admintools command stop_subcluster .

Graceful shutdown

The SHUTDOWN_WITH_DRAIN function drains a subcluster's client connections before shutting it down. The function first marks all nodes in the specified subcluster as draining. Work from existing user sessions continues on draining nodes, but the nodes refuse new client connections and are excluded from load-balancing operations. A dbadmin user can still connect to draining nodes. For more information about client connection draining, see Drain client connections .

To run the SHUTDOWN_WITH_DRAIN function, you must specify a timeout value. The function's behavior depends on the sign of the timeout value:

- Positive: The nodes drain until either all the existing connections close or the function reaches the runtime limit set by the timeout value. As soon as one of these conditions is met, the function sends a shutdown message to the subcluster and returns.

- Zero: The function immediately closes any active user sessions on the subcluster and then shuts down the subcluster and returns.
- Negative: The function marks the subcluster's nodes as draining and waits to shut down the subcluster until all active user sessions disconnect.

After all nodes in a draining subcluster are down, its nodes are automatically reset to a not draining status.

The following example demonstrates how you can use a positive timeout value to give active user sessions time to finish their work before shutting down the subcluster:

```
=> SELECT node_name, subcluster_name, is_draining, count_client_user_sessions, oldest_session_user FROM draining_status ORDER BY 1;
     node_name      |  subcluster_name   | is_draining | count_client_user_sessions | oldest_session_user
--------------------+--------------------+-------------+----------------------------+---------------------
 v_verticadb_node0001 | default_subcluster | f           |                          0 |
 v_verticadb_node0002 | default_subcluster | f           |                          0 |
 v_verticadb_node0003 | default_subcluster | f           |                          0 |
 v_verticadb_node0004 | analytics          | f           |                          1 | analyst
 v_verticadb_node0005 | analytics          | f           |                          0 |
 v_verticadb_node0006 | analytics          | f           |                          0 |
(6 rows)

=> SELECT SHUTDOWN_WITH_DRAIN('analytics', 300);
NOTICE 0:  Draining has started on subcluster (analytics)
NOTICE 0:  Begin shutdown of subcluster (analytics)
                        SHUTDOWN_WITH_DRAIN
-----------------------------------------------------------------------------------------------------------------------
Set subcluster (analytics) to draining state
Waited for 3 nodes to drain
Shutdown message sent to subcluster (analytics)

(1 row)
```

You can query the NODES system table to confirm that the subcluster shut down:

```
=> SELECT subcluster_name, node_name, node_state FROM nodes;
  subcluster_name   |      node_name       | node_state
--------------------+----------------------+------------
 default_subcluster | v_verticadb_node0001 | UP
 default_subcluster | v_verticadb_node0002 | UP
 default_subcluster | v_verticadb_node0003 | UP
 analytics          | v_verticadb_node0004 | DOWN
 analytics          | v_verticadb_node0005 | DOWN
 analytics          | v_verticadb_node0006 | DOWN
(6 rows)
```

If you want to see more information about the draining and shutdown events, such as whether all user sessions finished their work before the timeout, you can query the dc_draining_events table. In this case, the subcluster still had one active user session when the function reached timeout:

```
=> SELECT event_type, event_type_name, event_description, event_result, event_result_name FROM dc_draining_events;
 event_type |       event_type_name        |                        event_description                        | event_result | event_result_name
------------+------------------------------+-----------------------------------------------------------------+--------------+-------------------
          0 | START_DRAIN_SUBCLUSTER       | START_DRAIN for SHUTDOWN of subcluster (analytics)              |            0 | SUCCESS
          2 | START_WAIT_FOR_NODE_DRAIN    | Wait timeout is 300 seconds                                     |            4 | INFORMATIONAL
          4 | INTERVAL_WAIT_FOR_NODE_DRAIN | 1 sessions remain after 0 seconds                               |            4 | INFORMATIONAL
          4 | INTERVAL_WAIT_FOR_NODE_DRAIN | 1 sessions remain after 60 seconds                              |            4 | INFORMATIONAL
          4 | INTERVAL_WAIT_FOR_NODE_DRAIN | 1 sessions remain after 120 seconds                             |            4 | INFORMATIONAL
          4 | INTERVAL_WAIT_FOR_NODE_DRAIN | 1 sessions remain after 125 seconds                             |            4 | INFORMATIONAL
          4 | INTERVAL_WAIT_FOR_NODE_DRAIN | 1 sessions remain after 180 seconds                             |            4 | INFORMATIONAL
          4 | INTERVAL_WAIT_FOR_NODE_DRAIN | 1 sessions remain after 240 seconds                             |            4 | INFORMATIONAL
          4 | INTERVAL_WAIT_FOR_NODE_DRAIN | 1 sessions remain after 250 seconds                             |            4 | INFORMATIONAL
          4 | INTERVAL_WAIT_FOR_NODE_DRAIN | 1 sessions remain after 300 seconds                             |            4 | INFORMATIONAL
          3 | END_WAIT_FOR_NODE_DRAIN      | Wait for drain ended with 1 sessions remaining                  |            2 | TIMEOUT
          5 | BEGIN_SHUTDOWN_AFTER_DRAIN   | Staring shutdown of subcluster (analytics) following drain      |            4 | INFORMATIONAL
(12 rows)
```

After you restart the subcluster, you can query the DRAINING_STATUS system table to confirm that the nodes have reset their draining statuses to not draining:

```
=> SELECT node_name, subcluster_name, is_draining, count_client_user_sessions, oldest_session_user FROM draining_status ORDER BY 1;
     node_name      |  subcluster_name   | is_draining | count_client_user_sessions | oldest_session_user
--------------------+--------------------+-------------+----------------------------+---------------------
 v_verticadb_node0001 | default_subcluster | f           |                          0 |
 v_verticadb_node0002 | default_subcluster | f           |                          0 |
 v_verticadb_node0003 | default_subcluster | f           |                          0 |
 v_verticadb_node0004 | analytics          | f           |                          0 |
 v_verticadb_node0005 | analytics          | f           |                          0 |
 v_verticadb_node0006 | analytics          | f           |                          0 |
(6 rows)
```

Immediate shutdown

To shut down a subcluster immediately, call SHUTDOWN_SUBCLUSTER. The following example shuts down the analytics subcluster immediately, without checking for active client connections:

```
=> SELECT SHUTDOWN_SUBCLUSTER('analytics');
 SHUTDOWN_SUBCLUSTER
---------------------
Subcluster shutdown
(1 row)
```

admintools

You can use the stop_subcluster tool to stop a subcluster:

```
$ adminTools -t stop_subcluster -h
Usage: stop_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database whose subcluster is to be stopped
  -c SCNAME, --subcluster=SCNAME
                        Name of subcluster to be stopped
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  -n DRAIN_SECONDS, --drain-seconds=DRAIN_SECONDS
                        Seconds to wait for user connections to close.
                        Default value is 60 seconds.
                        When the time expires, connections will be forcibly closed
                        and the db will shut down.
  -F, --force           Force the subcluster to shutdown immediately,
                        even if users are connected.
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
```

By default, stop_subcluster calls SHUTDOWN_WITH_DRAIN to gracefully shut down the target subcluster. The shutdown process drains client connections from the subcluster before shutting it down.

The -n (--drain-seconds) option, which has a default value of 60 seconds, allows you to specify the number of seconds to wait before forcefully closing client connections and shutting down the subcluster. If you set a negative -n value, the subcluster is marked as draining but is not shut down until all active user sessions disconnect.

In the following example, the subcluster named analytics initially has an active client session, but the session closes before the timeout limit is reached and the subcluster shuts down:

```
$ admintools -t stop_subcluster -d verticadb -c analytics --password password --drain-seconds 200
--- Subcluster shutdown ---
Verifying subcluster 'analytics'
Node 'v_verticadb_node0004' will shutdown
Node 'v_verticadb_node0005' will shutdown
Node 'v_verticadb_node0006' will shutdown
Connecting to database to begin shutdown of subcluster 'analytics'
Shutdown will use connection draining.
Shutdown will wait for all client sessions to complete, up to 200 seconds
Then it will force a shutdown.
Poller has been running for 0:00:00.000022 seconds since 2022-07-28 12:18:04.891781

-------------------------------------------------------------
client_sessions    |node_count         |node_names
-------------------------------------------------------------
0                  |5                  |v_verticadb_node0002,v_verticadb_node0004,v_verticadb_node0003,v_verticadb_node0...
1                  |1                  |v_verticadb_node0005
STATUS: vertica.engine.api.db_client.module is still running on 1 host: nodeIP as of 2022-07-28 12:18:14. See /opt/vertica/log/adminTools.log for full details.
Poller has been running for 0:00:10.383018 seconds since 2022-07-28 12:18:04.891781

...

-------------------------------------------------------------
client_sessions    |node_count         |node_names
-------------------------------------------------------------
0                  |3                  |v_verticadb_node0002,v_verticadb_node0001,v_verticadb_node0003
down               |3                  |v_verticadb_node0004,v_verticadb_node0005,v_verticadb_node0006
Stopping poller drain_status because it was canceled
SUCCESS running the shutdown metafunction
Not waiting for processes to completely exit
Shutdown operation was successful
```

You can use the -F (or --force) option to shut down a subcluster immediately, without checking for active user sessions or draining the subcluster:

```
$ admintools -t stop_subcluster -d verticadb -c analytics --password password -F
--- Subcluster shutdown ---
Verifying subcluster 'analytics'
Node 'v_verticadb_node0004' will shutdown
Node 'v_verticadb_node0005' will shutdown
Node 'v_verticadb_node0006' will shutdown
Connecting to database to begin shutdown of subcluster 'analytics'
Running shutdown metafunction. Not using connection draining
STATUS: vertica.engine.api.db_client.module is still running on 1 host: 192.168.111.31 as of 2022-07-28 13:13:57. See /opt/vertica/log/adminTools.log for full details.
STATUS: vertica.engine.api.db_client.module is still running on 1 host: 192.168.111.31 as of 2022-07-28 13:14:07. See /opt/vertica/log/adminTools.log for full details.
SUCCESS running the shutdown metafunction
Not waiting for processes to completely exit
Shutdown operation was successful
```

If you want to shut down all subclusters in a database, see Stopping an Eon Mode Database .

Altering subcluster settings

There are several settings you can alter on a subcluster using the ALTER SUBCLUSTER statement. You can also switch a subcluster from a primary to a secondary subcluster, or from a secondary to a primary.

Renaming a subcluster

To rename an existing subcluster, use the ALTER SUBCLUSTER statement's RENAME TO clause:

```
=> ALTER SUBCLUSTER default_subcluster RENAME TO load_subcluster;
ALTER SUBCLUSTER

=> SELECT DISTINCT subcluster_name FROM subclusters;
  subcluster_name
-------------------
 load_subcluster
 analytics_cluster
(2 rows)
```

Changing the default subcluster

The default subcluster designates which subcluster Vertica adds nodes to if you do not explicitly specify a subcluster when adding nodes to the database. When you create a new database (or when a database is upgraded from a version prior to 9.3.0) the default_subcluster is the default. You can find the current default subcluster by querying the is_default column of the SUBCLUSTERS system table.

The following example demonstrates finding the default subcluster, and then changing it to the subcluster named analytics_cluster:

```
=> SELECT DISTINCT subcluster_name FROM SUBCLUSTERS WHERE is_default = true;
  subcluster_name
--------------------
 default_subcluster
(1 row)

=> ALTER SUBCLUSTER analytics_cluster SET DEFAULT;
ALTER SUBCLUSTER
=> SELECT DISTINCT subcluster_name FROM SUBCLUSTERS WHERE is_default = true;
  subcluster_name
-------------------
 analytics_cluster
(1 row)
```

Converting a subcluster from primary to secondary, or secondary to primary

You usually choose whether a subcluster is primary or secondary when creating it (see Creating subclusters for more information). However, you can switch a subcluster between the two settings after you have created it. You may want to change whether a subcluster is primary or secondary to impact the K-safety of your database. For example, if you have a single primary subcluster that has down nodes that you cannot easily replace, you can promote a secondary subcluster to primary to ensure losing another primary node will not cause your database to shut down. On the oither hand, you may choose to convert a primary subcluster to a secondary before eventually shutting it down. This conversion can prevent the database from losing K-Safety if the subcluster you are shutting down contains half or more of the total number of primary nodes in the database.

> **Note**
> You cannot promote or demote a subcluster containing the initiator node . You must be connected to a node in a subcluster other than the one you want to promote or demote.

To make a secondary subcluster into a primary subcluster, use the PROMOTE_SUBCLUSTER_TO_PRIMARY function:

```
=> SELECT DISTINCT subcluster_name, is_primary from subclusters;
  subcluster_name  | is_primary
-------------------+------------
 analytics_cluster | f
 load_subcluster   | t
(2 rows)


=> SELECT PROMOTE_SUBCLUSTER_TO_PRIMARY('analytics_cluster');
 PROMOTE_SUBCLUSTER_TO_PRIMARY
--------------------------------
 PROMOTE SUBCLUSTER TO PRIMARY
(1 row)


=> SELECT DISTINCT subcluster_name, is_primary from subclusters;
  subcluster_name  | is_primary
-------------------+------------
 analytics_cluster | t
 load_subcluster   | t
(2 rows)
```

Making a primary subcluster into a secondary subcluster is similar. Unlike converting a secondary subcluster to a primary, there are several issues that may prevent you from making a primary into a secondary. Vertica prevents you from making a primary into a secondary if any of the following is true:

- The subcluster contains a critical node .
- The subcluster is the only primary subcluster in the database. You must have at least one primary subcluster.
- The initiator node is a member of the subcluster you are trying to demote. You must call DEMOTE_SUBCLUSTER_TO_SECONDARY from another subcluster.

To convert a primary subcluster to secondary, use the DEMOTE_SUBCLUSTER_TO_SECONDARY function:

```
=> SELECT DISTINCT subcluster_name, is_primary from subclusters;
  subcluster_name  | is_primary
-------------------+------------
 analytics_cluster | t
 load_subcluster   | t
(2 rows)

=> SELECT DEMOTE_SUBCLUSTER_TO_SECONDARY('analytics_cluster');
 DEMOTE_SUBCLUSTER_TO_SECONDARY
---------------------------------
 DEMOTE SUBCLUSTER TO SECONDARY
(1 row)

=> SELECT DISTINCT subcluster_name, is_primary from subclusters;
  subcluster_name  | is_primary
-------------------+------------
 analytics_cluster | f
 load_subcluster   | t
(2 rows)
```

Removing subclusters

Removing a subcluster from the database deletes the subcluster from the Vertica catalog. During the removal, Vertica removes any nodes in the subcluster from the database. These nodes are still part of the database cluster, but are no longer part of the database. If you view your cluster in the MC, you will see these nodes with the status STANDBY. They can be added back to the database by adding them to another subcluster. See Creating subclusters and Adding New Nodes to a Subcluster .

Vertica places several restrictions on removing a subcluster:

- You cannot remove the default subcluster . If you want to remove the subcluster that is set as the default, you must make another subcluster the default. See Changing the Default Subcluster for details.
- You cannot remove the last primary subcluster in the database. Your database must always have at least one primary subcluster.

> **Note**
> Removing a subcluster can fail if the database is repartitioning. If this happens, you will see the error message "Transaction commit aborted because session subscriptions do not match catalog." Wait until the repartitioning is done before removing a subcluster.

To remove a subcluster, use the admintools command line db_remove_subcluster tool:

```
$ adminTools -t db_remove_subcluster -h
Usage: db_remove_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be modified
  -c SCNAME, --subcluster=SCNAME
                        Name of subcluster to be removed
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
  --skip-directory-cleanup
                        Caution: this option will force you to do a manual
                        cleanup. This option skips directory deletion during
                        remove subcluster. This is best used in a cloud
                        environment where the hosts being removed will be
                        subsequently discarded.
```

This example removes the subcluster named analytics_cluster:

```
$ adminTools -t db_remove_subcluster -d verticadb -c analytics_cluster -p 'password'
Found node v_verticadb_node0004 in subcluster analytics_cluster
Found node v_verticadb_node0005 in subcluster analytics_cluster
Found node v_verticadb_node0006 in subcluster analytics_cluster
Found node v_verticadb_node0007 in subcluster analytics_cluster
Waiting for rebalance shards. We will wait for at most 36000 seconds.
Rebalance shards polling iteration number [0], started at [17:09:35], time
   out at [03:09:35]
Attempting to drop node v_verticadb_node0004 ( 10.11.12.40 )
   Shutting down node v_verticadb_node0004
   Sending node shutdown command to '['v_verticadb_node0004', '10.11.12.40',
      '/vertica/data', '/vertica/data']'
   Deleting catalog and data directories
   Update admintools metadata for v_verticadb_node0004
   Eon mode detected. The node v_verticadb_node0004 has been removed from
      host 10.11.12.40. To remove the node metadata completely, please clean
      up the files corresponding to this node, at the communal location:
      s3://eonbucket/verticadb/metadata/verticadb/nodes/v_verticadb_node0004
Attempting to drop node v_verticadb_node0005 ( 10.11.12.50 )
   Shutting down node v_verticadb_node0005
   Sending node shutdown command to '['v_verticadb_node0005', '10.11.12.50',
      '/vertica/data', '/vertica/data']'
   Deleting catalog and data directories
   Update admintools metadata for v_verticadb_node0005
   Eon mode detected. The node v_verticadb_node0005 has been removed from
      host 10.11.12.50. To remove the node metadata completely, please clean
      up the files corresponding to this node, at the communal location:
      s3://eonbucket/verticadb/metadata/verticadb/nodes/v_verticadb_node0005
Attempting to drop node v_verticadb_node0006 ( 10.11.12.60 )
   Shutting down node v_verticadb_node0006
   Sending node shutdown command to '['v_verticadb_node0006', '10.11.12.60',
      '/vertica/data', '/vertica/data']'
   Deleting catalog and data directories
   Update admintools metadata for v_verticadb_node0006
   Eon mode detected. The node v_verticadb_node0006 has been removed from
      host 10.11.12.60. To remove the node metadata completely, please clean
      up the files corresponding to this node, at the communal location:
      s3://eonbucket/verticadb/metadata/verticadb/nodes/v_verticadb_node0006
Attempting to drop node v_verticadb_node0007 ( 10.11.12.70 )
   Shutting down node v_verticadb_node0007
   Sending node shutdown command to '['v_verticadb_node0007', '10.11.12.70',
      '/vertica/data', '/vertica/data']'
   Deleting catalog and data directories
   Update admintools metadata for v_verticadb_node0007
   Eon mode detected. The node v_verticadb_node0007 has been removed from
      host 10.11.12.70. To remove the node metadata completely, please clean
      up the files corresponding to this node, at the communal location:
      s3://eonbucket/verticadb/metadata/verticadb/nodes/v_verticadb_node0007
   Reload spread configuration
   Replicating configuration to all nodes
   Checking database state
   Node Status: v_verticadb_node0001: (UP) v_verticadb_node0002: (UP)
      v_verticadb_node0003: (UP)
Communal storage detected: syncing catalog
```

Depot management

The nodes of an Eon Mode database fetch data from communal storage as needed to process queries, and cache that data locally on disk. The cached data of all nodes within a subcluster comprise that cluster's *depot* . Vertica uses depots to facilitate query execution: when processing a query, Vertica first checks the current depot for the required data. If the data is unavailable, Vertica fetches it from communal storage and saves a copy in the depot

to expedite future queries. Vertica also uses the depot for load operations, caching newly-loaded data in the depot before uploading it to communal storage.

In this section

- Managing depot caching
- Resizing depot caching capacity

Managing depot caching

You can control depot caching in several ways:

- Configure gateway parameters so a depot caches only queried data or loaded data.
- Control fetching of queried data from communal storage.
- Manage eviction of cached data.
- Enable depot warming on new and restarted nodes.

You can monitor depot activity and settings with several V_MONITOR system tables, or with the Management Console .

> **Note**
> Depot caching is supported only on primary shard subscriber nodes.

Depot gateway parameters

Vertica depots can cache two types of data:

- Queried data: The depot facilitates query execution by fetching queried data from communal storage and caching it in the depot. The cached data remains available until it is evicted to make room for fresher data, or for data that is fetched for more recent queries.
- Loaded data: The depot expedites load operations such as COPY by temporarily caching data until it is uploaded to communal storage.

By default, depots are configured to cache both types of data.

Two configuration parameters determine whether a depot caches queried or loaded data:

Parameter	Settings
UseDepotForReads	Boolean: • 1 (default): Search the depot for the queried data; if not found, fetch the data from communal storage. • 0 : Bypass the depot and get queried data from communal storage.
UseDepotForWrites	Boolean: • 1 (default): Write loaded data to the depot, then upload files to communal storage. • 0 : Bypass the depot and write directly to communal storage.

Both parameters can be set at session, user and database levels.

If set on the session or user levels, these parameters can be used to segregate read and write activity on the depots of different subclusters. For example, parameters UseDepotForReads and UseDepotForWrites might be set as follows for users joe and rhonda :

```
=> SHOW USER joe ALL;
         name          | setting
-----------------------+---------
 UseDepotForReads      | 1
 UseDepotForWrites     | 0
(2 rows)
=> SHOW USER rhonda ALL;
         name          | setting
-----------------------+---------
 UseDepotForReads      | 0
 UseDepotForWrites     | 1
(2 rows)
```

Given these user settings, when joe connects to a Vertica subcluster, his session only uses the current depot to process queries; all load operations are uploaded to communal storage. Conversely, rhonda 's sessions only use the depot to process load operations; all queries must fetch their data from communal storage.

Depot fetching

If a depot is enabled to cache queried data (UseDepotForReads = 1), you can configure how it fetches data from communal storage with configuration parameter DepotOperationsForQuery . This parameter has three settings:

- ALL (default): Fetch file data from communal storage, if necessary displace existing files by evicting them from the depot.
- FETCHES : Fetch file data from communal storage only if space is available; otherwise, read the queried data directly from communal storage.
- NONE : Do not fetch file data to the depot, read the queried data directly from communal storage.

You can set fetching behavior at four levels, in ascending levels of precedence:

- Database: ALTER DATABASE...SET PARAMETER
- Per user: ALTER USER...SET PARAMETER
- Per session: ALTER SESSION...SET PARAMETER
- Per query: DEPOT_FETCH hint

For example, you can set DepotOperationsForQuery at the database level as follows:

```
=> ALTER DATABASE default SET PARAMETER DepotOperationsForQuery = FETCHES;
ALTER DATABASE
```

This setting applies to all database depots unless overridden at other levels. For example, the following ALTER USER statement specifies fetching behavior for a depot when it processes queries from user joe :

```
=> ALTER USER joe SET PARAMETER DepotOperationsForQuery = ALL;
ALTER USER
```

Finally, joe can override his own DepotOperationsForQuery setting by including the DEPOT_FETCH hint in individual queries:

```
SELECT /*+DEPOT_FETCH(NONE)*/ count(*) FROM bar;
```

Evicting depot data

In general, Vertica evicts data from the depot as needed to provide room for new data, and expedite request processing. Before writing new data to the depot, Vertica evaluates it as follows:

- Data fetched from communal storage: Vertica sizes the download and evicts data from the depot accordingly.
- Data uploaded from a DML operation such as COPY: Vertica cannot estimate the total size of the upload before it is complete, so it sizes individual buffers and evicts data from the depot as needed.

In both cases, Vertica assesses existing depot data and determines which objects to evict from the depot as follows, in descending order of precedence (most to least vulnerable):

1. Least recently used unpinned object evicted for any new object, pinned or unpinned.
2. Least recently used pinned object evicted for a new pinned object.

Pinning depot objects

You can set depot pinning policies on database objects to reduce their exposure to eviction. Pinning policies can be set on individual subclusters, or on the entire database, and at different levels of granularity—table, projection, and partitions:

Pinning of...	Supported by...
Tables	SET_DEPOT_PIN_POLICY_TABLE
Projections	SET_DEPOT_PIN_POLICY_PROJECTION
Partitions	SET_DEPOT_PIN_POLICY_PARTITION

By default, pinned objects are queued for download from communal storage as needed to execute a query or DML operation. SET_DEPOT_PIN_POLICY functions can specify to override this behavior and immediately queue newly pinned objects for download: set the last Boolean argument of the function to true .

In the following example, SET_DEPOT_PIN_POLICY_TABLE pins the data of table foo and specifies to queue the data immediately for download:

```
=> SELECT SET_DEPOT_PIN_POLICY_TABLE ('foo', 'default_subluster', true );
```

> **Tip**
> How soon Vertica downloads a pinned object from communal storage depends on a number of factors, including space availability and precedence of other pinned objects that are queued for download. You can force immediate download of queued objects by calling FINISH_FETCHING_FILES .

Usage guidelines

Pinning one or more objects on a depot affects its retention of fetched (queried) data and uploaded (newly loaded) data. If too much depot space is claimed by pinned objects, the depot might be unable to handle load operations on unpinned objects. In this case, set configuration parameter UseDepotForWrites to 0, so load operations are routed directly to communal storage for processing. Otherwise, load operations are liable to return with an error.

To minimize contention over depot usage, consider the following guidelines:

- Pin only those objects that are most active in DML operations and queries.
- Minimize the size of pinned data by setting policies at the smallest effective level—for example, pin only the data of a table's active partition .

Depot warming

On startup, the depots of new nodes are empty, while the depots of restarted nodes often contain stale data that must be refreshed. When depot warming is enabled, a node that is undergoing startup preemptively loads its depot with frequently queried and pinned data. When the node completes startup and begins to execute queries, its depot already contains much of the data it needs to process those queries. This reduces the need to fetch data from communal storage, and expedites query performance accordingly.

> **Note**
> Fetching data to a warming depot can delay node startup.

By default, depot warming is disabled (EnableDepotWarmingFromPeers = 0). A node executes depot warming as follows:

1. The node checks configuration parameter PreFetchPinnedObjectsToDepotAtStartup. If enabled (set to 1), the node:
   - Gets from the database catalog a list of all objects that are pinned on this node's subcluster.
   - Queues the pinned objects for fetching and calculates their total size.
2. The node checks configuration parameter EnableDepotWarmingFromPeers. If enabled (set to 1), the node:
   - Identifies a peer node in the same subcluster whose depot contents it can copy.
   - After taking into account all pinned objects, calculates how much space remains available in the warming depot.
   - Gets from the peer node a list of the most recently used objects that can fit in the depot.
   - Queues the objects for fetching.
3. If BackgroundDepotWarming is enabled (set to 1, default), the node loads queued objects into its depot while it is warming, and continues to do so in the background after the node becomes active and starts executing queries. Otherwise (BackgroundDepotWarming = 0), node activation is deferred until the depot fetches and loads all queued objects.

Monitoring the depot

You can monitor depot activity and settings with several V_MONITOR system tables.

Tip
You can also use the Management Console to monitor depot activity. For details, see Monitoring depot activity with MC

System table...	Shows...
DATA_READS	All storage locations that a query reads to obtain data.
DEPOT_EVICTIONS	Details about objects that were evicted from the depot.
DEPOT_FETCH_QUEUE	Pending depot requests for queried file data to fetch from communal storage.
DEPOT_FILES	Objects that are cached in database depots.
DEPOT_PIN_POLICIES	Objects —tables and table partitions—that are pinned to database depots.
DEPOT_SIZES	Depot caching capacity per node.
DEPOT_UPLOADS	Details about depot uploads to communal storage.

Resizing depot caching capacity

Each node in an Eon database caches depot data in a predefined storage location. The storage location path depends on your Vertica installation's filesystem. By default, each node in a cluster can use up to 60 percent of disk space on the storage location's filesystem to cache depot data. You can change caching capacity with ALTER_LOCATION_SIZE , by specifying to a fixed size or a percentage of total disk space. The function can specify a single node, a subcluster, or all nodes in the database cluster. You can increase depot caching capacity for each node up to 80 percent.

In the following example, ALTER_LOCATION_SIZE increases depot caching capacity to 80 percent of disk space on the storage location's filesystem. The function supplies an empty string as the second (*node-name*) argument, so the change applies to all nodes:

Important
By default, depot caching capacity cannot exceed 80 percent of disk space on the store location file system; attempts to set it to a higher value return an error. Vertica requires at least 20 percent of disk space for the catalog, Data Collector tables, and temp files.

```
=> SELECT node_name, location_label, location_path, max_size, disk_percent FROM storage_locations WHERE location_usage = 'DEPOT' ORDER BY
node_name;
    node_name     | location_label |      location_path      |  max_size   | disk_percent
------------------+----------------+-------------------------+-------------+--------------
 v_vmart_node0001 | auto-data-depot | /home/dbadmin/verticadb | 36060108800 | 70%
 v_vmart_node0002 | auto-data-depot | /home/dbadmin/verticadb | 36059377664 | 70%
 v_vmart_node0003 | auto-data-depot | /home/dbadmin/verticadb | 36060108800 | 70%
(3 rows)

=> SELECT alter_location_size('depot', '','80%');
 alter_location_size
---------------------
 depotSize changed.
(1 row)

=> SELECT node_name, location_label, location_path, max_size, disk_percent FROM storage_locations WHERE location_usage = 'DEPOT' ORDER BY
node_name;
    node_name     | location_label |      location_path      |  max_size   | disk_percent
------------------+----------------+-------------------------+-------------+--------------
 v_vmart_node0001 | auto-data-depot | /home/dbadmin/verticadb | 41211552768 | 80%
 v_vmart_node0002 | auto-data-depot | /home/dbadmin/verticadb | 41210717184 | 80%
 v_vmart_node0003 | auto-data-depot | /home/dbadmin/verticadb | 41211552768 | 80%
(3 rows)
```

Rescaling depot capacity

When a database is revived on an instance with greater or lesser disk space than it had previously, Vertica evaluates the depot size settings that were previously in effect. If depot size was specified as a percentage of available disk space, Vertica proportionately rescales depot capacity. For example, if depot caching capacity for a given node was set to 70 percent, the revived node applies that setting to the new disk space and adjusts depot caching capacity accordingly. If depot capacity was set to a fixed size, Vertica applies that setting, unless doing so will consume more than 80 percent of available disk space. In that case, Vertica automatically adjusts depot size as needed.

Scaling your Eon Mode database

One of the strengths of an Eon Mode database is its ability to grow or shrink to meet your workload demands. You can add nodes to and remove nodes from your database to meet changing workload demands. For an overview of why you would scale your database and how it affects queries, see Elasticity .

Scaling up your database by starting stopped nodes

The easiest way to scale up your database is to start any stopped nodes:

- To start individual nodes, see Stopping and starting nodes on MC .
- To start an entire subcluster that has been stopped, see Starting and stopping subclusters .

Scaling up your database by adding nodes

If you do not have stopped nodes in your database, or the stopped nodes are not in the subclusters where you want to add new nodes, then you can add new nodes to the database. In supported environments, you can use the MC to provision and add new nodes to your database in a single step. See Viewing and managing your cluster for more information.

You can also manually add new nodes:

- For databases running on AWS, see Add nodes to a cluster in AWS using Management Console .
- For databases running on-premises, or in other environments:
  1. Provision the hardware for your new nodes. Be sure to follow the steps in Before you install Vertica to verify your hardware is configured correctly.
  2. Follow the steps in Adding hosts to a cluster to add the new nodes to the database cluster.
  3. Add the node to your database by following the steps in Adding New Nodes to a Subcluster .

Controlling how Vertica uses your new nodes

New nodes can improve your database's performance in one of two ways:

- Increase the query throughput (the number of queries your database processes at the same time).
- Increase individual query performance (how fast each query runs).

See Elasticity for details on these performance improvements. You control how the new nodes improve your database's performance by choosing what subclusters you add them to. The following topics explain how to use scaling to improve throughput and query performance.

In this section

- Change the number of shards in the database
- Improving query throughput using subclusters
- Using elastic crunch scaling to improve query performance
- Manually choosing an ECS strategy

Change the number of shards in the database

The initial number of shards is set when you create a database. You might choose to change the number of shards in a database for the following reasons:

- Improve large subcluster performance. For example, if you have a 24-node subcluster that has 6 shards, the subcluster uses Elastic Crunch Scaling (ECS) to split the responsibility for processing the data in each shard among the nodes. Re-sharding the database to 24 shards avoids the necessity of ECS and improves performance as ECS is not as efficient as having a one-to-one shard to node ratio. For more information, see Using elastic crunch scaling to improve query performance .
- Reduce catalog size. If your catalog size has grown due to a high number of shards in your database, you might choose to reduce the number of shards.
- Improve performance after migrating from Enterprise Mode to Eon Mode. When you migrate your database from Enterprise Mode to Eon Mode, the number of shards in your Eon database is initially set to the number of nodes that you had in your Enterprise database. This default number of shards might not be ideal. For details, see Choosing the Number of Shards and the Initial Node Count .
- Scale your database effectively. To evenly distribute work among nodes, the number of nodes in the database should be a multiple, or even a divisor, of the number of shards. You might re-shard your database if you plan to scale the subclusters to a size that is incompatible with this guidance. For example, a database with seven shards should only have subclusters that have a multiple of seven nodes. Choosing a shard count

with more divisors, such as eight, gives you greater flexibility in choosing the number of nodes in a subcluster.

You should not re-shard your database every time you scale subclusters. While in progress, re-sharding might affect the database's performance. After re-sharding, the storage containers on the subcluster are not immediately aligned with the new shard subscription bounds. This misalignment adds overhead to query execution.

Re-sharding an Eon Mode database

To re-shard your database, call the RESHARD_DATABASE function with the new shard count as the argument. This function takes a global catalog lock, so avoid running it during busy periods or when performing heavy ETL loads. The runtime depends on the size of your catalog.

After RESHARD_DATABASE completes, the nodes in the cluster use the new catalog shard definitions. However, the re-sharding process does not immediately alter the storage containers in communal storage. The shards continue to point to the existing storage containers. For example, if you double the number of shards in your database, each storage container now has two associated shards. During queries, each node filters out the data in the storage containers that does not apply to its subscribed shard. This adds a small overheard to the query. Eventually, the Tuple Mover's background reflexive mergeout processes automatically update the storage containers so they align with the new shard definitions. You can call DO_TM_TASK to run a 'RESHARDMERGEOUT' task that has the Tuple Mover immediately realign the storage containers.

The following query returns the details of any storage containers that Tuple Mover has not yet realigned:

```
=> SELECT * FROM storage_containers WHERE original_segment_lower_bound IS NOT NULL AND original_segment_upper_bound IS NOT NULL;
```

Example

This example demonstrates the re-sharding process and how it affects shard assignments and storage containers. To illustrate the impact of re-sharding, the shard assignment and storage container details are compared before and after re-sharding. The following three queries return information about the database's shards, node subscriptions, and storage container catalog objects:

```
=> SELECT shard_name, lower_hash_bound, upper_hash_bound FROM shards ORDER BY shard_name;

shard_name  | lower_hash_bound | upper_hash_bound
------------+------------------+------------------
replica     |                  |
segment0001 |                0 |       1073741825
segment0002 |       1073741826 |       2147483649
segment0003 |       2147483650 |       3221225473
segment0004 |       3221225474 |       4294967295
(5 rows)

=> SELECT node_name, shard_name, is_primary, is_resubscribing, is_participating_primary FROM node_subscriptions;

node_name | shard_name  | is_primary | is_resubscribing | is_participating_primary
----------+-------------+------------+------------------+--------------------------
initiator | replica     | t          | f                | t
e0        | replica     | f          | f                | t
e1        | replica     | f          | f                | t
e2        | replica     | f          | f                | t
e0        | segment0002 | t          | f                | t
e1        | segment0003 | t          | f                | t
e2        | segment0004 | t          | f                | t
initiator | segment0001 | t          | f                | t
(8 rows)

=> SELECT node_name, projection_name, storage_oid, sal_storage_id, total_row_count, deleted_row_count, segment_lower_bound,
segment_upper_bound, shard_name FROM storage_containers WHERE projection_name = 't_super';

node_name | projection_name |    storage_oid    |                sal_storage_id                | total_row_count | deleted_row_count | segment_lower_bound |
segment_upper_bound | shard_name
----------+-----------------+-------------------+----------------------------------------------+-----------------+-------------------+---------------------+---------------------+------------
initiator | t_super         | 45035996273842990 | 022e836bff54b0aed318df2fe73b5afe00a0000000021b2d |               4 |                 0 |                   0 |
1073741825 | segment0001
e0        | t_super         | 49539595901213486 | 024bbf043c1ca3f5c7a86a423fc7e1e300b0000000021b2d |               3 |                 0 |          1073741826 |
2147483649 | segment0002
e1        | t_super         | 54043195528583990 | 02dac7dc405a1620c92bae1a17c7bbad00c0000000021b35 |               8 |                 0 |          2147483650 |
3221225473 | segment0003
e2        | t_super         | 54043195528583992 | 02dac7dc405a1620c92bae1a17c7bbad00c0000000021b31 |               6 |                 0 |          3221225474 |
4294967295 | segment0004
(4 rows)
```

The following call to RESHARD_DATABASE changes the number of shards to eight:

```
=> SELECT RESHARD_DATABASE(8);

              RESHARD_DATABASE
-------------------------------------------------------------
The database has been re-sharded from 4 shards to 8 shards
(1 row)
```

You can use the following query to view the database's new shard definitions:

```
=> SELECT shard_name, lower_hash_bound, upper_hash_bound FROM shards ORDER BY shard_name;

shard_name  | lower_hash_bound | upper_hash_bound
------------+------------------+------------------
replica     |                  |
segment0001 |                0 |        536870913
segment0002 |        536870914 |       1073741825
segment0003 |       1073741826 |       1610612737
segment0004 |       1610612738 |       2147483649
segment0005 |       2147483650 |       2684354561
segment0006 |       2684354562 |       3221225473
segment0007 |       3221225474 |       3758096385
segment0008 |       3758096386 |       4294967295
(9 rows)
```

The database now has eight shards. Because re-sharding cut the boundary range of each shard in half, each shard is responsible for about half as much of the communal storage data.

The following query returns the database's new node subscriptions:

```
=> SELECT node_name, shard_name, is_primary, is_resubscribing, is_participating_primary FROM node_subscriptions;

node_name | shard_name  | is_primary | is_resubscribing | is_participating_primary
----------+-------------+------------+------------------+--------------------------
initiator | replica     | t          | f                | t
e0        | replica     | f          | f                | t
e1        | replica     | f          | f                | t
e2        | replica     | f          | f                | t
initiator | segment0001 | t          | f                | t
e0        | segment0002 | t          | f                | t
e1        | segment0003 | t          | f                | t
e2        | segment0004 | t          | f                | t
initiator | segment0005 | t          | f                | t
e0        | segment0006 | t          | f                | t
e1        | segment0007 | t          | f                | t
e2        | segment0008 | t          | f                | t
(12 rows)
```

After re-sharding, each node now subscribes to two shards instead of one.

You can use the following query to see how re-sharding affected the database's storage container catalog objects:

```
=> SELECT node_name, projection_name, storage_oid, sal_storage_id, total_row_count, deleted_row_count, segment_lower_bound,
segment_upper_bound, shard_name FROM storage_containers WHERE projection_name = 't_super';

node_name | projection_name |    storage_oid    |                sal_storage_id                | total_row_count | deleted_row_count | segment_lower_bound |
segment_upper_bound | shard_name
----------+-----------------+-------------------+----------------------------------------------+-----------------+-------------------+---------------------+---------------------+------------
initiator | t_super         | 45035996273843145 | 02dac7dc405a1620c92bae1a17c7bbad00c0000000021b35 |               8 |                 0 |          2147483650 |
3221225473 | segment0005
initiator | t_super         | 45035996273843149 | 022e836bff54b0aed318df2fe73b5afe00a0000000021b2d |               4 |                 0 |                   0 |
1073741825 | segment0001
e0        | t_super         | 49539595901213641 | 02dac7dc405a1620c92bae1a17c7bbad00c0000000021b35 |               8 |                 0 |          2147483650 |
3221225473 | segment0006
e0        | t_super         | 49539595901213645 | 022e836bff54b0aed318df2fe73b5afe00a0000000021b2d |               4 |                 0 |                   0 |
1073741825 | segment0002
e1        | t_super         | 54043195528584141 | 02dac7dc405a1620c92bae1a17c7bbad00c0000000021b31 |               6 |                 0 |          3221225474 |
4294967295 | segment0007
e1        | t_super         | 54043195528584143 | 02dac7dc405a1620c92bae1a17c7bbad00c0000000021b31 |               6 |                 0 |          1073741826 |
2147483649 | segment0003
e2        | t_super         | 54043195528584137 | 024bbf043c1ca3f5c7a86a423fc7e1e300b0000000021b2d |               3 |                 0 |          3221225474 |
4294967295 | segment0008
e2        | t_super         | 54043195528584139 | 024bbf043c1ca3f5c7a86a423fc7e1e300b0000000021b2d |               3 |                 0 |          1073741826 |
2147483649 | segment0004
(8 rows)
```

The shards point to storage files with the same sal_storage_id as before the re-shard. Eventually, the TM's mergeout processes will automatically update the storage containers.

You can query the RESHARDING_EVENTS system table for information about current and historical resharding operations, such as a node's previous shard subscription bounds and the current status of the resharding operation:

```
=> SELECT node_name, running_status, old_shard_name, old_shard_lower_bound, old_shard_upper_bound FROM RESHARDING_EVENTS;
node_name | running_status  | old_shard_name |   old_shard_lower_bound   |   old_shard_upper_bound
----------+-----------------+----------------+---------------------------+-------------------------
e0        | Running         | segment0001    |                         0 |              1073741825
e0        | Running         | segment0002    |                1073741826 |              2147483649
e0        | Running         | segment0003    |                2147483650 |              3221225473
e0        | Running         | segment0004    |                3221225474 |              4294967295
e1        | Running         | segment0001    |                         0 |              1073741825
e1        | Running         | segment0002    |                1073741826 |              2147483649
e1        | Running         | segment0003    |                2147483650 |              3221225473
e1        | Running         | segment0004    |                3221225474 |              4294967295
initiator | Running         | segment0001    |                         0 |              1073741825
initiator | Running         | segment0002    |                1073741826 |              2147483649
initiator | Running         | segment0003    |                2147483650 |              3221225473
initiator | Running         | segment0004    |                3221225474 |              4294967295
(12 rows)
```

Improving query throughput using subclusters

Improving query throughput increases the number of queries your Eon Mode database processes at the same time. You are usually concerned about your database's throughput when your workload consists of many short-running queries. They are often referred to as "dashboard queries." This term describes type of workload you see when a large number of users have web-based dashboard pages open to monitor some sort of status. These dashboards tend to update frequently, using simpler, short-running queries instead of analytics-heavy long running queries.

The best way to improve your database's throughput is to add new subclusters to the database or start any stopped subclusters. Then distribute the client connections among these subclusters using connection load balancing policies. Subclusters independently process queries. By adding more subclusters, you improve your database's parallelism.

For the best performance, make the number of nodes in your subcluster the same as the number of shards in your database. If you choose to have less nodes than the number of shards, make the number of nodes an even divisor of the number of shards. When the number of shards is divisible by the number of nodes, the data in your database is equally divided among the nodes in the subcluster.

The easiest way of adding subclusters is to use the MC:

1. From the MC home page, click the database you want to add subclusters to.
2. Click **Manage** .
3. Click **Add Subcluster** .
4. Follow the steps in the wizard to add the subcluster. Normally, the only items you need to fill in are the subcluster name and the number of instances to add to it.

> **Note**
> The MC currently does not support creating instances on all platforms. For those platforms where the MC does not support instances, you can manually add subclusters. See Creating subclusters for more information.

Distributing clients among the throughput subclusters

To gain benefits from the added subclusters, you must have clients that will execute short-running queries connect to the nodes that the subclusters contain. Queries run only on the subcluster that contains the initiator node (the node that the client is connected to). Use connection load balancing policies to spread the connections across all of the subclusters you created to increase query throughput. See Connection load balancing policies for details.

The following example creates a load balancing policy that spreads client connections across two three-node subclusters named query_pool_a and query_pool_b. This example:

- Creates network addresses on the six nodes that are in the two subclusters.
- Creates a load balance group from all the nodes in the two subclusters.
- Creates the routing rule to redirect all incoming connections to the two subclusters.

```
=> CREATE NETWORK ADDRESS node04 ON v_verticadb_node0004 WITH '203.0.113.1';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node05 ON v_verticadb_node0005 WITH '203.0.113.2';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node06 ON v_verticadb_node0006 WITH '203.0.113.3';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node07 ON v_verticadb_node0007 WITH '203.0.113.4';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node08 ON v_verticadb_node0008 WITH '203.0.113.5';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node09 ON v_verticadb_node0009 WITH '203.0.113.6';
CREATE NETWORK ADDRESS

=> CREATE LOAD BALANCE GROUP query_subclusters WITH SUBCLUSTER query_pool_a,
   query_pool_b FILTER '0.0.0.0/0';
CREATE LOAD BALANCE GROUP
=> CREATE ROUTING RULE query_clients ROUTE '0.0.0.0/0' TO query_subclusters;
CREATE ROUTING RULE
```

Important

In cloud environments where clients will be connecting from outside the private network, use the external IP address for each node when creating the network addresses. Otherwise, external clients will not be able to connect to the nodes.

If you have a mix of internal and external clients, set up two network addresses for each node and add them to two separate load balancing groups: one for internal clients and another for external. Then create two routing rules one that routes the internal clients to the internal group, and another that routes the external clients to the external group. Make the routing rule for internal clients only apply to the virtual private networks where your internal clients will connect from (for example, 10.0.0.0/8). The routing rule for the external clients can use the 0.0.0.0/0 CIDR range (all IP addresses) as the incoming connection address range. The rules will work correctly together because the more-specific internal client routing rule takes precedence over the less-restrictive external client rule.

After creating the policy, any client that opts into load balancing is redirected to one of the nodes in the two subclusters. For example, when you connect to node 1 in the cluster (with the IP address 203.0.113.1) using vsql with the -C flag, you see output similar to this:

```
$ vsql -h 203.0.113.1 -U dbadmin -w mypassword -C
Welcome to vsql, the Vertica Analytic Database interactive terminal.

Type:  \h or \? for help with vsql commands
       \g or terminate with semicolon to execute query
       \q to quit

SSL connection (cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, protocol: TLSv1.2)

INFO: Connected using a load-balanced connection.
INFO: Connected to 203.0.113.7 at port 5433.
=>
```

Connection load balancing policies take into account nodes that are stopped when picking a node to handle a client connection. If you shut down one or more subclusters to save money during low-demand periods, you do not need to adjust your load balancing policy as long as some of the nodes are still up.

Using elastic crunch scaling to improve query performance

You can choose to add nodes to your database to improve the performance of complex long-running analytic queries. Adding nodes helps these queries run faster.

When you have more nodes in a subcluster than you have shards in your database, multiple nodes subscribe to each shard. To involve all of the nodes in the subcluster in queries, the Vertica query optimizer automatically uses a feature called Elastic Crunch Scaling (ECS). This feature splits the responsibility for processing the data in each shard among the nodes that subscribe to it. During a query, each node has less data to process and usually finishes the query faster.

For example, suppose you have a six-node subcluster in a three-shard database. In this subcluster, two nodes subscribe to each shard. When you execute a query, Vertica assigns each node roughly half of the data in the shard it subscribes to. Because all nodes in the subcluster participate in the query, the query usually finishes faster than if only half the nodes had participated.

ECS lets a subcluster that has more nodes than shards act as if the shard count in the database were higher. In a three-shard database, a six-node subcluster acts as if the database has six shards by splitting each shard in half. However, using ECS isn't as efficient as having a higher shard count. In practice, you will see slightly slower query performance on a six-node subcluster in a three shard database than you would see from a six-node subcluster in a six-shard database.

You can call RESHARD_DATABASE to change the number of shards in your database. If the new number of shards is greater than or equal to the number of nodes in the subcluster, the subcluster no longer uses ECS. This will generally lead to faster query performance. However, re-sharding produces a larger catalog size and storage containers that are initially misaligned with the new shard definitions. Until the storage containers are realigned, queries must filter out the data in the storage containers that is outside the new shard bounds. This adds a small overhead to queries. For details, see Change the number of shards in the database .

You can determine when the optimizer will use ECS in a subcluster by querying the V_CATALOG.SESSION_SUBSCRIPTIONS system table and look for nodes whose is_collaborating column is TRUE. Subclusters whose node count is less than or equal to the number of shards in the database only have participating nodes. Subclusters that have more nodes than the database's shard count assign the "extra" nodes the role of collaborators. The differences between the two types of nodes are not important for when you are executing queries. The two types just relate to how Vertica organizes the nodes to execute ECS-enabled queries.

This example shows how to get the list of nodes that are participating or collaborating in resolving queries for the current session:

```
=> SELECT node_name, shard_name, is_collaborating, is_participating
      FROM V_CATALOG.SESSION_SUBSCRIPTIONS
      WHERE is_participating = TRUE OR is_collaborating = TRUE
      ORDER BY shard_name, node_name;
     node_name       | shard_name  | is_collaborating | is_participating
---------------------+-------------+------------------+------------------
 v_verticadb_node0004 | replica     | f                | t
 v_verticadb_node0005 | replica     | f                | t
 v_verticadb_node0006 | replica     | t                | f
 v_verticadb_node0007 | replica     | f                | t
 v_verticadb_node0008 | replica     | t                | f
 v_verticadb_node0009 | replica     | t                | f
 v_verticadb_node0007 | segment0001 | f                | t
 v_verticadb_node0008 | segment0001 | t                | f
 v_verticadb_node0005 | segment0002 | f                | t
 v_verticadb_node0009 | segment0002 | t                | f
 v_verticadb_node0004 | segment0003 | f                | t
 v_verticadb_node0006 | segment0003 | t                | f
(12 rows)
```

You can see that nodes 4, 5, and 7 are participating, and nodes 6, 8, and 9 are collaborating.

You can also see that ECS is enabled by looking at an EXPLAIN plan for a query. At the top of the plan for an ECS-enabled query is the statement "this query involves non-participating nodes." These non-participating nodes are the collaborating nodes that are splitting the data in the shard with the participating nodes. The plan also lists the nodes taking part in the query.

This example shows an explain plan for an ECS-enabled query in a six-node subcluster in a three-shard database:

```
=> EXPLAIN SELECT sales_quantity, sales_dollar_amount, transaction_type, cc_name
    FROM online_sales.online_sales_fact
    INNER JOIN online_sales.call_center_dimension
    ON (online_sales.online_sales_fact.call_center_key
       = online_sales.call_center_dimension.call_center_key
       AND sale_date_key = 156)
    ORDER BY sales_dollar_amount DESC;

                                QUERY PLAN
-------------------------------------------------------------------------------
------------------------------
QUERY PLAN DESCRIPTION:
The execution of this query involves non-participating nodes. Crunch scaling
strategy preserves data segmentation
------------------------------

EXPLAIN SELECT sales_quantity, sales_dollar_amount, transaction_type, cc_name
FROM online_sales.online_sales_fact
INNER JOIN online_sales.call_center_dimension
ON (online_sales.online_sales_fact.call_center_key
   = online_sales.call_center_dimension.call_center_key
   AND sale_date_key = 156)
ORDER BY sales_dollar_amount DESC;

Access Path:
+-SORT [Cost: 6K, Rows: 754K] (PATH ID: 1)
|  Order: online_sales_fact.sales_dollar_amount DESC
|  Execute on: v_verticadb_node0007, v_verticadb_node0004, v_verticadb_node0005,
|     v_verticadb_node0006, v_verticadb_node0008, v_verticadb_node0009
| +---> JOIN MERGEJOIN(inputs presorted) [Cost: 530, Rows: 754K (202 RLE)] (PATH ID: 2)
| | |     Join Cond: (online_sales_fact.call_center_key = call_center_dimension.call_center_key)
| | |     Materialize at Output: online_sales_fact.sales_quantity,
| | |          online_sales_fact.sales_dollar_amount, online_sales_fact.transaction_type
| | |     Execute on: v_verticadb_node0007, v_verticadb_node0004,
| | |          v_verticadb_node0005, v_verticadb_node0006, v_verticadb_node0008,
| | |          v_verticadb_node0009
| | +-- Outer -> STORAGE ACCESS for online_sales_fact [Cost: 13, Rows: 754K (202 RLE)] (PATH ID: 3)
| | | |     Projection: online_sales.online_sales_fact_DBD_18_seg_vmart_b0
| | | |     Materialize: online_sales_fact.call_center_key
| | | |     Filter: (online_sales_fact.sale_date_key = 156)
| | | |     Execute on: v_verticadb_node0007, v_verticadb_node0004,
| | | |          v_verticadb_node0005, v_verticadb_node0006, v_verticadb_node0008,
| | | |          v_verticadb_node0009
| | | |     Runtime Filter: (SIP1(MergeJoin): online_sales_fact.call_center_key)
| | +-- Inner -> STORAGE ACCESS for call_center_dimension [Cost: 17, Rows: 200] (PATH ID: 4)
| | | |     Projection: online_sales.call_center_dimension_DBD_16_seg_vmart_b0
| | | |     Materialize: call_center_dimension.call_center_key, call_center_dimension.cc_name
| | | |     Execute on: v_verticadb_node0007, v_verticadb_node0004,
|               v_verticadb_node0005, v_verticadb_node0006, v_verticadb_node0008,
|               v_verticadb_node0009
. . .
```

Taking advantage of ECS

To take advantage of ECS, create a secondary subcluster where the number of nodes is a multiple of the number of shards in your database. For example, in a 12-shard database, create a subcluster that contains a multiple of 12 nodes such as 24 or 36. The number of nodes must be a multiple of the number of shards to evenly distribute the data across the nodes in the subcluster. See Subclusters for more information.

> **Note**
> Instead of creating a new subcluster, you can add more nodes to an existing secondary subcluster. Just be sure that the number of nodes in the subcluster is a multiple of the shard count.

Once you have created the subcluster, have users connect to it and run their analytic queries. Vertica automatically enables ECS in the subcluster because it has more nodes than there are shards in the database.

How the optimizer assigns data responsibilities to nodes

The optimizer has two strategies to choose from when dividing the data in a shard among its subscribing nodes. One strategy is optimized for queries that use data segmentation. Queries that contain a JOIN or GROUP BY clause rely on data segmentation. The other strategy is for queries that do not need segmentation.

By default, the optimizer automatically chooses the strategy to use. For most queries, the automatically-chosen strategy results in faster query performance. For some queries, you may want to manually override the strategy using hints. In a small number of queries, ECS does not help performance. In these cases, you can disable ECS. See Manually choosing an ECS strategy for details.

Manually choosing an ECS strategy

When the number of nodes in a subcluster is greater than the number of database shards, the Vertica query optimizer uses elastic crunch scaling (ECS) to involve all nodes in processing queries. For each shard, the optimizer divides responsibility for processing shard data among its subscribing nodes, using one of the following strategies:

Strategy	Description
I/O-optimized	Optimizer divides the list of ROS containers in the shard among the subscribing nodes. Use this strategy when nodes must fetch the data for the query from communal storage, rather than the depot. Nodes only fetch the ROS containers they need to resolve the query from communal storage, reducing the amount of data each needs to transfer from communal storage. Due to the arbitrary division of data among the nodes, this strategy does not support query optimizations that rely on data segmentation.
Compute-optimized	Optimizer uses data segmentation to assign portions to each subscribing node. The nodes scan the entire shard, but use sub-segment filtering to find their assigned segments of the data. Use this strategy when most data for the query is in the depot, because nodes must scan the entire contents of the shard. Because this strategy uses data segmentation, it supports optimizations such as local joins that the I/O-optimized strategy cannot.

The optimizer automatically chooses a strategy based on whether the query can take advantage of data segmentation. You can tell which strategy the optimizer chooses for a query by using EXPLAIN. The top of the plan explanation states whether ECS is preserving segmentation. For example, this simple query on a single table does not need to use segmentation, so it uses the I/O-optimized strategy:

```
=> EXPLAIN SELECT employee_last_name,
        employee_first_name,employee_age
        FROM employee_dimension
        ORDER BY employee_age DESC;

                        QUERY PLAN
--------------------------------------------------------------------------------
 ------------------------------
QUERY PLAN DESCRIPTION:
The execution of this query involves non-participating nodes.
Crunch scaling strategy does not preserve data segmentation
------------------------------
. . .
```

A more complex query using a JOIN results in ECS preserving data segmentation by using the compute-optimized strategy. The query plan tells you that segmentation is preserved:

```
=> EXPLAIN SELECT sales_quantity, sales_dollar_amount, transaction_type, cc_name
   FROM online_sales.online_sales_fact
   INNER JOIN online_sales.call_center_dimension
   ON (online_sales.online_sales_fact.call_center_key
      = online_sales.call_center_dimension.call_center_key
      AND sale_date_key = 156)
   ORDER BY sales_dollar_amount DESC;

                              QUERY PLAN
------------------------------------------------------------------------------
 ------------------------------
 QUERY PLAN DESCRIPTION:
 The execution of this query involves non-participating nodes.
 Crunch scaling strategy preserves data segmentation
 ------------------------------
 . . .
```

In most cases, the optimizer chooses the best strategy to use to split the data among the nodes subscribing to the same shard. However, you might occasionally find that some queries perform poorly. In these cases, the query can embed the ECSMODE hint to specify which strategy to use, or even disable ECS.

Setting the ECS strategy for individual queries

You can use the ECSMODE hint in a query to force the optimizer to use a specific ECS strategy (or disable ECS entirely). The ECSMODE hint takes one of the following arguments:

- AUTO : The optimizer chooses the strategy to use, useful only if ECS mode is set at the session level (see Setting the ECS Strategy for the Session or Database).
- IO_OPTIMIZED : Use I/O-optimized strategy.
- COMPUTE_OPTIMIZED : Use compute-optimized strategy.
- NONE : Disable use of ECS for this query. Only participating nodes are involved in query execution; collaborating nodes are not.

The following example shows the query plan for a simple single-table query that is forced to use the compute-optimized strategy:

```
=> EXPLAIN SELECT /*+ECSMode(COMPUTE_OPTIMIZED)*/ employee_last_name,
          employee_first_name,employee_age
          FROM employee_dimension
          ORDER BY employee_age DESC;

                              QUERY PLAN
------------------------------------------------------------------------------
 ------------------------------
 QUERY PLAN DESCRIPTION:
 The execution of this query involves non-participating nodes.
 Crunch scaling strategy preserves data segmentation
 ------------------------------
 . . .
```

This example disable ECS in a six-node cluster in a three-shard database:

```
=> EXPLAIN SELECT /*+ECSMode(NONE)*/ employee_last_name,
           employee_first_name,employee_age
           FROM employee_dimension
           ORDER BY employee_age DESC;

                                QUERY PLAN
-------------------------------------------------------------------------------
 ------------------------------
 QUERY PLAN DESCRIPTION:
 ------------------------------

 EXPLAIN SELECT /*+ECSMode(NONE)*/ employee_last_name,
           employee_first_name,employee_age
           FROM employee_dimension
           ORDER BY employee_age DESC;

 Access Path:
 +-SORT [Cost: 243, Rows: 10K] (PATH ID: 1)
 |  Order: employee_dimension.employee_age DESC
 |  Execute on: v_verticadb_node0007, v_verticadb_node0004, v_verticadb_node0005
 | +---> STORAGE ACCESS for employee_dimension [Cost: 71, Rows: 10K] (PATH ID: 2)
 | |      Projection: public.employee_dimension_DBD_8_seg_vmart_b0
 | |      Materialize: employee_dimension.employee_first_name,
 | |      employee_dimension.employee_last_name, employee_dimension.employee_age
 | |      Execute on: v_verticadb_node0007, v_verticadb_node0004,
 | |         v_verticadb_node0005
 . . .
```

Note that this query plan lacks the "this query involves non-participating nodes" statement, indicating that it does not use ECS. It also lists just three participating nodes. These nodes are marked as participating in the V_CATALOG.SESSION_SUBSCRIPTIONS system table.

Setting the ECS strategy for the session or database

You can use the ECSMode configuration parameter to set the ECS strategy for the current session. This parameter accepts the same values as the ECSMODE hint except NONE , which is valid only for individual queries.

The following example demonstrates using the configuration parameter to force a simple query to use the COMPUTE_OPTIMIZED strategy. It then sets the parameter back to its default value of AUTO :

```
=> EXPLAIN SELECT employee_first_name,employee_age
   FROM employee_dimension ORDER BY employee_age DESC;

                           QUERY PLAN
------------------------------------------------------------------------------
 ------------------------------
 QUERY PLAN DESCRIPTION:
 The execution of this query involves non-participating nodes.
 Crunch scaling strategy does not preserve data segmentation
 ------------------------------
. . .

=> ALTER SESSION SET ECSMode = 'COMPUTE_OPTIMIZED';
ALTER SESSION
=> EXPLAIN SELECT employee_first_name,employee_age
   FROM employee_dimension ORDER BY employee_age DESC;


                           QUERY PLAN
------------------------------------------------------------------------------
 ------------------------------
 QUERY PLAN DESCRIPTION:
 The execution of this query involves non-participating nodes.
 Crunch scaling strategy preserves data segmentation
 ------------------------------
 . . .

=> ALTER SESSION SET ECSMode = 'AUTO';
ALTER SESSION
```

Individual query hints override the session-level settings. This example sets the session default to use COMPUTE_OPTIMIZED , then restores the default behavior for a query by using the ECSMode hint with the value AUTO:

```
=> ALTER SESSION SET ECSMode = 'COMPUTE_OPTIMIZED';
ALTER SESSION
=> EXPLAIN SELECT /*+ECSMode(AUTO)*/ employee_first_name,employee_age
  FROM employee_dimension ORDER BY employee_age DESC;


                           QUERY PLAN
------------------------------------------------------------------------------
 ------------------------------
 QUERY PLAN DESCRIPTION:
 The execution of this query involves non-participating nodes.
 Crunch scaling strategy does not preserve data segmentation
 ------------------------------
```

Note that setting the ECSMode hint to AUTO let the optimizer pick the I/O-optimized strategy (which does not preserve segmentation) instead of using the compute-optimized strategy set at the session level.

You can also set the ECS strategy at the database level using ALTER DATABASE . However, doing so overrides the Vertica optimizer's settings for all users in all subclusters that use ECS. Before setting the ECS strategy at the database level, verify that the majority of the queries run by all users of the ECS-enabled subclusters must have the optimizer's default behavior overridden. If not, then use the session or query-level settings to override the optimizer for just the queries that benefit from a specific strategy.

Subcluster sandboxing

Sandboxing enables you to spin-off a secondary subcluster from an existing cluster, resulting in two mutually isolated clusters that share the same data but do not interfere with each other. The sandboxed subcluster inherits the state of the catalog and data of the main cluster at the time of sandbox creation . As soon as the sandbox is active, the catalog and data of the two clusters are independent and can diverge. Within each cluster, you can perform standard database operations and queries, such as creating new tables or loading libraries, without affecting the other cluster. For example, dropping a table in the sandboxed subcluster does not drop the table in the main cluster, and vice versa.

Sandboxes support many use cases, including the following:

- Testing a new version of Vertica without needing to spin-up a new cluster and reload data. After creating a sandbox, you can upgrade the sandboxed subcluster and experiment with the new functionality. To rejoin the sandboxed subcluster to the main cluster, you just need to downgrade the Vertica version and perform the necessary unsandboxing tasks.
- Trying out features without compromising the consistency of the main cluster. For instance, you could spin-off a sandbox and experiment with external tables using data stored by Apache Iceberg .
- Sharing data with another team by giving them access to a sandboxed subcluster. This gives the other team the necessary data access, but keeps the changes separate from the main cluster. Anything the other team changes in the sandbox, such as dropping tables, would not propagate back to the main cluster.

After removing the sandbox and performing necessary cleanup tasks, the subcluster can rejoin the main cluster.

In this section

- Creating sandboxes
- Removing sandboxes

Creating sandboxes

In order to create a sandbox for a secondary subcluster, all nodes in the subcluster must have a status of UP and collectively provide full-subscription coverage for all shards.

To sandbox a subcluster, use the sandbox_subcluster admintools command:

```
$ adminTools -t sandbox_subcluster -h
Usage: sandbox_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be modified
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  -c SCNAME, --subcluster=SCNAME
                        Name of subcluster to be sandboxed
  -b SBNAME, --sandbox=SBNAME
                        Name of the sandbox
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
```

At command runtime, under a global catalog lock (GCLX), the nodes in the specified subcluster create a checkpoint of the catalog. When these nodes auto-restart in the sandbox, they form a new primary cluster that uses the data and catalog checkpoint from the main cluster. After the nodes successfully restart, the sandboxed subcluster and the main cluster are mutually isolated and can diverge.

While the nodes in the main cluster sync their metadata to */path-to-communal-storage/`metadata`/db_name* , the nodes in the sandbox sync to */path-to-communal-storage/`metadata`/sandbox_name* .

> **Note**
> Though the sandboxed cluster and main cluster are mutually isolated, the nodes in both clusters remain in the same Spread ring. However, if the main cluster and sandboxed cluster partition from each other—which can be caused by events like Spread communication problems—the two clusters are not affected.

You can perform standard database operations and queries, such as loading data or creating new tables, in either cluster without affecting the other cluster. For example, dropping a table in the sandboxed subcluster does not drop the table in the main cluster, and vice versa.

Because both clusters reference the same data files, neither cluster can delete files that existed at the time of sandbox creation. However, the sandbox can remove files that it creates after spin-off from the main cluster. Files in the main cluster are queued for removal, but they are not processed until all active sandboxes are removed.

You cannot nest sandboxes, remove a sandboxed subcluster from the database, or add or remove nodes to an existing sandbox, but you can have multiple individual sandboxes active at the same time.

Examples

The following example sandboxes the sc02 secondary subcluster into a sandbox named sand :

```
$ admintools -t sandbox_subcluster -d verticadb -p password -c sc_02 -b sand

Validating sandboxing conditions

Sandboxing subcluster sc_02 as sand...
Subcluster 'sc_02' has been sandboxed to 'sand'. It is going to auto-restart and re-form.

Checking for sandboxed nodes to be UP...
   Node Status: v_verticadb_node0004: (DOWN) v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
   Node Status: v_verticadb_node0004: (UP) v_verticadb_node0005: (UP) v_verticadb_node0006: (UP)

Sandboxing complete. Subcluster analytics is ready for use
```

If you query the NODES system table from the main cluster, you can see that the sc_02 nodes have a status of UNKNOWN and are listed as member of the sand sandbox:

```
=> SELECT node_name, subcluster_name, node_state, sandbox FROM NODES;
     node_name      |  subcluster_name   | node_state | sandbox
--------------------+--------------------+------------+---------
 v_verticadb_node0001 | default_subcluster | UP         |
 v_verticadb_node0002 | default_subcluster | UP         |
 v_verticadb_node0003 | default_subcluster | UP         |
 v_verticadb_node0004 | sc_02              | UNKNOWN    | sand
 v_verticadb_node0005 | sc_02              | UNKNOWN    | sand
 v_verticadb_node0006 | sc_02              | UNKNOWN    | sand
(6 rows)
```

> **Note**
> If the sandboxed and main clusters are sufficiently incompatible or are Spread-partitioned from each other, the status of the sandboxed nodes might appear as DOWN in the above query.

When you issue the same query on one of the sandboxed nodes, the table shows that the sandboxed nodes are UP and the nodes from the main cluster are UNKNOWN, confirming that the cluster is successfully sandboxed and isolated from the main cluster:

```
=> SELECT node_name, subcluster_name, node_state, sandbox FROM NODES;
     node_name      |  subcluster_name   | node_state | sandbox
--------------------+--------------------+------------+---------
 v_verticadb_node0001 | default_subcluster | UNKNOWN    |
 v_verticadb_node0002 | default_subcluster | UNKNOWN    |
 v_verticadb_node0003 | default_subcluster | UNKNOWN    |
 v_verticadb_node0004 | sc_02              | UP         | sand
 v_verticadb_node0005 | sc_02              | UP         | sand
 v_verticadb_node0006 | sc_02              | UP         | sand
(6 rows)
```

You can now perform standard database operations in either cluster without impacting the other cluster. For instance, if you create a machine learning dataset named train_data in the sandboxed subcluster, the new table does not propagate to the main cluster:

```
--In the sandboxed subcluster
=> CREATE TABLE train_data(time timestamp, Temperature float);
CREATE TABLE

=> COPY train_data FROM LOCAL 'daily-min-temperatures.csv' DELIMITER ',';
 Rows Loaded
-------------
3650
(1 row)

=> SELECT * FROM train_data LIMIT 5;
        time         | Temperature
---------------------+-------------
 1981-01-27 00:00:00 |        19.4
 1981-02-20 00:00:00 |        15.7
 1981-02-27 00:00:00 |        17.5
 1981-03-04 00:00:00 |          16
 1981-04-24 00:00:00 |        11.5
(5 rows)

--In the main cluster
=> SELECT * FROM train_data LIMIT 5;
ERROR 4566:  Relation "train_data" does not exist
```

Similarly, if you drop a table in the main cluster, the table is not subsequently dropped in the sandboxed cluster:

```
--In the main cluster
=> SELECT * FROM transaction_data LIMIT 5;
 first_name | last_name |   store    |  cost   | fraud
------------+-----------+------------+---------+-------
 Adam       | Rice      | Gembucket | 8757.35 | FALSE
 Alan       | Gordon    | Wrapsafe  | 3874.48 | FALSE
 Albert     | Harvey    | Treeflex  | 1558.27 | FALSE
 Andrea     | Bryant    | Greenlam  |  1148.2 | FALSE
 Andrew     | Simmons   | Home Ing  | 8400.03 | FALSE
(5 rows)

=> DROP TABLE transaction_data;
DROP TABLE

--In the sandboxed subcluster
=> SELECT * FROM transaction_data LIMIT 5;
 first_name | last_name |   store    |  cost   | fraud
------------+-----------+------------+---------+-------
 Adam       | Rice      | Gembucket | 8757.35 | FALSE
 Alan       | Gordon    | Wrapsafe  | 3874.48 | FALSE
 Albert     | Harvey    | Treeflex  | 1558.27 | FALSE
 Andrea     | Bryant    | Greenlam  |  1148.2 | FALSE
 Andrew     | Simmons   | Home Ing  | 8400.03 | FALSE
(5 rows)
```

If you decide to upgrade your sandboxed cluster, you can confirm that the main cluster and sandboxed cluster are running two different Vertica versions by comparing their build_info values in the NODES system table:

```
=> SELECT node_name, subcluster_name, node_state, sandbox, build_info FROM NODES;
     node_name      |  subcluster_name   | node_state | sandbox | build_info
--------------------+--------------------+------------+---------+------------
 v_verticadb_node0001 | default_subcluster | UP         |         | v12.0.4-0
 v_verticadb_node0002 | default_subcluster | UP         |         | v12.0.4-0
 v_verticadb_node0003 | default_subcluster | UP         |         | v12.0.4-0
 v_verticadb_node0004 | sc_02              | UNKNOWN    | sand    | v12.0.4-1
 v_verticadb_node0005 | sc_02              | UNKNOWN    | sand    | v12.0.4-1
 v_verticadb_node0006 | sc_02              | UNKNOWN    | sand    | v12.0.4-1
(6 rows)
```

See also

- SANDBOX_SUBCLUSTER
- Removing sandboxes

Removing sandboxes

To remove a sandbox from a subcluster and return that subcluster to the main cluster, you can run the unsandbox_subcluster admintools command:

```
$ adminTools -t unsandbox_subcluster -h
Usage: unsandbox_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be modified
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  -c SCNAME, --subcluster=SCNAME
                        Name of subcluster to be un-sandboxed
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
```

> **Note**
> If you upgraded the Vertica version of the sandboxed subcluster, you must downgrade the version of the subcluster before rejoining it to the main cluster.

The unsandbox_subcluster command stops the nodes in the sandboxed subcluster, changes the metadata in the main cluster that designates the specified subcluster as sandboxed, wipes the node's local catalogs, and then restarts the nodes. After the nodes restart, they rejoin the main cluster and inherit the current state of the main cluster's catalog. The nodes should then be back to their normal state and can be used as expected.

Because the sandbox synced its metadata to the same communal storage location as the main cluster, you must remove the metadata files that were created in the sandbox. Those files can be found by replacing the name of the database in the path to the metadata with the name of the sandbox—for instance, */path-to-communal/`metadata`/sandbox_name* instead of */path-to-communal/`metadata`/db_name* . Removing these files helps avoid problems that might arise from reusing the same sandbox name.

If there are no more active sandboxes, the main cluster can resume the processing of data queued for deletion. To remove any data created in the sandbox, you can run the CLEAN_COMMUNAL_STORAGE function.

You can also unsandbox a subcluster using the UNSANDBOX_SUBCLUSTER meta-function, but you must manually stop the nodes, wipe their catalog subdirectories, run the function, and restart the nodes.

Examples

The following command unsandboxes the sc02 secondary subcluster from the sand sandbox. The command stops the nodes in sc02 , wipes the nodes catalogs, and then restarts the nodes. After the nodes restart, they should rejoin the main cluster and be ready for normal use:

```
$ admintools -t unsandbox_subcluster -d verticadb -p vertica -c analytics
Stopping subcluster nodes for unsandboxing...
Sending signal 'TERM' to ['192.168.111.34', '192.168.111.35', '192.168.111.36']
Successfully sent signal 'TERM' to hosts ['192.168.111.34', '192.168.111.35', '192.168.111.36'].
Details:
Host: 192.168.111.34 - Success - PID: 267860 Signal TERM
Host: 192.168.111.35 - Success - PID: 285917 Signal TERM
Host: 192.168.111.36 - Success - PID: 245272 Signal TERM

Checking for processes to be down
All processes are down.
Details:
Host 192.168.111.34 Success process 267860 is down
Host 192.168.111.35 Success process 285917 is down
Host 192.168.111.36 Success process 245272 is down



Unsandboxing Subcluster analytics...
STATUS: vertica.engine.api.db_client.module is still running on 1 host: 192.168.111.31 as of 2023-03-01 13:23:37. See /opt/vertica/log/adminTools.log for full details.
STATUS: vertica.engine.api.db_client.module is still running on 1 host: 192.168.111.31 as of 2023-03-01 13:23:47. See /opt/vertica/log/adminTools.log for full details.
STATUS: vertica.engine.api.db_client.module is still running on 1 host: 192.168.111.31 as of 2023-03-01 13:23:57. See /opt/vertica/log/adminTools.log for full details.
STATUS: vertica.engine.api.db_client.module is still running on 1 host: 192.168.111.31 as of 2023-03-01 13:24:07. See /opt/vertica/log/adminTools.log for full details.
Subcluster 'analytics' has been unsandboxed. If wiped out and restarted, it should be able to rejoin the cluster.

Removing Catalog directory contents from subcluster nodes...

Catalog cleanup complete!

Restarting unsandboxed nodes to re-join the main cluster...
    Restarting host [192.168.111.34] with catalog [v_verticadb_node0004_catalog]
    Restarting host [192.168.111.35] with catalog [v_verticadb_node0005_catalog]
    Restarting host [192.168.111.36] with catalog [v_verticadb_node0006_catalog]
    Issuing multi-node restart
    Starting nodes:
        v_verticadb_node0004 (192.168.111.34)
        v_verticadb_node0005 (192.168.111.35)
        v_verticadb_node0006 (192.168.111.36)
    Starting Vertica on all nodes. Please wait, databases with a large catalog may take a while to initialize.
    Node Status: v_verticadb_node0004: (DOWN) v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
    Node Status: v_verticadb_node0004: (DOWN) v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
    Node Status: v_verticadb_node0004: (UP) v_verticadb_node0005: (UP) v_verticadb_node0006: (UP)
Syncing catalog on verticadb with 2000 attempts.

Unsandboxed nodes have restarted successfully and joined the main cluster and are ready to use
```

When the admintools command completes, you can query the NODES system table to confirm that the previously sandboxed nodes are UP and are no longer members of sand :

```
=> SELECT node_name, subcluster_name, node_state, sandbox FROM NODES;
     node_name      |  subcluster_name   | node_state | sandbox
--------------------+--------------------+------------+---------
 v_verticadb_node0001 | default_subcluster | UP         |
 v_verticadb_node0002 | default_subcluster | UP         |
 v_verticadb_node0003 | default_subcluster | UP         |
 v_verticadb_node0004 | sc_02              | UP         |
 v_verticadb_node0005 | sc_02              | UP         |
 v_verticadb_node0006 | sc_02              | UP         |
(6 rows)
```

If there are no more active sandboxes, you can run the CLEAN_COMMUNAL_STORAGE function to remove any data created in the sandbox. You should also remove the sandbox's metadata from the shared communal storage location, which can be found at */path-to-communal-storage/`metadata`/sandbox_name* . The following example removes the sandbox's metadata from an S3 bucket and then calls CLEAN_COMMUNAL_STORAGE to cleanup any data from the sandbox:

```
$ aws s3 rm /path-to-communal/metadata/sandbox_name

SELECT CLEAN_COMMUNAL_STORAGE('true');
                    CLEAN_COMMUNAL_STORAGE
-------------------------------------------------------------------
 CLEAN COMMUNAL STORAGE
Total leaked files: 143
Files have been queued for deletion.
Check communal_cleanup_records for more information.

(1 row)
```

See also

- UNSANDBOX_SUBCLUSTER
- CLEAN_COMMUNAL_STORAGE

Local caching of storage containers

The Vertica execution engine uses the StorageMerge operator to read data from storage containers in cases where it is important to read container data in its projection-specified sort order. This is particularly useful for operations that must preserve the sort order of the data that is read from multiple storage containers, before merging it into a single storage container. Common operations that enforce sort order include mergeout , and some queries with ORDER BY clauses—for example CREATE TABLE...AS , where the query includes an ORDER BY clause.

The execution engine typically allocates multiple threads to the StorageMerge operator. Each thread is assigned a single Scan operator to open and read container contents. If the number of containers to read is greater than the number of available threads, the execution engine is likely to assign individual Scan operators to multiple containers. In this case, Scan operators might need to switch among different containers and reopen them multiple times before all required data is fetched and assembled. Doing so is especially problematic when reading storage containers on remote filesystems such as S3. The extra overhead incurred by reopening and reading remote storage containers can significantly impact performance and usage costs.

You can configure your database so the execution engine caches on local disk the data of S3 storage containers that require multiple opens. The size of temp space allocated per query to the StorageMerge operator for caching is set by configuration parameter StorageMergeMaxTempCacheMB . By default, this configuration parameter is set to -1 (unlimited). If caching requests exceed temp space limits or available disk space, Vertica caches as much container data as it can, and then reads from S3.

> **Note**
> The actual temp space that is allocated is the lesser of these settings:
>
> - StorageMergeMaxTempCacheMB
> - A user's TEMPSPACECAP setting
> - The session TEMPSPACECAP setting

To turn off caching, set StorageMergeMaxTempCacheMB to 0.

Managing an Eon Mode database in MC

Vertica Management Console (MC), a database health and activity monitoring tool, provides in-browser wizards you can follow to deploy Vertica cluster instances and create an Eon Mode database on them. You can also use MC to manage and monitor resources that are specific to Eon Mode:

- Node and shard subscriptions
- Depot storage and activity

See also

- Management Console

Stopping and starting an Eon Mode cluster

Stopping your Eon Mode database and cluster using the MC

When running an Eon Mode database in the cloud, you usually want to stop the nodes running your database when you stop the database. Stopping your nodes avoids wasting money. The nodes aren't needed while the database is down.

> **Note**
> Instead of shutting down your entire database, you can still save money by shutting down subclusters that aren't being used. This technique lets your database continue to run and load data, while reducing the hourly cost. See Starting and stopping subclusters for more information.

The easiest way to stop both your database and the nodes that run it is to use the MC:

1. From the MC home page, click **View Your Infrastructure** .
2. In row labeled Databases, click the database you want to stop.
3. In the popup, click **Stop** .
4. Click **OK** to confirm you want to stop the database.
5. Once your database has stopped, in the row labeled Clusters, click the entry for the cluster running the database you just stopped.
6. In the popup, click **Manage** .
7. In the ribbon at the top of the cluster view, click **Stop Cluster** .
8. In the dialog box, check the **I would like to stop all instances in the cluster** box and click **Stop Cluster** .

Manually stopping the database and cluster

To manually stop your database and cluster, first stop your database using one of the following methods:

- Use the stop_db admintools command to stop the database. See Stopping the database .
- Use the SHUTDOWN or SHUTDOWN_WITH_DRAIN function to stop the database.

Once you have stopped the database, you can stop your nodes. If you are in a cloud environment, see your cloud provider's documentation for instruction on stopping nodes.

Starting your cluster and database using the MC

To start your database cluster and database:

1. From the MC home, click **View Infrastructure** .
2. In the Clusters row, click the cluster that runs the database you want to start.
3. In the pop-up, click **Manage** .
4. In the ribbon at the top of the cluster's page, click **Start Cluster** .
5. Check the **I would like to start all instances in the cluster** box and click **Start Cluster** .

Starting the cluster automatically starts the database.

Manually starting your cluster and database

To manually start your cluster and database:

1. Start the nodes in your database cluster. If you are running in the cloud, see your cloud provider's documentation on how to start instances.
2. Connect to one of the nodes in the cluster and use the admintools menus or command line to start your database. See Starting the database for instructions.

Terminating an Eon Mode database cluster

When you terminate an Eon Mode database's cluster, you free its resources. In a cloud environment, terminating the cluster deletes the instances that ran the database's nodes. In an on-premises database, terminating the cluster usually means repurposing physical hardware for other uses. See Stopping, starting, terminating, and reviving Eon Mode database clusters for more information.

Terminating an Eon Mode database's cluster does not affect the data it stores. The data remains stored in the communal storage location. As long as you do not delete the communal storage location, you can revive the database onto a new Eon Mode cluster. See Reviving an Eon Mode database cluster for more information.

Important
Vertica persists catalog data in communal storage, updating it every few minutes. Nonetheless, before shutting down your database you should make sure your metadata is up to date on communal storage. To do so, see Synchronizing metadata .

Terminating an Eon Mode cluster using Management Console

Management Console provides the easiest way to terminate an Eon Mode cluster. You must follow a two-step process: first stop the database, then terminate the cluster:

1. If you have not yet synchronized the database's catalog, follow the steps in Synchronizing metadata .
2. From the Management Console home page, click **View Your Infrastructure** .
3. In the row labeled Databases, click the database whose cluster you want to terminate.
4. In the popup, click **Stop** .
5. Click **OK** to confirm you want to stop the database.
6. After the database stops, in the row labeled Clusters, click the entry for the cluster you want to terminate.
7. In the popup, click **Manage** .
8. In the ribbon at the top of the cluster view, click **Advanced** and then select **Terminate Cluster** .
9. In the dialog box:
   - Check **I understand that terminating a cluster will terminate all instances in the cluster**
   - Click **Terminate Cluster** .

Manually terminating an Eon Mode cluster

To manually terminate your Eon Mode cluster:

1. If you have not yet synchronized the database's catalog, follow the steps in Synchronizing metadata .
2. Stop the database using one of the following methods:
   - Use admintools .
   - Use the SHUTDOWN meta-function.
3. Terminate the database node instances. If you are in a cloud environment, see your cloud provider's documentation for instructions on terminating instances. For on-premises database clusters, you can repurpose the systems that were a part of the cluster.

See also

- Reviving an Eon Mode database cluster
- Stopping, starting, terminating, and reviving Eon Mode database clusters
- Eon Mode architecture
- Adding and removing nodes from subclusters
- Altering subcluster settings

Reviving an Eon Mode database cluster

If you have terminated your Eon Mode database's cluster, but have not deleted the database's communal storage, you can revive your database. Reviving the database restores it to its pre-shutdown state. The revival process requires creating a new database cluster and configuring it to use the database's communal storage location. See Stopping, starting, terminating, and reviving Eon Mode database clusters for more information.

> **Note**
> You cannot revive sandboxed subclusters. If you call the revive_db admintools command on a cluster with both sandboxed and unsandboxed subclusters, the nodes in the unsandboxed subclusters start as expected, but the nodes in the sandboxed subclusters remain down. Attempting to revive only a sandboxed subcluster returns an error.

You can also use the revive process to restart a database when its nodes do not have persistent local storage. You may choose to configure your node's instances in your cloud-based Eon Mode cluster with non-persistent local storage to reduce cost. Cloud providers such as AWS and GCP charge less for instances when they are not required to retain data when you shut them down.

You revive a database using either the Management Console or admintools. The MC and admintools offer different revival methods:

- The MC always revives onto a newly-provision cluster that it creates itself. It cannot revive onto an existing cluster. Use the MC to revive a database when you do not have a cluster already provisioned for your database.
- admintools only revives onto an existing database cluster. You can manually create a cluster to revive your database. See Set up Vertica on-premises .
  You can also revive a database whose hosts use instance storage where data is not persistently stored between shutdowns. In this case, admintools treats the existing database cluster as a new cluster, because the hosts do not contain the database's catalog data.
- Currently, only admintools lets you revive just the primary subclusters in a database cluster. This option is useful if you want to revive the minimum number of nodes necessary to start your database. See Reviving Only Primary Subclusters below.
  The MC always revives the entire database cluster.

> **Note**
> You cannot revive a database from a communal storage location that is currently running on another cluster. The revive process fails if it detects that there is a cluster already running the database. Having two instances of a database running on separate clusters using the same communal storage location leads to data corruption.

Reviving using the Management Console

You can use a wizard in the Management Console to provision a new cluster and revive a database onto it from a browser. For details, see:

- Reviving an Eon Mode database on AWS in MC
- Reviving an Eon Mode database on GCP in MC

Revive using admintools

You can use admintools to revive your Eon Mode database on an existing cluster.

Cluster requirements

This existing cluster must:

- Have the same version (or later version) of Vertica installed on it. You can repurpose an existing Vertica cluster whose database you have shut down. Another option is to create a cluster from scratch by manually installing Vertica (see Set up Vertica on-premises).
- Contain a number of hosts in the cluster that is equal to or greater than either:
  - The total number of nodes that the database cluster had when it shut down.
  - The total number of primary nodes the database cluster had when it shut down. When you supply a cluster that matches the number of primary nodes in the database, admintools revives just the primary nodes.

When reviving, you supply admintools with a list of the hosts in the cluster to revive the database onto. The number of hosts in this list must match either the total number of nodes or the number of primary nodes in the database when it shut down. If the number of nodes you supply does not match either of these values, admintools returns an error.

You do not need to use all of the hosts in the cluster to revive the database. You can revive a database onto a subset of the hosts in the cluster. But you must have at least enough hosts to revive all of the primary nodes.

For example, suppose you want to revive a database that had 16 nodes when it was shut down, with four of those nodes being primary nodes. In that case, you can revive:

- Just the primary nodes onto a cluster that contains at least four nodes.
- All of the 16 nodes onto a cluster that contains at least 16 nodes.

You may choose to revive your database onto a cluster with more nodes that is necessary in cases where you want to quickly add new nodes. You may also want to revive just the primary nodes in a database onto a larger cluster. In this case, you can use the extra nodes in the cluster to start one or more secondary subclusters.

Required database information

To revive the database, you must know:

- The name of the database to revive (note that the database name is case sensitive)
- The version of Vertica that created the database, so you can use the same or later version
- The total number of all nodes or the number of primary nodes in the database when it shut down
- The URL and credentials for the database's communal storage location
- The user name and password of the database administrator
- The IP addresses of all hosts in the cluster you want to revive onto

If you do not know what version of Vertica created the database or are unsure how many nodes it had, see Getting Database Details From a Communal Storage Location below.

Required database settings

Before starting the revive process, verify the following conditions are true for your Eon Mode environment:

Eon environment	Revived database requirements
All	• The uppermost directories of the catalog , data , and depot directories on all nodes exist and are owned by the database dbadmin • The cluster has no other database running on it
Azure	If your database does not use Azure managed identities to authenticate with the communal storage blob container, the following values must be set: • AzureStorageCredentials • AzureStorageEndpointConfig See Azure Blob Storage object store for details.
S3: AWS, on-premises	The following configuration parameters are set: • AWSEndpoint • AWSRegion (AWS only) • AWSAuth / IAM role • AWSEnableHttps Important If migrating to an on-premises database, set configuration parameter AWSEnableHttps to be compatible with the database TLS setup: AWSEnableHttps=1 if using TLS, otherwise 0. If settings are incompatible, the migration returns with an error.
GCP	The following configuration parameters are set: • GCSAuth • GCSEnableHttps (if not using the default value) • GCSEndpoint (if not using the default value)

Getting database details from a communal storage location

To revive a database, you must know:

- The version of Vertica that created it (so you can use the same or a later version)
- The total number of nodes (when reviving both primary and secondary nodes) or primary nodes (when just reviving the primary nodes) in the database's cluster when it shut down.

If you do not know these details, you can determine them based on the contents of the communal storage location.

If you are not sure which version of Vertica created the database stored in a communal storage location, examine the cluster_config.json file. This file is stored in the communal storage location in the folder named metadata/ *databasename* . For example, suppose you have a database named mydb stored in the communal storage location s3://mybucket/mydb . Then you can download and examine the file s3://mybucket/mydb/metadata/mydb/cluster_config.json .

In the cluster_config.json , the Vertica version that created the database is stored with the JSON key named DatabaseVersion near the top of the file:

```
{
  "CatalogTruncationVersion" : 804,
  "ClusterLeaseExpiration" : "2020-12-21 21:52:31.005936",
  "Database" : {
     "branch" : "",
     "name" : "verticadb"
  },
  "DatabaseVersion" : "v10.1.0",
  "GlobalSettings" : {
     "TupleMoverServices" : -33,
     "appliedUpgrades" : [
. . .
```

In this example, you can revive the storage location using Vertica version 10.1.0 or later.

If you do not know how many nodes or primary nodes the cluster had when it shut down, use the --display-only option of the admintools revive_db tool. Adding this option prevents admintools from reviving the database. Instead, it validates the files in the communal storage and reports details about the nodes that made up the database cluster. Parts of this report show the total number of nodes in the cluster and the number of primary nodes:

```
$ admintools -t revive_db --display-only --communal-storage-location \
          s3://mybucket/verticadb -d verticadb
Attempting to retrieve file: [s3://mybucket/verticadb/metadata/verticadb/cluster_config.json]

Validated 6-node database verticadb defined at communal storage s3://mybucket/verticadb.

Expected layout of database after reviving from communal storage: s3://mybucket/verticadb

== Communal location details: ==
{
 "communal_storage_url": "s3://mybucket/verticadb",
 "num_shards": "3",
 "depot_path": "/vertica/data",
   . . .
]

Number of primary nodes: 3
```

You can use grep to find just the relevant lines in the report:

```
$ admintools -t revive_db --display-only --communal-storage-location \
          s3://mybucket/verticadb -d verticadb | grep  'Validated\|primary nodes'
Validated 6-node database verticadb defined at communal storage s3://mybucket/verticadb.
Number of primary nodes: 3
```

Creating a parameter file

For Eon Mode deployments that are not on AWS, you must create a configuration file to pass the parameters listed in the table in the previous section to admintools. Traditionally this file is named auth_params.conf although you can choose any file name you want.

For on-premises Eon Mode databases, this parameter file is the same one you used when initially installing the database. See the following links for instructions on creating a parameter file for the communal storage solution you are using for your database:

- Pure Storage FlashBlade
- HDFS
- MinIO

For databases running on Microsoft Azure, the parameter file is only necessary if your database does not use managed identities. This file is the same format that you use to manually install an Eon Mode database. See Manually create an Eon Mode database on Azure for more information.

To revive an Eon Mode database on GCP manually, create a configuration file to hold the GCSAuth parameter and optionally, the GCSEnableHttp parameter.

You must supply the GCSAuth parameter to enable Vertica to read from the communal storage location stored in GCS. The value for this parameter is the HMAC access key and secret:

```
GCSAuth = HMAC_access_key:HMAC_secret_key
```

See Creating an HMAC Key for more information about HMAC keys.

If your Eon Mode database does not use encryption when accessing communal storage on GCS, then disable HTTPS access by adding the following line to auth_params.conf :

```
GCSEnableHttps = 0
```

Running the revive_db tool

Use the admintools revive_db tool to revive the database:

1. Use SSH to access a cluster host as an administrator.
2. Depending on your environment, run one of the following admintools commands:
   - AWS:

     ```
     $ admintools -t revive_db \
      --communal-storage-location=s3://communal_store_path \
      -s host1,... -d database_name
     ```

     Important

     If you revive an on-premises Eon Mode database to AWS, check the controlmode setting in /opt/vertica/config/admintools.conf . This setting must be compatible with the network messaging requirements of your Eon implementation. AWS relies on unicast messaging, which is compatible with a controlmode setting of point-to-point (pt2pt). If the source database controlmode setting was broacast and you migrate to S3/AWS communal storage, you must change controlmode with admintools:

     ```
     $ admintools -t re_ip -d dbname -T
     ```

   - On-premises and other environments:

     ```
     $ admintools -t revive_db -x auth_params.conf \
      --communal-storage-location=storage-schema://communal_store_path \
      -s host1_ip,... -d database_name
     ```

This example revives a six-node on-premises database:

```
$ admintools -t revive_db -x auth_params.conf \
  --communal-storage-location=s3://mybucket/mydir \
  -s 172.16.116.27,172.16.116.28,172.16.116.29,172.16.116.30,\
  172.16.116.31,172.16.116.32 -d VMart
```

The following example demonstrates reviving a three-node database hosted on GCP:

```
$ admintools -t revive_db -x auth_params.conf \
--communal-storage-location gs://mybucket/verticadb \
-s 10.142.0.35,10.142.0.38,10.142.0.39 -d VerticaDB

Attempting to retrieve file:
   [gs://mybucket/verticadb/metadata/VerticaDB/cluster_config.json]
Validated 3-node database VerticaDB defined at communal storage
  gs://mybucket/verticadb .
Cluster lease has expired.
Preparation succeeded all hosts
Calculated necessary addresses for all nodes.
Starting to bootstrap nodes. Please wait, databases with a large
  catalog may take a while to initialize.
>>Calling bootstrap on node v_verticadb_node0002 (10.142.0.38)
>>Calling bootstrap on node v_verticadb_node0003 (10.142.0.39)
Load Remote Catalog succeeded on all hosts
Database revived successfully.
```

Reviving only primary subclusters

You can revive just the primary subclusters in an Eon Mode database. Make the list of hosts you pass to the admintools revive_db tool's --hosts (or -s) argument match the number of primary nodes that were in the database when it shut down. For example, if you have a six-node Eon Mode database that had three primary nodes, you can revive just the primary nodes by supplying three hosts in the --hosts argument:

```
$ admintools -t revive_db --communal-storage-location=s3://verticadb -d verticadb \
             -x auth_params.conf --hosts node01,node02,node03
Attempting to retrieve file: [s3://verticadb/metadata/verticadb/cluster_config.json]
Consider reviving to only primary nodes: communal storage indicates 6 nodes, while
  3 nodes were specified

Validated 3-node database verticadb defined at communal storage s3://verticadb.
Cluster lease has expired.
Preparation succeeded all hosts

Calculated necessary addresses for all nodes.
Starting to bootstrap nodes. Please wait, databases with a large catalog may take a
  while to initialize.
>>Calling bootstrap on node v_verticadb_node0002 (192.168.56.103)
>>Calling bootstrap on node v_verticadb_node0003 (192.168.56.104)
Load Remote Catalog succeeded on all hosts

Database revived successfully.
```

In a database where you have revived only the primary nodes, the secondary nodes are down. Their IP address is set to 0.0.0.0 so they are not part of the database. For example, querying the NODES system table in the database revived in the previous example shows the secondary nodes are all down:

```
=> SELECT node_name,node_state,node_address,subcluster_name FROM NODES;
     node_name       | node_state |  node_address  |  subcluster_name
---------------------+------------+----------------+--------------------
 v_verticadb_node0001 | UP         | 192.168.56.102 | default_subcluster
 v_verticadb_node0002 | UP         | 192.168.56.103 | default_subcluster
 v_verticadb_node0003 | UP         | 192.168.56.104 | default_subcluster
 v_verticadb_node0004 | DOWN       | 0.0.0.0        | analytics
 v_verticadb_node0005 | DOWN       | 0.0.0.0        | analytics
 v_verticadb_node0006 | DOWN       | 0.0.0.0        | analytics
```

Note

Secondary nodes that have not been revived may cause error messages if your database has the large cluster feature enabled. (See Large cluster for more information about the large cluster feature.)

For example, adding a node to a secondary subcluster can fail if the new node would be assigned a control node that has not been revived. In this case, Vertica reports that adding the node failed because the control node has an invalid IP address.

If you encounter errors involving control nodes with invalid IP addresses, consider reviving the unrevived secondary subcluster, as explained below.

Because Vertica considers these unrevived nodes to be down, it may not allow you to remove them or remove their subcluster while they are in their unrevived state. The best way to remove the nodes or the secondary subcluster is to revive them first.

Reviving unrevived secondary subclusters

If you revived just the primary subclusters in your database, you can later choose to revive some or all of the secondary subclusters. Your cluster must have hosts that are not nodes in the database that Vertica can use to revive the unrevived nodes. If your cluster does not have enough of these non-node hosts, you can add more hosts. See Adding hosts to a cluster .

You revive a secondary subcluster by using the admintools' restart_subcluster tool. You supply it with the list of hosts in the --hosts argument where the nodes will be revived. The number of hosts in this list must match the number of nodes in the subcluster. You must revive all nodes in the subcluster at the same time. If you pass restart_subcluster a list with fewer or more hosts than the number of nodes defined in the subcluster, it returns an error.

The follow example demonstrates reviving the secondary subcluster named analytics shown in the previous examples.

```
$ admintools -t restart_subcluster -d verticadb --hosts node04,node05,node06 \
          -p 'password' -c analytics
Updating hostnames of nodes in subcluster analytics.
   Replicating configuration to all nodes
   Generating new configuration information and reloading spread
Hostnames of nodes in subcluster analytics updated successfully.
*** Restarting subcluster for database verticadb ***
   Restarting host [192.168.56.105] with catalog [v_verticadb_node0004_catalog]
   Restarting host [192.168.56.106] with catalog [v_verticadb_node0005_catalog]
   Restarting host [192.168.56.107] with catalog [v_verticadb_node0006_catalog]
   Issuing multi-node restart
   Starting nodes:
      v_verticadb_node0004 (192.168.56.105)
      v_verticadb_node0005 (192.168.56.106)
      v_verticadb_node0006 (192.168.56.107)
   Starting Vertica on all nodes. Please wait, databases with a large catalog may take a while to initialize.
   Node Status: v_verticadb_node0004: (DOWN) v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
   Node Status: v_verticadb_node0004: (DOWN) v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
   Node Status: v_verticadb_node0004: (DOWN) v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
   Node Status: v_verticadb_node0004: (INITIALIZING) v_verticadb_node0005: (INITIALIZING) v_verticadb_node0006: (INITIALIZING)
   Node Status: v_verticadb_node0004: (UP) v_verticadb_node0005: (UP) v_verticadb_node0006: (UP)
Syncing catalog on verticadb with 2000 attempts.
```

See also

- Stopping, starting, terminating, and reviving Eon Mode database clusters
- Terminating an Eon Mode database cluster
- Eon Mode architecture
- Adding and removing nodes from subclusters
- Altering subcluster settings

Synchronizing metadata

An Eon Mode database maintains its catalog, which contains all database metadata, in communal storage. Vertica uses this metadata when it revives the database, so it is important that the catalog is always up to date. Vertica automatically synchronizes the catalog at regular intervals as specified by the configuration parameter CatalogSyncInterval —by default, set to five minutes.

In general, it is not necessary to monitor the synchronization process or change it. One exception applies: before shutting down a database that you intend to revive or replicate, it is good practice to verify that the catalog contains all recent changes, and if necessary synchronize it manually.

Verifying catalog status

You can verify the synchronization status of the database catalog in two ways, depending on whether the database is running.

If the database is running, query and compare these two system tables:

- CATALOG_SYNC_STATE : Shows how recently each node synchronized its catalog to communal storage, and the version that it synchronized.
- CATALOG_TRUNCATION_STATUS : Shows the latest synchronization status of the database catalog. Catalog synchronization is up to date when columns TRUNCATION_CATALOG_VERSION and CURRENT_CATALOG_VERSION are the same.

If the database is not currently running, check the following JSON file on communal storage:

/metadata/ *database-name* /cluster_config.json

The catalog truncation version and timestamp in this file indicate when Vertica last synchronized the database catalog.

Manually synchronizing the database datalog

If necessary, call SYNC_CATALOG to synchronize the catalog immediately with all nodes or a specific node:

```
=> SELECT sync_catalog();
```

Customizing synchronization intervals

By default, Vertica checks for catalog changes every five minutes. Occasionally, you might want to change this setting temporarily—for example, set it to a high value in order to take a snapshot of the current bucket contents:

```
=> ALTER DATABASE DEFAULT SET CatalogSyncInterval = 300;
```

Important
If you set CatalogSyncInterval to high value for a specific task, be sure to revert it to its default setting immediately after the task is complete:

```
=> ALTER DATABASE DEFAULT CLEAR PARAMETER CatalogSyncInterval;
```

Containerized Vertica

Vertica Eon Mode leverages container technology to meet the needs of modern application development and operations workflows that must deliver software quickly and efficiently across a variety of infrastructures. Containerized Vertica supports Kubernetes with automation tools to help maintain the state of your environment with minimal disruptions and manual intervention.

Containerized Vertica provides the following benefits:

- **Performance** : Eon Mode separates compute from storage , which provides the optimal architecture for stateful, containerized applications. Eon Mode subclusters can target specific workloads and scale elastically according to the current computational needs.
- **High availability** : Vertica containers provide a consistent, repeatable environment that you can deploy quickly. If a database host or service fails, you can easily replace the resource.
- **Resource utilization** : A container is a runtime environment that packages an application and its dependencies in an isolated process. This isolation allows containerized applications to share hardware without interference, providing granular resource control and cost savings.
- **Flexibility** : Kubernetes is the de facto container orchestration platform. It is supported by a large ecosystem of public and private cloud providers.

Containerized Vertica ecosystem

Vertica provides various tools and artifacts for production and development environments. The containerized Vertica ecosystem includes the following:

- **Vertica Helm chart** : Helm is a Kubernetes package manager that bundles into a single package the YAML manifests that deploy Kubernetes objects. Download Vertica Helm charts from the Vertica Helm Charts Repository .
- **Custom Resource Definition** (CRD): A CRD is a shared global object that extends the Kubernetes API with your custom resource types. You can use a CRD to instantiate a custom resource (CR), a deployable object with a desired state. Vertica provides CRDs that deploy and support the Eon Mode architecture on Kubernetes.
- **VerticaDB Operator** : The operator is a custom controller that monitors the state of your CR and automates administrator tasks. If the current state differs from the declared state, the operator works to correct the current state.
- **Admission controller** : The admission controller uses a webhook that the operator queries to verify changes to mutable states in a CR.
- **VerticaDB vlogger** : The vlogger is a lightweight image used to deploy a sidecar utility container. The sidecar sends logs from vertica.log in the Vertica server container to standard output on the host node to simplify log aggregation.
- **Vertica Community Edition (CE) image** : The CE image is the containerized version of the limited Enterprise Mode Vertica community edition (CE) license. The CE image provides a test environment consisting of an example database and developer tools.
  In addition to the pre-built CE image, you can build a custom CE image with the tools provided in the Vertica one-node-ce GitHub repository .
- **Communal Storage Options** : Vertica supports a variety of public and private cloud storage providers. For a list of supported storage providers, see Containerized environments .
- **UDx development tools** : The UDx-container GitHub repository provides the tools to build a container that packages the binaries, libraries, and compilers required to create C++ Vertica user-defined extensions. For additional details about extending Vertica in C++, see C++ SDK .

In this section

- Containerized Vertica on Kubernetes
- Vertica images
- VerticaDB operator
- Configuring communal storage
- Custom resource definitions
- Custom resource definition parameters
- Subclusters on Kubernetes
- Upgrading Vertica on Kubernetes
- Hybrid Kubernetes clusters
- Generating a custom resource from an existing Eon Mode database
- Backup and restore containerized Vertica

- Troubleshooting your Kubernetes cluster

Containerized Vertica on Kubernetes

Kubernetes is an open-source container orchestration platform that automatically manages infrastructure resources and schedules tasks for containerized applications at scale. Kubernetes achieves automation with a declarative model that decouples the application from the infrastructure. The administrator provides Kubernetes the desired state of an application, and Kubernetes deploys the application and works to maintain its desired state. This frees the administrator to update the application as business needs evolve, without worrying about the implementation details.

An application consists of resources, which are stateful objects that you create from Kubernetes resource types. Kubernetes provides access to resource types through the Kubernetes API, an HTTP API that exposes resource types as endpoints. The most common way to create a resource is with a YAML-formatted manifest file that defines the desired state of the resource. You use the kubectl command line tool to request a resource instance of that type from the Kubernetes API. In addition to the default resource types, you can extend the Kubernetes API and define your own resource types as a Custom Resource Definition (CRD).

To manage the infrastructure, Kubernetes uses a host to run the control plane, and designates one or more hosts as worker nodes. The control plane is a collection of services and controllers that maintain the desired state of Kubernetes objects and schedule tasks on worker nodes. Worker nodes complete tasks that the control plane assigns. Just as you can create a CRD to extend the Kubernetes API, you can create a custom controller that maintains the state of your custom resources (CR) created from the CRD.

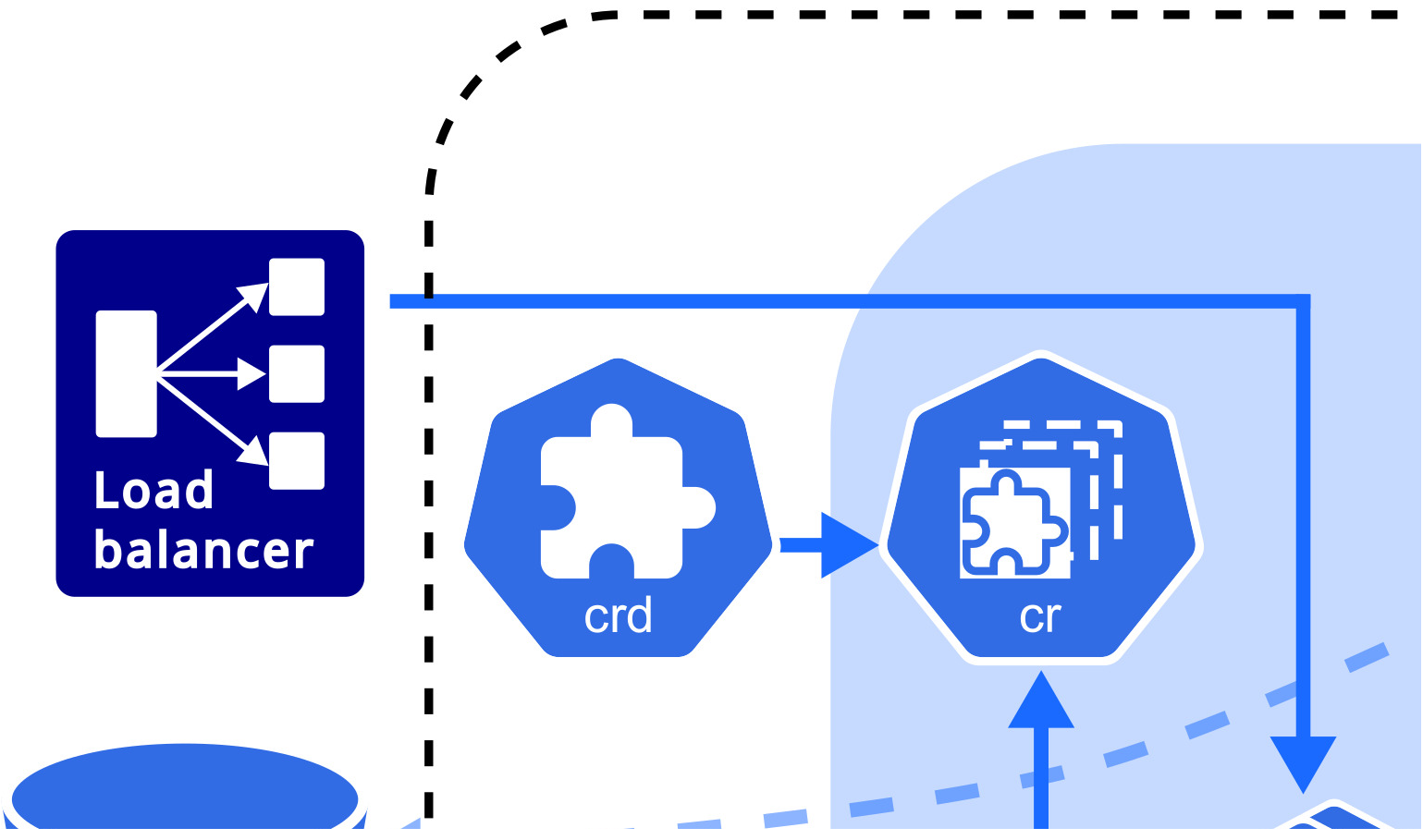
Vertica custom resource definition and custom controller

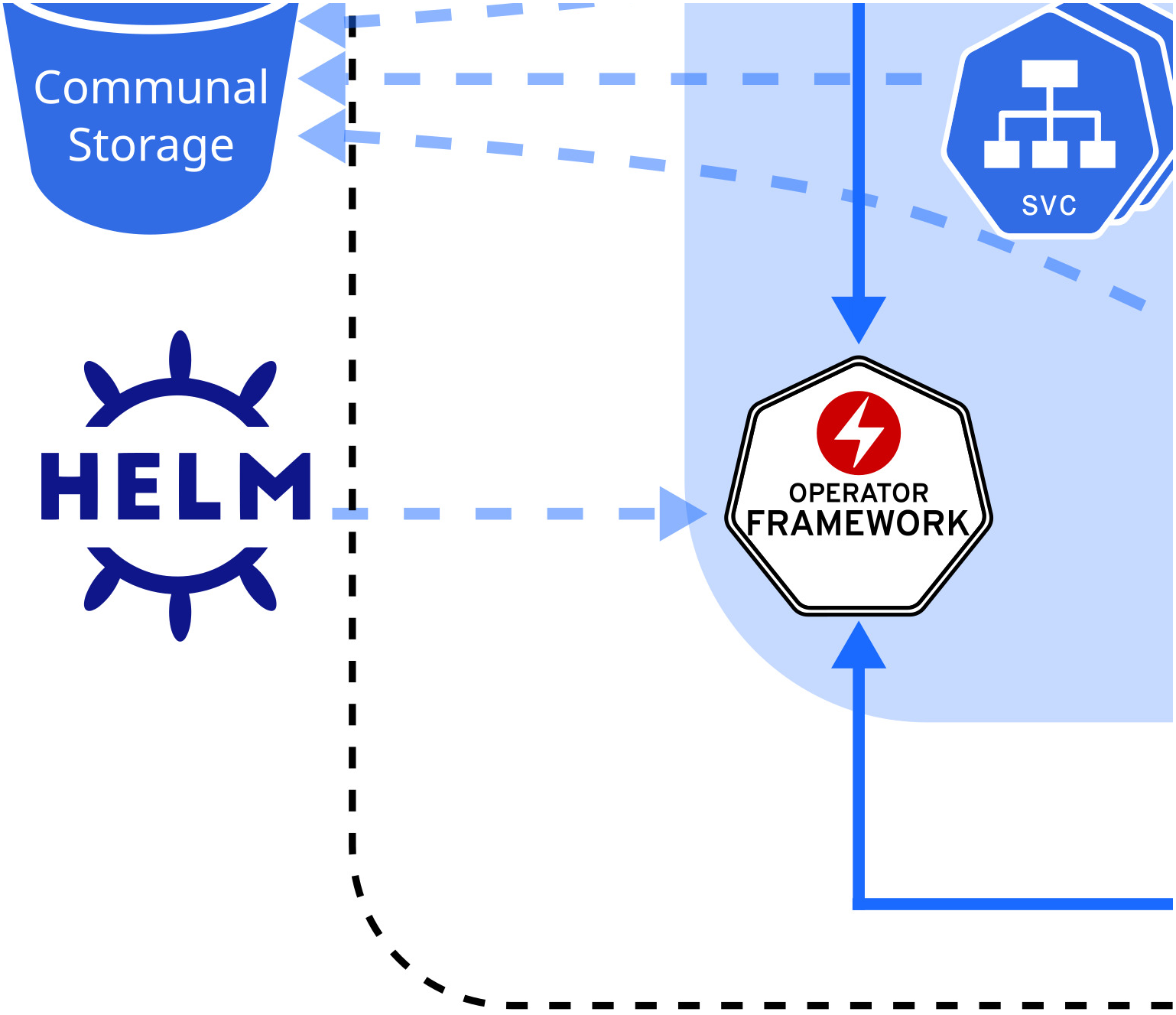
The VerticaDB CRD extends the Kubernetes API so that you can create custom resources that deploy an Eon Mode database as a StatefulSet. In addition, Vertica provides the VerticaDB operator, a custom controller that maintains the desired state of your CR and automates lifecycle tasks. The result is a self-healing, highly-available, and scalable Eon Mode database that requires minimal manual intervention.

To simplify deployment, Vertica packages the CRD and the operator in Helm charts. A Helm chart bundles manifest files into a single package to create multiple resource type objects with a single command.

Custom resource definition architecture

The Vertica CRD creates a StatefulSet, a workload resource type that persists data with ephemeral Kubernetes objects. The following diagram describes the Vertica CRD architecture:





VerticaDB operator

The operator is a namespace-scoped custom controller that maintains the state of custom objects and automates administrator tasks. The operator watches objects and compares their current state to the desired state declared in the custom resource . When the current state does not match the desired state, the operator works to restore the objects to the desired state.

In addition to state maintenance, the operator:

- Installs Vertica
- Creates an Eon Mode database
- Upgrades Vertica
- Revives an existing Eon Mode database
- Restarts and reschedules DOWN pods
- Scales subclusters
- Manages services for pods
- Monitors pod health
- Handles load balancing for internal and external traffic

To validate changes to the custom resource, the operator queries the admission controller, a webhook that provides rules for mutable states in a custom resource.

Vertica makes the operator and admission controller available through OperatorHub.io or as a Helm chart. For details about installing the operator and the admission controller with both methods, see Installing the Vertica DB operator.

Vertica pod

A pod is essentially a wrapper around one or more logically-grouped containers. These containers consume the host node resources in a shared execution environment. In addition to sharing resources, a pod extends the container to interact with Kubernetes services. For example, you can assign labels to associate pods to other objects, and you can implement affinity rules to schedule pods on specific host nodes.

DNS names provide continuity between pod lifecycles. Each pod is assigned an ordered and stable DNS name that is unique within its cluster. When a Vertica pod fails, the rescheduled pod uses the same DNS name as its predecessor. If a pod needs to persist data between lifecycles, you can mount a custom volume in its filesystem.

Rescheduled pods require information about the environment to become part of the cluster. This information is provided by the Downward API. Environment information, such as the superuser password Secret, is mounted in the /etc/podinfo directory.

Sidecar container

Pods run multiple containers to tightly couple containers that contribute to the same process. The Vertica pod allows a sidecar, a utility container that can access and perform utility tasks for the Vertica server process.

For example, logging is a common utility task. Idiomatic Kubernetes practices retrieve logs from standard output and standard error on the host node for log aggregation. To facilitate this practice, Vertica offers the vlogger sidecar image that sends the contents of vertica.log to standard output on the host node.

If a sidecar needs to persist data, you can mount a custom volume in the sidecar filesystem.

For implementation details, see VerticaDB CRD.

Persistent storage

A pod is an ephemeral, immutable object that requires access to external storage to persist data between lifecycles. To persist data, the operator uses the following API resource types:

- **StorageClass** : Represents an external storage provider. You must create a StorageClass object separately from your custom resource and set this value with the local.storageClassName configuration parameter.
- **PersistentVolume (PV)** : A unit of storage that mounts in a pod to persist data. You dynamically or statically provision PVs. Each PV references a StorageClass.
- **PersistentVolumeClaim (PVC)** : The resource type that a pod uses to describe its StorageClass and storage requirements.

A pod mounts a PV in its filesystem to persist data, but a PV is not associated with a pod by default. However, the pod is associated with a PVC that includes a StorageClass in its storage requirements. When a pod requests storage with a PVC, the operator observes this request and then searches for a PV that meets the storage requirements. If the operator locates a PV, it binds the PVC to the PV and mounts the PV as a volume in the pod. If the operator does not locate a PV, it must either dynamically provision one, or the administrator must manually provision one before the operator can bind it to a pod.

PVs persist data because they exist independently of the pod life cycle. When a pod fails or is rescheduled, it has no effect on the PV. When you delete a VerticaDB, the VerticaDB operator automatically deletes any PVCs associated with that VerticaDB instance.

For additional details about StorageClass, PersistentVolume, and PersistentVolumeClaim, see the Kubernetes documentation.

StorageClass requirements

The StorageClass affects how the Vertica server environment and operator function. For optimum performance, consider the following:

- If you do not set the local.storageClassName configuration parameter, the operator uses the default storage class. If you use the default storage class, confirm that it meets storage requirements for a production workload.
- Select a StorageClass that uses a recommended storage format type as its fsType.
- Use dynamic volume provisioning. The operator requires on-demand volume provisioning to create PVs as needed.

Local volume mounts

The operator mounts a single PVC in the /home/dbadmin/local-data/ directory of each pod to persist data. Each of the following subdirectories is a sub-path into the volume that backs the PVC:

- /catalog : Optional subdirectory that you can create if your environment requires a catalog location that is separate from the local data. You can customize this path with the local.catalogPath parameter.
  By default, the catalog is stored in the /data subdirectory.
- /data : Stores any temporary files, and the catalog if local.catalogPath is not set. You can customize this path with the local.dataPath parameter.

- /depot : Improves depot warming in a rescheduled pod. You can customize this path with the local.depotPath parameter .

> **Note**
> You can change the volume type for the /depot with the local.depotVolume parameter . By default, this parameter is set to PersistentVolume , and the operator creates the /depot sub-path. If local.depotVolume is not set to PersistentVolume , the operator does not create the sub-path.

- /opt/vertica/config : Persists the contents of the configuration directory between restarts.
- /opt/vertica/log : Persists log files between pod restarts.

> **Note**
> Kubernetes assigns each custom resource a unique identifier. The volume mount paths include the unique identifier between the mount point and the subdirectory. For example, the full path to the /data directory is /home/dbadmin/local-data/ *uid* /data .

By default, each path mounted in the /local-data directory are owned by the dbadmin user and the verticadb group. For details, see Linux users created by Vertica .

Custom volume mounts

You might need to persist data between pod lifecycles in one of the following scenarios:

- An external process performs a task that requires long-term access to the Vertica server data.
- Your custom resource includes a sidecar container in the Vertica pod.

You can mount a custom volume in the Vertica pod or sidecar filesystem. To mount a custom volume in the Vertica pod, add the definition in the spec section of the CR. To mount the custom volume in the sidecar, add it in an element of the sidecars array.

The CR requires that you provide the volume type and a name for each custom volume. The CR accepts any Kubernetes volume type. The volumeMounts.name value identifies the volume within the CR, and has the following requirements and restrictions:

- It must match the volumes.name parameter setting.
- It must be unique among all volumes in the /local-data , /podinfo , or /licensing mounted directories.

For instructions on how to mount a custom volume in either the Vertica server container or in a sidecar, see VerticaDB CRD .

Service objects

Vertica on Kubernetes provides two service objects: a headless service that requires no configuration to maintain DNS records and ordered names for each pod, and a load balancing service that manages internal traffic and external client requests for the pods in your cluster.

Load balancing services

Each subcluster uses a single load balancing service object. You can manually assign a name to a load balancing service object with the subclusters[i].serviceName parameter in the custom resource. Assigning a name is useful when you want to:

- Direct traffic from a single client to multiple subclusters.
- Scale subclusters by workload with more flexibility.
- Identify subclusters by a custom service object name.

To configure the type of service object, use the subclusters[i].serviceType parameter in the custom resource to define a Kubernetes service type . Vertica supports the following service types:

- **ClusterIP** : The default service type. This service provides internal load balancing, and sets a stable IP and port that is accessible from within the subcluster only.
- **NodePort** : Provides external client access. You can specify a port number for each host node in the subcluster to open for client connections.
- **LoadBalancer** : Uses a cloud provider load balancer to create NodePort and ClusterIP services as needed. For details about implementation, see the Kubernetes documentation and your cloud provider documentation.

Important

To prevent performance issues during heavy network traffic, Vertica recommends that you set --proxy-mode to iptables for your Kubernetes cluster.

Because native Vertica load balancing interferes with the Kubernetes service object, Vertica recommends that you allow the Kubernetes services to manage load balancing for the subcluster. You can configure the native Vertica load balancer within the Kubernetes cluster, but you receive unexpected results. For example, if you set the Vertica load balancing policy to ROUNDROBIN, the load balancing appears random.

For additional details about Kubernetes services, see the official Kubernetes documentation .

Security considerations

Vertica on Kubernetes supports both TLS and mTLS for communications between resource objects. You must manually configure TLS in your environment. For details, see TLS protocol .

The VerticaDB operator manages changes to the certificates. If you update an existing certificate, the operator replaces the certificate in the Vertica server container. If you add or delete a certificate, the operator reschedules the pod with the new configuration.

The subsequent sections detail internal and external connections that require TLS for secure communications.

Admission controller webhook certificates

The VerticaDB operator Helm chart includes the admission controller, a webhook that communicates with the Kubernetes API server to validate changes to a resource object. Because the API server communicates over HTTPS only, you must configure TLS certificates to authenticate communications between the API server and the webhook.

The method you use to install the VerticaDB operator determines how you manage TLS certificates for the admission controller:

- OperatorHub.io: Runs on the Operator Lifecycle Manager (OLM) and automatically creates and mounts a self-signed certificate for the webhook. This installation method does not require additional action.
- Helm charts: Manually manage admission TLS certificates with the webhook.certSource Helm chart parameter .

For details about each installation method, see Installing the Vertica DB operator .

Communal storage certificates

Supported storage locations authenticate requests with a self-signed certificate authority (CA) bundle. For TLS configuration details for each provider, see Configuring communal storage .

Client-server certificates

You might require multiple certificates to authenticate external client connections to the load balancing service object . You can mount one or more custom certificates in the Vertica server container with the certSecrets custom resource parameter . Each certificate is mounted in the container at /certs/ *cert-name* / *key* .

For details, see VerticaDB CRD .

Prometheus metrics certificates

Vertica integrates with Prometheus to scrape metrics about the VerticaDB operator. The operator SDK framework enforces role-based access control (RBAC) to the metrics with a proxy sidecar that uses self-signed certificates to authenticate requests for authorized service accounts. If you run Prometheus outside of Kubernetes, you cannot authenticate with a service account, so you must provide the proxy sidecar with custom TLS certificates.

For details, see Prometheus integration .

System configuration

As a best practice, make system configurations on the host node so that pods inherit those settings from the host node. This strategy eliminates the need to provide each pod a privileged security context to make system configurations on the host.

To manually configure host nodes, refer to the following sections:

- Automatically configured operating system settings
- Manually configured operating system settings

The dbadmin account must use one of the authentication techniques described in Dbadmin authentication access .

Vertica images

The following table describes Vertica server and automation tool images:

Image	Description

Vertica Kubernetes minimal image	Optimized for Kubernetes. This image includes the following UDx development capabilities: • C++ • Python **Image names** : • vertica/vertica-k8s:12.0.4-0-minimal • vertica/vertica-k8s:latest
Vertica Kubernetes (Full image)	Optimized for Kubernetes and includes the following packages for machine learning and UDx development : • TensorFlow • Java 8 • C++ • Python **Image name** : vertica/vertica-k8s:12.0.4-0
Vertica Kubernetes (No keys)	Optimized for Kubernetes and includes the following packages for machine learning and UDx development : • TensorFlow • Java 8 • C++ • Python Important This image does not include static SSH keys for internal communication between the pods. You must provide these keys to the custom resource. For details, see VerticaDB CRD . **Image name** : vertica/vertica-k8s:12.0.4-0-nokeys
Vertica Community Edition	A single-node Enterprise Mode image for test environments. For more information, see Vertica community edition (CE) . This image includes the following UDx development capabilities: • C++ • Python **Image name** : vertica/vertica-ce:12.0.4-0
VerticaDB Operator	The operator monitors the state of your custom resources and automates lifecycle tasks for Vertica on Kubernetes. For installation instructions, see Installing the Vertica DB operator . **Image name** : vertica/verticadb-operator:1.11.1
Vertica vlogger	Lightweight image for sidecar logging. The vlogger sends the contents of vertica.log to stdout on the host node. For implementation details, see VerticaDB CRD . **Image name** : vertica/vertica-logger:1.0.0

Important
The Apache Kafka and Apache Spark integrations require an image that includes Java 8.

Creating a custom Vertica image

The Creating a Vertica Image tutorial in the Vertica Integrator's Guide provides a line-by-line description of the Dockerfile hosted on GitHub . You can add dependencies to replicate your development and production environments.

Python container UDx

The Vertica images with Python UDx development capabilities include the vertica_sdk package and the Python Standard Library.

If your UDx depends on a Python package that is not included in the image, you must make the package available to the Vertica process during runtime. You can either mount a volume that contains the package dependencies, or you can create a custom Vertica server image.

Important
When you load the UDx library with CREATE LIBRARY, the DEPENDS clause must specify the location of the Python package in the Vertica server container filesystem.

Use the Python Package Index to download Python package source distributions.

Mounting Python libraries as volumes

You can mount a Python package dependency as a volume in the Vertica server container filesystem. A Python UDx can access the contents of the volume at runtime.

1. Download the package source distribution to the host machine.
2. On the host machine, extract the tar file contents into a mountable volume:
   ```
   $ tar -xvf lib-name.version.tar.gz -C /path/to/py-dependency-vol
   ```
3. Mount the volume that contains the extracted source distribution in the custom resource (CR). The following snippet mounts the py-dependency-vol volume in the Vertica server container:
   ```
   spec:
     ...
     volumeMounts:
     - name: nfs
       mountPath: /path/to/py-dependency-vol
     volumes:
     - name: nfs
       nfs:
         path: /nfs
         server: nfs.example.com
     ...
   ```
   For details about mounting custom volumes in a CR, see VerticaDB CRD.

Adding a Python library to a custom Vertica image

Create a custom image that includes any Python package dependencies in the Vertica server base image.

For a comprehensive guide about creating a custom Vertica image, see the Creating a Vertica Image tutorial in the Vertica Integrator's Guide.

1. Download the package source distribution on the machine that builds the container.
2. Create a Dockerfile that includes the Python source distribution:
   ```
   FROM vertica/vertica-k8s:latest
   ADD lib-name.version.tar.gz /path/to/target-dir
   ...
   ```
   In the preceding example, the ADD command automatically extracts the contents of the tar file into the target-dir directory.
3. Build the Dockerfile:
   ```
   $ docker build . -t image-name:tag
   ```
4. Push the image to a container registry so that you can add the image to a Vertica custom resource:
   ```
   $ docker image push registry-host:port/registry-username/image-name:tag
   ```

VerticaDB operator

The Vertica operator automates error-prone and time-consuming tasks that a Vertica on Kubernetes administrator must otherwise perform manually. The operator:

- Installs Vertica
- Creates an Eon Mode database
- Upgrades Vertica
- Revives an existing Eon Mode database
- Restarts and reschedules DOWN pods

- Scales subclusters
- Manages services for pods
- Monitors pod health
- Handles load balancing for internal and external traffic

The Vertica operator is a Go binary that uses the SDK operator framework . It runs in its own pod, and is namespace-scoped to limit any failures to the objects in its namespace.

For details about installing and upgrading the operator, see Installing the Vertica DB operator .

Monitoring desired state

Each namespace is allowed one operator pod that acts as a custom controller and monitors the state of the custom resource objects within that namespace. The operator uses the control loop mechanism to reconcile state changes by investigating state change notifications from the custom resource instance, and periodically comparing the current state with the desired state.

If the operator detects a change in the desired state, it determines what change occurred and reconciles the current state with the new desired state. For example, if the user deletes a subcluster from the custom resource instance and successfully saves the changes, the operator deletes the corresponding subcluster objects in Kubernetes.

Validating state changes

The verticadb-operator Helm chart includes an admission controller, which uses a webhook to prevent invalid state changes to the custom resource. When you save a change to a custom resource, the admission controller webhook queries a REST endpoint that provides rules for mutable states in a custom resource. If a change violates the state rules, the admission controller prevents the change and returns an error. For example, it returns an error if you try to save a change that violates K-Safety .

Limitations

The operator has the following limitations:

- You must manually configure TLS. For details, see Containerized Vertica on Kubernetes .
- Vertica recommends that you do not use the Large cluster feature. If a control nodes fails, it might cause more than half of the database nodes to fail. This results in the database losing quorum.
- Backup and Restore is a manual process.
- Importing and exporting data between a cluster outside of Kubernetes requires that you expose the service with the NodePort or LoadBalancer service type and properly configure the network.
  Important
  When configuring the network to import or export data, you must assign each node a static IP export address. When pods are rescheduled to different nodes, you must update the static IP address to reflect the new node.

  See Configuring the Network to Import and Export Data for more information.

In this section

- Installing the Vertica DB operator
- Upgrading the VerticaDB operator
- Helm chart parameters
- Red Hat OpenShift integration
- Prometheus integration

Installing the Vertica DB operator

The custom resource definition (CRD), VerticaDB operator , and admission controller work together to maintain the state of your environment and automate tasks:

- The CRD extends the Kubernetes API to provide custom objects. It serves as a blueprint for custom resource (CR) instances that specify the desired state of your environment.
- The VerticaDB operator is a custom controller that monitors CR instances to maintain the desired state of VerticaDB objects. You can deploy one VerticaDB operator per namespace, and the operator monitors only the VerticaDB objects within that namespace.
- The admission controller is a webhook that queries a REST endpoint to verify changes to mutable states in a CR instance.

Prerequisites

- Kubernetes 1.21.1 and higher
- Helm 3.5.0 and higher
- kubectl command line tool

Installation options

Vertica provides two separate options to install the VerticaDB operator and admission controller:

- Helm charts . Helm chart installations include operator logging levels and log rotation policy. For details, see Helm chart parameters .
- OperatorHub.io

> **Note**
> Each install option has its own workflow that is incompatible with the other option. For example, you cannot install the VerticaDB operator with the Helm charts, and then deploy an operator in the same environment using OperatorHub.io.

Helm charts

Vertica packages VerticaDB operator and admission controller in a Helm chart . Vertica on Kubernetes allows one operator instance per namespace.

Important

Vertica recommends that you use Kubernetes 1.21.1 or later. Earlier versions require that you add the kubernetes.io/metadata.name= *namespace-name* label to each namespace that contains an operator.

Configuring TLS for the admission controller

Before you can install the VerticaDB Helm chart, you must configure TLS for the admission controller. The admission controller uses a webhook that requires TLS certificates for data encryption. Use the webhook.certSource Helm chart parameter to manage the TLS certificates.

By default, webhook.certSource is set to internal , a setting that generates a self-signed certificate before starting the admission controller. There are two additional settings that require manual configuration before you install the operator:

- secret : You generate custom certificates before you create the Helm chart and store them in a Secret . This option requires that you set the webhook.tlsSecret Helm chart parameter .
- cert-manager : Deprecated. You install the cert-manager operator, and it generates self-signed certificates. When the certificate nears expiration, cert-manager automatically handles private key rotation.

Defining custom certificates

Custom certificates require a TLS key that sets the Subjective Alternative Name (SAN) using the admission controller webhook's fully-qualified domain name (FDQN). You can set the SAN in a configuration file with the following format:

```
[alt_names]
DNS.1 = verticadb-operator-webhook-service.namespace.svc
DNS.2 = verticadb-operator-webhook-service.namespace.svc.cluster.local
```

For more information about TLS and Vertica, see TLS protocol .

When you install the VerticaDB operator and admission controller Helm chart, you can pass parameters to customize the Helm chart. Conceal custom certificates in a Secret before you pass them as parameters. The following command creates a Secret that stores the TLS key, TLS certificate, and CA certificate:

```
$ kubectl create secret generic tls-secret --from-file=tls.key=/path/to/tls.key --from-file=tls.crt=/path/to/tls.crt --from-file=ca.crt=/path/to/ca.crt
```

Use tls-secret when you install the VerticaDB operator and admission controller Helm chart. For a detailed example, see Helm chart parameters .

Installing cert-manager

> **Deprecated**
> This TLS certificate management method is deprecated and will be removed in a future release.

cert-manager is available as a YAML manifest in a GitHub repository:

1. Use kubectl to install cert-manager:
   ```
   $ kubectl apply -f https://github.com/jetstack/cert-manager/releases/download/v1.5.3/cert-manager.yaml
   ```
   Installation might take a few minutes.
2. Verify the cert-manager installation:

```
$ kubectl get pods --namespace cert-manager
NAME                                       READY   STATUS    RESTARTS   AGE
cert-manager-7dd5854bb4-skks7              1/1     Running   5          12d
cert-manager-cainjector-64c949654c-9nm2z   1/1     Running   5          12d
cert-manager-webhook-6bdffc7c9d-b7r2p      1/1     Running   5          12d
```

3. When you install the Helm chart, set the webhook.certSource Helm chart parameter to cert-manager :

```
$ helm install operator-name --namespace namespace --create-namespace vertica-charts/verticadb-operator \
  --set webhook.certSource=cert-manager
```

For additional details about cert-manager install verification, see the cert-manager documentation .

Granting operator privileges

Optionally, you can authorize a user without cluster administrator privileges to install the operator in a specific namespace. You can grant these operator privileges with a preconfigured Kubernetes service account .

Vertica leverages Kubernetes RBAC to authorize a service account with operator privileges to perform operator actions. You can grant these privileges to a Role resource type, then define a RoleBinding resource type that associates that Role with a ServiceAccount.

After the cluster administrator binds that ServiceAccount to a namespace, any user can perform operator actions if they install the Helm chart with the ServiceAccount.

Cluster administrator set up

The cluster administrator creates a namespace and then binds to it a service account with the required operator privileges:

1. Install the CRDs from the vertica-kubernetes GitHub repository:

```
$ kubectl apply -f https://github.com/vertica/vertica-kubernetes/releases/latest/download/verticadbs.vertica.com-crd.yaml
$ kubectl apply -f https://github.com/vertica/vertica-kubernetes/releases/latest/download/verticaautoscalers.vertica.com-crd.yaml
```

2. Create a namespace:

```
$ kubectl create namespace namespace
```

3. Apply the ServiceAccount, Roles, and RoleBindings required to grant operator privileges to a service account.
   The following command applies operator-rbac.yaml , a sample file that defines the required operator privileges:

```
$ kubectl -n namespace apply -f https://github.com/vertica/vertica-kubernetes/releases/latest/download/operator-rbac.yaml
```

4. Verify the changes with kubectl get :
   - ServiceAccount:

     ```
     $ kubectl get serviceaccounts -n namespace
     NAME                                 SECRETS   AGE
     default                              1         71m
     verticadb-operator-controller-manager   1         69m
     ```

   - Roles in the correct namespace:

     ```
     $ kubectl get roles -n namespace
     NAME                                    CREATED AT
     verticadb-operator-leader-election-role   2022-04-14T16:26:53Z
     verticadb-operator-manager-role           2022-04-14T16:26:53Z
     ```

   - RoleBindings in the correct namespace:

     ```
     $ kubectl get rolebinding -n namespace
     NAME                                          ROLE                                         AGE
     verticadb-operator-leader-election-rolebinding   Role/verticadb-operator-leader-election-role   73m
     verticadb-operator-manager-rolebinding           Role/verticadb-operator-manager-role           73m
     ```

Non-cluster administrator installation

Any user can perform operator actions if they use the serviceAccountOverride parameter to install the helm chart with the ServiceAccount with privileges.

1. Add the Vertica helm charts to your repository. The following command installs the CRD Helm chart and names it vertica-charts for future reference:

```
$ helm repo add vertica-charts https://vertica.github.io/charts
```

2. Update your Helm repository to ensure that you are using the latest version of your repository:

```
$ helm repo update
```

3. Install the operator:

```
$ helm install vdb-op -n namespace vertica-charts/verticadb-operator \
  --skip-crds \
  --set webhook.enable=false \
  --set prometheus.createProxyRBAC=false \
  --set skipRoleAndRoleBindingCreation=true \
  --set serviceAccountNameOverride=verticadb-operator-controller-manager
```

Installing the helm chart

Before you can install the Helm chart, you must select a method to configure TLS for the admission controller .

The following install steps use custom certificates:

1. Add the Vertica helm charts to your repository. The following command installs the CRD Helm chart and names it vertica-charts for future reference:

```
$ helm repo add vertica-charts https://vertica.github.io/charts
```

2. Update your Helm repository to ensure that you are using the latest version of your repository:

```
$ helm repo update
```

3. Install the operator Helm chart. The following examples demonstrate the most common Helm chart configurations. For details about the Helm chart options and parameters, see Helm chart parameters .

> **Note**
> Each of the following commands include the --create-namespace option to create the provided namespace if it does not exist. If you do not provide the namespace during install, Helm installs the operator in the current namespace that is defined in the kubectl configuration file.

Enter one of the following commands to customize your Helm chart installation:

- Default configuration. The following command requires cluster administrator privileges:

```
$ helm install operator-name --namespace namespace --create-namespace vertica-charts/verticadb-operator
```

- Custom certificates. Pass custom certificates with the webhook.caBundle , webhook.certSource , and webhook.tlsSecret . The following command requires cluster administrator privileges, and uses the tls-secret Secret created in Defining Custom Certificates :

```
$ helm install operator-name --namespace namespace --create-namespace vertica-charts/verticadb-operator \
  --set webhook.certSource=secret \
  --set webhook.tlsSecret=tls-secret
```

- Service account override. Use service accounts to allow users without cluster administrator privileges to install the operator. Pass the service account with the serviceAccountNameOverride parameter:

```
$ helm install operator-name --namespace namespace --create-namespace vertica-charts/verticadb-operator \
    --set serviceAccountNameOverride=service-account-name
```

  For details, see Granting Operator Installation Privileges .

- Do not install the admission controller webhook. Deploying the webhook requires cluster-scoped privileges that are not required to install the operator. If you use a service account that is granted the privileges required to install the operator but not the webhook, provide the service account with serviceAccountNameOverride , and set webhook.enable to false to deploy only the operator:

```
$ helm install operator-name --namespace namespace --create-namespace vertica-charts/verticadb-operator \
    --set serviceAccountNameOverride=service-account-name
    --set webhook.enable=false
```

> **Caution**
> Webhooks prevent invalid state changes to the custom resource. Running Vertica on Kubernetes without webhook validations might result in invalid state transitions.

For additional details about helm install , see the official documentation .

OperatorHub.io

OperatorHub.io is a registry that allows vendors to share Kubernetes operators. Each vendor must adhere to packaging guidelines to simplify user adoption.

To install the VerticaDB operator from OperatorHub.io, navigate to the Vertica operator page and follow the install instructions.

Upgrading the VerticaDB operator

Vertica supports two separate options to upgrade the VerticaDB operator:

- OperatorHub.io
- Helm Charts

> **Note**
> You must upgrade the operator with the same option that you selected for installation. For example, you cannot install the VerticaDB operator with Helm charts, and then upgrade the operator in the same environment using OperatorHub.io.

Prerequisites

- Complete Installing the Vertica DB operator

OperatorHub.io

The Operator Lifecycle Manager (OLM) operator manages upgrades for OperatorHub.io installations. You can configure the OLM operator to upgrade the VerticaDB operator manually or automatically with the Subscription object's spec.installPlanApproval parameter.

Automatic upgrade

To configure automatic version upgrades, set spec.installPlanApproval to Automatic , or omit the setting entirely. When the OLM operator refreshes the catalog source, it installs the new VerticaDB operator automatically.

Manual upgrade

Upgrade the VerticaDB operator manually to approve version upgrades for specific install plans. To manually upgrade, set spec.installPlanApproval parameter to Manual and complete the following:

1. Verify if there is an install plan that requires approval to proceed with the upgrade:
   ```
   $ kubectl get installplan
   NAME CSV APPROVAL APPROVED
   install-ftcj9 verticadb-operator.v1.7.0 Manual false
   install-pw7ph verticadb-operator.v1.6.0 Manual true
   ```
   The command output shows that the install plan install-ftcj9 for VerticaDB operator version 1.7.0 is not approved.
2. Approve the install plan with a patch command:
   ```
   $ kubectl patch installplan install-ftcj9 --type=merge --patch='{"spec": {"approved": true}}'
   installplan.operators.coreos.com/install-ftcj9 patched
   ```

After you set the approval, the OLM operator silently upgrades the VerticaDB operator. To monitor its progress, inspect the STATUS column of the Subscription object:

```
$ kubectl describe subscription subscription-object-name
```

Helm charts

The CRD is included when you install the Helm chart, but the helm install command does not overwrite an existing CRD. To upgrade the operator, you must update the CRD with the manifest from the GitHub repository. Upgrading the operator with the CRD requires the following prerequisites:

- Cluster administrator privileges
- Complete Installing the Vertica DB operator

Additionally, you must upgrade the VerticaAutoscaler CRD and the EventTrigger CRD , even if you do not use either in your environment. These CRDs are installed with the operator and maintained as separate YAML manifests. Upgrade the VerticaAutoscaler and EventTrigger to ensure that your operator is upgraded completely.

Use kubectl apply to upgrade the CRDs:

1. Upgrade the VerticaDB operator CRD:

```
$ kubectl apply -f https://github.com/vertica/vertica-kubernetes/releases/latest/download/verticadbs.vertica.com-crd.yaml
```

2. Upgrade the VerticaAutoscaler CRD:

```
$ kubectl apply -f https://github.com/vertica/vertica-kubernetes/releases/latest/download/verticaautoscalers.vertica.com-crd.yaml
```

3. Upgrade the EventTrigger CRD:

```
$ kubectl apply -f https://github.com/vertica/vertica-kubernetes/releases/latest/download/eventtriggers.vertica.com-crd.yaml
```

4. Upgrade the Helm chart:

```
$ helm upgrade operator-name --wait vertica-charts/verticadb-operator
```

Helm chart parameters

The following table describes the available settings for the VerticaDB operator and admission controller Helm chart.

Parameter	Description
affinity	Applies rules that constrain the VerticaDB operator to specific nodes. It is more expressive than nodeSelector . If this parameter is not set, then the operator uses no affinity setting.
image.name	The name of the image that runs the operator. **Default:** vertica/verticadb-operator: *version*
imagePullSecrets	A list of Secrets that store credentials to authenticate to the private container repository specified by image.repo and rbac_proxy_image . For details, see Specifying ImagePullSecrets in the Kubernetes documentation.
image.repo	The server that hosts the repository that contains image.name . Use this parameter for deployments that require control over a private hosting server, such as an air-gapped operator. Use this parameter with rbac_proxy_image.name and rbac_proxy_image.repo . **Default:** docker.io
logging.filePath	The path to a log file in the VerticaDB operator filesystem. If this value is not specified, Vertica writes logs to standard output. **Default:** Empty string (' ') that indicates standard output.
logging.level	Minimum logging level. This parameter accepts the following values: • debug • info • warn • error **Default:** info
logging.maxFileSize	When logging.filePath is set, the maximum size in MB of the logging file before log rotation occurs. **Default:** 500
logging.maxFileAge	When logging.filePath is set, the maximum age in days of the logging file before log rotation deletes the file. **Default:** 7

logging.maxFileRotation	When logging.filePath is set, the maximum number of files that are kept in rotation before the old ones are removed. **Default:** 3
nameOverride	Sets the prefix for the name assigned to all objects that the Helm chart creates. If this parameter is not set, each object name begins with the name of the Helm chart, verticadb-operator .

nodeSelector	Provides control over which nodes are used to schedule the operator pod. If this is not set, the node selector is omitted from the operator pod when it is created. To set this parameter, provide a list of key/value pairs.
	
	The following example schedules the operator only on nodes that have the region=us-east label:
	
	`nodeSelector:`
	`  region: us-east`

priorityClassName	The PriorityClass name assigned to the operator pod. This affects where the pod is scheduled.
prometheus.createProxyRBAC	When set to true, creates role-based access control (RBAC) rules that authorize access to the operator's /metrics endpoint for the Prometheus integration . **Default** : true
prometheus.createServiceMonitor	**Deprecated** This parameter is deprecated and will be removed in a future release. When set to true, creates the ServiceMonitor custom resource for the Prometheus operator. You must install the Prometheus operator before you set this to true and install the Helm chart. For details, see the Prometheus operator GitHub repository . **Default** : false
prometheus.expose	Configures the operator's /metrics endpoint for the Prometheus integration . The following options are valid: • EnableWithAuthProxy: Creates a new service object that exposes an HTTPS /metrics endpoint. The RBAC proxy controls access to the metrics. • EnableWithoutAuth: Creates a new service object that exposes an HTTP /metrics endpoint that does not authorize connections. Any client with network access can read the metrics. • Disable: Prometheus metrics are not exposed. **Default** : EnableWithAuthProxy

prometheus.tlsSecret	Secret that contains the TLS certificates for the Prometheus /metrics endpoint. You must create this Secret in the same namespace that you deployed the Helm chart. The Secret requires the following values: • tls.key: TLS private key • tls.crt: TLS certificate for the private key • ca.crt: Certificate authority (CA) certificate To ensure that the operator uses the certificates in this parameter, you must set prometheus.expose to EnableWithAuthProxy . If prometheus.expose is not set to EnableWithAuthProxy , then this parameter is ignored, and the RBAC proxy sidecar generates its own self-signed certificate.
rbac_proxy_image.name	The name of the Kubernetes RBAC proxy image that performs authorization. Use this parameter for deployments that require authorization by a proxy server, such as an air-gapped operator. Use this parameter with image.repo and rbac_proxy_image.repo . **Default:** kubebuilder/kube-rbac-proxy:v0.11.0
rbac_proxy_image.repo	The server that hosts the repository that contains rbac_proxy_image.name . Use this parameter for deployments that perform authorization by a proxy server, such as an air-gapped operator. Use this parameter with image.repo and rbac_proxy_image.name . **Default:** gcr.io

resources.limits and resources.requests	The resource requirements for the operator pod.
	
	resources.limits is the maximum amount of CPU and memory that an operator pod can consume from its host node.
	
	resources.requests is the maximum amount of CPU and memory that an operator pod can request from its host node.
	
	Defaults :
	
	resources:
	  limits:
	  cpu: 100m
	  memory: 750Mi
	  requests:
	  cpu: 100m
	  memory: 20Mi

serviceAccountNameOverride	Service account that identifies any pods in the cluster for apiserver access. A cluster administrator can create a service account that grants the privileges required to install the operator so that users without cluster administrator privileges can install the Helm chart. To correctly control access, the service account's Roles and RoleBindings must exist before you add the service account to the CR. If these are not set, the Vertica Helm chart creates and uses a service account. Vertica provides the required Roles and RoleBindings as GitHub release artifacts . **Default:** Empty string ("")

skipRoleAndRoleBindingCreation	Determines whether the Helm chart creates any Roles or RoleBindings to authorize service accounts with VerticaDB operator privileges. When set to true, the Helm chart does not create any Roles or RoleBindings. This allows a user that cannot create Roles and RoleBindings to install the Helm chart. Vertica provides the required Roles and RoleBindings as GitHub release artifacts . The service account that installs the Helm chart must exist, and you must set serviceAccountNameOverride to that service account. **Default** : false
tolerations	Any taints and tolerations that influence where the operator pod is scheduled.
webhook.caBundle	A PEM-encoded certificate authority (CA) bundle that validates the webhook's server certificate. If this is not set, the webhook uses the system trust roots on the apiserver. **Deprecated** This parameter is deprecated and will be removed in a future release. To add a CA bundle, see webhook.tlsSecret . If webhook.caBundle is set and the webhook.tlsSecret Secret contains a ca.crt key, then the webhook.tlsSecret CA value takes precedence.
webhook.certSource	Determines how TLS certificates are provided for the admission controller webhook. This parameter accepts the following values: • internal: The VerticaDB operator internally generates a self-signed, 10-year expiry certificate before starting the managing controller. When the certificate expires, you must manually restart the operator pod to create a new certificate. • cert-manager: The cert-manager operator generates self-signed certificates and automatically handles private key rotation when the certificate nears expiration. **Deprecated** This TLS certificate management method is deprecated and will be removed in a future release. • secret: You generate the custom certificates before you create the Helm chart and store them in a Secret . This option requires that you set webhook.tlsSecret . If webhook.tlsSecret is set, then this option is implicitly selected. **Default** : internal For details, see Installing the Vertica DB operator .

webhook.enable	Determines if the Helm chart installs the admission controller webhooks for the VerticaDB custom resource and VerticaAutoscaler. If you do not have the privileges required to install the admission controller, set this value to false to deploy the operator only. This parameter enables or disables both webhooks. You cannot enable one webhook and disable the other. **Caution** Webhooks prevent invalid state changes to the custom resource. Running Vertica on Kubernetes without webhook validations might result in invalid state transitions. **Default:** true
webhook.tlsSecret	Secret that contains a PEM-encoded certificate authority (CA) bundle and its keys. The CA bundle validates the webhook's server certificate. If this is not set, the webhook uses the system trust roots on the apiserver. This Secret includes the following keys for the CA bundle: • tls.key • ca.crt • tls.crt

Red Hat OpenShift integration

Red Hat OpenShift is a hybrid cloud platform that provides enhanced security features and greater control over the Kubernetes cluster. In addition, OpenShift provides the OperatorHub, a catalog of operators that meet OpenShift requirements.

For comprehensive instructions about the OpenShift platform, refer to the Red Hat OpenShift documentation .

> **Note**
> If your Kubernetes cluster is in the cloud or on a managed service, each Vertica node must operate in the same availability zone.

Enhanced security with security context constraints

OpenShift requires that each deployment uses a security context constraint (SCC) to enforce enhanced security measures. The SCC lets administrators control the privileges of the pods in a cluster. For example, you can restrict namespace access for specific users in a multi-user environment.

Default SCCs

OpenShift provides default SCCs that provide a range of security features without manual configuration. Vertica on Kubernetes supports the privileged SCC, the most relaxed default SCC. The privileged SCC allows Vertica to assign user and group IDs to the Kubernetes objects in the cluster. In addition, the privileged SCC has the following Linux capabilities that enable internal SSH communication between the pods:

- SYS_CHROOT
- AUDIT_WRITE

Anyuid-extra custom SCC

Vertica provides anyuid-extra , a custom SCC that you can create that extends the anyuid SCC. The anyuid-extra SCC runs Vertica with more restrictions than the privileged SSC. For example, if you do not have the privileges to grant the privileged SCC, you can create the anyuid-extra SCC and add it to your Vertica workloads service account.

For installation details, see Creating a Custom SCC with anyuid-extra .

Installing the operator

The VerticaDB operator is a community operator that is maintained by Vertica. Each operator available in the OperatorHub must adhere to requirements defined by the Operator Lifecycle Manager (OLM). To meet these requirements, vendors must provide a cluster service version (CSV) manifest for each operator. Vertica provides a CSV for each version of the VerticaDB operator available in the OpenShift OperatorHub.

The VerticaDB operator supports OpenShift versions 4.8 and higher.

You must have cluster-admin privileges on your OpenShift account to install the VerticaDB operator. For detailed installation instructions, refer to the OpenShift documentation .

Installing the operator in multiple OpenShift namespaces

By default, the OpenShift user interface (UI) installs the VerticaDB operator in a single OpenShift namespace. In some circumstances, you might require that the operator watch and manage resource objects across multiple OpenShift namespaces.

Prequisites :

- OpenShift CLI tools
- kubectl command line tool

The following steps add the VerticaDB operator to an additional namespace:

1. Create a YAML-formatted OperatorGroup object file. The following example creates file named operatorgroup.yaml:

```
apiVersion: operators.coreos.com/v1alpha2
kind: OperatorGroup
metadata:
 name: vertica-operatorgroup
 namespace: $NAMESPACE
spec:
 targetNamespaces:
 - $NAMESPACE
```

   In the previous command, $NAMESPACE is the namespace where you want to install the operator.
2. Create the OperatorGroup object:

```
$ oc apply -f operatorgroup.yaml
```

3. Create a YAML-formatted Subscription object file to subscribe a namespace to an operator. The following example creates a file named sub.yaml:

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
 name: verticadb-operator
 namespace: $NAMESPACE
spec:
 channel: stable
 name: verticadb-operator
 source: community-operators
 sourceNamespace: openshift-marketplace
```

4. Create the Subscription object:

```
$ oc apply -f sub.yaml
```

   After you create the Subscription object, the OLM is aware of the operator.
5. Use kubectl get to view the installation progress in a separate shell:

```
$ kubectl get -n $NAMESPACE clusterserviceversion -w --selector operators.coreos.com/verticadb-operator.$NAMESPACE
```

When the installation is complete, you can manage the operator from the UI.

Creating a custom SCC with anyuid-extra

Before you can create an operator, you must create the anyuid-extra SCC and add it to your Vertica workloads service account. The Vertica anyuid-extra SCC manifest is available on the Vertica GitHub repository.

1. Create the custom SCC using the anyuid-extra YAML-formatted manifest:

```
$ kubectl apply -f https://github.com/vertica/vertica-kubernetes/releases/latest/download/custom-scc.yaml
```

   For detailed instructions, refer to the OpenShift documentation .
2. Execute the following command to add the custom SCC to your Vertica workloads service account:

```
$ oc adm policy add-scc-to-user -n $NAMESPACE -z verticadb-operator-controller-manager anyuid-extra
```

   In the previous command, $NAMESPACE is the namespace with the operator installation.

By default, the anyuid-extra has a priority setting of 10, so it is automatically selected instead of the default privileged SCC. For additional details about the priority setting, refer to the OpenShift documentation .

Deploying Vertica on OpenShift

After you installed the VerticaDB operator and added a supported SCC to your Vertica workloads service account, you can deploy Vertica on OpenShift.

For details about installing OpenShift in supported environments, see the OpenShift Container Platform installation overview .

Before you deploy Vertica on OpenShift, create the required Secrets to store sensitive information. For details about Secrets and OpenShift, see the OpenShift documentation . For guidance on deploying a Vertica custom resource, see VerticaDB CRD .

Prometheus integration

Vertica on Kubernetes integrates with Prometheus to scrape time series metrics about the VerticaDB operator. These metrics create a detailed model of your application over time, which provides valuable performance and troubleshooting insights. Prometheus exposes these metrics with an HTTP endpoint to facilitate internal and external communications and service discovery in microservice and containerized architectures.

Prometheus requires that you set up targets—metrics that you want to monitor. Each target is exposed on the operator's /metrics endpoint, and Prometheus periodically scrapes that endpoint to collect target data. The operator supports the operator SDK framework , which requires that an authorization proxy impose role-based-access control (RBAC) to access operator metrics. To increase flexibility, Vertica provides the following options to access the /metrics endpoint with Prometheus:

- Use a sidecar container as an RBAC proxy to authorize connections.
- Expose the /metrics endpoint to external connections without RBAC.
- Disable Prometheus entirely.

Vertica provides Helm chart parameters and YAML manifests to configure each option.

> **Note**
> If you installed the VerticaDB operator with OperatorHub.io , you can use the Prometheus integration with the default Helm chart settings. OperatorHub.io installations cannot configure any Helm chart parameters.

Prerequisites

- Complete Installing the Vertica DB operator .
- Install the kubectl command line tool.

Access metrics with RBAC

The operator SDK framework requires that operators use an authorization proxy for metrics access. Because the operator sends metrics to localhost only, Vertica meets these requirements with a sidecar container with localhost access that enforces RBAC.

RBAC rules are cluster-scoped, and the sidecar authorizes connections from clients associated with a service account that has the correct ClusterRole and ClusterRoleBindings. Vertica provides the following example manifests:

- verticadb-operator-proxy-role-cr : ClusterRole that has TokenReviews and SubjectAccessReviews access so that the sidecar can verify privileges on connections.
- verticadb-operator-proxy-rolebinding-crb : ClusterRoleBinding that associates the ClusterRole that verifies sidecar privileges to a service account.
- verticadb-operator-metrics-reader-cr : ClusterRole that allows HTTP GET requests on the /metrics endpoint for non-Kubernetes resources.
- verticadb-operator-metrics-reader-crb : ClusterRoleBinding that associates the metrics reader ClusterRole with a service account.

For additional details about ClusterRoles and ClusterRoleBindings, see the Kubernetes documentation .

Create RBAC rules

> **Note**
> This section details how to create RBAC rules for environments that require that you set up ClusterRole and ClusterRoleBinding objects outside of the Helm chart installation.

The following steps create the ClusterRole and ClusterRoleBindings objects that grant access to the /metrics endpoint to a non-Kubernetes resource such as Prometheus. Because RBAC rules are cluster-scoped, you must create or add to an existing ClusterRoleBinding:

1. Create a ClusterRoleBinding that binds the role for the RBAC sidecar proxy with a service account:
    - Create a ClusterRoleBinding:

```
$ kubectl create clusterrolebinding verticadb-operator-proxy-rolebinding \
  --clusterrole=verticadb-operator-proxy-role \
  --serviceaccount=namespace:serviceaccount
```

- Add a service account to an existing ClusterRoleBinding:

```
$ kubectl patch clusterrolebinding verticadb-operator-proxy-rolebinding \
  --type='json' \
  -p='[{"op": "add", "path": "/subjects/-", "value": {"kind": "ServiceAccount", "name": "serviceaccount","namespace": "namespace" } }]'
```

2. Create a ClusterRoleBinding that binds the role for the non-Kubernetes object to the RBAC sidecar proxy service account:
   - Create a ClusterRoleBinding:

```
$ kubectl create clusterrolebinding verticadb-operator-metrics-reader \
  --clusterrole=verticadb-operator-metrics-reader \
  --serviceaccount=namespace:serviceaccount \
  --group=system:authenticated
```

   - Bind the service account to an existing ClusterRoleBinding:

```
$ kubectl patch clusterrolebinding verticadb-operator-metrics-reader \
  --type='json' \
  -p='[{"op": "add", "path": "/subjects/-", "value": {"kind": "ServiceAccount", "name": "serviceaccount","namespace": "namespace"},
{"op":"add","path":"/subjects/-","value":{"kind": "Group", "name": "system:authenticated"} }]'
```

```
$ kubectl patch clusterrolebinding verticadb-operator-metrics-reader \
  --type='json' \
  -p='[{"op": "add", "path": "/subjects/-", "value": {"kind": "ServiceAccount", "name": "serviceaccount","namespace": "namespace" } }]'
```

When you install the Helm chart , the ClusterRole and ClusterRoleBindings are created automatically. By default, the prometheus.expose parameter is set to EnableWithProxy, which creates the service object and exposes the operator's /metrics endpoint.

For details about creating a sidecar container, see VerticaDB CRD .

Service object

Vertica provides a service object verticadb-operator-metrics-service to access the Prometheus /metrics endpoint. The VerticaDB operator does not manage this service object. By default, the service object uses the ClusterIP service type to support RBAC.

Connect to the /metrics endpoint at port 8443 with the following path:

```
https://verticadb-operator-metrics-service.namespace.svc.cluster.local:8443/metrics
```

Bearer token authentication

Kubernetes authenticates requests to the API server with service account credentials. Each pod is associated with a service account and has the following credentials stored in the filesystem of each container in the pod:

- Token at /var/run/secrets/kubernetes.io/serviceaccount/token
- Certificate authority (CA) bundle at /var/run/secrets/kubernetes.io/serviceaccount/ca.crt

Use these credentials to authenticate to the /metrics endpoint through the service object. You must use the credentials for the service account that you used to create the ClusterRoleBindings.

For example, the following cURL request accesses the /metrics endpoint. Include the --insecure option only if you do not want to verify the serving certificate:

```
$ curl --insecure --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt -H "Authorization: Bearer $(cat
/var/run/secrets/kubernetes.io/serviceaccount/token)" https://verticadb-operator-metrics-service.vertica:8443/metrics
```

For additional details about service account credentials, see the Kubernetes documentation .

TLS client certificate authentication

Some environments might prevent you from authenticating to the /metrics endpoint with the service account token. For example, you might run Prometheus outside of Kubernetes. To allow external client connections to the /metrics endpoint, you have to supply the RBAC proxy sidecar with TLS certificates.

You must create a Secret that contains the certificates, and then use the prometheus.tlsSecret Helm chart parameter to pass the Secret to the RBAC proxy sidecar when you install the Helm chart. The following steps create the Secret and install the Helm chart:

1. Create a Secret that contains the certificates:

   ```
   $ kubectl create secret generic metrics-tls --from-file=tls.key=/path/to/tls.key --from-file=tls.crt=/path/to/tls.crt --from-file=ca.crt=/path/to/ca.crt
   ```

2. Install the Helm chart with prometheus.tlsSecret set to the Secret that you just created:

   ```
   $ helm install operator-name --namespace namespace --create-namespace vertica-charts/verticadb-operator \
     --set prometheus.tlsSecret=metrics-tls
   ```

   The prometheus.tlsSecret parameter forces the RBAC proxy to use the TLS certificates stored in the Secret. Otherwise, the RBAC proxy sidecar generates its own self-signed certificate.

After you install the Helm chart, you can authenticate to the /metrics endpoint with the certificates in the Secret. For example:

```
$ curl --key tls.key --cert tls.crt --cacert ca.crt https://verticadb-operator-metrics-service.vertica.svc:8443/metrics
```

Prometheus operator integration (optional)

Vertica on Kubernetes integrates with the Prometheus operator , which provides custom resources (CRs) that simplify targeting metrics. Vertica supports the ServiceMonitor CR that discovers the VerticaDB operator automatically, and authenticates requests with a bearer token .

The ServiceMonitor CR is available as a release artifact in our GitHub repository . See Helm chart parameters for details about the prometheus.createServiceMonitor parameter.

Access metrics without RBAC

You might have an environment that does not require privileged access to Prometheus metrics. For example, you might run Prometheus outside of Kubernetes.

To allow external access to the /metrics endpoint with HTTP, set prometheus.expose to EnableWithoutAuth. For example:

```
$ helm install operator-name --namespace namespace --create-namespace vertica-charts/verticadb-operator \
  --set prometheus.expose=EnableWithoutAuth
```

Service object

Vertica provides a service object verticadb-operator-metrics-service to access the Prometheus /metrics endpoint. The VerticaDB operator does not manage this service object. By default, the service object uses the ClusterIP service type, so you must change the serviceType for external client access. The service object's fully-qualified domain name (FQDN) is as follows:

```
verticadb-operator-metrics-service.namespace.svc.cluster.local
```

Connect to the /metrics endpoint at port 8443 with the following path:

```
http://verticadb-operator-metrics-service.namespace.svc.cluster.local:8443/metrics
```

Disabling prometheus

To disable Prometheus, set the prometheus.expose Helm chart parameter to Disable. The following is an example command:

```
$ helm install operator-name --namespace namespace --create-namespace vertica-charts/verticadb-operator \
  --set prometheus.expose=Disable
```

For details about Helm install commands, see Installing the Vertica DB operator .

Configuring communal storage

Vertica on Kubernetes supports a variety of communal storage providers to accommodate your storage requirements. Each storage provider uses authentication methods that conceal sensitive information so that you can declare that information in your Custom Resource (CR) without exposing any literal values.

> **Note**
> If your Kubernetes cluster is in the cloud or on a managed service, each Vertica node must operate in the same availability zone.

AWS S3 or S3-Compatible storage

Vertica on Kubernetes supports multiple authentication methods for Amazon Web Services (AWS) communal storage locations and private cloud S3 storage such as MinIO .

For additional details about Vertica and AWS, see Vertica on Amazon Web Services .

Secrets authentication

To connect to an S3-compatible storage location, create a Secret to store both your communal access and secret key credentials. Then, add the Secret, path, and S3 endpoint to the CR spec.

1. The following command stores both your S3-compatible communal access and secret key credentials in a Secret named s3-creds :

```
$ kubectl create secret generic s3-creds --from-literal=accesskey=accesskey --from-literal=secretkey=secretkey
```

2. Add the Secret to the communal section of the CR spec:

```
spec:
  ...
  communal:
    credentialSecret: s3-creds
    endpoint: https://path/to/s3-endpoint
    path: s3://bucket-name/key-name
    ...
```

For a detailed description of an S3-compatible storage implementation, see VerticaDB CRD .

IAM profile authentication

Identify and access management (IAM) profiles manage user identities and control which services and resources a user can access. IAM authentication to Vertica on Kubernetes reduces the number of manual updates when you rotate your access keys.

The IAM profile must have read and write access to the communal storage. The IAM profile is associated with the EC2 instances that run worker nodes.

1. Create an EKS node group using a Node IAM role with a policy that allows read and write access to the S3 bucket used for communal storage.
2. Deploy the VerticaDB operator in a namespace. For details, see Installing the Vertica DB operator .
3. Create a VerticaDB custom resource (CR) , and omit the communal.credentialSecret field:

```
spec:
  ...
  communal:
    endpoint: https://path/to/s3-endpoint
    path: s3://bucket-name/key-name
```

When the Vertica server accesses the communal storage location, it uses the policy associated to the EKS node.

For additional details about authenticating to Vertica with an IAM profile, see AWS authentication .

IRSA profile authentication

Important
This authentication method requires an image running Vertica server version 12.0.3 or later.

You can use IAM roles for service accounts (IRSA) to associate an IAM role with a Kubernetes service account. You must set the IAM policies for the Kubernetes service account, and then pods running that service account have the IAM policies.

Before you begin, complete the following prerequisites:

- Configure the EKS cluster's control plane. For details, see the Amazon documentation .
- Create a bucket policy that has access to the S3 communal storage bucket. For details, see the Amazon documentation .

1. Create an EKS node group using a Node IAM role that does not have S3 access.
2. Use eksctl to create the IAM OpenID Connect (OIDC) provider for your EKS cluster:

```
$ eksctl utils associate-iam-oidc-provider --cluster cluster --approve
2022-10-07 08:31:37 [ℹ]  will create IAM Open ID Connect provider for cluster "cluster" in "us-east-1"
2022-10-07 08:31:38 [✔]  created IAM Open ID Connect provider for cluster "cluster" in "us-east-1"
```

3. Create the Kubernetes namespace where you deploy the VerticaDB operator:

```
$ kubectl create ns vertica
namespace/vertica created
```

4. Use eksctl to create a Kubernetes service account in the vertica namespace. When you create a service account with eksctl , you can attach an IAM policy that allows S3 access:

```
$ eksctl create iamserviceaccount --name my-serviceaccount --namespace vertica --cluster cluster --attach-policy-arn arn:aws:iam::profile:policy/policy --approve
2022-10-07 08:38:32 [ℹ]  1 iamserviceaccount (vertica/my-serviceaccount) was included (based on the include/exclude rules)
2022-10-07 08:38:32 [!]  serviceaccounts that exist in Kubernetes will be excluded, use --override-existing-serviceaccounts to override
2022-10-07 08:38:32 [ℹ]  1 task: {
    2 sequential sub-tasks: {
        create IAM role for serviceaccount "vertica/my-serviceaccount",
        create serviceaccount "vertica/my-serviceaccount",
    } }2022-10-07 08:38:32 [ℹ]  building iamserviceaccount stack "eksctl-cluster-addon-iamserviceaccount-vertica-my-serviceaccount"
2022-10-07 08:38:33 [ℹ]  deploying stack "eksctl-cluster-addon-iamserviceaccount-vertica-my-serviceaccount"
2022-10-07 08:38:33 [ℹ]  waiting for CloudFormation stack "eksctl-cluster-addon-iamserviceaccount-vertica-my-serviceaccount"
2022-10-07 08:39:03 [ℹ]  waiting for CloudFormation stack "eksctl-cluster-addon-iamserviceaccount-vertica-my-serviceaccount"
2022-10-07 08:39:04 [ℹ]  created serviceaccount "vertica/my-serviceaccount"
```

5. Install the VerticaDB operator, and set the service account:

```
$ helm install vdb-op --namespace vertica vertica-charts/verticadb-operator --set serviceAccountNameOverride=my-serviceaccount
```

6. Create a VerticaDB custom resource (CR) , and omit the communal.credentialSecret field. When pods are created, they use the service account that has a policy that provides access to the S3 communal storage:

```
apiVersion: vertica.com/v1beta1
kind: VerticaDB
metadata:
  name: irsadb
spec:
  image: vertica/vertica-k8s:12.0.3-0
  communal:
    path: "s3://path/to/s3-endpoint
    endpoint: https://s3.amazonaws.com
  subclusters:
    - name: sc
      size: 3
```

Server-side encryption

Important
Vertica supports S3 server-side encryption in versions 12.0.1 and higher.

If your S3 communal storage uses server-side encryption (SSE), you must configure the encryption type when you create the CR. Vertica supports the following types of SSE:

- SSE-S3
- SSE-KMS
- SSE-C

For details about Vertica support for each encryption type, see S3 object store .

The following tabs provide examples on how to implement each SSE type. For details about the parameters, see Custom resource definition parameters .

SSE-S3

SSE-KMS

SSE-C

```
apiVersion: vertica.com/v1beta1
kind: VerticaDB
metadata:
  name: verticadb
spec:
  communal:
    path: "s3://bucket-name"
    s3ServerSideEncryption: SSE-S3
```

Google Cloud Storage

Authenticating to Google Cloud Storage (GCS) requires your hash-based message authentication code (HMAC) access and secret keys, and the path to your GCS bucket. For details about HMAC keys, see Eon Mode on GCP prerequisites .

1. The following command stores your HMAC access and secret key in a Secret named gcs-creds :

```
$ kubectl create secret generic gcs-creds --from-literal=accesskey=accessKey --from-literal=secretkey=secretkey
```

2. Add the Secret and the path to the GCS bucket that contains your Vertica database to the communal section of the CR spec:

```
spec:
  ...
  communal:
    credentialSecret: gcs-creds
    path: gs://bucket-name/path/to/database-name
    ...
```

For additional details about Vertica and GCS, see Vertica on Google Cloud Platform .

Azure Blob Storage

Micosoft Azure provides a variety of options to authenticate to Azure Blob Storage location. Depending on your environment, you can use one of the following combinations to store credentials in a Secret :

- accountName and accountKey
- accountName and shared access signature (SAS)

If you use an Azure storage emulator such as Azurite in a tesing environment, you can authenticate with accountName and blobStorage values.

Important
Vertica does not officially support Azure storage emulators as a communal storage location.

1. The following command stores accountName and accountKey in a Secret named azb-creds :

```
$ kubectl create secret generic azb-creds --from-literal=accountKey=accessKey --from-literal=accountName=accountName
```

   Alternately, you could store your accountName and your SAS credentials in azb-creds :

```
$ kubectl create secret generic azb-creds --from-literal=sharedAccessSignature=sharedAccessSignature --from-literal=accountName=accountName
```

2. Add the Secret and the path that contains your AZB storage bucket to the communal section of the CR spec:

```
spec:
  ...
  communal:
    credentialSecret: azb-creds
    path: azb://accountName/bucket-name/database-name
    ...
```

For details about Vertica and authenticating to Microsoft Azure, see Eon Mode on GCP prerequisites .

Hadoop file storage

Connect to Hadoop Distributed Filesystem (HDFS) communal storage with the standard webhdfs scheme, or the swebhdfs scheme for wire encryption. In addition, you must add your HDFS configuration files in a ConfigMap , a Kubernetes object that stores data in key-value pairs. You can optionally configure Kerberos to authenticate connections to your HDFS storage location.

The following example uses the swebhdfs wire encryption scheme that requires a certificate authority (CA) bundle in the CR spec.

1. The following command stores a PEM-encoded CA bundle in a Secret named hadoop-cert :

```
$ kubectl create secret generic hadoop-cert --from-file=ca-bundle.pem
```

2. HDFS configuration files are located in the /etc/hadoop directory. The following command creates a ConfigMap named hadoop-conf :

```
$ kubectl create configmap hadoop-conf --from-file=/etc/hadoop
```

3. Add the configuration values to the communal and certSecrets sections of the spec:

```
spec:
  ...
  communal:
    path: "swebhdfs://path/to/database"
    hadoopConfig: hadoop-conf
    caFile: /certs/hadoop-cert/ca-bundle.pem
  certSecrets:
    - name: hadoop-cert
  ...
```

The previous example defines the following:

- communal.path : The path to the database, using the wire encryption scheme. Enclose the path in double quotes.
- communal.hadoopConfig : The ConfigMap storing the contents of the /etc/hadoop directory.
- communal.caFile : The mount path in the container filesystem containing the CA bundle used to create the hadoop-cert Secret.
- certSecrets.name : The Secret containing the CA bundle.

For additional details about HDFS and Vertica, see Apache Hadoop integration .

Kerberos authentication (optional)

Vertica authenticates connections to HDFS with Kerberos. The Kerberos configuration between Vertica on Kubernetes is the same as between a standard Eon Mode database and Kerberos, as described in Kerberos authentication .

1. The following command stores the krb5.conf and krb5.keytab files in a Secret named krb5-creds :

   ```
   $ kubectl create secret generic krb5-creds --from-file=kerberos-conf=/etc/krb5.conf --from-file=kerberos-keytab=/etc/krb5.keytab
   ```

   Consider the following when managing the krb5.conf and krb5.keytab files in Vertica on Kubernetes:

   - Each pod uses the same krb5.keytab file, so you must update the krb5.keytab file before you begin any scaling operation.
   - When you update the contents of the krb5.keytab file, the operator updates the mounted files automatically, a process that does not require a pod restart.
   - The krb5.conf file must include a [domain_realm] section that maps the Kubernetes cluster domain to the Kerberos realm. The following example maps the default .cluster.local domain to a Kerberos realm named EXAMPLE.COM:

     ```
     [domain_realm]
       .cluster.local = EXAMPLE.COM
     ```

2. Add the Secret and additional Kerberos configuration information to the CR:

   ```
   spec:
     ...
     communal:
       path: "swebhdfs://path/to/database"
       hadoopConfig: hadoop-conf
       kerberosServiceName: verticadb
       kerberosRealm: EXAMPLE.COM
     kerberosSecret: krb5-creds
     ...
   ```

The previous example defines the following:

- communal.path : The path to the database, using the wire encryption scheme. Enclose the path in double quotes.
- communal.hadoopConfig : The ConfigMap storing the contents of the /etc/hadoop directory.
- communal.kerberosServiceName : The service name for the Vertica principal.
- communal.kerberosRealm : The realm portion of the principal.
- kerberosSecret : The Secret containing the krb5.conf and krb5.keytab files.

For a complete definition of each of the previous values, see Custom resource definition parameters .

Custom resource definitions

The custom resource definition (CRD) is a shared global object that extends the Kubernetes API beyond the standard resource types. The CRD serves as a blueprint for custom resource (CR) instances. You create CRs that specify the desired state of your environment, and the operator monitors the CR to maintain state for the objects within its namespace.

In this section

- VerticaDB CRD
- VerticaAutoscaler CRD
- EventTrigger CRD

VerticaDB CRD

The VerticaDB custom resource definition (CRD) deploys an Eon Mode database. Each subcluster is a StatefulSet , a workload resource type that persists data with ephemeral Kubernetes objects.

A VerticaDB custom resource (CR) requires a primary subcluster and a connection to a communal storage location to persist its data. The VerticaDB operator monitors the CR to maintain its desired state and validate state changes.

The following sections provide a YAML-formatted manifest that defines the minimum required fields to create a VerticaDB CR, and each subsequent section implements a production-ready recommendation or best practice using custom resource parameters. For a comprehensive list of all parameters and their definitions, see custom resource parameters .

Prerequisites

- Complete Installing the Vertica DB operator .
- Configure a dynamic volume provisioner .
- Confirm that you have the resources to deploy objects you plan to create.
- Optionally, acquire a Vertica license. By default, the Helm chart deploys the free Community Edition license. This license limits you to a three-node cluster and 1TB data.
- Configure a supported communal storage location with an empty communal path bucket.
- Understand Kubernetes Secrets . Secrets conceal sensitive information in your custom resource.

> **Note**
> Instead of creating a Secret with kubectl, you can manually base64 encode a string on the command line and then add the encoded output to a Secrets manifest.
>
> For example, pass the string value to the echo command, and pipe the output to the base64 command to encode the value. In the echo command, include the -n option so that it does not append a newline character:
>
> ```
> $ echo -n 'secret-value' | base64
> c2VjcmV0LXZhbHVl
> ```
>
> For detailed steps about creating the manifest and applying it to a namespace, see the Kubernetes documentation .

Minimal manifest

At minimum, a VerticaDB CR requires a connection to an empty communal storage bucket and a primary subcluster definition. The operator is namespace-scoped, so make sure that you apply the CR manifest in the same namespace as the operator.

The following VerticaDB CR connects to S3 communal storage and deploys a three-node primary subcluster on three nodes. This manifest serves as the starting point for all implementations detailed in the subsequent sections:

```
apiVersion: vertica.com/v1beta1
kind: VerticaDB
metadata:
  name: cr-name
spec:
  licenseSecret: vertica-license
  superuserPasswordSecret: su-password
  communal:
    path: "s3://bucket-name/key-name"
    endpoint: "https://path/to/s3-endpoint"
    credentialSecret: s3-creds
    region: region
  subclusters:
    - name: primary
      size: 3
  shardCount: 6
```

The following sections detail the minimal manifest's CR parameters, and how to create the CR in the current namespace.

Required fields

Each VerticaDB manifest begins with required fields that describe the version, resource type, and metadata:

- apiVersion : The API group and Kubernetes API version in api-group/version format.
- kind : The resource type. VerticaDB is the name of the Vertica custom resource type.
- metadata : Data that identifies objects in the namespace.
  - name : The name of this CR object. Provide a unique metadata.name value so that you can identify the CR and its resources in its namespace.

spec definition

The spec field defines the desired state of the CR. The operator control loop compares the spec definition to the current state and reconciles any differences.

Nest all fields that define your StatefulSet under the spec field.

Add a license

By default, the Helm chart pulls the free Vertica Community Edition (CE) image . The CE image has a restricted license that limits you to a three-node cluster and 1TB of data.

To add your license so that you can deploy more nodes and use more data, store your license in a Secret and add it to the manifest:

1. Create a Secret from your Vertica license file:
   ```
   $ kubectl create secret generic vertica-license --from-file=license.dat=/path/to/license-file.dat
   ```
2. Add the Secret to the licenseSecret field:
   ```
   ...
   spec:
     licenseSecret: vertica-license
     ...
   ```

The licenseSecret value is mounted in the Vertica server container in the /home/dbadmin/licensing/mnt directory.

Add password authentication

The superuserPasswordSecret field enables password authentication for the database. You must define this field when you create the CR—you cannot define a password for an existing database.

To create a database password, conceal it in a Secret before you add it to the mainfest:

1. Create a Secret from a literal string. You must use password as the key:
   ```
   $ kubectl create secret generic su-passwd --from-literal=password=password-value
   ```
2. Add the Secret to the superuserPasswordSecret field:

```
...
spec:
  ...
  superuserPasswordSecret: su-password
```

Connect to communal storage

Vertica on Kubernetes supports multiple communal storage locations . For implementation details for each communal storage location, see Configuring communal storage .

This CR connects to an S3 communal storage location. Define your communal storage location with the communal field:

```
...
spec:
  ...
  communal:
    path: "s3://bucket-name/key-name"
    endpoint: "https://path/to/s3-endpoint"
    credentialSecret: s3-creds
    region: region
  ...
```

This manifest sets the following parameters:

- credentialSecret : The Secret that contains your communal access and secret key credentials.
  The following command stores both your S3-compatible communal access and secret key credentials in a Secret named s3-creds :
  ```
  $ kubectl create secret generic s3-creds --from-literal=accesskey=accesskey --from-literal=secretkey=secretkey
  ```

  > **Note**
  > Omit credentialSecret for environments that authenticate to S3 communal storage with Identity and Access Management (IAM) or IAM roles for service accounts (IRSA)—these methods do not require that you store your credentials in a Secret. For details, see Configuring communal storage .

- endpoint : The S3 endpoint URL.
- path : The location of the S3 storage bucket, in S3 bucket notation. This bucket must exist before you create the custom resource. After you create the custom resource, you cannot change this value.
- region : The geographic location of the communal storage resources. This field is valid for AWS and GCP only. If you set the wrong region, you cannot connect to the communal storage location.

Define a primary subcluster

Each CR requires a primary subcluster or it returns an error. At minimum, you must define the name and size of the subcluster:

```
...
spec:
  ...
  subclusters:
    - name: primary
      size: 3
  ...
```

This manifest sets the following parameters:

- name : The name of the subcluster.
- size : The number of pods in the subcluster.

When you define a CR with a single subcluster, the operator designates it as the primary subcluster. If your manifest includes multiple subclusters, you must use the isPrimary parameter to identify the primary subcluster. For example:

```
spec:
  ...
  subclusters:
    - name: primary
      size: 3
      isPrimary: true
    - name: secondary
      size: 3
```

For additional details about primary and secondary subclusters, see Subclusters .

Set the shard count

shardCount specifies the number of shards in the database, which determines how subcluster nodes subscribe to communal storage data. You cannot change this value after you instantiate the CR. When you change the number of pods in a subcluster or add or remove a subcluster, the operator rebalances shards automatically .

Vertica recommends that the shard count equals double the number of nodes in the cluster. Because this manifest creates a three-node cluster with one Vertica server container per node, set shardCount to 6 :

```
...
spec:
  ...
  shardCount: 6
```

For guidance on selecting the shard count, see Configuring your Vertica cluster for Eon Mode . For details about limiting each node to one Vertica server container, see Node affinity .

Apply the manifest

After you define the minimal manifest in a YAML-formatted file, use kubectl to create the VerticaDB CR. The following command creates a CR in the current namespace:

```
$ kubectl apply -f minimal.yaml
verticadb.vertica.com/cr-name created
```

After you apply the manifest, the operator creates the primary subcluster, connects to the communal storage, and creates the database. You can use kubectl wait to see when the database is ready:

```
$ kubectl wait --for=condition=DBInitialized=True vdb/cr-name --timeout=10m
verticadb.vertica.com/cr-name condition met
```

Specify an image

Each time the operator launches a container, it pulls the image for the most recently released Vertica version from the Vertica Dockerhub repository. Vertica recommends that you explicitly set the image that the operator pulls for your CR. For a list of available Vertica images, see the Vertica Dockerhub registry .

To run a specific image version, set the image parameter in docker-registry-hostname/image-name:tag format:

```
spec:
  ...
  image: vertica/vertica-k8s:version
```

When you specify an image other than the latest , the operator pulls the image only when it is not available locally. You can control when the operator pulls the image with the imagePullPolicy custom resource parameter .

Important

If your environment uses the Vertica Kubernetes (No keys) image, you must provide SSH keys for internal communication between the pods. This requires that you add the keys as a Secret with sshSecret parameter:

```
$ kubectl create secret generic ssh-keys --from-file=/path/to/ssh/keys
```

You can add this Secret with the sshSecret parameter:

```
spec:
  ...
  sshSecret: ssh-keys
```

Communal storage authentication

Your communal storage validates HTTPS connections with a self-signed certificate authority (CA) bundle. You must make the CA bundle's root certificate available to each Vertica server container so that the communal storage can authenticate requests from your subcluster.

This authentication requires that you set the following parameters:

- certSecrets : Adds a Secret that contains the root certificate.
  This parameter is a list of Secrets that encrypt internal and external communications for your CR. Each certificate is mounted in the Vertica server container filesystem in the /certs/ *Secret-name* / *cert-name* directory.
- communal.caFile : Makes the communal storage location aware of the mount path that stores the certificate Secret.

Complete the following to add these parameters to the manifest:

1. Create a Secret that contains the PEM-encoded root certificate. The following command creates a Secret named aws-cert :
   ```
   $ kubectl create secret generic aws-cert --from-file=root-cert.pem
   ```
2. Add the certSecrets and communal.caFile parameters to the manifest:
   ```
   spec:
     ...
     communal:
       ...
       caFile: /certs/aws-cert/root_cert.pem
     certSecrets:
       - name: aws-cert
   ```

Now, the communal storage authenticates requests with the /certs/aws-cert/root_cert.pem file, whose contents are stored in the aws-cert Secret.

External client connections

Each subcluster communicates with external clients and internal pods through a service object. To configure the service object to accept external client connections, set the following parameters:

- serviceName : Assigns a custom name to the service object. A custom name lets you identify it among multiple subclusters.
  Service object names use the *metadata.name-serviceName* naming convention.
- serviceType : Defines the type of the subcluster service object.
  By default, a subcluster uses the ClusterIP serviceType, which sets a stable IP and port that is accessible from within Kubernetes only. In many circumstances, external client applications need to connect to a subcluster that is fine-tuned for that specific workload. For external client access, set the serviceType to NodePort or LoadBalancer .

  > **Note**
  > The LoadBalancer service type is an external service type that is managed by your cloud provider. For implementation details, refer to the Kubernetes documentation and your cloud provider's documentation.

- serviceAnnotations : Assigns a custom annotation to the service object for implementation-specific services.

Add these external client connection parameters under the subclusters field:

```
spec:
  ...
  subclusters:
    ...
    serviceName: connections
    serviceType: LoadBalancer
    serviceAnnotations:
      service.beta.kubernetes.io/load-balancer-source-ranges: 10.0.0.0/24
```

This example creates a LoadBalancer service object named verticadb-connections . The serviceAnnotations parameter defines the CIDRs that can access the network load balancer (NLB). For additional details, see the AWS Load Balancer Controller documentation.

> **Note**
> If you run your CR on Amazon Elastic Kubernetes Service (EKS), Vertica recommends the AWS Load Balancer Controller . To use the AWS Load Balancer Controller, apply the following annotations:
>
> ```
> serviceAnnotations:
>   service.beta.kubernetes.io/aws-load-balancer-type: external
>   service.beta.kubernetes.io/aws-load-balancer-nlb-target-type: ip
>   service.beta.kubernetes.io/aws-load-balancer-scheme: internet-facing
> ```
>
> For longer-running queries, you might need to configure TCP keepalive settings.

For additional details about Vertica and service objects, see Containerized Vertica on Kubernetes .

Authenticate clients

You might need to connect applications or command-line interface (CLI) tools to your VerticaDB CR. You can add TLS certificates that authenticate client requests with the certSecrets parameter:

1. Create a Secret that contains your TLS certificates. The following command creates a Secret named mtls :
   ```
   $ kubectl create secret generic mtls --from-file=mtls=/path/to/mtls-cert
   ```
2. Add the Secret to the certSecrets parameter :
   ```
   spec:
     ...
     certSecrets:
       ...
       - name: mtls
   ```
   This mounts the TLS certificates in the /certs/mtls/mtls-cert directory.

Sidecar logger

A sidecar is a utility container that runs in the same pod as your main application container and performs a task for that main application's process. The VerticaDB CR uses a sidecar container to handle logs for the Vertica server container. You can use the vertica-logger image to add a sidecar that sends logs from vertica.log to standard output on the host node for log aggregation.

Add a sidecar with the sidecars parameter. This parameter accepts a list of sidecar definitions, where each element specifies the following:

- name : Name of the sidecar. name indicates the beginning of a sidecar element.
- image : Image for the sidecar container.

The following example adds a single sidecar container that shares a pod with each Vertica server container:

```
spec:
  ...
  sidecars:
    - name: sidecar-container
      image: sidecar-image:latest
```

This configuration persists logs only for the lifecycle of the container. To persist log data between pod lifecycles, you must mount a custom volume in the sidecar filesystem.

Persist logs with a volume

An external service that requires long-term access to Vertica server data should use a volume to persist that data between pod lifecycles. For details about volumes, see the Kubernetes documentation .

The following parameters add a volume to your CR and mounts it in a sidecar container:

- volumes : Make a custom volume available to the CR so that you can mount it in a container filesystem. This parameter requires a name value and a volume type.
- sidecars[i].volumeMounts : Mounts one or more volumes in the sidecar container filesystem. This parameter requires a name value and a

mountPath value that defines where the volume is mounted in the sidecar container.

> **Note**
> Vertica also provides a spec.volumeMounts parameter so you can mount volumes for other use cases. This parameter behaves like sidecars[i].volumeMounts , but it mounts volumes in the Vertica server container filesystem.
>
> For details, see Custom resource definition parameters .

The following example creates a volume of type emptyDir , and mounts it in the sidecar-container filesystem:

```
spec:
  ...
  volumes:
    - name: sidecar-vol
      emptyDir: {}
  ...
  sidecars:
    - name: sidecar-container
      image: sidecar-image:latest
      volumeMounts:
        - name: sidecar-vol
          mountPath: /path/to/sidecar-vol
```

Resource limits and requests

You should limit the amount of CPU and memory resources that each host node allocates for the Vertica server pod, and set the amount of resources each pod can request.

To control these values, set the following parameters under the subclusters.resources field:

- limits.cpu : Maximum number of CPUs that each server pod can consume.
- limits.memory : Maximum amount of memory that each server pod can consume.
- requests.cpu : Number CPUs that each pod requests from the host node.
- requests.memory : Amount of memory that each pod requests from a PV.

When you change resource settings, Kubernetes restarts each pod with the updated settings.

> **Note**
> Select resource settings that your host nodes can accommodate. When a pod is started or rescheduled, Kubernetes searches for host nodes with enough resources available to start the pod. If there is not a host node with enough resources, the pod **STATUS** stays in **Pending** until the resources become available.
>
> For guidance on setting production limits and requests, see Recommendations for Sizing Vertica Nodes and Clusters .

As a best practice, set resource.limits.* and resource.requests.* to equal values so that the pods are assigned to the Guaranteed Quality of Service (QoS) class . Equal settings also provide the best safeguard against the Out Of Memory (OOM) Killer in constrained environments.

The following example allocates 32 CPUs and 96 gigabytes of memory on the host node, and limits the requests to the same values. Because the limits.* and requests.* values are equal, the pods are assigned the Guaranteed QoS class:

```
spec:
  ...
  subclusters:
    ...
    resources:
      limits:
        cpu: 32
        memory: 96Gi
      requests:
        cpu: 32
        memory: 96Gi
```

Node affinity

Kubernetes affinity and anti-affinity settings control which resources the operator uses to schedule pods. As a best practice, you should set affinity to ensure that a single node does not serve more than one Vertica pod.

The following example creates an anti-affinity rule that schedules only one Vertica server pod per node:

```
spec:
  ...
  subclusters:
    ...
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
            - key: app.kubernetes.io/name
              operator: In
              values:
              - vertica
          topologyKey: "kubernetes.io/hostname"
```

The following provides a detailed explanation about all settings in the previous example:

- affinity : Provides control over pod and host scheduling using labels.
- podAntiAffinity : Uses pod labels to prevent scheduling on certain resources.
- requiredDuringSchedulingIgnoredDuringExecution : The rules defined under this statement must be met before a pod is scheduled on a host node.
- labelSelector : Identifies the pods affected by this affinity rule.
- matchExpressions : A list of pod selector requirements that consists of a key , operator , and values definition. This matchExpression rule checks if the host node is running another pod that uses a vertica label.
- topologyKey : Defines the scope of the rule. Because this uses the hostname topology label, this applies the rule in terms of pods and host nodes.

For additional details, see the Kubernetes documentation .

VerticaAutoscaler CRD

The VerticaAutoscaler custom resource (CR) is a HorizontalPodAutoscaler that automatically scales resources for existing subclusters using one of the following strategies:

- Subcluster scaling for short-running dashboard queries.
- Pod scaling for long-running analytic queries.

The VerticaAutoscaler CR scales using resource or custom metrics . Vertica manages subclusters by workload, which helps you pinpoint the best metrics to trigger a scaling event. To maintain data integrity, the operator does not scale down unless all connections to the pods are drained and sessions are closed.

For details about the algorithm that determines when the VerticaAutoscaler scales, see the Kubernetes documentation .

Additionally, the VerticaAutoscaler provides a webhook to validate state changes. By default, this webhook is enabled. You can configure this webhook with the webhook.enable Helm chart parameter .

Examples

The examples in this section use the following VerticaDB custom resource. Each example uses CPU to trigger scaling:

```
apiVersion: vertica.com/v1beta1
kind: VerticaDB
metadata:
  name: dbname
spec:
  communal:
    path: "path/to/communal-storage"
    endpoint: "path/to/communal-endpoint"
    credentialSecret: credentials-secret
  subclusters:
    - name: primary1
      size: 3
      isPrimary: true
      serviceName: primary1
      resources:
        limits:
          cpu: "8"
        requests:
          cpu: "4"
```

Prerequisites

- Complete Installing the Vertica DB operator .
- Install the kubectl command line tool.
- Complete VerticaDB CRD .
- Confirm that you have the resources to scale.

> **Note**
> By default, the custom resource uses the free Community Edition (CE) license. This license allows you to deploy up to three nodes with a maximum of 1TB of data. To add resources beyond these limits, you must add your Vertica license to the custom resource as described in VerticaDB CRD .

- Set a value for the metric that triggers scaling. For example, if you want to scale by CPU utilization, you must set CPU limits and requests .

Subcluster scaling

Automatically adjust the number of subclusters in your custom resource to fine-tune resources for short-running dashboard queries. For example, increase the number of subclusters to increase throughput. For more information, see Improving query throughput using subclusters .

All subclusters share the same service object, so there are no required changes to external service objects. Pods in the new subcluster are load balanced by the existing service object.

The following example creates a VerticaAutoscaler custom resource that scales by subcluster when the VerticaDB uses 50% of the node's available CPU:

1. Define the VerticaAutoscaler custom resource in a YAML-formatted manifest:

   ```
   apiVersion: vertica.com/v1beta1
   kind: VerticaAutoscaler
   metadata:
     name: autoscaler-name
   spec:
     verticaDBName: dbname
     scalingGranularity: Subcluster
     serviceName: primary1
   ```

2. Create the VerticaAutoscaler with the kubectl autoscale command:

   ```
   $ kubectl autoscale verticaautoscaler autoscaler-name --cpu-percent=50 --min=3 --max=12
   ```

   The previous command creates a HorizontalPodAutoscaler object that:
   - Sets the target CPU utilization to 50%.
   - Scales to a minimum of three pods in one subcluster, and 12 pods in four subclusters.

Pod scaling

For long-running, analytic queries, increase the pod count for a subcluster. For additional information about Vertica and analytic queries, see Using elastic crunch scaling to improve query performance .

When you scale pods in an Eon Mode database, you must consider the impact on database shards. For details, see Shards and subscriptions .

The following example creates a VerticaAutoscaler custom resource that scales by pod when the VerticaDB uses 50% of the node's available CPU:

1. Define the VerticaAutoScaler custom resource in a YAML-formatted manifest:

```
apiVersion: vertica.com/v1beta1
kind: VerticaAutoscaler
metadata:
  name: autoscaler-name
spec:
  verticaDBName: dbname
  scalingGranularity: Pod
  serviceName: primary1
```

2. Create the autoscaler instance with the kubectl autoscale command:

```
$ kubectl autoscale verticaautoscaler autoscaler-name --cpu-percent=50 --min=3 --max=12
```

   The previous command creates a HorizontalPodAutoscaler object that:
   - Sets the target CPU utilization to 50%.
   - Scales to a minimum of three pods in one subcluster, and 12 pods in four subclusters.

Event monitoring

To view the VerticaAutoscaler object, use the kubetctl describe hpa command:

```
$ kubectl describe hpa autoscaler-name
Name:                                                  as
Namespace:                                               vertica
Labels:                                                <none>
Annotations:                                             <none>
CreationTimestamp:                                           Tue, 12 Apr 2022 15:11:28 -0300
Reference:                                             VerticaAutoscaler/as
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):  0% (9m) / 50%
Min replicas:                                          3
Max replicas:                                          12
VerticaAutoscaler pods:                                      3 current / 3 desired
Conditions:
  Type           Status  Reason            Message
  ----           ------  ------            -------
  AbleToScale    True    ReadyForNewScale  recommended size matches current size
  ScalingActive  True    ValidMetricFound  the HPA was able to successfully calculate a replica count from cpu resource utilization (percentage of request)
  ScalingLimited  False  DesiredWithinRange  the desired count is within the acceptable range
```

When a scaling event occurs, you can view the admintools commands to scale the cluster. Use kubectl to view the StatefulSets:

```
$ kubectl get statefulsets
NAME                                         READY   AGE
db-name-as-instance-name-0                           0/3     71s
db-name-primary1                                   3/3     39m
```

Use kubectl describe to view the executing commands:

```
$ kubectl describe vdb dbname | tail
  Upgrade Status:
Events:
  Type    Reason                 Age   From               Message
  ----    ------                 ----  ----               -------
  Normal  ReviveDBStart          41m   verticadb-operator  Calling 'admintools -t revive_db'
  Normal  ReviveDBSucceeded      40m   verticadb-operator  Successfully revived database. It took 25.255683916s
  Normal  ClusterRestartStarted  40m   verticadb-operator  Calling 'admintools -t start_db' to restart the cluster
  Normal  ClusterRestartSucceeded 39m  verticadb-operator  Successfully called 'admintools -t start_db' and it took 44.713787718s
  Normal  SubclusterAdded        10s   verticadb-operator  Added new subcluster 'as-0'
  Normal  AddNodeStart           9s    verticadb-operator  Calling 'admintools -t db_add_node' for pod(s) 'db-name-as-instance-name-0-0, db-name-as-instance-name-0-1, db-name-as-instance-name-0-2'
```

EventTrigger CRD

The EventTrigger custom resource definition (CRD) runs a task when the condition of a Kubernetes object changes to a specified status. EventTrigger extends the Kubernetes Job , a workload resource that creates pods, runs a task, then cleans up the pods after the task completes.

Prerequisites

- Deploy a VerticaDB operator .
- Confirm that you have the resources to deploy objects you plan to create.

Limitations

The EventTrigger CRD has the following limitations:

- It can monitor a condition status on only one VerticaDB custom resource (CR).
- You can match only one condition status.
- The EventTrigger and the object that it watches must exist within the same namespace.

Creating an EventTrigger

An EventTrigger resource defines the Kubernetes object that you want to watch, the status condition that triggers the Job, and a pod template that contains the Job logic and provides resources to complete the Job.

This example creates a YAML-formatted file named eventtrigger.yaml . When you apply eventtrigger.yaml to your VerticaDB CR, it creates a single-column database table when the VerticaDB CR's DBInitialized condition status changes to True :

```
$ kubectl describe vdb verticadb-name
Status:
  ...
  Conditions:
    ...
    Last Transition Time:  transition-time
    Status:                True
    Type:                  DBInitialized
```

The following fields form the spec , which defines the desired state of the EventTrigger object:

- references : The Kubernetes object whose condition status you want to watch.
- matches : The condition and status that trigger the Job.
- template : Specification for the pods that run the Job after the condition status triggers an event.

The following steps create an EventTrigger CR:

1. Add the apiVersion , kind , and metadata.name required fields :

   ```
   apiVersion: vertica.com/v1beta1
   kind: EventTrigger
   metadata:
     name: eventtrigger-example
   ```

2. Begin the spec definition with the references field. The object field is an array whose values identify the VerticaDB CR object that you want to watch. You must provide the VerticaDB CR's apiVersion , kind , and name :

```
spec:
  references:
  - object:
      apiVersion: vertica.com/v1beta1
      kind: VerticaDB
      name: verticadb-example
```

3. Define the matches field that triggers the Job. EventTrigger can match only one condition:

```
spec:
  ...
  matches:
  - condition:
      type: DBInitialized
      status: "True"
```

   The preceding example defines the following:
   - condition.type : The condition that the operator watches for state change.
   - condition.status : The status that triggers the Job.
4. Add the template that defines the pod specifications that run the Job after matches.condition triggers an event.
   A pod template requires its own spec definition, and it can optionally have its own metadata. The following example includes metadata.generateName , which instructs the operator to generate a unique, random name for any pods that it creates for the Job. The trailing dash (-) separates the user-provided portion from the generated portion:

```
spec:
  ...
  template:
    metadata:
      generateName: create-user-table-
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: main
            image: "vertica/vertica-k8s:latest"
            command: ["/opt/vertica/bin/vsql", "-h", "verticadb-sample-defaultsubcluster", "-f", "CREATE TABLE T1 (C1 INT);"]
```

   The remainder of the spec defines the following:
   - restartPolicy : When to restart all containers in the pod.
   - containers : The containers that run the Job.
     - name : The name of the container.
     - image : The image that the container runs.
     - command : An array that contains a command, where each element in the array combines to form a command. The final element creates the single-column SQL table.

Apply the manifest

After you create the EventTrigger, apply the manifest in the same namespace as the VerticaDB CR:

```
$ kubectl apply -f eventtrigger.yaml

eventtrigger.vertica.com/eventtrigger-example created
configmap/create-user-table-sql created
```

After you create the database, the operator runs a Job that creates a table. You can check the status with kubectl get job :

```
$ kubectl get job
NAME                COMPLETIONS   DURATION   AGE
create-user-table   1/1           4s         7s
```

Verify that the table was created in the logs:

```
$ kubectl logs create-user-table-guid
CREATE TABLE
```

Complete file reference

```
apiVersion: vertica.com/v1beta1
kind: EventTrigger
metadata:
  name: eventtrigger-example
spec:
  references:
  - object:
      apiVersion: vertica.com/v1beta1
      kind: VerticaDB
      name: verticadb-example
  matches:
  - condition:
      type: DBInitialized
      status: "True"
  template:
    metadata:
      generateName: create-user-table-
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: main
            image: "vertica/vertica-k8s:latest"
            command: ["/opt/vertica/bin/vsql", "-h", "verticadb-sample-defaultsubcluster", "-f", "CREATE TABLE T1 (C1 INT);"]
```

Monitoring an EventTrigger

The following table describes the status fields that help you monitor an EventTrigger CR:

Status Field	Description
references[].apiVersion	Kubernetes API version of the object that the EventTrigger CR watches.
references[].kind	Type of object that the EventTrigger CR watches.
references[].name	Name of the object that the EventTrigger CR watches.
references[].namespace	Namespace of the object that the EventTrigger CR watches. The EventTrigger and the object that it watches must exist within the same namespace.
references[].uid	Generated UID of the reference object. The operator generates this identifier when it locates the reference object.
references[].resourceVersion	Current resource version of the object that the EventTrigger watches.
references[].jobNamespace	If a Job was created for the object that the EventTrigger watches, the namespace of the Job.
references[].jobName	If a Job was created for the object that the EventTrigger watches, the name of the Job.

Custom resource definition parameters

The following table describes the available settings for the Vertica Custom Resource Definitions .

VerticaDB

Parameter	Description

annotations	Custom annotations added to all of the objects that the operator creates. Each annotation is encoded as an environment variable in the Vertica server container. The following values are accepted: • Letters • Numbers • Underscores Invalid character values are converted to underscore characters. For example: vertica.com/git-ref: 1234abcd Is converted to: VERTICA_COM_GIT_REF=1234abcd **Note** Enclose integer values in double quotes (""), or the admission controller returns an error.
autoRestartVertica	Determines if the operator restarts the Vertica process when the process is not running. Set this parameter to false when performing manual maintenance that requires a DOWN database. This prevents the operator from interfering with the database state. **Default:** true
certSecrets	A list of Secrets for custom TLS certificates. Each certificate is mounted in the container at /certs/ *cert-name* / *key* . For example, a PEM-encoded CA bundle named root_cert.pem and concealed in a Secret named aws-cert is mounted in /certs/aws-cert/root_cert.pem . If you update the certificate after you add it to a custom resource, the operator updates the value automatically. If you add or delete a certificate, the operator reschedules the pod with the new configuration. For implementation details, see VerticaDB CRD .

communal.additionalConfig	Sets one or more configuration parameters in the CR:
	
	`spec:`
	`  communal:`
	`  additionalConfig:`
	`  *config-param*: "*value*"`
	`  ...`
	`...`
	
	Configuration parameters are set only when the database is initialized. After the database is initialized, changes to this parameter have no effect in the server.
	
	Important
	Configuration parameters in the CR have the following requirements and behaviors:
	
	• If you set an invalid configuration parameter, the Vertica server process does not start. For example, the server does not start if you misspell a parameter name or if the configuration parameter is not supported by the Vertica server version.
	• If communal.addtitionalConfig sets a configuration parameter that the operator sets with a CR parameter, the operator ignores the communal.addtitionalConfig setting. For example, the communal.endpoint parameter sets the AWSEndpoint S3 parameter. If you set communal.endpoint and also set AWSEndpoint with communal.addtitionalConfig , the operator enforces the communal.endpoint setting.

communal.caFile	The mount path in the container filesystem to a CA certificate file that validates HTTPS connections to a communal storage endpoint. Typically, the certificate is stored in a Secret and included in certSecrets . For details, see VerticaDB CRD .
communal.credentialSecret	The name of the Secret that stores the credentials for the communal storage endpoint. For implementation details for each supported communal storage location , see Configuring communal storage . This parameter is optional when you authenticate to an S3-compatible endpoint with an Identity and Access Management (IAM) profile.
communal.endpoint	A communal storage endpoint URL. The endpoint must begin with either the http:// or https:// protocol. For example: https://path/to/endpoint You cannot change this value after you create the custom resource instance. This setting is required when initPolicy is set to Create or Revive .

communal.s3ServerSideEncryption	Server-side encryption type used when reading from or writing to S3. The value depends on which type of encryption at rest is configured for S3. This parameter accepts the following values: • SSE-S3 • SSE-KMS : Requires that you pass the key identifier with the communal.additionalConfig parameter. • SSE-C : Requires that you pass the client key with the communal.s3SSECustomerKeySecret parameter. You cannot change this value after you create the custom resource instance. For implementation examples of all encryption types, see Configuring communal storage . For details about each encryption type, see S3 object store . **Default** : Empty string (""), no encryption
communal.s3SSECustomerKeySecret	If s3ServerSideEncryption is set to SSE-C , a Secret containing the client key for S3 access with the following requirements: • The Secret must be in the same namespace as the CR. • You must set the client key contents with the clientKey field. The client key must use one of the following formats: • 32-character plaintext • 44-character base64-encoded For additional implementation details, see Configuring communal storage .
communal.hadoopConfig	A ConfigMap that contains the contents of the /etc/hadoop directory. This is mounted in the container to configure connections to a Hadoop Distributed File System (HDFS) communal path.
communal.includeUIDInPath	When set to true , the operator includes in the path the unique identifier (UID) that Kubernetes assigns to the VerticaDB object. Including the UID creates a unique database path so that you can reuse the communal path in the same endpoint. **Default:** false
communal.kerberosRealm	The realm portion of the Vertica Kerberos principal. This value is set in the KerberosRealm database parameter during boostrapping.
communal.kerberosServiceName	The service name portion of the Vertica Kerberos principal. This value is set in the KerberosServiceName database parameter during bootstrapping.
communal.path	The path to the communal storage bucket. For example: s3://bucket-name/key-name You must create this bucket before you create the Vertica database. The following initPolicy values determine how to set this value: • Create : The path must be empty. • Revive : The path cannot be empty. You cannot change this value after you create the custom resource.

communal.region	The geographic location where the communal storage resources are located. If you do not set the correct region, the configuration fails. You might experience a delay because Vertica retries several times before failing. This setting is valid for Amazon Web Services (AWS) and Google Cloud Platform (GCP) only. Vertica ignores this setting for other communal storage providers. **Default:** • AWS: us-east-1 • GCP: US-EAST1
dbName	The database name. When initPolicy is set to Revive or ScheduleOnly , this must match the name of the source database. **Default:** vertdb
encryptSpreadComm	Sets the EncryptSpreadComm security parameter to configure Spread encryption for a new Vertica database. The VerticaDB operator ignores this parameter unless you set initPolicy to Create. This parameter accepts the following values: • vertica : Enables Spread encryption. Vertica generates the Spread encryption key for the database cluster. • Empty string (""): Clears encryption. **Default** : Empty string ("")
ignoreClusterLease	Ignore the cluster lease when executing a revive or start_db . **Default:** false **Caution** If another system is using the same communal storage, setting ignoreClusterLease to true results in data corruption.
image	The image that defines the Vertica server container's runtime environment. If the container is hosted in a private container repository, this name must include the path to the repository. When you update the image, the operator stops and restarts the cluster. **Default:** vertica/vertica-k8s:latest
imagePullPolicy	Determines how often Kubernetes pulls the image for an object. For details, see Updating Images in the Kubernetes documentation. **Default:** If the image tag is latest , the default is Always . Otherwise, the default is IfNotPresent .
imagePullSecrets	A list of Secrets that store credentials for authentication to a private container repository. For details, see Specifying imagePullSecrets in the Kubernetes documentation.

initPolicy	How to initialize the Vertica database in Kubernetes. This parameter accepts the following values: • Create : Forces the creation of a new database for the custom resource. • CreateSkipPackageInstall : Same as Create , but does not install any default packages to quickly create a database. To install default packages, see the admintools install_packages command. **Note** CreateSkipPackageInstall is available in Vertica version 12.0.1 and later. • Revive : Initializes an existing Eon Mode database as a StatefulSet with the revive command. For information about Revive , see Generating a custom resource from an existing Eon Mode database . • ScheduleOnly : Schedules a subcluster for a Hybrid Kubernetes cluster .
kerberosSecret	The Secret that stores the following values for Kerberos authentication to Hadoop Distributed File System (HDFS): • krb5.conf: Contains Kerberos configuration information. • krb5.keytab: Contains credentials for the Vertica Kerberos principal. This file must be readable by the file owner that is running the process. The default location for each of these files is the /etc directory.
kSafety	Sets the fault tolerance for the cluster. The operator supports setting this value to 0 or 1 only. For details, see K-safety . You cannot change this value after you create the custom resource. **Default:** 1
labels	Custom labels added to all of the objects that the operator creates.
licenseSecret	The Secret that contains the contents of license files. The Secret must share a namespace with the custom resource (CR). Each of the keys in the Secret is mounted as a file in /home/dbadmin/licensing/mnt . If this value is set when the CR is created, the operator installs one of the licenses automatically, choosing the first one alphabetically. If you update this value after you create the custom resource, you must manually install the Secret in each Vertica pod.

livenessProbeOverride	Overrides default livenessProbe settings that indicate whether the container is running. The VerticaDB operator sets or updates the liveness probe in the StatefulSet.
	For example, the following object overrides the default initialDelaySeconds , periodSeconds , and failureThreshold settings:
	`spec:`
	`...`
	`  livenessProbeOverride:`
	`    initialDelaySeconds: 120`
	`    periodSeconds: 15`
	`    failureThreshold: 8`
	For a detailed list of the available probe settings, see the Kubernetes documentation .

local.catalogPath	Optional parameter that sets a custom path in the container filesystem for the catalog, if your environment requires that the catalog is stored in a location separate from the local data. If initPolicy is set to Revive or ScheduleOnly , local.catalogPath for the new database must match local.catalogPath for the source database.
local.dataPath	The path in the container filesystem for the local data. If local.catalogPath is not set, the catalog is stored in this location. If initPolicy is set to Revive or ScheduleOnly , the dataPath for the new database must match the dataPath for the source database. **Default:** /data
local.depotPath	The path in the container filesystem that stores the depot. If initPolicy is set to Revive or ScheduleOnly , the depotPath for the new database must match the depotPath for the source database. **Default:** /depot
local.depotVolume	The type of volume to use for the depot. This parameter accepts the following values: • PersistentVolume : A PersistentVolume is used to store the depot data. This volume type persists depot data between pod lifecycles. • EmptyDir : A volume of type emptyDir is used to store the depot data. When the pod is removed from a node, the contents of the volume are deleted. If a container crashes, the depot data is unaffected. Important You cannot change the depot volume type on an existing database. If you want to change this setting, you must create a new custom resource. For details about each volume type, see the Kubernetes documentation . **Default** : PersistentVolume
local.requestSize	The minimum size of the local data volume when selecting a PersistentVolume (PV). If local.storageClass allows volume expansion, the operator automatically increases the size of the PV when you change this setting. It expands the size of the depot if the following conditions are met: • local.storageClass is set to PersistentVolume . • Depot storage is allocated using a percentage of the total disk space rather than a unit, such as a gigabyte. If you decrease this value, the operator does not decrease the size of the PV or the depot. **Default:** 500 Gi
local.storageClass	The StorageClass for the PersistentVolumes that persist local data between pod lifecycles. Select this value when defining the persistent volume claim (PVC). By default, this parameter is not set. The PVC in the default configuration uses the default storage class set by Kubernetes.

podSecurityContext	Overrides any pod-level security context. This setting is merged with the default context for the pods in the cluster. For details about the available settings for this parameter, see the Kubernetes documentation .

readinessProbeOverride	Overrides default readinessProbe settings that indicate whether the Vertica pod is ready to accept traffic. The VerticaDB operator sets or updates the readiness probe in the StatefulSet.
	
	For example, the following object overrides the default timeoutSeconds and periodSeconds settings:
	
	`spec:`
	`...`
	`  readinessProbeOverride:`
	`    initialDelaySeconds: 0`
	`    periodSeconds: 10`
	`    failureThreshold: 3`
	
	For a detailed list of the available probe settings, see the Kubernetes documentation .

reviveOrder	The order of nodes during a revive operation. Each entry contains the subcluster index, and the number of pods to include from the subcluster.
	
	For example, consider a database with the following setup:
	
	`- v_db_node0001: subcluster A`
	`- v_db_node0002: subcluster A`
	`- v_db_node0003: subcluster B`
	`- v_db_node0004: subcluster A`
	`- v_db_node0005: subcluster B`
	`- v_db_node0006: subcluster B`
	
	If the subclusters[] list is defined as {'A', 'B'}, the revive order is as follows:
	
	`- {subclusterIndex:0, podCount:2} # 2 pods from subcluster A`
	`- {subclusterIndex:1, podCount:1} # 1 pod from subcluster B`
	`- {subclusterIndex:0, podCount:1} # 1 pod from subcluster A`
	`- {subclusterIndex:1, podCount:2} # 2 pods from subcluster B`
	
	This parameter is used only when initPolicy is set to Revive .

restartTimeout	When restarting pods, the number of seconds before admintools times out. **Default:** 0. The operator uses the 20 minutes default used by admintools.

securityContext	Sets any additional security context for the Vertica server container. This setting is merged with the security context value set for the VerticaDB Operator.
	
	For example, if you need a core file for the Vertica server process, you can set the privileged property to true to elevate the server privileges on the host node:
	
	`spec:`
	`  ...`
	`  securityContext:`
	`    privileged: true`
	
	For additional information about generating a core file, see Metrics gathering . For details about this parameter, see the Kubernetes documentation .

shardCount	The number of shards in the database. You cannot update this value after you create the custom resource. For more information about database shards and Eon Mode, see Configuring your Vertica cluster for Eon Mode .

sidecars[]	One or more optional utility containers that complete tasks for the Vertica server container. Each sidecar entry is a fully-formed container spec, similar to the container that you add to a Pod spec.
	
	The following example adds a sidecar named vlogger to the custom resource:
	
	`spec:`
	`  ...`
	`  sidecars:`
	`  - name: vlogger`
	`  image: vertica/vertica-logger:1.0.0`
	`  volumeMounts:`
	`  - name: my-custom-vol`
	`  mountPath: /path/to/custom-volume`
	
	volumeMounts.name is the name of a custom volume. This value must match volumes.name to mount the custom volume in the sidecar container filesystem. See volumes for additional details.
	
	For implementation details, see VerticaDB CRD .

sidecars[i].volumeMounts	List of custom volumes and mount paths that persist sidecar container data. Each volume element requires a name value and a mountPath . To mount a volume in the Vertica sidecar container filesystem, volumeMounts.name must match the volumes.name value for the corresponding sidecar definition, or the webhook returns an error. For implementation details, see VerticaDB CRD .
sshSecret	A Secret that contains SSH credentials that authenticate connections to a Vertica server container. Example use cases include the following: • Authenticate communication between an Eon Mode database and custom resource in a hybrid architecture. • For environments that run the Vertica Kubernetes (No keys) image, pass the custom resource user-provided SSH keys for internal communication between Vertica pods. The Secret requires the following values: • id_rsa • id_rsa.pub • authorized_keys For details, see Hybrid Kubernetes clusters .

startupProbeOverride	Overrides the default startupProbe settings that indicate whether the Vertica process is started in the container. The VerticaDB operator sets or updates the startup probe in the StatefulSet.
	
	For example, the following object overrides the default initialDelaySeconds , periodSeconds , and failureThreshold settings:
	
	`spec:`
	`...`
	`  startupProbeOverride:`
	`    initialDelaySeconds: 30`
	`    periodSeconds: 10`
	`    failureThreshold: 117`
	`    timeoutSeconds: 5`
	
	For a detailed list of the available probe settings, see the Kubernetes documentation .

subclusters[i].affinity	Applies rules that constrain the Vertica server pod to specific nodes. It is more expressive than nodeSelector . If this parameter is not set, then the pods use no affinity setting. In production settings, it is a best practice to configure affinity to run one server pod per host node. For configuration details, see VerticaDB CRD .
subclusters[i].externalIPs	Enables the service object to attach to a specified external IP . If not set, the external IP is empty in the service object.
subclusters[i].isPrimary	Indicates whether the subcluster is primary or secondary. Each database must have at least one primary subcluster. **Default:** true
subclusters[i].loadBalancerIP	When subcluster[i].serviceType is set to LoadBalancer , assigns a static IP to the load balancing service. **Default:** Empty string ("")
subclusters[i].name	The subcluster name. This is a required setting. If you change the name of an existing subcluster, the operator deletes the old subcluster and creates a new one with the new name. Kubernetes derives names for the subcluster Statefulset, service object, and pod from the subcluster name. For additional details about Kubernetes and subcluster naming conventions, see Subclusters on Kubernetes .
subclusters[i].nodePort	When subclusters[i].serviceType is set to NodePort , this parameter enables you to define the port that is opened at each node for external client connections. The port must be within the defined range allocated by the control plane (ports 30000-32767). If you do not manually define a port number, Kubernetes chooses the port automatically.

subclusters[i].nodeSelector	Provides control over which nodes are used to schedule each pod. If this is not set, the node selector is left off the pod when it is created. To set this parameter, provide a list of key/value pairs.
	
	The following example schedules server pods only at nodes that have the disktype=ssd and region=us-east labels:
	
	`subclusters:`
	`  - name: defaultsubcluster`
	`    nodeSelector:`
	`      disktype: ssd`
	`      region: us-east`

subclusters[i].priorityClassName	The PriorityClass name assigned to pods in the StatefulSet. This affects where the pod gets scheduled.
subclusters[i].resources.limits	The resource limits for pods in the StatefulSet, which sets the maximum amount of CPU and memory that each server pod can consume. Vertica recommends that you set these values equal to subclusters[i].resources.requests to ensure that the pods are assigned to the guaranteed QoS class . This reduces the possibility that pods are chosen by the out of memory (OOM) Killer. For more information, see Recommendations for Sizing Vertica Nodes and Clusters in the Vertica Knowledge Base.
subclusters[i].resources.requests	The resource requests for pods in the StatefulSet, which sets the maximum amount of CPU and memory that each server pod can consume. Vertica recommends that you set these values equal to subclusters[i].resources.limits to ensure that the pods are assigned to the guaranteed QoS class . This reduces the possibility that pods are chosen by the out of memory (OOM) Killer. For more information, see Recommendations for Sizing Vertica Nodes and Clusters in the Vertica Knowledge Base.
subclusters[i].serviceAnnotations	Custom annotations added to implementation-specific services. Managed Kubernetes use service annotations to configure services such as network load balancers, virtual private cloud (VPC) subnets, and loggers.

subclusters[i].serviceName	Identifies the service object that directs client traffic to the subcluster. Assign a single service object to multiple subclusters to process client data with one or more subclusters. For example:
	
	`spec:`
	`  ...`
	`  subclusters:`
	`  - name: subcluster-1`
	`  size: 3`
	`  serviceName: connections`
	`  - name: subcluster-2`
	`  size: 3`
	`  serviceName: connections`
	
	The previous example creates a service object named *metadata.name* -connections that load balances client traffic among its assigned subclusters.
	
	For implementation details, see VerticaDB CRD .

subclusters[i].serviceType	Identifies the type of Kubernetes service to use for external client connectivity. The default is type is ClusterIP, which sets a stable IP and port that is accessible only from within Kubernetes itself. Depending on the service type, you might need to set nodePort or externalIPs in addition to this configuration parameter. **Default:** ClusterIP
subclusters[i].size	The number of pods in the subcluster. This determines the number of Vertica nodes in the subcluster. Changing this number deletes or schedules new pods. The minimum size of a subcluster is 1. The subclusters kSafety setting determines the minimum and maximum size of the cluster. **Note** By default, the Vertica container uses the Vertica community edition (CE) license. The CE license limits subclusters to 3 Vertica nodes and a maximum of 1TB of data. Use the licenseSecret parameter to add your Vertica license. For instructions about how to create the license Secret, see VerticaDB CRD .
subclusters[i].tolerations	Any taints and tolerations used to influence where a pod is scheduled.

superuserPasswordSecret	The Secret that contains the database superuser password. Create this Secret before deployment.
	
	If you do not create this Secret before deployment, there is no password authentication for the database.
	
	The Secret must use a key named password :
	
	kubectl create secret generic su-passwd --from-literal=password= *secret-password*
	
	The following text adds this Secret to the custom resource:
	
	`db:`
	`  superuserSecretPassword: su-passwd`

temporarySubclusterRouting.names	Specifies an existing subcluster that accepts traffic during an online upgrade. The operator routes traffic to the first subcluster that is online. For example:
	
	`spec:`
	`  ...`
	`  temporarySubclusterRouting:`
	`  names:`
	`  - subcluster-2`
	`  - subcluster-1`
	
	In the previous example, the operator selects subcluster-2 during the upgrade, and then routes traffic to subcluster-1 when subcluster-2 is down. As a best practice, use secondary subclusters when rerouting traffic.
	
	Note
	By default, the operator selects an existing subcluster to receive rerouted client traffic even if you do not specify a subcluster with this parameter.

temporarySubclusterRouting.template	Instructs the operator create a new secondary subcluster during an Online upgrade. The operator creates the subcluster when the upgrade begins and deletes it when the upgrade completes.
	
	To define a temporary subcluster, provide a name and size value. For example:
	
	`spec:`
	`  ...`
	`  temporarySubclusterRouting:`
	`  template:`
	`  name: transient`
	`  size: 1`

upgradePolicy	Determines how the operator upgrades Vertica server versions. Accepts the following values: • Offline: The operator stops the cluster to prevent multiple versions from running simultaneously. • Online: The cluster continues to operator during a rolling update. The data is in read-only mode while the operator upgrades the image for the primary subcluster. The Online setting has the following restrictions: ◦ The cluster must currently run Vertica server version 11.1.0 or higher. ◦ If you have only one subcluster, you must configure temporarySubclusterRouting.template to create a new secondary subcluster during the Online upgrade. Otherwise, the operator performs an Offline upgrade, regardless of the setting. • Auto: The operator selects either Offline or Online depending on the configuration. The operator selects Online if all of the following are true: ◦ A license Secret exists. ◦ K-Safety is 1. ◦ The cluster is currently running Vertica version 11.1.0 or higher. **Default:** Auto
upgradeRequeueTime	During an online upgrade, the number of seconds that the operator waits to complete work for any resource that was requeued during the reconciliation loop. **Default:** 30 seconds
volumeMounts	List of custom volumes and mount paths that persist Vertica server container data. Each volume element requires a name value and a mountPath . To mount a volume in the Vertica server container filesystem, volumeMounts.name must match the volumes.name value defined in the spec definition, or the webhook returns an error. For implementation details, see VerticaDB CRD .
volumes	List of custom volumes that persist Vertica server container data. Each volume element requires a name value and a volume type. volumes accepts any Kubernetes volume type. To mount a volume in a filesystem, volumes.name must match the volumeMounts.name value for the corresponding volume mount, or the webhook returns an error. For implementation details, see VerticaDB CRD .

VerticaAutoScaler

Parameter	Description
verticaDBName	Required. Name of the VerticaDB CR that the VerticaAutoscaler CR scales resources for.

scalingGranularity	Required. The scaling strategy. This parameter accepts one of the following values: • Subcluster: Create or delete entire subclusters. To create a new subcluster, the operator uses a template or an existing subcluster with the same serviceName. • Pod: Increase or decrease the size of an existing subcluster. **Default** : Subcluster
serviceName	Required. Refers to the subclusters[i].serviceName for the VerticaDB CR. VerticaAutoscaler uses this value as a selector when scaling subclusters together.

template	When scalingGranularity is set to Subcluster, you can use this parameter to define how VerticaAutoscaler scales the new subcluster. The following is an example:
	`spec:`
	`  verticaDBName: dbname`
	`  scalingGranularity: Subcluster`
	`  serviceName: service-name`
	`  template:`
	`  name: autoscaler-name`
	`  size: 2`
	`  serviceName: service-name`
	`  isPrimary: false`
	If you set template.size to 0, VerticaAutoscaler selects as a template an existing subcluster that uses *service-name* .
	This setting is ignored when scalingGranularity is set to Pod.

EventTrigger

Parameter	Description
matches[].condition.status	The status portion of the status condition match. The operator watches the condition specified by matches[].condition.type on the EventTrigger reference object. When that condition changes to the status specified in this parameter, the operator runs the task defined in the EventTrigger.
matches[].condition.type	The condition portion of the status condition match. The operator watches this condition on the EventTrigger reference object. When this condition changes to the status specified with matches[].condition.status , the operator runs the task defined in the EventTrigger.
references[].object.apiVersion	Kubernetes API version of the object that the EventTrigger watches.
references[].object.kind	The type of object that the EventTrigger watches.
references[].object.name	The name of the object that the EventTrigger watches.
references[].object.namespace	Optional. The namespace of the object that the EventTrigger watches. The object and the EventTrigger CR must exist within the same namespace. If omitted, the operator uses the same namespace as the EventTrigger.
template	Full spec for the Job that EventTrigger runs when references[].condition.type and references[].condition.status are found for a reference object. For implementation details, see EventTrigger CRD .

Subclusters on Kubernetes

Eon Mode uses subclusters for workload isolation and scaling. The Vertica operator provides tools to direct external client communications to specific subclusters, and automate scaling without stopping your database.

The custom resource definition (CRD) provides parameters that allow you to fine-tune each subcluster for specific workloads. For example, you can increase the subcluster size setting for increased throughput, or adjust the resource requests and limits to manage compute power. When you create a custom resource instance, the operator deploys each subcluster as a StatefulSet . Each StatefulSet has a service object, which allows an external client to connect to a specific subcluster.

Naming conventions

Kubernetes derives names for the subcluster Statefulset, service object, and pod from the subcluster name. This naming convention tightly couples the subcluster objects to help Kubernetes manage the cluster effectively. If you want to rename a subcluster, you must delete it from the CRD and redefine it so that the operator can create new objects with a derived name.

Kubernetes forms an object's fully qualified domain name (FQDN) with its resource type name, so resource type names must follow FQDN naming conventions . The underscore character ("_") does not follow FQDN rules, but you can use it in the subcluster name. Vertica converts each underscore to a hyphen (" - ") in the FQDN for any object name derived from the subcluster name. For example, Vertica generates a default subcluster and names it default_subcluster , and then converts the corresponding portion of the derived object's FQDN to default-subcluster .

For additional naming guidelines, see the Kubernetes documentation .

External client connections

External clients can target specific subclusters that are fine-tuned to handle their workload. Each subcluster has a service object that handles external connections. To target multiple subclusters with a single service object, assign each subcluster the same spec.subclusters.serviceName value in the custom resource (CR). For implementation details, see VerticaDB CRD .

The operator performs health monitoring that checks if the Vertica daemon is running on each pod. If it is, then the operator allows the service object to route traffic to the pod.

By default, the service object derives its name from the custom resource name and the associated subcluster and uses the *customResourceName-subclusterName* format. Use the subclusters[i].serviceName CR parameter to override the default naming format and use the *metadata.name-serviceName* format.

Vertica supports the following service object types:

- **ClusterIP** : The default service type. This service provides internal load balancing, and sets a stable IP and port that is accessible from within the subcluster only.
- **NodePort** : Provides external client access. You can specify a port number for each host node in the subcluster to open for client connections.
- **LoadBalancer** : Uses a cloud provider load balancer to create NodePort and ClusterIP services as needed. For details about implementation, see the Kubernetes documentation and your cloud provider documentation.

For configuration details, see VerticaDB CRD .

Managing internal and external workloads

The Vertica StatefulSet is associated with an external service object. All external client requests are sent through this service object and load balanced among the pods in the cluster.

Import and export

Importing and exporting data between a cluster outside of Kubernetes requires that you expose the service with the NodePort or LoadBalancer service type and properly configure the network.

Important
When importing or exporting data, each node must have a static IP address. Rescheduled pods might be on different host nodes, so you must monitor and update the static IP addresses to reflect the new node.

For more information, see Configuring the Network to Import and Export Data.

In this section
- Scaling subclusters

Scaling subclusters

The operator enables you to scale the number of subclusters, and the number of pods per subcluster automatically. This allows you to utilize or conserve resources depending on the immediate needs of your workload.

The following sections explain how to scale resources for new workloads. For details about scaling resources for existing workloads, see VerticaAutoscaler CRD .

Prerequisites

- Complete Installing the Vertica DB operator .
- Install the kubectl command line tool.
- Complete VerticaDB CRD .
- Confirm that you have the resources to scale.

> **Note**
> By default, the custom resource uses the free Community Edition (CE) license. This license allows you to deploy up to three nodes with a maximum of 1TB of data. To add resources beyond these limits, you must add your Vertica license to the custom resource as described in VerticaDB CRD .

Scaling the number of subclusters

Adjust the number of subclusters in your custom resource to fine-tune resources for short-running dashboard queries. For example, increase the number of subclusters to increase throughput. For more information, see Improving query throughput using subclusters .

1. Use kubectl edit to open your default text editor and update the YAML file for the specified custom resource. The following command opens a custom resource named vdb for editing:
   ```
   $ kubectl edit vdb
   ```
2. In the spec section of the custom resource, locate the subclusters subsection. Begin the isPrimary field to define a new subcluster. The isPrimary field accepts a boolean that specifies whether the subcluster is a primary or secondary. Because there is already a primary subcluster in our custom resource, enter false :
   ```
   spec:
   ...
    subclusters:
    ...
    - isPrimary: false
   ```
3. Follow the steps in VerticaDB CRD to complete the subcluster definition. The following completed example adds a secondary subcluster for dashboard queries:
   ```
   spec:
   ...
    subclusters:
    - isPrimary: true
      name: primary-subcluster
    ...
    - isPrimary: false
      name: dashboard
      nodePort: 32001
      resources:
        limits:
          cpu: 32
          memory: 96Gi
        requests:
          cpu: 32
          memory: 96Gi
      serviceType: NodePort
      size: 3
   ```
4. Save and close the custom resource file. You receive a message similar to the following when you successfully update the file: verticadb.vertica.com/vertica-db edited
5. Use the kubectl wait command to monitor when the new pods are ready:

```
$ kubectl wait --for=condition=Ready pod --selector app.kubernetes.io/name=verticadb --timeout 180s
pod/vdb-dashboard-0 condition met
pod/vdb-dashboard-1 condition met
pod/vdb-dashboard-2 condition met
```

Scaling the pods in a subcluster

For long-running, analytic queries, increase the pod count for a subcluster. See Using elastic crunch scaling to improve query performance .

1. Use kubectl edit to open your default text editor and update the YAML file for the specified custom resource. The following command opens a custom resource named vdb for editing:
   ```
   $ kubectl edit verticadb
   ```
2. Update the subclusters.size value to 6:
   ```
   spec:
   ...
    subclusters:
    ...
    - isPrimary: false
      ...
      size: 6
   ```
   Shards are rebalanced automatically.
3. Save and close the custom resource file. You receive a message similar to the following when you successfully update the file: verticadb.vertica.com/verticadb edited
4. Use the kubectl wait command to monitor when the new pods are ready:
   ```
   $ kubectl wait --for=condition=Ready pod --selector app.kubernetes.io/name=verticadb --timeout 180s
   pod/vdb-subcluster1-3 condition met
   pod/vdb-subcluster1-4 condition met
   pod/vdb-subcluster1-5 condition met
   ```

Removing a subcluster

Remove a subcluster when it is no longer needed, or to preserve resources.

Important
Because each custom resource instance requires a primary subcluster, you cannot remove all subclusters.

1. Use kubectl edit to open your default text editor and update the YAML file for the specified custom resource. The following command opens a custom resource named vdb for editing:
   ```
   $ kubectl edit verticadb
   ```
2. In the subclusters subsection nested under spec , locate the subcluster that you want to delete. Delete the element in the subcluster array represents the subcluster that you want to delete. Each element is identified by a hyphen (-).
3. After you delete the subcluster and save, you receive a message similar to the following:
   ```
   verticadb.vertica.com/verticadb edited
   ```

Upgrading Vertica on Kubernetes

The operator automates Vertica server version upgrades for a custom resource (CR). Use the upgradePolicy setting in the CR to determine whether your cluster remains online or is taken offline during the version upgrade.

> **Note**
> Vertica recommends using incremental upgrade paths . The operator validates the Vertica version before proceeding with the upgrade.

Prerequisites

Before you begin, complete the following:

- Installing the Vertica DB operator .
- VerticaDB CRD .

Setting the policy

The upgradePolicy CR parameter setting determines how the operator upgrades Vertica server versions. It provides the following options:

Setting	Description
Offline	The operator shuts down the cluster to prevent multiple versions from running simultaneously. The operator performs all server version upgrades using the Offline setting in the following circumstances: • You have only one subcluster • You are upgrading from a Vertica server version prior to version 11.1.0
Online	The cluster continues to operate during an online upgrade. The data is in read-only mode while the operator upgrades the image for the primary subcluster.
Auto	The default setting. The operator selects either Offline or Online depending on the configuration. The operator performs an Online upgrade if all of the following are true: • A license Secret exists • K-Safety is 1 • The cluster is currently running a Vertica version 11.1.0 or higher If the current configuration does not meet all of the previous requirements, the operator performs an Offline upgrade.

Set the reconcile loop iteration time

During an upgrade, the operator runs the reconcile loop to compare the actual state of the objects to the desired state defined in the CR. The operator requeues any unfinished work, and the reconcile loop compares states with a set period of time between each reconcile iteration. Set the upgradeRequeueTime parameter to determine the amount of time between each reconcile loop iteration.

Routing client traffic during an online upgrade

During an online upgrade, the operator begins by upgrading the Vertica server version in the primary subcluster to form a cluster with the new version. When the operator restarts the primary nodes, it places the secondary subclusters in read-only mode. Next, the operator upgrades any secondary subclusters one at a time. During the upgrade for any subcluster, all client connections are drained, and traffic is rerouted to either an existing subcluster or a temporary subcluster.

Online upgrades require more than one subcluster so that the operator can reroute client traffic for the subcluster while it is upgrading. By default, the operator selects which subcluster receives the rerouted traffic using the following rules:

- When rerouting traffic for the primary subcluster, the operator selects the first secondary subcluster defined in the CR.
- When restarting the first secondary subcluster after the upgrade, the operator selects the first subcluster that is defined in the CR that is up.
- If no secondary subclusters exist, you cannot perform an online upgrade. The operator selects the first primary subcluster defined in the CR and performs an offline upgrade.

Routing client traffic to an existing subcluster

You might want to control which subclusters handle rerouted client traffic due to subcluster capacity or licensing limitations. You can set the temporarySubclusterRouting.names parameter to specify an existing subcluster to receive the rerouted traffic:

```
spec:
  ...
  temporarySubclusterRouting:
    names:
      - subcluster-2
      - subcluster-1
```

In the previous example, subcluster-2 accepts traffic when the other subcluster-1 is offline. When subcluster-2 is down, subcluster-1 accepts its traffic.

Routing client traffic to a temporary subcluster

To create a temporary subcluster that exists for the duration of the upgrade process, use the temporarySubclusterRouting.template parameter to provide a name and size for the temporary subcluster:

```
spec:
  ...
  temporarySubclusterRouting:
    template:
      name: transient
      size: 3
```

If you choose to upgrade with a temporary subcluster, ensure that you have the necessary resources.

Upgrading the Vertica server version

After you set the upgradePolicy and optionally configure temporary subcluster routing, use the kubectl command line tool to perform the upgrade and monitor its progress.

> **Note**
> Online upgrades require that you upgrade from Vertica server image for 11.1.0 and higher.

The following steps perform an online version upgrade:

1. Set the upgrade policy. The following command uses the kubectl patch command to set the upgradePolicy value to Online:
   ```
   $ kubectl patch verticadb cluster-name --type=merge --patch '{"spec": {"upgradePolicy": "Online"}}'
   ```
2. Update the image value in the CR with kubectl patch :
   ```
   $ kubectl patch verticadb cluster-name --type=merge --patch '{"spec": {"image": "vertica/vertica-k8s:new-version"}}'
   ```
3. Use kubectl wait to wait until the operator acknowledges the new image and begins upgrade mode:
   ```
   $ kubectl wait --for=condition=ImageChangeInProgress=True vdb/cluster-name --timeout=180s
   ```
4. Use kubectl wait to wait until the operator leaves upgrade mode:
   ```
   $ kubectl wait --for=condition=ImageChangeInProgress=False vdb/cluster-name --timeout=800s
   ```

Viewing the upgrade process

To view the current phase of the upgrade process, use kubectl get to inspect the upgradeStatus status field:

```
$ kubectl get vdb -n namespacedatabase-name -o jsonpath='{.status.upgradeStatus}{"\n"}'
Restarting cluster with new image
```

To view the entire upgrade process, use kubectl describe to list the events the operator generated during the upgrade:

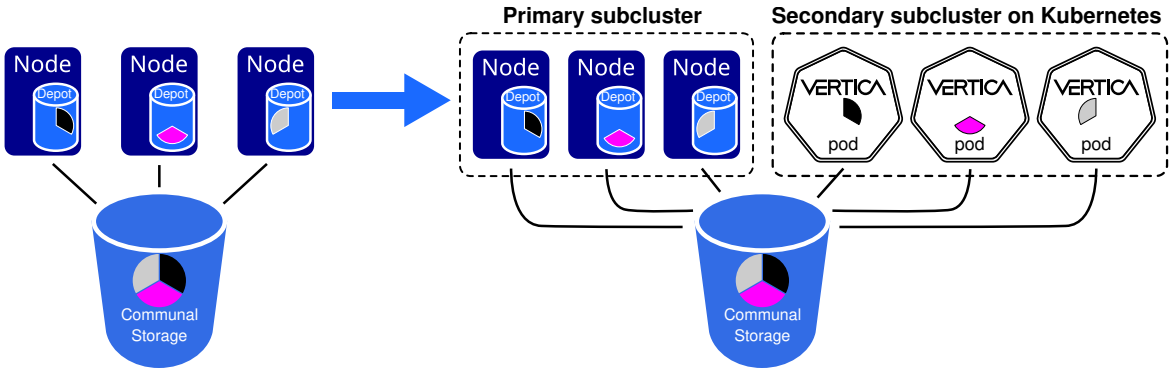
```
$ kubectl describe vdb cluster-name

...
Events:
  Type    Reason                  Age    From                Message
  ----    ------                  ----   ----                -------
  Normal  UpgradeStart            5m10s  verticadb-operator  Vertica server upgrade has started.  New image is 'vertica-k8s:new-version'
  Normal  ClusterShutdownStarted  5m12s  verticadb-operator  Calling 'admintools -t stop_db'
  Normal  ClusterShutdownSucceeded 4m08s  verticadb-operator  Successfully called 'admintools -t stop_db' and it took 56.22132s
  Normal  ClusterRestartStarted   4m25s  verticadb-operator  Calling 'admintools -t start_db' to restart the cluster
  Normal  ClusterRestartSucceeded  25s    verticadb-operator  Successfully called 'admintools -t start_db' and it took 240s
  Normal  UpgradeSucceeded        5s     verticadb-operator  Vertica server upgrade has completed successfully
```

Hybrid Kubernetes clusters

An Eon Mode database can run hosts separate from the database and within Kubernetes. This architecture is useful in the following scenarios:

- Leveraging Kubernetes tooling to quickly create a secondary subcluster for a database.
- Creating an isolated sandbox environment to run ad hoc queries on a communal dataset.
- Experimenting with the Vertica on Kubernetes performance overhead without migrating your primary subcluster into Kubernetes.

Define the Kubernetes portion of a hybrid architecture with a custom resource (CR) . The custom resource has no knowledge of Vertica hosts that exist separately from the custom resource. This limits the operator's functionality and requires that you manually complete some tasks that the operator automates for a standard Vertica on Kubernetes custom resource.



Requirements and restrictions

The hybrid Kubernetes architecture has the following requirements and restrictions:

- Hybrid Kubernetes clusters require a tool that enables Border Gateway Protocol (BGP) so that pods are accessible to your on-premises subcluster for external communication. For example, you can use the Calico CNI plugin to enable BGP .
- You cannot use network address translation (NAT) between the Kubernetes pods and the on-premises cluster.

Operator limitations

In a hybrid architecture, the operator has no visibility outside of the custom resource. This limited visibility means that the operator cannot interact with the Eon Mode database or the primary subcluster. Within the scope of the custom resource, the operator automates only the following:

- Schedules pods based on the manifest.
- Creates service objects for the subcluster.
- Creates a PersistentVolumeClaim (PVC) that persists data for each pod.
- Executes the restart_node administration tool command if the Vertica server process is not running. To override this default behavior, set the autoRestartVertica custom resource parameter to false .

Defining a hybrid cluster

To define a hybrid cluster, you must set up SSH communications between the Eon Mode nodes and containers, and then define the hybrid CR.

SSH between environments

In an Eon Mode database, nodes communicate through SSH. Vertica containers use SSH with a static key. Because the CR has no knowledge of any of the Eon Mode hosts, you must make the containers aware of the Eon Mode SSH keys.

You can create a Secret for the CR that stores SSH credentials for both the Eon Mode database and the Vertica container. The Secret must contain the following:

- id_rsa: private key shared among the pods.
- id_rsa.pub: public key shared among the pods.
- authorized_keys: file that contains the following keys:
  - id_rsa.pub for pod-to-pod traffic.
  - public key of on-premises root account.
  - public key of on-prem dbadmin account.

The following command creates a Secret named ssh-key that stores these SSH credentials. The Secret persists between life cycles to allow secure connections between the on-premises nodes and the CR:

```
$ kubectl create secret generic ssh-keys --from-file=$HOME/.ssh
```

Hybrid CR definition

Create a custom resource to define a subcluster that runs outside your standard Eon Mode database:

```
apiVersion: vertica.com/v1beta1
kind: VerticaDB
metadata:
  name: hybrid-secondary-sc
spec:
  image: vertica/vertica-k8s:latest
  initPolicy: ScheduleOnly
  sshSecret: ssh-keys
  local:
    dataPath: /data
    depotPath: /depot
  dbName: vertdb
  subclusters:
    - name: sc1
      size: 3
    - name: sc2
      size: 3
```

In the previous example:

- initPolicy : Hybrid clusters require that you set this to ScheduleOnly .
- sshSecret : The Secret that contains SSH keys that authenticate connections to Vertica hosts outside the CR.
- local : Required. The values persist data to the PersistentVolume (PV). These values must match the directory locations in the Eon Mode database that is associated with the Kubernetes pods.
- dbName : This value must match the name of the standard Eon Mode database that is associated with this subcluster.
- subclusters : Definition for each subcluster.

> **Note**
> Hybrid custom resources ignore configuration parameters that control settings outside the scope of the hybrid subcluster, such as the communal.* and the subclusters[i].isPrimary parameters.

For complete implementation details, see VerticaDB CRD . For details about each setting, see Custom resource definition parameters .

Maintaining quorum

If quorum is lost, you must manually restart the cluster with admintools :

```
$ /opt/vertica/bin/admintools -t restart_db --database database-name;
```

For details about maintaining quorum, see Data integrity and high availability in an Eon Mode database .

Scaling the Kubernetes subcluster

When you scale a hybrid cluster, you add nodes from the primary subcluster to the secondary subcluster on Kubernetes.

HDFS with Kerberos authentication

If you are scaling a cluster that authenticates Hadoop file storage (HDFS) data with Kerberos, you must alter the database configuration before you scale.

In the default configuration, the Vertica server process running in the Kubernetes pods cannot access the HDFS data due to incorrect permissions on the keytab file mounted in the pod. This requires that you set the KerberosEnableKeytabPermissionCheck Kerberos parameter :

1. Set the KerberosEnableKeytabPermissionCheck configuration parameter to 0 :
   ```
   => ALTER DATABASE DEFAULT SET KerberosEnableKeytabPermissionCheck = 0;
   WARNING 4324:  Parameter KerberosEnableKeytabPermissionCheck will not take effect until database restart
   ALTER DATABASE
   ```
2. Restart the cluster with admintools so that the new setting takes effect:
   ```
   $ /opt/vertica/bin/admintools -t restart_db --database database-name;
   ```

For additional details about Vertica on Kubernetes and HDFS, see Configuring communal storage .

Scale the subcluster

When you add nodes from the primary subcluster to the secondary subcluster on Kubernetes, you must set up the configuration directory for the new nodes and change operator behavior during the scaling event:

1. Execute the update_vertica script to set up the configuration directory. Vertica on Kubernetes requires the following configuration options for update_vertica :

```
$ /opt/vertica/sbin/update_vertica \
    --accept-eula \
    --add-hosts host-list \
    --dba-user-password dba-user-password \
    --failure-threshold NONE \
    --no-system-configuration \
    --point-to-point \
    --data-dir /data-dir \
    --dba-user dbadmin \
    --no-package-checks \
    --no-ssh-key-install
```

2. Set autoRestartVertica to false so that the operator does not interfere with the scaling operation:

```
$ kubectl patch vdb database-name --type=merge --patch='{"spec": {"autoRestartVertica": false}}'
```

3. Add the new nodes with the admintools db_add_node option:

```
$ /opt/vertica/bin/admintools \
 -t db_add_node \
 --hosts host-list \
 --database database-name\
 --subcluster sc-name \
 --noprompt
```

   For details, see Adding and removing nodes from subclusters .
4. After the scaling operation, set autoRestartVertica back to true :

```
$ kubectl patch vdb database-name --type=merge --patch='{"spec": {"autoRestartVertica": true}}'
```

Generating a custom resource from an existing Eon Mode database

To simplify Vertica on Kubernetes adoption, Vertica provides the vdb-gen migration tool that revives an existing Eon Mode database as a StatefulSet in Kubernetes. vdb-gen generates a custom resource (CR) from an existing Eon Mode database by connecting to the database and writing to standard output.

The vdb-gen tool is available for download as a release artifact in the vertica-kubernetes GitHub repository.

Use the -h flag to view a full list of the available vdb-gen options, including options for debugging and working with environment variables. The following steps generate a CR using basic commands:

1. Execute vdb-gen and redirect the output to a YAML-formatted file:

```
$ vdb-gen --password secret --name mydb 10.20.30.40 vertdb > vdb.yaml
```

   The previous command uses the following flags and values:
   - password: The existing database superuser secret password.
   - name: The name of the new custom resource object.
   - 10.20.30.40: The IP address of the existing database
   - vertdb: The name of the existing Eon Mode database.
   - vdb.yaml: The YAML formatted file that contains the custom resource definition generated by the vdb-gen tool.
2. Use the admintools stop_db command to stop the existing database:

```
$ /opt/vertica/bin/admintools -t stop_db -d vertdb
```

   Wait for the cluster lease to expire before continuing. For details, see Reviving an Eon Mode database cluster .
3. Apply the YAML-formatted manifest that was generated by the vdb-gen tool:

```
$ kubectl apply -f vdb.yaml
verticadb.vertica.com/mydb created
```

> **Note**
> For performance purposes, do not apply the manifest to resources that already contain a Vertica on Kubernetes install.

4. The operator creates the StatefulSet, installs Vertica on each pod, and runs revive . To view the events generated for the new database, use kubectl describe :

```
$ kubectl describe vdb mydb
```

Backup and restore containerized Vertica

In Vertica on Kubernetes, backup and restore operations use the same components and tooling as non-containerized environments, including the following:

- Configuration file that specifies environment details and custom options
- Backup location that was prepared by an init script
- vbr utility

Containerized backup and restore operations require that you make these components and tools available to the Vertica server process running within a pod. The following sections describe strategies that back up and restore your VerticaDB custom resource (CR) with Kubernetes objects.

For comprehensive backup and restore documentation, see Backing up and restoring the database .

Prerequisites

- Complete Installing the Vertica DB operator
- Create a VerticaDB CR object
- The kubectl command line tool
- Review Setting up backup locations and complete necessary steps

Sample configuration file

The vbr configuration file defines parameters that the vbr utility uses to execute backup and restore tasks. For details, see the following:

- vbr configuration file reference
- Sample vbr configuration files

To define a vbr configuration file in Kubernetes, you can create a ConfigMap whose data field defines vbr configuration values. After you create the ConfigMap object in your Kubernetes environment, you can run vbr commands from within a pod that has access to the ConfigMap.

The following backup-configmap.yaml manifest creates a configuration file named backup.ini that backs up to an S3 bucket:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: backup-configmap
data:
  backup-host: |
    backup-pod-dns
  backup.ini: |
    [CloudStorage]
    cloud_storage_backup_path = s3://backup-bucket/database-backup-path
    cloud_storage_backup_file_system_path = [backup-pod-dns]:/opt/vertica/config/

    [Database]
    dbName = database-name

    [Misc]
    tempDir = /tmp/vbr
    restorePointLimit = 7
    objectRestoreMode = coexist
```

To create the ConfigMap object, apply the manifest to your Kubernetes environment:

```
$ kubectl apply -f backup-configmap.yaml
```

backup-host definition

In the sample configuration file , *backup-pod-dns* is a portion of the pod's fully qualified domain name (FQDN). Vertica on Kubernetes creates a headless service object that constructs the FQDN for each object. The DNS format for each pod is as follows:

```
podName.headlessServiceName
```

> **Note**
> The headless service name always matches the VerticaDB CR name.

The *podName* portion of the DNS is itself constructed from Kubernetes object names. For example, the following is a complete pod DNS:

```
vdb-main-0.vdb
```

In the preceding example:

- vdb : VerticaDB CR name
- main : Subcluster name
- 0 : StatefulSet ordinal index
- vdb : Headless service name (always identical to the VerticaDB CR name)

To access a pod from outside the namespace, append the namespace to the pod DNS:

```
podName.headlessService.namespace
```

For additional details, see the Kubernetes documentation .

Mount the configuration file

After you define a ConfigMap with vbr configuration information, you must make it available to the Vertica pods that can execute the vbr utility. You can mount the ConfigMap as a volume in a VerticaDB CR instance. For details about mounting volumes in the VerticaDB CR, see Mounting custom volumes .

Cloud storage locations require access to information that you cannot provide in your configuration file , such as environment variables . You can set environment variables in your CR with annotations .

The following mounted-vbr-config.yaml manifest mounts a backup-config ConfigMap object in the Vertica container's /vbr directory:

```
apiVersion: vertica.com/v1beta1
kind: VerticaDB
metadata:
  name: verticadb
spec:
  annotations:
    VBR_BACKUP_STORAGE_SECRET_ACCESS_KEY: "access-key"
    VBR_BACKUP_STORAGE_ACCESS_KEY_ID: "access-key-id"
    VBR_BACKUP_STORAGE_ENDPOINT_URL: "https://path/to/backup/storage"
    VBR_COMMUNAL_STORAGE_SECRET_ACCESS_KEY: "access-key"
    VBR_COMMUNAL_STORAGE_ACCESS_KEY_ID: "access-key-id"
    VBR_COMMUNAL_STORAGE_ENDPOINT_URL: "https://path/to/communal/storage"
  communal:
    endpoint: https://path/to/s3-endpoint
    path: "s3://bucket/database-path"
    includeUIDInPath: true
  image: vertica/vertica-k8s:version
  subclusters:
  - isPrimary: true
    name: main
  volumeMounts:
  - name: backup-configmap
    mountPath: /vbr
  volumes:
  - name: backup-configmap
    configMap:
      name: backup-configmap
```

To mount the ConfigMap, apply the manifest

```
$ kubectl apply -f mounted-vbr-config.yaml
```

After you apply the manifest, each Vertica pod restarts, and the new backup volume is mounted.

Prepare the backup location

Before you can run a backup, you must prepare the backup location with the vbr init command. This command initializes a directory on the backup host to receive and store Vertica backup data. You need to initialize a backup location only once. For details, see Setting up backup locations .

The following backup-init.yaml manifest creates a pod to initialize the backup-host defined in the sample configuration file :

```
apiVersion: v1
kind: Pod
metadata:
  name: backup-init
spec:
  restartPolicy: OnFailure
  containers:
    - name: main
      image: vertica/vertica-k8s:version
      command:
        - bash
        - -c
        - "ssh -o 'StrictHostKeyChecking no' -i /home/dbadmin/.ssh/id_rsa dbadmin@$BACKUP_HOST '/opt/vertica/bin/vbr -t init --cloud-force-init --config-file /vb
      env:
        - name: BACKUP_HOST
          valueFrom:
            configMapKeyRef:
              key: backup-host
              name: backup-configmap
```

Apply the manifest to initialize the backup location:

```
$ kubectl create -f backup-init.yaml
```

Run a backup

Your organization might run backups as needed or on a schedule. The following sections use the sample configuration file ConfigMap to demonstrate both scenarios.

On-demand backups

In some circumstances, you might need to run backup operations as needed. You can create a Kubernetes Job to run an on-demand backup. The following backup-on-demand.yaml manifest creates a Job object that executes a backup:

```
apiVersion: batch/v1
kind: Job
metadata:
  generateName: vertica-backup-
spec:
  template:
    spec:
      restartPolicy: OnFailure
      containers:
        - name: main
          image: vertica/vertica-k8s:version
          command:
            - bash
            - -c
            - "ssh -o 'StrictHostKeyChecking no' -i /home/dbadmin/.ssh/id_rsa dbadmin@$BACKUP_HOST '/opt/vertica/bin/vbr -t backup --config-file /vbr/backup.i
          env:
            - name: BACKUP_HOST
              valueFrom:
                configMapKeyRef:
                  key: backup-host
                  name: backup-configmap
```

Each time that you want to create a new backup, execute the following command:

```
$ kubectl create -f backup-on-demand.yaml
```

Scheduled backups

You might need to schedule a backup at a fixed time or interval. You can run the backup as a Kubenetes CronJob object that schedules a Kubernetes Job as specified in Cron format .

The following backup-cronjob.yaml manifest runs a daily backup at 2:00 AM:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  generateName: vertica-backup-
spec:
  schedule: "00 2 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
            - name: main
              image: vertica/vertica-k8s:version
              command:
                - bash
                - -c
                - "ssh -o 'StrictHostKeyChecking no' -i /home/dbadmin/.ssh/id_rsa dbadmin@$BACKUP_HOST '/opt/vertica/bin/vbr -t backup --config-file /vbr/backu
              env:
                - name: BACKUP_HOST
                  valueFrom:
                    configMapKeyRef:
                      key: backup-host
                      name: backup-configmap
```

To schedule the backup, create the CronJob object:

```
$ kubectl create -f backup-cronjob.yaml
```

Restore from a backup

You can create a Kubernetes Job to restore database objects from a backup. For comprehensive documentation about the vbr restore task, see Restoring backups .

The following restore-on-demand-job.yaml manifest creates a Job object that restores a database:

```
apiVersion: batch/v1
kind: Job
metadata:
  generateName: vertica-restore-
spec:
  template:
    spec:
      restartPolicy: OnFailure
      containers:
        - name: main
          image: vertica/vertica-k8s:version
          command:
            - bash
            - -c
            - "ssh -o 'StrictHostKeyChecking no' -i /home/dbadmin/.ssh/id_rsa dbadmin@$BACKUP_HOST '/opt/vertica/bin/vbr -t restore --config-file /vbr/backup.i
          env:
            - name: BACKUP_HOST
              valueFrom:
                configMapKeyRef:
                  key: backup-host
                  name: backup-configmap
```

The restore process requires that you stop the database, run the restore operation, and then restart the database. This workflow requires additional steps in a containerized environment because Kubernetes has components that monitor and maintain the desired state of the database. You must temporarily adjust some settings to provide time for the restore operation to complete. For details about the settings in this section, see Custom resource definition parameters .

The following steps change the environment for the resource process and then restore the original values:

1. Update the CR to extend the livenessProbe timeout. This timeout triggers a container restart when it expires. The default livenessProbe timeout is about two and half minutes, which does not provide enough time to restore the database. The following patch command uses the livenessProberOverride parameter to set the timeout to about 20 minutes:

```
$ kubectl patch vdb customResourceName --type=json --patch '[ { "op": "add", "path": "/spec/livenessProbeOverride", "value": {"initialDelaySeconds": 60, "
```

2. Delete the StatefulSet for each subcluster so that the pods are restarted with the new livenessProberOverride setting:

```
$ kubectl delete statefulset customResourceName-subclusterName
```

3. Wait until the pods restart and the new pod IPs are present in admintools.conf :

```
$ kubectl wait --for=condition=Ready=True pod --selector=app.kubernetes.io/instance=customResourceName --timeout=10m
```

4. Set autoRestartVertica to false so that the Vertica server process does not automatically restart when you stop the database:

```
$ kubectl patch vdb customResourceName --type=merge --patch '{"spec": {"autoRestartVertica": false}}'
```

5. Access a shell in a host that is running a Vertica pod, and stop the database with admintools :

```
$ kubectl exec -it hostname -- admintools -t stop_db -d database-name
```

   After you stop the database, wait for the cluster lease to expire.

   > **Note**
   > In some scenarios, estimating when the cluster lease expires is difficult. If the restore fails, it logs when the lease expires.
   >
   > You can also experiment with the restartPolicy and backoff failure policy in the Job spec to control how many times to retry the restore.

6. Apply the manifest to run a Job that restores the backup:

```
$ kubectl create -f restore-on-demand-job.yaml
```

7. After the Job completes, use patch to reset the livenessProbe timeout to its default setting:

```
$ kubectl patch vdb customResourceName --type=json --patch '[{ "op": "remove", "path": "/spec/livenessProbeOverride" }]'
```

8. Set autoRestartVertica back to true to reset the restart behavior to its state before the restore operation:

```
$ kubectl patch vdb customResourceName --type=merge --patch '{"spec": {"autoRestartVertica": true}}'
```

9. To speed up the restart process, delete the StatefulSet for each subcluster. The restart speed was affected when you increased the livenessProbeOverride setting:

```
$ kubectl delete statefulset customResourceName-subclusterName
```

10. Wait for the Vertica server to restart:

```
$ kubectl wait --for=condition=Ready=True pod --selector=app.kubernetes.io/instance=customResourceName --timeout=10m
```

Troubleshooting your Kubernetes cluster

These tips can help you avoid issues related to your Vertica on Kubernetes deployment and troubleshoot any problems that occur.

Download the kubectl command line tool to debug your Kubernetes resources.

In this section

- General cluster and database
- Helm charts
- Metrics gathering
- Security
- VerticaAutoscaler

General cluster and database

Inspect objects to diagnose issues

When you deploy a custom resource (CR) , you might encounter a variety of issues. To pinpoint an issue, use the following commands to inspect the objects that the CR creates:

kubectl get returns basic information about deployed objects:

```
$ kubectl get pods -n namespace
$ kubectl get statefulset -n namespace
$ kubectl get pvc -n namespace
$ kubectl get event
```

kubectl describe returns detailed information about deployed objects:

```
$ kubectl describe pod pod-name -n namespace
$ kubectl describe statefulset name -n namespace
$ kubectl describe custom-resource-name -n namespace
```

Verify updates to a custom resource

Because the operator takes time to perform tasks, updates to the custom resource are not effective immediately. Use the kubectl command line tool to verify that changes are applied.

You can use the kubectl wait command to wait for a specified condition. For example, the operator uses the ImageChangeInProgress condition to provide an upgrade status. After you begin the image version upgrade , wait until the operator acknowledges the upgrade and sets this condition to True:

```
$ kubectl wait --for=condition=ImageChangeInProgress=True vdb/cluster-name —-timeout=180s
```

After the upgrade begins, you can wait until the operator leaves upgrade mode and sets this condition to False:

```
$ kubectl wait --for=condition=ImageChangeInProgress=False vdb/cluster-name —-timeout=800s
```

For more information about kubectl wait, see the kubectl reference documentation .

Pods are running but the database is not ready

When you check the pods in your cluster, the pods are running but the database is not ready:

```
$ kubectl get pods
NAME                                                READY   STATUS    RESTARTS   AGE
vertica-crd-sc1-0                                   0/1     Running   0          12m
vertica-crd-sc1-1                                   0/1     Running   1          12m
vertica-crd-sc1-2                                   0/1     Running   0          12m
verticadb-operator-controller-manager-5d9cdc9b8-kw9nv   2/2     Running   0          24m
```

To find the root cause of the issue, use kubectl logs to check the operator manager. The following example shows that the communal storage bucket does not exist:

```
$ kubectl logs -l app.kubernetes.io/name=verticadb-operator -c manager -f
2021-08-04T20:03:00.289Z	INFO	controllers.VerticaDB	ExecInPod entry {"verticadb": "default/vertica-crd", "pod": {"namespace": "default", "name": "vertica-crd-sc1-0"}, "command": "bash -c ls -l /opt/vertica/config/admintools.conf && grep '^node\\|^v_\\|^host' /opt/vertica/config/admintools.conf "}
2021-08-04T20:03:00.369Z	INFO	controllers.VerticaDB	ExecInPod stream	{"verticadb": "default/vertica-crd", "pod": {"namespace": "default", "name": "vertica-crd-sc1-0"}, "err": null, "stdout": "-rw-rw-r-- 1 dbadmin verticadba 1243 Aug  4 20:00 /opt/vertica/config/admintools.conf\nhosts = 10.244.1.5,10.244.2.4,10.244.4.6\nnode0001 = 10.244.1.5,/data,/data\nnode0002 = 10.244.2.4,/data,/data\nnode0003 = 10.244.4.6,/data,/data\n", "stderr": ""}
2021-08-04T20:03:00.369Z	INFO	controllers.VerticaDB	ExecInPod entry {"verticadb": "default/vertica-crd", "pod": {"namespace": "default", "name": "vertica-crd-sc1-0"}, "command": "/opt/vertica/bin/admintools -t create_db --skip-fs-checks --hosts=10.244.1.5,10.244.2.4,10.244.4.6 --communal-storage-location=s3://newbucket/db/26100df1-93e5-4e64-b665-533e14abb67c --communal-storage-params=/home/dbadmin/auth_parms.conf --sql=/home/dbadmin/post-db-create.sql --shard-count=12 --depot-path=/depot --database verticadb --force-cleanup-on-failure --noprompt --password ******* "}
2021-08-04T20:03:00.369Z	DEBUG	controller-runtime.manager.events	Normal	{"object": {"kind":"VerticaDB","namespace":"default","name":"vertica-crd","uid":"26100df1-93e5-4e64-b665-533e14abb67c","apiVersion":"vertica.com/v1beta1","resourceVersion":"11591"}, "reason": "CreateDBStart", "message": "Calling 'admintools -t create_db'"}
2021-08-04T20:03:17.051Z	INFO	controllers.VerticaDB	ExecInPod stream	{"verticadb": "default/vertica-crd", "pod": {"namespace": "default", "name": "vertica-crd-sc1-0"}, "err": "command terminated with exit code 1", "stdout": "Default depot size in use\nDistributing changes to cluster.\n\tCreating database verticadb\nBootstrap on host 10.244.1.5 return code 1 stdout '' stderr 'Logged exception in writeBufferToFile: RecvFiles failed in closing file [s3://newbucket/db/26100df1-93e5-4e64-b665-533e14abb67c/verticadb_rw_access_test.txt]: The specified bucket does not exist. Writing test data to file s3://newbucket/db/26100df1-93e5-4e64-b665-533e14abb67c/verticadb_rw_access_test.txt failed.\\nTesting rw access to communal location s3://newbucket/db/26100df1-93e5-4e64-b665-533e14abb67c/ failed\\n'\n\nError: Bootstrap on host 10.244.1.5 return code 1 stdout '' stderr 'Logged exception in writeBufferToFile: RecvFiles failed in closing file [s3://newbucket/db/26100df1-93e5-4e64-b665-533e14abb67c/verticadb_rw_access_test.txt]: The specified bucket does not exist. Writing test data to file s3://newbucket/db/26100df1-93e5-4e64-b665-533e14abb67c/verticadb_rw_access_test.txt failed.\\nTesting rw access to communal location s3://newbucket/db/26100df1-93e5-4e64-b665-533e14abb67c/ failed\\n'\n\n", "stderr": ""}
2021-08-04T20:03:17.051Z	INFO	controllers.VerticaDB	aborting reconcile of VerticaDB {"verticadb": "default/vertica-crd", "result": {"Requeue":true,"RequeueAfter":0}, "err": null}
2021-08-04T20:03:17.051Z	DEBUG	controller-runtime.manager.events	Warning {"object": {"kind":"VerticaDB","namespace":"default","name":"vertica-crd","uid":"26100df1-93e5-4e64-b665-533e14abb67c","apiVersion":"vertica.com/v1beta1","resourceVersion":"11591"}, "reason": "S3BucketDoesNotExist", "message": "The bucket in the S3 path 's3://newbucket/db/26100df1-93e5-4e64-b665-533e14abb67c' does not exist"}
```

Create an S3 bucket for the cluster:

```
$ S3_BUCKET=newbucket
$ S3_CLUSTER_IP=$(kubectl get svc | grep minio | head -1 | awk '{print $3}')
$ export AWS_ACCESS_KEY_ID=minio
$ export AWS_SECRET_ACCESS_KEY=minio123
$ aws s3 mb s3://$S3_BUCKET --endpoint-url http://$S3_CLUSTER_IP
make_bucket: newbucket
```

Use kubectl get pods to verify that the cluster uses the new S3 bucket and the database is ready:

```
$ kubectl get pods
NAME                                              READY   STATUS    RESTARTS   AGE
minio-ss-0-0                                      1/1     Running   0          18m
minio-ss-0-1                                      1/1     Running   0          18m
minio-ss-0-2                                      1/1     Running   0          18m
minio-ss-0-3                                      1/1     Running   0          18m
vertica-crd-sc1-0                                 1/1     Running   0          20m
vertica-crd-sc1-1                                 1/1     Running   0          20m
vertica-crd-sc1-2                                 1/1     Running   0          20m
verticadb-operator-controller-manager-5d9cdc9b8-kw9nv   2/2     Running   0          63m
```

Database is not available

After you create a custom resource instance, the database is not available. The kubectl get *custom-resource* command does not display information:

```
$ kubectl get vdb
NAME         AGE   SUBCLUSTERS   INSTALLED   DBADDED   UP
vertica-crd   4s
```

Use kubectl describe *custom-resource* to check the events for the pods to identify any issues:

```
$ kubectl describe vdb
Name:         vertica-crd
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  vertica.com/v1beta1
Kind:         VerticaDB
Metadata:
  ...
  Superuser Password Secret:  su-passwd
Events:
  Type     Reason                         Age              From               Message
  ----     ------                         ----             ----               -------
  Warning  SuperuserPasswordSecretNotFound  5s (x12 over 15s)  verticadb-operator  Secret for superuser password 'su-passwd' was not found
```

In this circumstance, the custom resource uses a Secret named su-passwd to store the Superuser Password Secret , but there is no such Secret available. Create a Secret named su-passwd to store the Secret:

```
$ kubectl create secret generic su-passwd --from-literal=password=sup3rs3cr3t
secret/su-passwd created
```

Use kubectl get *custom-resource* to verify the issue is resolved:

```
$ kubectl get vdb
NAME         AGE   SUBCLUSTERS   INSTALLED   DBADDED   UP
vertica-crd  89s   1             0           0         0
```

Image pull failure

You receive an ImagePullBackOff error when you deploy a Vertica cluster with Helm charts, but you do not pre-pull the Vertica image from the local registry server:

```
$ kubectl describe pod pod-name-0
...
Events:
  Type     Reason          Age                From               Message
  ----     ------          ----               ----               -------
  ...
  Warning  Failed          2m32s              kubelet            Failed to pull image "k8s-rhel7-01:5000/vertica-k8s:default-1": rpc error: code = Unknown desc =
context canceled
  Warning  Failed          2m32s              kubelet            Error: ErrImagePull
  Normal   BackOff         2m32s              kubelet            Back-off pulling image "k8s-rhel7-01:5000/vertica-k8s:default-1"
  Warning  Failed          2m32s              kubelet            Error: ImagePullBackOff
  Normal   Pulling         2m18s (x2 over 4m22s)  kubelet            Pulling image "k8s-rhel7-01:5000/vertica-k8s:default-1"
```

This occurs because the Vertica image size is too big to pull from the registry while deploying the Vertica cluster. Execute the following command on a Kubernetes host:

```
$ docker image list | grep vertica-k8s
k8s-rhel7-01:5000/vertica-k8s default-1 2d6f5d3d90d6 9 days ago 1.55GB
```

To solve this issue, complete one of the following:

- Pull the Vertica images on each node before creating the Vertica StatefulSet:
  ```
  $ NODES=`kubectl get nodes | grep -v NAME | awk '{print $1}'`
  $ for node in $NODES; do ssh $node docker pull $DOCKER_REGISTRY:5000/vertica-k8s:$K8S_TAG; done
  ```
- Use the reduced-size vertica/vertica-k8s:latest image for the Vertica server.

Pending pods due to insufficient CPU

If your host nodes do not have enough resources to fulfill the resource request from a pod, the pod stays in pending status.

> **Note**
> As a best practice, do not request the maximum amount of resources available on a host node to leave resources for other processes on the

host node.

In the following example, the pod requests 40 CPUs on the host node, and the pod stays in **Pending** :

```
$ kubectl describe pod cluster-vertica-defaultsubcluster-0
...
Status:         Pending
...
Containers:
  server:
    Image:       docker.io/library/vertica-k8s:default-1
    Ports:       5433/TCP, 5434/TCP, 22/TCP
    Host Ports:  0/TCP, 0/TCP, 0/TCP
    Command:
      /opt/vertica/bin/docker-entrypoint.sh
      restart-vertica-node
    Limits:
      memory:  200Gi
    Requests:
      cpu: 40
      memory:  200Gi
...
Events:
  Type     Reason            Age    From               Message
  ----     ------            ----   ----               -------
  Warning  FailedScheduling  3h20m  default-scheduler  0/5 nodes are available: 5 Insufficient cpu.
```

To confirm the resources available on the host node. The following command confirms that the host node has only 40 allocatable CPUs:

```
$ kubectl describe node host-node-1
...
Conditions:
  Type             Status  LastHeartbeatTime                 LastTransitionTime                Reason                       Message
  ----             ------  -----------------                 ------------------                ------                       -------
  MemoryPressure   False   Sat, 20 Mar 2021 22:39:10 -0400   Sat, 20 Mar 2021 13:07:02 -0400   KubeletHasSufficientMemory   kubelet has sufficient
memory available
  DiskPressure     False   Sat, 20 Mar 2021 22:39:10 -0400   Sat, 20 Mar 2021 13:07:02 -0400   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Sat, 20 Mar 2021 22:39:10 -0400   Sat, 20 Mar 2021 13:07:02 -0400   KubeletHasSufficientPID      kubelet has sufficient PID
available
  Ready            True    Sat, 20 Mar 2021 22:39:10 -0400   Sat, 20 Mar 2021 13:07:12 -0400   KubeletReady                 kubelet is posting ready status
Addresses:
  InternalIP:  172.19.0.5
  Hostname:    eng-g9-191
Capacity:
  cpu:                40
  ephemeral-storage:  285509064Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             263839236Ki
  pods:               110
Allocatable:
  cpu:                40
  ephemeral-storage:  285509064Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             263839236Ki
  pods:               110
...
Non-terminated Pods:          (3 in total)
  Namespace                   Name                                CPU Requests  CPU Limits  Memory Requests  Memory Limits  AGE
  ---------                   ----                                ------------  ----------  ---------------  -------------  ---
  default                     cluster-vertica-defaultsubcluster-0    38 (95%)      0 (0%)      200Gi (79%)      200Gi (79%)    51m
  kube-system                 kube-flannel-ds-8brv9                  100m (0%)     100m (0%)   50Mi (0%)        50Mi (0%)      9h
  kube-system                 kube-proxy-lgjhp                       0 (0%)        0 (0%)      0 (0%)           0 (0%)         9h
...
```

To correct this issue, reduce the resource.requests in the subcluster to values lower than the maximum allocatable CPUs. The following example uses a YAML-formatted file named patch.yaml to lower the resource requests for the pod:

```
$ cat patch.yaml
spec:
  subclusters:
    - name: defaultsubcluster
      resources:
        requests:
          memory: 238Gi
          cpu: "38"
        limits:
          memory: 238Gi
$ kubectl patch vdb cluster-vertica —type=merge --patch “$(cat patch.yaml)”
verticadb.vertica.com/cluster-vertica patched
```

Pending pod after node removed

When you remove a host node from your Kubernetes cluster, a Vertica pod might stay in pending status if the pod uses a PersistentVolume (PV) that has a node affinity rule that prevents the pod from running on another node.

To resolve this issue, you must verify that the pods are pending because of an affinity rule, and then use the vdb-gen tool to revive the entire cluster.

First, determine if the pod is pending because of a node affinity rule. This requires details about the pending pod, the PersistentVolumeClaim (PVC) associated with the pod, and the PersistentVolume (PV) associated with the PVC:

1. Use kubectl describe to return details about the pending pod:

```
$ kubectl describe pod pod-name
...
Events:
  Type     Reason            Age               From               Message
  ----     ------            ----              ----               -------
  Warning  FailedScheduling  28s (x2 over 48s)  default-scheduler  0/2 nodes are available: 1 node(s) had untolerated taint
{node.kubernetes.io/unschedulable: }, 1 node(s) had volume node affinity conflict, 1 node(s) were unschedulable. preemption: 0/2 nodes are available: 2
Preemption is not helpful for scheduling.
```

   The Message column verifies that the pod was not scheduled due a volume node affinity conflict .
2. Get the name of the PVC associated with the pod:

```
$ kubectl get pod -o jsonpath='{.spec.volumes[0].persistentVolumeClaim.claimName}{"\n"}' pod-name
local-data-pod-name
```

3. Use the PVC to get the PV. PVs are associated with nodes:

```
$ kubectl get pvc -o jsonpath='{.spec.volumeName}{"\n"}' local-data-pod-name
pvc-1926ae96-574d-4433-99b4-ec9ab0e5e497
```

4. Use the PV to get the name of the node that has the affinity rule:

```
$ kubectl get pv -o jsonpath='{.spec.nodeAffinity.required.nodeSelectorTerms[0].matchExpressions[0].values[0]}{"\n"}' pvc-1926ae96-574d-4433-99b4-
ec9ab0e5e497
ip-10-20-30-40.ec2.internal
```

5. Verify that the node with the affinity rule is the node that was removed from the Kubernetes cluster.

Next, you must revive the entire cluster to get all pods running again. When you revive the cluster, you create new PVCs that restore the association between each pod and a PV to satisfy the node affinity rule.

While you have nodes running in the cluster, you can use the vdb-gen tool to generate a manifest and revive the database:

1. Download the vdb-gen tool from the vertica-kubernetes GitHub repository:

```
$ wget https://github.com/vertica/vertica-kubernetes/releases/latest/download/vdb-gen
```

2. Copy the tool into a pod that has a running Vertica process:

```
$ kubectl cp vdb-gen pod-name:/tmp/vdb-gen
```

3. The vdb-gen tool requires the database name, so retrieve it with the following command:

```
$ kubectl get vdb -o jsonpath='{.spec.dbName}{"\n"}' v
database-name
```

4. Run the vdb-gen tool with the database name. The following command runs the tool and pipes the output to a file named revive.yaml :

```
$ kubectl exec -i pod-name -- bash -c "chmod +x /tmp/vdb-gen && /tmp/vdb-gen --ignore-cluster-lease --name v localhost database-name | tee
/tmp/revive.yaml"
```

5. Copy revive.yaml to your local machine so that you can use it after you remove the cluster:

```
$ kubectl cp pod-name:/tmp/revive.yaml revive.yaml
```

6. Save the current VerticaDB Custom Resource (CR). For example, the following command saves a CR named vertdb to a file named orig.yaml :

```
$ kubectl get vdb vertdb -o yaml > orig.yaml
```

7. Update revive.yaml with parts of orig.yaml that vdb-gen did not capture. For example, custom resource limits.
8. Delete the existing Vertica cluster:

```
$ kubectl delete vdb vertdb
verticadb.vertica.com "vertdb" deleted
```

9. Delete all PVCs that are associated with the deleted cluster.
   1. Retrieve the PVC names. A PVC name uses the *dbname* - *subcluster* - *podindex* format:

```
$ kubectl get pvc
NAME                   STATUS   VOLUME                                     CAPACITY ACCESS MODES   STORAGECLASS   AGE
local-data-vertdb-sc-0   Bound    pvc-e9834c18-bf60-4a4b-a686-ba8f7b601230   1Gi      RWO           local-path     34m
local-data-vertdb-sc-1   Bound    pvc-1926ae96-574d-4433-99b4-ec9ab0e5e497   1Gi      RWO           local-path     34m
local-data-vertdb-sc-2   Bound    pvc-4541f7c9-3afc-47f0-8d04-67fac370ee88   1Gi      RWO           local-path     34m
```

2. Delete the PVCs:

```
$ kubectl delete pvc local-data-vertdb-sc-0 local-data-vertdb-sc-1 local-data-vertdb-sc-2
persistentvolumeclaim "local-data-vertdb-sc-0" deleted
persistentvolumeclaim "local-data-vertdb-sc-1" deleted
persistentvolumeclaim "local-data-vertdb-sc-2" deleted
```

10. Revive the database with revive.yaml :

```
$ kubectl apply -f revive.yaml
verticadb.vertica.com/vertdb created
```

After the revive completes, all Vertica pods are running, and PVCs are recreated on new nodes. Wait for the operator to start the database.

Deploying to Istio

Vertica does not officially support Istio because the Istio sidecar port requirement conflicts with the port that Vertica requires for internal node communication. However, you can deploy Vertica on Kubernetes to Istio with changes to the Istio InboundInterceptionMode setting. Vertica provides access to this setting with annotations on the VerticaDB CR.

REDIRECT mode

REDIRECT mode is the default InboundInterceptionMode setting, and it requires that you disable network address translation (NAT) on port 5434, the port that the pods use for internal communication. Disable NAT on this port with the excludeInboundPorts annotation:

```
apiVersion: vertica.com/v1beta1
kind: VerticaDB
metadata:
  name: vdb
spec:
  annotations:
    traffic.sidecar.istio.io/excludeInboundPorts: "5434"
```

TPROXY mode

Another option is TPROXY mode, which permits both encrypted and unencrypted traffic. Set this mode with the interceptionMode annotation:

```
apiVersion: vertica.com/v1beta1
kind: VerticaDB
metadata:
  name: vdb
spec:
  annotations:
    sidecar.istio.io/interceptionMode: TPROXY
```

By default, TPROXY mode permits both encrypted and unencrypted traffic. To disable unencrypted traffic, apply a PeerAuthentication CR that implements strict mTLS:

Important
If you use strict mTLS, you must use operator version 1.11.0 or higher .

```
$ kubectl apply -n namespace -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
spec:
  mtls:
    mode: STRICT
EOF
```

Helm charts

Helm install failure

When you install the VerticaDB operator and admission controller Helm chart, the helm install command might return the following error:

```
$ helm install vdb-op vertica-charts/verticadb-operator
Error: INSTALLATION FAILED: unable to build kubernetes objects from release manifest: [unable to recognize "": no matches for kind "Certificate" in version "cert-manager.io/v1", unable to recognize "": no matches for kind "Issuer" in version "cert-manager.io/v1"]
```

The error indicates that you have not met the TLS prerequisite for the admission controller webhook. To resolve this issue, install cert-manager or configure custom certificates. The following steps install cert-manager.

1. Install the cert-manager YAML manifest:
   ```
   $ kubectl apply -f https://github.com/jetstack/cert-manager/releases/download/v1.5.3/cert-manager.yaml
   ```
2. Verify the cert-manager installation.
   If you try to install the Helm chart immediately after you install cert-manager, you might receive the following error:
   ```
   $ helm install vdb-op vertica-charts/verticadb-operator
   Error: failed to create resource: Internal error occurred: failed calling webhook "webhook.cert-manager.io": failed to call webhook: Post "https://cert-manager-webhook.cert-manager.svc:443/mutate?timeout=10s": dial tcp 10.96.232.154:443: connect: connection refused
   ```

   You receive this error because cert-manager needs time to create its pods and register the webhook with the cluster. Wait a few minutes, and then verify the cert-manager installation with the following command:
   ```
   $ kubectl get pods --namespace cert-manager
   NAME                                       READY   STATUS    RESTARTS   AGE
   cert-manager-7dd5854bb4-skks7              1/1     Running   5          12d
   cert-manager-cainjector-64c949654c-9nm2z   1/1     Running   5          12d
   cert-manager-webhook-6bdffc7c9d-b7r2p      1/1     Running   5          12d
   ```

   For additional details about cert-manager install verification, see the cert-manager documentation.
3. After you verify the cert-manager installation, you must uninstall the Helm chart and then reinstall:
   ```
   $ helm uninstall vdb-op
   $ helm install vdb-op vertica-charts/verticadb-operator
   ```

For additional information, see Installing the Vertica DB operator.

Custom certificate helm install error

If you use custom certificates when you install the operator with the Helm chart, the helm install or kubectl apply command might return an error similar to the following:

```
$ kubectl apply -f ../operatorcrd.yaml
Error from server (InternalError): error when creating "../operatorcrd.yaml": Internal error occurred: failed calling webhook "mverticadb.kb.io": Post "https://verticadb-operator-webhook-service.namespace.svc:443/mutate-vertica-com-v1beta1-verticadb?timeout=10s": x509: certificate is valid for ip-10-0-21-169.ec2.internal, test-bastion, not verticadb-operator-webhook-service.default.svc
```

You receive this error when the TLS key's Domain Name System (DNS) or Subject Alternate Name (SAN) is incorrect. To correct this error, define the DNS and SAN in a configuration file in the following format:

```
commonName = verticadb-operator-webhook-service.namespace.svc
...
[alt_names]
DNS.1 = verticadb-operator-webhook-service.namespace.svc
DNS.2 = verticadb-operator-webhook-service.namespace.svc.cluster.local
```

For additional details, see Installing the Vertica DB operator.

Metrics gathering

Adding and testing the vlogger sidecar

Vertica provides the vlogger image that sends logs from vertica.log to standard output on the host node for log aggregation.

To add the sidecar to the CR, add an element to the spec.sidecars definition:

```
spec:
  ...
  sidecars:
    - name: vlogger
      image: vertica/vertica-logger:1.0.0
```

To test the sidecar, run the following command and verify that it returns logs:

```
$ kubectl logs pod-name -c vlogger

2021-12-08 14:39:08.538 DistCall Dispatch:0x7f3599ffd700-c000000000997e [Txn
2021-12-08 14:40:48.923 INFO New log
2021-12-08 14:40:48.923 Main Thread:0x7fbbe2cf6280 [Init] <INFO> Log /data/verticadb/v_verticadb_node0002_catalog/vertica.log opened; #1
2021-12-08 14:40:48.923 Main Thread:0x7fbbe2cf6280 [Init] <INFO> Processing command line: /opt/vertica/bin/vertica -D
/data/verticadb/v_verticadb_node0002_catalog -C verticadb -n v_verticadb_node0002 -h 10.20.30.40 -p 5433 -P 4803 -Y ipv4
2021-12-08 14:40:48.923 Main Thread:0x7fbbe2cf6280 [Init] <INFO> Starting up Vertica Analytic Database v11.0.2-20211201
2021-12-08 14:40:48.923 Main Thread:0x7fbbe2cf6280 [Init] <INFO>
2021-12-08 14:40:48.923 Main Thread:0x7fbbe2cf6280 [Init] <INFO> vertica(v11.0.2) built by @re-docker5 from
master@a44ffabdf3f05e8d104426506b088192f741c485 on 'Wed Dec  1 06:10:34 2021' $BuildId$
2021-12-08 14:40:48.923 Main Thread:0x7fbbe2cf6280 [Init] <INFO> CPU architecture: x86_64
2021-12-08 14:40:48.923 Main Thread:0x7fbbe2cf6280 [Init] <INFO> 64-bit Optimized Build
2021-12-08 14:40:48.923 Main Thread:0x7fbbe2cf6280 [Init] <INFO> Compiler Version: 7.3.1 20180303 (Red Hat 7.3.1-5)
2021-12-08 14:40:48.923 Main Thread:0x7fbbe2cf6280 [Init] <INFO> LD_LIBRARY_PATH=/opt/vertica/lib
2021-12-08 14:40:48.923 Main Thread:0x7fbbe2cf6280 [Init] <INFO> LD_PRELOAD=
2021-12-08 14:40:48.925 Main Thread:0x7fbbe2cf6280 <LOG> @v_verticadb_node0002: 00000/5081: Total swap memory used: 0
2021-12-08 14:40:48.925 Main Thread:0x7fbbe2cf6280 <LOG> @v_verticadb_node0002: 00000/4435: Process size resident set: 28651520
2021-12-08 14:40:48.925 Main Thread:0x7fbbe2cf6280 <LOG> @v_verticadb_node0002: 00000/5075: Total Memory free + cache: 59455180800
2021-12-08 14:40:48.925 Main Thread:0x7fbbe2cf6280 [Txn] <INFO> Looking for catalog at: /data/verticadb/v_verticadb_node0002_catalog/Catalog
...
```

Core file for Vertica server container process

In some circumstances, you might need to examine a core file that contains information about the Vertica server container process:

1. For the custom resource securityContext value, set the privileged property to true :

   ```
   apiVersion: vertica.com/v1beta1
   kind: VerticaDB
   ...
   spec:
     ...
     securityContext:
       privileged: true
   ```

2. On the host machine, verify that /proc/sys/kernel/core_pattern is set to core :

   ```
   $ cat /proc/sys/kernel/core_pattern
   core
   ```

   The /proc/sys/kernel/core_pattern file is not namespaced, so setting this value affects all containers running on that host.

When Vertica generates a core, the machine writes a message to vertica.log that indicates where you can locate the core file.

Security

Custom PodSecurityPolicy errors

Vertica on Kubernetes requires the following Linux capabilities that enable SSH communications between the pods:

- SYS_CHROOT
- AUDIT_WRITE

In some circumstances, these capabilities might conflict with custom security policy restrictions and cause errors. For example:

```
$ kubectl describe statefulset subcluster-name
...
Events:
  Type     Reason        Age                  From                  Message
  ----     ------        ----                 ----                  -------
  Warning  FailedCreate  29m (x73 over 15h)   statefulset-controller  create Pod subcluster-name-0 in StatefulSet subcluster-name failed error: pods
"subcluster-name-0" is forbidden: PodSecurityPolicy: unable to admit pod: [spec.containers[0].securityContext.capabilities.add: Invalid value:
"AUDIT_WRITE": capability may not be added spec.containers[0].securityContext.capabilities.add: Invalid value: "SYS_CHROOT": capability may not be
added]
```

When a similar error is returned, you must update your PodSecurityPolicy. For details, see the Kubernetes documentation .

VerticaAutoscaler

Cannot find CPU metrics with VerticaAutoscaler

You might notice that your VerticaAutoScaler is not scaling correctly according to CPU utilization:

```
$ kubectl get hpa
NAME              REFERENCE                        TARGETS        MINPODS   MAXPODS   REPLICAS   AGE
autoscaler-name   VerticaAutoscaler/autoscaler-name   <unknown>/50%   3         12        0          19h

$ kubectl describe hpa
Warning: autoscaling/v2beta2 HorizontalPodAutoscaler is deprecated in v1.23+, unavailable in v1.26+; use autoscaling/v2 HorizontalPodAutoscaler
Name: autoscaler-name
Namespace: namespace
Labels: <none>
Annotations: <none>
CreationTimestamp: Thu, 12 May 2022 10:25:02 -0400
Reference: VerticaAutoscaler/autoscaler-name
Metrics: ( current / target )
resource cpu on pods (as a percentage of request): <unknown> / 50%
Min replicas: 3
Max replicas: 12
VerticaAutoscaler pods: 3 current / 0 desired
Conditions:
Type Status Reason Message
---- ------ ------ -------
AbleToScale True SucceededGetScale the HPA controller was able to get the target's current scale
ScalingActive False FailedGetResourceMetric the HPA was unable to compute the replica count: failed to get cpu utilization: unable to get metrics for resource cpu: unable to fetch metrics from resource metrics API: the server could not find the requested resource (get pods.metrics.k8s.io)
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Warning FailedGetResourceMetric 7s horizontal-pod-autoscaler failed to get cpu utilization: unable to get metrics for resource cpu: unable to fetch metrics from resource metrics API: the server could not find the requested resource (get pods.metrics.k8s.io)
Warning FailedComputeMetricsReplicas 7s horizontal-pod-autoscaler invalid metrics (1 invalid out of 1), first error is: failed to get cpu utilization: unable to get metrics for resource cpu: unable to fetch metrics from resource metrics API: the server could not find the requested resource (get pods.metrics.k8s.io)
```

You receive this error because the metrics server is not installed:

```
$ kubectl top nodes
error: Metrics API not available
```

To install the metrics server:

1. Download the components.yaml file:
   ```
   $ kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
   ```
2. Optionally, disable TLS:
   ```
   $ if ! grep kubelet-insecure-tls components.yaml; then
     sed -i 's/- args:/- args:\n - --kubelet-insecure-tls/' components.yaml;
   ```
3. Apply the YAML file:
   ```
   $ kubectl apply -f components.yaml
   ```
4. Verify that the metrics server is running:
   ```
   $ kubectl get svc metrics-server -n namespace
   NAME             TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
   metrics-server   ClusterIP   10.105.239.175   <none>        443/TCP   19h
   ```

CPU request error with VerticaAutoscaler

You might receive an error that states:

failed to get cpu utilization: missing request for cpu

You get this error because you must set resource limits on all containers, including sidecar containers. To correct this error:

1. Verify the error:

```
$ kubectl get hpa
NAME                REFERENCE                          TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
autoscaler-name     VerticaAutoscaler/autoscaler-name   <unknown>/50%   3         12        0          19h

$ kubectl describe hpa
Warning: autoscaling/v2beta2 HorizontalPodAutoscaler is deprecated in v1.23+, unavailable in v1.26+; use autoscaling/v2 HorizontalPodAutoscaler
Name: autoscaler-name
Namespace: namespace
Labels: <none>
Annotations: <none>
CreationTimestamp: Thu, 12 May 2022 15:58:31 -0400
Reference: VerticaAutoscaler/autoscaler-name
Metrics: ( current / target )
resource cpu on pods (as a percentage of request): <unknown> / 50%
Min replicas: 3
Max replicas: 12
VerticaAutoscaler pods: 3 current / 0 desired
Conditions:
Type Status Reason Message
---- ------ ------ -------
AbleToScale True SucceededGetScale the HPA controller was able to get the target's current scale
ScalingActive False FailedGetResourceMetric the HPA was unable to compute the replica count: failed to get cpu utilization: missing request for cpu
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Warning FailedGetResourceMetric 4s (x5 over 64s) horizontal-pod-autoscaler failed to get cpu utilization: missing request for cpu
Warning FailedComputeMetricsReplicas 4s (x5 over 64s) horizontal-pod-autoscaler invalid metrics (1 invalid out of 1), first error is: failed to get cpu utilization: missing request for cpu
```

2. Add resource limits to the CR:

```
$ cat /tmp/vdb.yaml
apiVersion: vertica.com/v1beta1
kind: VerticaDB
metadata:
  name: vertica-vdb
spec:
  sidecars:
    - name: vlogger
      image: vertica/vertica-logger:latest
      resources:
        requests:
          memory: "100Mi"
          cpu: "100m"
        limits:
          memory: "100Mi"
          cpu: "100m"
  communal:
    credentialSecret: communal-creds
    endpoint: https://endpoint
      path: s3://bucket-location
  dbName: verticadb
  image: vertica/vertica-k8s:latest
  subclusters:
  - isPrimary: true
    name: sc1
    resources:
      requests:
        memory: "4Gi"
        cpu: 2
      limits:
        memory: "4Gi"
        cpu: 2
    serviceType: ClusterIP
    serviceName: sc1
    size: 3
  upgradePolicy: Auto
```

3. Apply the update:

```
$ kubectl apply -f /tmp/vdb.yaml
verticadb.vertica.com/vertica-vdb created
```

When you set a new CPU resource limit, Kubernetes reschedules each pod in the StatefulSet in a rolling update until all pods have the updated CPU resource limit.

Data exploration

If the data you need to analyze is already well-specified in a schema, such as if it was exported from another database, then you can usually proceed to defining tables and loading the data . Often, however, your initial data is less clear or is in a format that does not require a consistent schema, like JSON. Before you can define database tables for your data, you must explore that data in more detail and make decisions about how to represent it in tables.

During data exploration, you develop a schema and decide what kind of table to use in your production database. In some cases, you might choose to use external tables to describe data, and in other cases you might choose to load data into the database. Either way, over time, you might need to make changes if the schema evolves . For example, new data might add columns or change the data type of existing columns. You should expect to revisit your data schema over time. When you do, use the techniques described in the following sections.

In this section

- Develop a schema
- Table types
- Schema changes

Develop a schema

Vertica provides two general approaches to developing a table definition during data exploration without resorting to inspecting data files by hand: inference and flex tables.

If your data is in Parquet, ORC, JSON, or Avro format, you can use a Vertica function to inspect a sample file and automatically generate a "first draft" of a table definition. The output indicates where the function could not make a decision, so that you can focus on those areas as you refine the definition. Using this approach, you iteratively test sample data, refine the table definition, and test more data until you arrive at a satisfactory table definition.

Alternatively, you can use flex tables, a special type of table designed for exploring data with an unknown or varying schema. You can use flex functions to explore this data and materialize individual columns in your table. When you have identified the columns that should consistently be part of your table, you can then define a regular (non-flex) table.

In this section

- Inference
- Flex tables

Inference

If your data is in Parquet, ORC, JSON, or Avro format, you can use a Vertica function to inspect a sample file and automatically generate a "first draft" of a table definition. You can then refine this definition, such as by specifying VARCHAR lengths, NUMERIC precision, and ARRAY bounds. If the data format supports it, you can ignore columns in the data that you do not need for your table.

When you have an initial table definition, you can load more data. New data might have different properties, so it's important to adjust your table definition in light of more data. Test the table definition with enough data to feel confident before putting it into production.

You might find it easier to do initial exploration with external tables, whose definitions are more easily modified.

Create first draft of table

The INFER_TABLE_DDL function inspects a data file and returns a CREATE TABLE or CREATE EXTERNAL TABLE statement based on its contents. The column definitions in the returned statement can be incomplete, and sometimes the function cannot determine a column's data type. The function also uses the column names as they appear in the data, even if they violate Vertica restrictions. For all of these reasons, you must review and adjust the proposed definition.

In the following example, the input path contains data for a product catalog. Note the warnings about data types:

```
=> SELECT INFER_TABLE_DDL('/data/products/products.parquet'
    USING PARAMETERS format = 'parquet', table_name = 'products');

WARNING 9311:  This generated statement contains one or more varchar/varbinary columns which default to length 80

     INFER_TABLE_DDL

-------------------------------------------------------------------------
 create table "products"(
  "id" int,
  "msrp" numeric(6,2),
  "name" varchar,
  "description" varchar,
  "vendors" Array[Row(
    "name" varchar,
    "price" numeric(6,2)
  )]
);
(1 row)
```

The table definition contains several VARCHAR values with unspecified lengths. The Vertica default length for VARCHAR is 80, and for a native table, Vertica allocates memory based on the maximum length. If a value is unlikely to be that large, such as for a name, you can specify a smaller size to improve query performance. Conversely, if a value could be much longer, such as the description field here, you can specify a larger size so that Vertica does not truncate the value.

For numeric values, INFER_TABLE_DDL suggests precision and scale based on the data in the sample file. As with VARCHAR, values that are too high cause more memory to be allocated. In addition, computations could produce unexpected results by using the extra precision instead of rounding.

Arrays, by default, are unbounded, and Vertica allocates memory to accommodate the maximum binary size. To reduce this footprint, you can specify a maximum number of elements for an array or its maximum binary size. For more information about array bounds, see Limits on Element Count and Collection Size on the ARRAY reference page.

In this example, you might make the following changes:

- Shorten the product name and vendor name fields, for example VARCHAR(32).
- Lengthen the description field, for example VARCHAR(1000).
- Accept the suggested numeric precision and scale.
- Specify a maximum number of elements for the vendors array.

With these changes, the table definition becomes:

```
=> CREATE TABLE products(
    id INT,
    msrp NUMERIC(6,2),
    name VARCHAR(32),
    description VARCHAR(1000),
    vendors ARRAY[ROW(name VARCHAR(32), price NUMERIC(6,2)),16]
    );
```

As a sanity check, start by loading the data file you used for INFER_TABLE_DDL. All rows should load successfully. The following query output has added carriage returns for readability:

```
=> COPY products FROM '/data/products/products.parquet' PARQUET;
 Rows Loaded
-------------
           1
(1 row)

=> \x
=> SELECT * FROM products;
-[ RECORD 1 ]----------------------------------------------------------------------------------------------------------------------
id          | 1064
msrp        | 329.99
name        | 4k 48in TV
description | something long goes here
vendors     | [{"name":"Amazon","price":"299.99"},{"name":"Best Buy","price":"289.99"},{"name":"Bob's Hardware","price":"309.99"}]
```

Test with data

INFER_TABLE_DDL bases its recommendations on a small sample of your data. As you load more data you might find outliers. Before you use your new table in production, test it with more data.

In the following example, a data file contains values that do not satisfy the table definition:

```
=> COPY products FROM '/data/products/products2.parquet' PARQUET;

WARNING 9738:  Some rows were rejected by the parquet parser
 Rows Loaded
-------------
           2
(1 row)

=> \x
=> SELECT * FROM products;
-[ RECORD 1 ]----------------------------------------------------------------------------------------------------------------------
id          | 1064
msrp        | 329.99
name        | 4k 48in TV
description | something long goes here
vendors     | [{"name":"Amazon","price":"299.99"},{"name":"Best Buy","price":"289.99"},{"name":"Bob's Hardware","price":"309.99"}]
-[ RECORD 2 ]----------------------------------------------------------------------------------------------------------------------
id          | 1271
msrp        | 8999.99
name        | professional kitchen appliance s
description | product description goes here
vendors     | [{"name":"Amazon","price":"8299.99"},{"name":"Designer Solutions","price":"8999.99"}]
```

If you compare the query output to the previous query, one immediate difference is the width of the name column. The value in the second row uses the full VARCHAR(32) length and appears to be truncated.

By default, COPY truncates VARCHAR values that are too long instead of rejecting them. The warning from the COPY statement indicates that in addition to the name length, there is another problem that caused data not to be loaded.

COPY can report rejected data to a (different) table . You can also direct COPY to enforce VARCHAR lengths instead of truncating:

```
=> COPY products FROM '/data/products/products2.parquet' PARQUET
ENFORCELENGTH REJECTED DATA AS TABLE products_rejected;
 Rows Loaded
-------------
           1
(1 row)
```

Note that only one row of data is loaded this time instead of two, because ENFORCELENGTH rejects the too-long string.

You can query the rejections table for more information:

```
=> \x

=> SELECT rejected_reason, rejected_data FROM products_rejected;

-[ RECORD 1 ]---+-----------------------------------------------------------------
rejected_reason | The 34-byte value is too long for type varchar(32), column 3 (name)
rejected_data   | professional kitchen appliance set
-[ RECORD 2 ]---+-----------------------------------------------------------------
rejected_reason | In column 2: Value (11999.99) exceeds range of type NUMERIC(6,2)
rejected_data   | 11999.99
```

The first rejection is caused by the long string. The second is caused by a price that exeeds the defined scale. You can increase both using ALTER TABLE:

```
=> ALTER TABLE products ALTER COLUMN name SET DATA TYPE VARCHAR(50);

=> ALTER TABLE products ALTER COLUMN msrp SET DATA TYPE NUMERIC(8,2);
```

With these changes, the data file now loads without errors:

```
=> TRUNCATE TABLE products;

=> DROP TABLE products_rejected;

=> COPY products FROM '/data/products/products2.parquet' PARQUET
ENFORCELENGTH REJECTED DATA AS TABLE products_rejected;

 Rows Loaded
-------------
           3
(1 row)

=> \x
=> SELECT * FROM products;
-[ RECORD 1 ]----------------------------------------------------------------------------------------------------------------
id          | 1064
msrp        | 329.99
name        | 4k 48in TV
description | something long goes here
vendors     | [{"name":"Amazon","price":"299.99"},{"name":"Best Buy","price":"289.99"},{"name":"Bob's Hardware","price":"309.99"}]
-[ RECORD 2 ]----------------------------------------------------------------------------------------------------------------
id          | 1183
msrp        | 11999.99
name        | home theatre system
description | product description...
vendors     | [{"name":"Amazon","price":"8949.99"},{"name":"Bob's Hardware","price":"9999.99"}]
-[ RECORD 3 ]----------------------------------------------------------------------------------------------------------------
id          | 1271
msrp        | 8999.99
name        | professional kitchen appliance set
description | product description goes here
vendors     | [{"name":"Amazon","price":"8299.99"},{"name":"Designer Solutions","price":"8999.99"}]

=> SELECT rejected_reason, rejected_data FROM products_rejected;
(No rows)
```

Continue to load more data using the ENFORCELENGTH and REJECTED DATA options, adjusting your table definition as needed, until the rate of new problems drops to an acceptable level. Most data sets of any size contain some anomalies that will be rejected. You can use the REJECTMAX option to set a limit on rejections per data load; if the limit is exceeded, COPY aborts the load.

Data formats

INFER_TABLE_DDL supports data in Parquet, ORC, Avro, and JSON formats. The first three formats enforce a schema on each file, but JSON data can contain any mix of fields and data types. When inspecting JSON data, INFER_TABLE_DDL can report multiple options. The iterative process of loading samples, examining the results, and adjusting the table definition is the same, but some details are different for JSON.

For data in other formats, including delimited text data, you cannot use INFER_TABLE_DDL. You can examine sample data and create your own initial table definition, and then test iteratively. Alternatively, you can use a flex table to explore the data.

Flex tables

Flex tables are designed for data with an unknown or varying schema. While flex tables generally do not meet the performance needs of a production system, you can use them to explore data and develop a table definition. You can use flex tables with all data formats, including delimited (CSV).

A flex table has a special column, named __raw__, that contains all fields from data you load into it. Each row is independent and might have different fields in this column. You can use flex functions to explore this data and materialize individual columns in your table. When you have identified the columns that should consistently be part of your table, you can then define a regular (non-flex) table.

You can iterate by loading and exploring small amounts of data, as described in Inference. However, flex tables allow you to load larger amounts of data without producing errors and rejections, so you might choose to start with a larger sample.

Column discovery

The simplest flex table does not specify any columns. In this example, the first thousand rows are pulled from a much larger data set:

```
=> CREATE FLEX TABLE reviews_flex();

=> COPY reviews_flex FROM '/home/data/reviews/sample_1000.json' PARSER FJSONPARSER();
 Rows Loaded
-------------
        1000
(1 row)
```

The COMPUTE_FLEXTABLE_KEYS function extrapolates field names and candidate data types from the raw data. The more data you have loaded the more work this function needs to do and the longer it takes to finish. If the function runs slowly for you, try a smaller sample size.

```
=> SELECT COMPUTE_FLEXTABLE_KEYS('reviews_flex');
             COMPUTE_FLEXTABLE_KEYS
---------------------------------------------------
 Please see public.reviews_flex_keys for updated keys
(1 row)

SELECT * FROM reviews_flex_keys;
  key_name   | frequency | data_type_guess
-------------+-----------+-----------------
 user_id     |      1000 | Varchar(44)
 useful      |      1000 | Integer
 text        |      1000 | Varchar(9878)
 stars       |      1000 | Numeric(5,2)
 review_id   |      1000 | Varchar(44)
 funny       |      1000 | Integer
 date        |      1000 | Timestamp
 cool        |      1000 | Integer
 business_id |      1000 | Varchar(44)
(9 rows)
```

In this example, all 1000 rows have all of the fields. This suggests that the larger data set is probably consistent, particularly if the sample was taken from the middle. In this case, the data was exported from another database. Had the data been more varied, you would see different numbers of occurrences in the keys table.

COMPUTE_FLEXTABLE_KEYS, unlike INFER_TABLE_DDL, proposes lengths for VARCHAR values and precision and scale for NUMERIC values. The function uses the largest values found in the data in the table. Future data could exceed these limits, so make sure you are using representative data or a large-enough sample.

If you query a flex table using SELECT *, the results include the raw column. This column is in a binary format, so querying it is not especially useful. To see a sample of the data, query the fields explicitly, as if they were columns. The following example omits the text of the review, which COMPUTE_FLEXTABLE_KEYS reported as potentially quite large (VARCHAR(9878)):

```
=> \x
SELECT user_id, useful, stars, review_id, funny, date, cool, business_id
FROM reviews_flex ORDER BY date DESC LIMIT 3;
-[ RECORD 1 ]----------------------
user_id     | mY8vqRndQ_jHeI-kUZc-rw
useful      | 0
stars       | 5.0
review_id   | viMRTXxVGZa_E0J5l3fD5A
funny       | 0
date        | 2021-01-28 01:25:04
cool        | 0
business_id | LsR1rENf9596Oo07QFjtaQ
-[ RECORD 2 ]----------------------
user_id     | vLKy-xJoyeSAVhPbG8-Pfg
useful      | 0
stars       | 5.0
review_id   | 8Bcq-HIlpdDJ773AkNAUeg
funny       | 0
date        | 2021-01-27 16:08:53
cool        | 0
business_id | ljNN92X-WRVzhkHeIgnIlA
-[ RECORD 3 ]----------------------
user_id     | yixdPMh2UidS3qOquE1Yow
useful      | 0
stars       | 5.0
review_id   | h_z6g8lOrvUgsLg-bklxVg
funny       | 0
date        | 2021-01-27 15:58:42
cool        | 0
business_id | WlTPQwhKyEDogIXQfP0uYA
```

This sample shows several integer fields with values of 0. These fields (useful, funny, and cool) appear to be counters indicating how many people found the review to have those qualities. To see how often they are used, which might help you decide whether to include them in your table, you can use the COUNT function:

```
=> SELECT COUNT(useful) FROM reviews_flex WHERE useful > 0;
 COUNT
-------
   548
(1 row)

=> SELECT COUNT(funny) FROM reviews_flex WHERE funny > 0;
 COUNT
-------
   234
(1 row)

=> SELECT COUNT(cool) FROM reviews_flex WHERE cool > 0;
 COUNT
-------
   296
(1 row)
```

All of the fields in this data except text are fairly small. How much of the review text you want to include depends on how you plan to use the data. If you plan to analyze the text, such as to perform sentiment analysis, you might set a larger size. If you don't plan to use the text, you might either ignore the text entirely or set a small size and allow the data to be truncated on load:

```
=> CREATE TABLE reviews(
   review_id VARCHAR(44), business_id VARCHAR(44), user_id VARCHAR(44),
   stars NUMERIC(5,2), date TIMESTAMP,
   useful INT, funny INT, cool INT);

=> COPY reviews FROM '/home/data/reviews/reviews_set1.json' PARSER FJSONPARSER() NO COMMIT;
WARNING 10596:  Warning in UDx call in user-defined object [FJSONParser], code:
0, message: Data source contained keys which did not match table schema
HINT:  SELECT key, sum(num_instances) FROM v_monitor.udx_events WHERE event_type
 = 'UNMATCHED_KEY' GROUP BY key
 Rows Loaded
-------------
       10000
(1 row)
```

The JSON parser warns about keys found in the data that are not part of the table definition. Some other parsers require you to consume all of the fields. Consult the reference pages for specific parsers for more information about whether and how you can ignore fields.

To find out what unmatched fields appeared in the data, query the UDX_EVENTS system table:

```
=> SELECT key, SUM(num_instances) FROM UDX_EVENTS
WHERE event_type='UNMATCHED_KEY' AND transaction_id = CURRENT_TRANS_ID()
GROUP BY key;
 key  |  SUM
------+-------
 text | 10000
(1 row)
```

The UDX_EVENTS table shows that the only unmatched field is the previously-seen text field. You can safely commit the load transaction. When you have tested with enough data to feel confident, you can suppress the parser warning as explained on FJSONPARSER .

Complex types

Consider the following JSON data in a flex table:

```
=> CREATE FLEX TABLE products_flex();
COPY products_flex FROM '/home/data/products/sample.json' PARSER FJSONPARSER();
 Rows Loaded
-------------
           3
(1 row)

=> SELECT COMPUTE_FLEXTABLE_KEYS('products_flex');
                COMPUTE_FLEXTABLE_KEYS
-------------------------------------------------------------
 Please see public.products_flex_j_keys for updated keys
(1 row)

=> SELECT * FROM products_flex_keys;
  key_name   | frequency |   data_type_guess
-------------+-----------+---------------------
 description |         3 | Varchar(58)
 id          |         3 | Integer
 msrp        |         3 | Numeric(10,3)
 name        |         3 | Varchar(68)
 vendors     |         3 | long varbinary(298)
(5 rows)
```

The vendors field has a type of LONG VARBINARY, but the name of the column implies a collection. You can use the MAPTOSTRING function to sample the data:

```
=> SELECT MAPTOSTRING(__raw__) FROM products_flex LIMIT 1;

                                    MAPTOSTRING
-------------------------------------------------------------------------------
{
        "description": "something long goes here",
        "id": "1064",
        "msrp": "329.99",
        "name": "4k 48in TV",
        "vendors": {
                "0.name": "Amazon",
                "0.price": "299.99",
                "1.name": "Best Buy",
                "1.price": "289.99",
                "2.name": "Bob's Hardware",
                "2.price": "309.99"
        }
}
(1 row)
```

By default, the JSON parser "flattens" JSON maps (key/value pairs). In this example, the value of the vendors field is presented as a set of six key/value pairs, but the key names (*number* .name and *number* .price) indicate that the data is more structured. Disabling this flattening makes the true structure of the JSON data more clear:

```
=> TRUNCATE TABLE products_flex;

=> COPY products_flex FROM '/home/data/products/sample.json'
PARSER FJSONPARSER(flatten_maps=false);
 Rows Loaded
-------------
           3
(1 row)

=> SELECT MAPTOSTRING(__raw__) FROM products_flex LIMIT 1;

                                    MAPTOSTRING
-------------------------------------------------------------------------------
{
        "description": "something long goes here",
        "id": "1064",
        "msrp": "329.99",
        "name": "4k 48in TV",
        "vendors": {
                "0": {
                        "name": "Amazon",
                        "price": "299.99"
                },
                "1": {
                        "name": "Best Buy",
                        "price": "289.99"
                },
                "2": {
                        "name": "Bob's Hardware",
                        "price": "309.99"
                }
        }
}
(1 row)
```

This data indicates that the vendors field holds an array of structs with two fields, name and price. Using this information and the computed keys for the other fields in the data, you can define a table with strongly-typed columns:

```
=> CREATE TABLE products(
   id INT, name VARCHAR(100), msrp NUMERIC(8,2), description VARCHAR(1000),
   vendors ARRAY[ROW(name VARCHAR(50), price NUMERIC(8,2)),16]);
```

After loading data, you can use an array function to confirm that the collection was loaded correctly:

```
=> COPY products FROM '/home/data/products/sample.json'
PARSER FJSONPARSER(flatten_maps=false);
 Rows Loaded
-------------
           3
(1 row)

=> SELECT EXPLODE(vendors) OVER() FROM products;
 position |                        value
----------+-----------------------------------------------------
        0 | {"name":"Amazon","price":"299.99"}
        1 | {"name":"Best Buy","price":"289.99"}
        2 | {"name":"Bob's Hardware","price":"309.99"}
        0 | {"name":"Amazon","price":"8949.99"}
        1 | {"name":"Bob's Hardware","price":"9999.99"}
        0 | {"name":"Amazon","price":"8299.99"}
        1 | {"name":"Designer Solutions","price":"8999.99"}
(7 rows)
```

Handling limitations of some file formats

The JSON format supports arrays, structs, and combinations of the two. Some formats are more limited. Delimited (CSV) data can represent arrays of scalar types, but not structs. Further, not all parsers that you can use with flex tables support complex types.

Consider the following delimited data:

```
id|msrp|name|vendors|prices
1064|329.99|4k 48in TV|[Amazon,Best Buy]|[299.99,289.99]
1183|11999.99|home theatre system|[Amazon,Bob's Hardware]|[8949.99,9999.99]
1271|8999.99|professional kitchen appliance set|[Amazon,Designer Solutions]|[829
9.99,8999.99]
```

You can load this data into a flex table using FDELIMITEDPARSER , but the parser misinterprets the arrays:

```
=> CREATE FLEX TABLE catalog_flex();

=> COPY catalog_flex FROM '/home/data/catalog/sample.csv' PARSER FDELIMITEDPARSER();
 Rows Loaded
-------------
           3
(1 row)

=> SELECT MAPTOSTRING(__raw__) FROM catalog_flex LIMIT 1;
                                        MAPTOSTRING
-------------------------------------------------------------------------------------
 {
        "id": "1064",
        "msrp": "329.99",
        "name": "4k 48in TV",
        "prices": "[299.99,289.99]",
        "vendors": "[Amazon,Best Buy]"
}
(1 row)

=> SELECT COMPUTE_FLEXTABLE_KEYS('catalog_flex');
               COMPUTE_FLEXTABLE_KEYS
-----------------------------------------------------
 Please see public.catalog_flex_keys for updated keys
(1 row)

=> SELECT * FROM catalog_flex_keys;
 key_name | frequency | data_type_guess
----------+-----------+-----------------
 id       |         3 | Integer
 msrp     |         3 | Numeric(10,3)
 name     |         3 | Varchar(68)
 prices   |         3 | Interval
 vendors  |         3 | Varchar(54)
(5 rows)
```

The parser has mistaken the arrays as strings, and COMPUTE_FLEXTABLE_KEYS has mistaken an array of numbers for an INTERVAL and an array of strings as a VARCHAR. These are limitations of flex tables used with certain file formats, not of Vertica in general. If you see unusual results, try adding real columns to the flex table:

```
=> CREATE FLEX TABLE catalog_flex(
        vendors ARRAY[VARCHAR(50), 10],
        prices ARRAY[NUMERIC(10,2), 10] );
WARNING 5727:  Sort clause contains a Array attribute, catalog_flex.prices - data loads may be slowed significantly
HINT:  Consider using a non-long type column for sorting or adding at least one such column to the table
```

For data exploration you need not be concerned with warnings such as this one. When you are ready to define a production table, heed any warnings you receive.

After loading the data, you can confirm that the arrays were loaded correctly:

```
=> SELECT EXPLODE(vendors, prices USING PARAMETERS explode_count=2) OVER()
AS (v_idx,vendor,p_idx,price)
FROM catalog_flex;
 v_idx |       vendor       | p_idx |  price
-------+--------------------+-------+---------
     0 | Amazon             |     0 |  299.99
     0 | Amazon             |     1 |  289.99
     1 | Best Buy           |     0 |  299.99
     1 | Best Buy           |     1 |  289.99
     0 | Amazon             |     0 | 8949.99
     0 | Amazon             |     1 | 9999.99
     1 | Bob's Hardware     |     0 | 8949.99
     1 | Bob's Hardware     |     1 | 9999.99
     0 | Amazon             |     0 | 8299.99
     0 | Amazon             |     1 | 8999.99
     1 | Designer Solutions |     0 | 8299.99
     1 | Designer Solutions |     1 | 8999.99
(12 rows)
```

Create table definition

You can continue to use flex tables to explore data, creating real columns for complex types as you encounter them. FDELIMITEDPARSER is specific to flex tables; for native tables (created with CREATE TABLE), DELIMITED is the default parser:

```
=> CREATE TABLE catalog(
  id INT, msrp NUMERIC(8,2), name VARCHAR(100),
  vendors ARRAY[VARCHAR(50), 10], prices ARRAY[NUMERIC(10,2), 10] );

=> COPY catalog FROM '/home/data/catalog/sample.csv';
 Rows Loaded
-------------
           3
(1 row)

=> SELECT * FROM catalog;
  id  |   msrp   |               name               |            vendors            |        prices
------+----------+----------------------------------+-------------------------------+----------------------
 1064 |   329.99 | 4k 48in TV                       | ["Amazon","Best Buy"]         | ["299.99","289.99"]
 1183 | 11999.99 | home theatre system              | ["Amazon","Bob's Hardware"]   | ["8949.99","9999.99"]
 1271 |  8999.99 | professional kitchen appliance set | ["Amazon","Designer Solutions"] | ["8299.99","8999.99"]
(3 rows)
```

Table types

Vertica has several types of tables. During initial exploration, your focus should be on whatever is most expedient for you; you will modify or replace the tables you build at this stage before you put them in production. As you get closer to production-ready definitions, pay more attention to the specific considerations for each table type.

Native (ROS) tables

Native tables store their data in ROS format in the database. After you have finalized table definitions, you can optimize the performance of your most important or frequent queries by creating projections.

If you are exploring data that is in multiple locations, using a native table for exploration can be easier than using external tables. With a native table, you can load data from many sources into the database. If multiple people are exploring the same data together, you do not need to grant additional permissions for an external file system or object store. You only need one person to have permission to read the files being loaded.

On the other hand, if you make a change to a native table definition that is not compatible with existing data, the change fails:

```
=> SELECT name FROM catalog;
               name
-------------------------------------
 4k 48in TV
 home theatre system
 professional kitchen appliance set
(3 rows)

=> ALTER TABLE catalog ALTER COLUMN name SET DATA TYPE VARCHAR(10);
ROLLBACK 2378:  Cannot convert column "name" to type "varchar(10)"
HINT:  Verify that the data in the column conforms to the new type
```

To make an incompatible change, you can create a new table and then use INSERT with SELECT to transform and copy existing data to the new table. For example, you can cast existing data to new types:

```
=> CREATE TABLE catalog2(
  id INT, msrp NUMERIC(8,2), name VARCHAR(10),
  vendors ARRAY[VARCHAR(50), 2],
  prices ARRAY[NUMERIC(10,2), 2] );

--- cannot directly insert:
=> INSERT INTO catalog2 SELECT * FROM catalog;
ERROR 4800:  String of 19 octets is too long for type Varchar(10)

-- violating array bounds is also an error:
=> INSERT INTO catalog2
SELECT id, msrp, name::VARCHAR(10), vendors, prices FROM catalog;
ERROR 9227:  Output array isn't big enough
DETAIL:  Type limit is 2 elements, but value has 3 elements

--- cast to truncate:
=> INSERT INTO catalog2
SELECT id, msrp, name::VARCHAR(10),
       vendors::ARRAY[VARCHAR(50), 2],
       prices::ARRAY[NUMERIC(10,2), 2] FROM catalog;
 OUTPUT
--------
      3
(1 row)

=> SELECT * FROM catalog2;
  id  |   msrp   |    name    |            vendors             |        prices

------+----------+------------+--------------------------------+----------------------
 1064 |   329.99 | 4k 48in TV | ["Amazon","Best Buy"]          | ["299.99","289.99"]
 1183 | 11999.99 | home theat | ["Amazon","Bob's Hardware"]    | ["8949.99","9999.99"]
 1271 |  8999.99 | profession | ["Amazon","Designer Solutions"] | ["8299.99","8999.99"]
(3 rows)
```

External tables

With an external table, Vertica reads data at query time instead of storing it in the database. External tables are a good choice when you do not need the performance of native tables, for example for older data or partitions that you query less frequently.

External tables are particularly useful when the external data is growing or changing, because you do not need to reload data yourself. However, this means you discover structural changes such as changed column types or new columns at query time rather than at load time.

You can drop an external table and recreate it to change its definition. Because the database does not store the data, replacing a table is not a destructive operation. During data exploration, you can treat external tables as lightweight and disposable.

Because the data is read at query time, you must ensure that your users have and retain permission to read the data in its original location. Depending on where the data is stored, you might need to take additional steps to manage access, such as creating AWS IAM roles on S3.

For more information, see How external tables differ from native tables .

Flex tables

Flex tables are designed for data with an unknown or varying schema. They are a good choice for exploring data with varying fields or columns. JSON and delimited formats, in particular, do not embed a schema, so these formats are particularly prone to variation.

You can use flex tables as either native tables (CREATE FLEX TABLE) or external tables (CREATE FLEX EXTERNAL TABLE).

Flex tables trade flexibility for reduced performance and are not a good choice for production environments. You can use flex tables to explore data and then create standard (non-flex) native or external tables to deploy.

Schema changes

After you develop a schema for a table and continue to load data, you might find that some new data diverges from that schema. Sometimes new fields appear, and sometimes fields change data types or stop appearing. Depending on how volatile you expect your data to be, you might decide to load what you can and ignore the rest, or you might decide to update your table definition to handle the changes.

Typically, COPY rejects data that does not match the definition and loads the rest. You can monitor these rejections and adjust your table definition, as described in Test with Data . Parsers for some formats have additional considerations:

- ORC: if the table definition does not include all columns in the table, the load aborts with an error.
- Parquet: by default, the table definition must include all columns as for ORC. Alternatively, you can use loose schema matching to load only columns that are part of the table definition, ignoring others. Type mismatches that cannot be coerced are errors.
- JSON and Avro: the parser loads only the columns that are part of the table definition. If the data contains other columns, the parsers emit warnings. Type mismatches that cannot be coerced are errors.

Parquet

If you load Parquet data that contains extra columns, by default the load fails with an error:

```
=> CREATE TABLE orders_summary(orderid INT, accountid INT);

=> COPY orders_summary FROM '/data/orders.parquet' PARQUET;
ERROR 9135:  Attempt to load 2 columns from a parquet source [/data/orders.parquet] that has 3 columns
```

If you do not know what the extra columns are, you can use inference to inspect a sample file. If the new fields are important, you can use ALTER TABLE to add the new columns to the table definition and then load the data.

If the new fields are not important and you want to ignore them, you can use loose schema matching. Loose schema matching means that the parser loads columns whose names match the table definition and ignores any others. Enable this with the do_soft_schema_match_by_name parameter in the PARQUET parser:

```
=> COPY orders_summary FROM '/data/orders.parquet'
   PARQUET(do_soft_schema_match_by_name='true');
 Rows Loaded
-------------
           3
(1 row)
```

Loose schema matching allows you to ignore extra data. Missing data and mismatched data types are still errors. If loads fail with these errors, you can use inference to inspect the data and then alter your table definition.

JSON and Avro

By default, the JSON and Avro parsers skip fields in the data that are not part of the table definition and proceed with the rest of the load. This means that the JSON and Avro parsers are more flexible in the face of variability in the data, but also that you must monitor your loads more closely for new fields.

When creating tables that will use JSON or Avro data, a good first step is to infer a table definition from a sample file:

```
=> SELECT INFER_TABLE_DDL ('/data/rest1.json'
   USING PARAMETERS table_name='restaurants', format='json');
WARNING 9311:  This generated statement contains one or more varchar/varbinary columns which default to length 80
...

=> CREATE TABLE restaurants(
       name VARCHAR(50),
       cuisine VARCHAR(25),
       location ARRAY[ROW(city VARCHAR(20), state VARCHAR(2)),50],
       hours ROW(open TIME, close TIME),
       menu ARRAY[ROW(item VARCHAR(50), price NUMERIC(8,2)),100]);

=> COPY restaurants FROM '/data/rest1.json' PARSER FJSONPARSER();
 Rows Loaded
-------------
           2
(1 row)
```

A second data file has some new fields. These parsers load what can be loaded and warn about the new data:

```
=> COPY restaurants FROM '/data/rest2.json' PARSER FJSONPARSER() NO COMMIT;
WARNING 10596:  Warning in UDx call in user-defined object [FJSONParser], code: 0, message:
Data source contained keys which did not match table schema
HINT:  SELECT key, sum(num_instances) FROM v_monitor.udx_events WHERE event_type = 'UNMATCHED_KEY' GROUP BY key
 Rows Loaded
-------------
           2
(1 row)
```

Loads that use the NO COMMIT option, as in this example, can recover from errors. Without NO COMMIT, COPY loads the data, skipping the new fields. If you change your table definition to handle the new fields and repeat the load, the table has two rows for each record, one with the new fields and one without. Use NO COMMIT to experiment, and use ROLLBACK to recover.

New fields are logged in the UDX_EVENTS system table:

```
--- from the HINT:
=> SELECT key, SUM(num_instances) FROM v_monitor.UDX_EVENTS
   WHERE event_type = 'UNMATCHED_KEY' AND transaction_id=CURRENT_TRANS_ID()
   GROUP BY key;
          key           | SUM
------------------------+-----
 chain                  |   1
 menu.elements.calories |   7
(2 rows)

=> ROLLBACK;
```

The data file has two new fields, one at the top level and one in the existing menu complex-type column. To find out their data types, you can inspect the file or use INFER_TABLE_DDL:

```
=> SELECT INFER_TABLE_DDL ('/data/rest2.json'
   USING PARAMETERS table_name='restaurants', format='json');
WARNING 9311:  This generated statement contains one or more varchar/varbinary columns which default to length 80
                        INFER_TABLE_DDL
-----------------------------------------------------------------------------------------------------------------------
 Candidate matched 1/1 of total files(s):
create table "restaurants"(
  "chain" bool,
  "cuisine" varchar,
  "hours" Row(
    "close" time,
    "open" time
  ),
  "location" Array[Row(
    "city" varchar,
    "state" varchar
  )],
  "menu" Array[Row(
    "calories" int,
    "item" varchar,
    "price" numeric(precision, scale)
  )],
  "name" varchar
);

(1 row)
```

You can use ALTER TABLE to add the new column and alter the menu column. With these changes, the data can be loaded without errors:

```
=> ALTER TABLE restaurants ADD COLUMN chain BOOLEAN;

=> ALTER TABLE restaurants ALTER COLUMN menu
SET DATA TYPE ARRAY[ROW(item VARCHAR(50), price NUMERIC(8,2), calories INT),100];

--- repeat the load:
COPY restaurants FROM '/data/rest2.json' PARSER FJSONPARSER();
 Rows Loaded
-------------
           2
(1 row)
```

ALTER TABLE affects future loads. Data already in the table has null values for the added column and field:

```
=> SELECT EXPLODE(name, chain, menu) OVER() FROM restaurants;
     name     | chain | position |                          value
--------------+-------+----------+-------------------------------------------------------------
 Pizza House  |       |        0 | {"item":"cheese pizza","price":"7.99","calories":null}
 Pizza House  |       |        1 | {"item":"spinach pizza","price":"8.99","calories":null}
 Pizza House  |       |        2 | {"item":"garlic bread","price":"4.99","calories":null}
 Sushi World  |       |        0 | {"item":"maki platter","price":"21.95","calories":null}
 Sushi World  |       |        1 | {"item":"tuna roll","price":"4.95","calories":null}
 Greasy Spoon | f     |        0 | {"item":"scrambled eggs","price":"3.99","calories":350}
 Greasy Spoon | f     |        1 | {"item":"grilled cheese","price":"3.95","calories":500}
 Greasy Spoon | f     |        2 | {"item":"tuna melt","price":"5.95","calories":600}
 Greasy Spoon | f     |        3 | {"item":"french fries","price":"1.99","calories":350}
 Pasta World  | t     |        0 | {"item":"cheese pizza","price":"7.99","calories":1200}
 Pasta World  | t     |        1 | {"item":"spinach pizza","price":"8.99","calories":900}
 Pasta World  | t     |        2 | {"item":"garlic bread","price":"4.99","calories":250}
(12 rows)
```

If you are only interested in certain fields and don't care about new fields found in data loads, you can use the suppress_warnings parameter on the JSON or Avro parser to ignore them. If you suppress warnings, you can separately check for new fields from time to time with INFER_TABLE_DDL or by loading sample data with warnings and NO COMMIT.

Data load

Vertica provides many ways to read data. You can load data into the database from a variety of sources, optionally transforming it in various ways. You can read data in place in its original format using external tables. You can use streaming, and you can import data from other Vertica databases. See Common use cases for an introduction.

Most data-loading operations, including external tables, revolve around the COPY statement, which has many options. This book focuses on COPY-based reads (data load and external tables). Other data-loading options supported by Vertica are described elsewhere:

- Apache Kafka integration
- Apache Spark integration
- Using the HCatalog Connector (HCatalog is part of HDFS)
- Database export and import
- Clients: Batch inserts using JDBC prepared statements , Using batch inserts (ODBC)
- Inserting individual rows and queried data into a table: INSERT/INSERT...SELECT

In this section

- Common use cases
- Introduction to the COPY statement
- Global and column-specific options
- Specifying where to load data from
- Data formats
- Complex types
- Schema evolution
- Handling Non-UTF-8 input
- Transforming data during loads
- Distributing a load
- Using transactions to stage a load
- Handling messy data
- Monitoring COPY loads and metrics
- Using load scripts
- Troubleshooting data loads
- Working with external data

Common use cases

Vertica supports a variety of use cases for reading data. Some of the most common are summarized here with links for more information. This is not a complete list of capabilities.

The COPY statement is central to loading data. See Introduction to the COPY statement for an overview of its use.

Loading data from files

You might have data, perhaps quite a bit of data, that you want to load into Vertica. These files might reside on shared storage, in the cloud, or on local nodes, and might be in a variety of formats.

For information about source locations, see Specifying where to load data from . To handle data in different formats you specify a *parser* ; for more information about the options, see Data formats .

You are not limited to loading data "as-is"; you can also transform it during load. See Transforming data during loads .

Loading data from other services

Apache Kafka is a platform for streaming data. Vertica supports streaming data to and from Kafka. See Apache Kafka integration .

Apache Spark is a cluster-computing framework for distributed data. Vertica supports connecting to Spark for data. See Apache Spark integration .

You can copy data directly from another Vertica cluster, instead of exporting to files and then loading those files. See Database export and import .

Read data where it exists (don't import)

Instead of loading data into Vertica, you can read it in place using external tables. External tables can be advantageous in the following cases:

- If you want to explore data, such as in a data lake, before selecting data to load into Vertica.
- If you are one of several consumers sharing the same data, for example in a data lake, then reading it in place eliminates concerns about whether query results are up to date. There's only one copy, so all consumers see the same data.
- If your data changes rapidly but you do not want to stream it into Vertica, you can instead query the latest updates automatically.
- If you have lower-priority data in Vertica that you still want to be able to query.

When you query an external table, Vertica loads the data it needs from the external source. The Parquet and ORC columnar formats are optimized for this kind of load, so using external tables does not necessarily have a large effect on performance compared to loading data into Vertica native tables.

For more information about using external tables, see Working with external data .

Complex types

Some data formats support complex types such as arrays and structs (sets of property-value pairs). You can use strong typing to define columns using the ARRAY and ROW types (including nesting) in native and external tables. See Complex types . Alternatively, you can define tables using flexible (schemaless) complex types without fully specifying them. You can load flexible complex types in the Parquet, ORC, JSON, and Avro formats; see Flexible complex types .

Unknown or evolving schema

Sometimes the schema for the data you want to load is unknown or changes over time, particularly with JSON data which does not embed a schema in data files. There are two primary ways to explore data with an unknown schema, explained in detail in Data exploration :

- You can inspect sample data files and derive an initial table definition using the INFER_TABLE_DDL function. For tables with many or complex columns, using this function can save time and reduce manual effort.
- You can use flex tables to load schemaless data as blobs and inspect it using flex functions. Flex tables are best used for initial exploration of heterogeneous or poly-structured data, not in production databases, because the query-time extraction affects performance.

If your data changes over time, some parsers emit warnings about mismatches or new columns. You can use ALTER TABLE to modify the table definition as the data evolves. For more information and an example, see Schema evolution .

Messy data

Sometimes data is not clean; values might not match the declared data types, or required values might be missing, or the parser might not be able to interpret a row for other reasons. You might still want to be able to load and explore this data. You can specify how error-tolerant to be and where to record information about rejected data using parameters to the COPY statement. For more information, see Handling messy data .

Introduction to the COPY statement

Use the the COPY statement to load data. COPY is a large and versatile statement with many parameters; for all of the details, see the reference page. In its simplest form, COPY copies data from a source to a table, as follows:

```
=> COPY target-table FROM data-source;
```

You also use COPY when defining an external table:

```
=> CREATE EXTERNAL TABLE target-table (...) AS COPY FROM data-source;
```

Source data can be a data stream or a file path. For more about the FROM clause, see Specifying where to load data from .

You can specify many details about a data load, including:

- Global and column-specific options
- Data formats and compression
- Which built-in parser to use, or which user-defined source, filters, or parser to use
- How to distribute the data load among database nodes (Distributing a load)
- How to transform data during loading (Transforming data during loads)
- What to do with data that could not be loaded (Handling messy data)

For a complete list of parameters, see Parameters .

Permissions

Generally, only a superuser can use the COPY statement to bulk-load data. Non-supersuers can use COPY in certain cases:

- To load from a stream on the host (such as STDIN) rather than a file (see Streaming data via JDBC).
- To load with the FROM LOCAL option.
- To load into a storage location where the user has been granted permission.
- To use a user-defined-load function for which the user has permission.

A non-superuser can also perform a batch load with a JDBC prepared statement , which invokes COPY to load data as a background task.

Users must also have read permission to the source from which the data is to be loaded.

Global and column-specific options

You can specify some COPY options globally, for the entire COPY statement, or limit their scope to a column. For example, in the following COPY statement, the first column is delimited by '|' but the others are delimited by commas.

```
=> COPY employees(id DELIMITER '|', name, department) FROM ... DELIMITER ',';
```

You could specify a different default for null inputs for one column:

```
=> COPY employees(id, name, department NULL 'General Admin') FROM ... ;
```

Alternatively, you can use the COLUMN OPTION parameter to specify column-specific parameters instead of enumerating the columns:

```
=> COPY employees COLUMN OPTION (department NULL 'General Admin') FROM ... ;
```

Where both global and column-specific values are provided for the same parameter, the column-specific value governs those columns and the global one governs others.

All parameters can be used globally. The description of each parameter indicates whether it can be restricted to specific columns.

Specifying where to load data from

Each COPY statement requires either a FROM clause to indicate the location of the file or files being loaded or a SOURCE clause when using a user-defined source. For more about the SOURCE clause, see Parameters . This section covers use of the FROM clause.

Loading from a specific path

Use the *path-to-data* argument to indicate the location of one or more files to load. You can load data from the following locations:

- The local file system.
- NFS, through a mount point on the local file system.
- S3 object store , Google Cloud Storage (GCS) object store , and Azure Blob Storage object store . If a data file you want to read resides in an S3 bucket that uses server-side encryption, you might need to set additional parameters .
- HDFS file system . If a data file you want to read resides on an HDFS cluster that uses Kerberos authentication, Vertica uses the current user's principal, session doAs user, or session delegation token. See Accessing kerberized HDFS data for more information about these options.

If the path is a URL, you must use URL encoding for the '%' character. Otherwise, URL encoding ('%NN' where NN is a two-digit hexadecimal number) is permitted but not required.

When copying from the local file system, the COPY statement expects to find files in the same location on every node that participates in the query. If you are using NFS, then you can create an NFS mount point on each node. Doing so allows all database nodes to participate in the load for better performance without requiring files to be copied to all nodes.

Treat NFS mount points as local files in paths:

```
=> COPY sales FROM '/mount/sales.dat' ON ANY NODE;
```

You can specify more than one path in the same COPY statement, as in the following example.

```
=> COPY myTable FROM 'webhdfs:///data/sales/01/*.dat', 'webhdfs:///data/sales/02/*.dat',
   'webhdfs:///data/sales/historical.dat';
```

For files in HDFS, you can specify a name service (hadoopNS). In this example, the COPY statement is part of the definition of an external table:

```
=> CREATE EXTERNAL TABLE users (id INT, name VARCHAR(20))
   AS COPY FROM 'webhdfs://hadoopNS/data/users.csv';
```

If *path-to-data* resolves to a storage location on a local file system, and the user invoking COPY is not a superuser, these permissions are required:

- The storage location must have been created with the USER option (see CREATE LOCATION).
- The user must already have been granted READ access to the storage location where the file or files exist, as described in GRANT (storage location) .

Vertica prevents symbolic links from allowing unauthorized access.

Loading with wildcards (glob)

You can invoke COPY for a large number of files in a shared directory with a single statement such as:

```
=> COPY myTable FROM '/data/manyfiles/*.dat' ON ANY NODE;
```

The glob (*) must indicate a set of files, not directories. The following statement fails if /data/manyfiles contains any subdirectories:

```
=> COPY myTable FROM '/data/manyfiles/*' ON ANY NODE;
```

If /data/manyfiles contains subdirectories that contain files, you can use a glob in the path for the subdirectories:

```
=> COPY myTable FROM '/data/manyfiles/*/*' ON ANY NODE;
```

Sometimes a directory structure partitions the data, as in the following example:

```
path/created=2016-11-01/*
path/created=2016-11-02/*
path/created=2016-11-03/*
path/...
```

You still use a glob to read the data, but you can also read the partition values themselves (creation date, in this case) as table columns. See Partitioned file paths .

Using a wildcard with the ON ANY NODE clause expands the file list on the initiator node. This command then distributes the individual files among all nodes, so that the COPY workload is evenly distributed across the entire cluster. ON ANY NODE is the default for all file systems other than Linux.

Loading from a Vertica client

Use COPY LOCAL to load files on a client system to the Vertica database. For example, to copy a GZIP file from your local client, use a command such as this:

```
=> COPY store.store_dimension FROM LOCAL '/usr/files/my_data/input_file' GZIP;
```

You can use a comma-separated list to load multiple files of the same compression type. COPY LOCAL then concatenates the files into a single file, so you cannot combine files with different compression types in the list. When listing multiple files, be sure to specify the type of every input file, such as BZIP, as shown:

```
=>COPY simple_table FROM LOCAL 'input_file.bz' BZIP, 'input_file.bz' BZIP;
```

You can load data from a local client from STDIN, as follows:

```
=> COPY simple_table FROM LOCAL STDIN;
```

Loading from Kafka or Spark

For information about streaming data from Kafka, see Apache Kafka integration .

For information about using Vertica with Spark data, see Apache Spark integration .

Loading data from an IDOL CFS client

The IDOL Connector Framework Server (CFS) VerticaIndexer feature lets CFS clients connect to your Vertica database using ODBC. After it is connected, CFS uses COPY...FROM LOCAL statements to load IDOL document metadata into an existing flex table. For more information, see the Using flex tables for IDOL data section in Using Flex Tables.

Data formats

COPY supports many data formats, detailed in the sections that follow. You specify a data format by specifying a parser.

By default, COPY uses the DELIMITED parser (Delimited data) to load raw data into the database. Raw input data must be in UTF-8, delimited text format. Other parsers support other data formats.

The syntax for specifying which parser to use varies. The description of each parser includes this information.

The same COPY statement cannot mix raw data types that require different parsers, such as NATIVE and FIXEDWIDTH . You can, however, load data of different formats, using different parsers, into the same table using separate COPY statements.

For information about verifying input data formats, see Handling Non-UTF-8 input .

All parsers described in this section can be used with conventional tables (those created with CREATE TABLE or CREATE EXTERNAL TABLE). Some also support Flex tables (CREATE FLEX TABLE). For more information specific to Flex tables, see Using flex table parsers .

All parsers support all primitive data types and several support one-dimensional arrays of primitive types. Some parsers support other complex types. See the documentation of individual parsers for information about supported types.

In this section

- Delimited data
- Binary (native) data
- Native varchar data
- Fixed-width format data
- ORC data
- Parquet data
- JSON data
- Avro data
- Matches from regular expressions
- Common event format (CEF) data

Delimited data

If you do not specify another parser, Vertica defaults to the DELIMITED parser . You can specify the delimiter, escape characters, how to handle null values, and other parameters in the COPY statement.

The following example shows the default behavior, in which the delimiter character is '|'

```
=> CREATE TABLE employees (id INT, name VARCHAR(50), department VARCHAR(50));
CREATE TABLE

=> COPY employees FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 42|Sheldon Cooper|Physics
>> 17|Howard Wolowitz|Astronomy
>> \.

=> SELECT * FROM employees;
 id |      name       |  department
----+-----------------+--------------
 17 | Howard Wolowitz | Astrophysics
 42 | Sheldon Cooper  | Physics
(2 rows)
```

By default, collection values are delimited by brackets and elements are delimited by commas. Collections must be one-dimensional arrays or sets of scalar types.

```
=> CREATE TABLE researchers (id INT, name VARCHAR, grants ARRAY[VARCHAR], values ARRAY[INT]);
CREATE TABLE

=> COPY researchers FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 42|Sheldon Cooper|[US-7376,DARPA-1567]|[65000,135000]
>> 17|Howard Wolowitz|[NASA-1683,NASA-7867,SPX-76]|[16700,85000,45000]
>> \.

=> SELECT * FROM researchers;
 id |      name       |              grants               |       values
----+-----------------+-----------------------------------+--------------------
 17 | Howard Wolowitz | ["NASA-1683","NASA-7867","SPX-76"] | [16700,85000,45000]
 42 | Sheldon Cooper  | ["US-7376","DARPA-1567"]           | [65000,135000]
(2 rows)
```

To use a special character as a literal, prefix it with an escape character. For example, to include a literal backslash (\ ) in the loaded data (such as when including a file path), use two backslashes (\\ ). COPY removes the escape character from the input when it loads escaped characters.

When loading delimited data, two consecutive delimiters indicate a null value, unless the NULL parameter is set otherwise. The final delimiter is optional. For example, the following input is valid for the previous table:

```
=> COPY employees FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 45|Raj|
>> 21|Leonard||
>> \.

=> SELECT * FROM employees;
 id |  name   |  department
----+---------+--------------
 21 | Leonard |
 42 | Raj     |
(2 rows)
```

By default, if the data has too few values, the load fails. You can use the TRAILING NULLCOLS option to accept any number of missing columns and treat their values as null.

Vertica assumes that data is in the UTF-8 encoding.

The options specific to the DELIMITED parser and their default values are:

Option	Default
DELIMITER	|
ENCLOSED BY	"
ESCAPE	\
NULL	'' (empty string)
COLLECTIONOPEN	[
COLLECTIONCLOSE	]
COLLECTIONDELIMITER	,
COLLECTIONNULLELEMENT	null
COLLECTIONENCLOSE	" (double quote)
TRAILING NULLCOLS	(none)

To load delimited data into a Flex table, use the FDELIMITEDPARSER parser.

Changing the column separator (DELIMITER)

The default COPY delimiter is a vertical bar ('|'). The DELIMITER is a single ASCII character used to separate columns within each record of an input source. Between two delimiters, COPY interprets all string data in the input as characters. Do not enclose character strings in quotes, because quote characters are also treated as literals between delimiters.

You can define a different delimiter using any ASCII value in the range E'\000' to E'\177' inclusive. For instance, if you are loading CSV data files, and the files use a comma (',') character as a delimiter, you can change the default delimiter to a comma. You cannot use the same character for both the DELIMITER and NULL options.

If the delimiter character is among a string of data values, use the ESCAPE AS character ('\' by default) to indicate that the delimiter should be treated as a literal.

The COPY statement accepts empty values (two consecutive delimiters) as valid input data for CHAR and VARCHAR data types. COPY stores empty columns as an empty string (''). An empty string is not equivalent to a NULL string.

To indicate a non-printing delimiter character (such as a tab), specify the character in extended string syntax (E'...'). If your database has StandardConformingStrings enabled, use a Unicode string literal (U&'...'). For example, use either E'\t' or U&'\0009' to specify tab as the delimiter.

The following example loads data from a comma-separated file:

```
=> COPY employees FROM ... DELIMITER ',';
```

In the following example, the first column has a column-specific delimiter:

```
=> COPY employees(id DELIMITER ':', name, department) FROM ... DELIMITER ',';
```

Changing collection delimiters (COLLECTIONDELIMITER, COLLECTIONOPEN, COLLECTIONCLOSE)

The DELIMITER option specifies the value that separates columns in the input. For a column with a collection type (ARRAY or SET), a delimiter is also needed between elements of the collection. In addition, the collection itself has start and end markers. By default, collections are enclosed in brackets and elements are delimited by commas, but you can change these values.

In the following example, collections are enclosed in braces and delimited by periods.

```
=> COPY researchers FROM STDIN COLLECTIONOPEN '{' COLLECTIONCLOSE '}' COLLECTIONDELIMITER '.';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 19|Leonard|{"us-1672"."darpa-1963"}|{16200.16700}
>> \.

=> SELECT * FROM researchers;
 id |      name       |               grants               |       values
----+-----------------+------------------------------------+--------------------
 17 | Howard Wolowitz | ["NASA-1683","NASA-7867","SPX-76"] | [16700,85000,45000]
 42 | Sheldon Cooper  | ["US-7376","DARPA-1567"]           | [65000,135000]
 19 | Leonard         | ["us-1672","darpa-1963"]           | [16200,16700]
(3 rows)
```

Changing the character enclosing column or collection values (ENCLOSED BY, COLLECTIONENCLOSE)

The ENCLOSED BY parameter lets you set an ASCII character to delimit characters to embed in string values. The enclosing character is not considered to be part of the data if and only if it is the first and last character of the input. You can use any ASCII value in the range E'\001' to E'\177' inclusive (any ASCII character except NULL: E'\000') for the ENCLOSED BY value. Using double quotation marks (") is common, as shown in the following example.

```
=> COPY employees FROM STDIN ENCLOSED BY '"';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 21|Leonard|Physics
>> 42|"Sheldon"|"Physics"
>> 17|Rajesh "Raj" K|Astronomy
>> \.

=> SELECT * FROM employees;
 id |      name       |  department
----+-----------------+--------------
 17 | Rajesh "Raj" K | Astronomy
 21 | Leonard        | Physics
 42 | Sheldon        | Physics
(3 rows)
```

Notice that while ENCLOSED BY is a double quote, the embedded quotes in Rajesh's name are treated as part of the data because they are not the first and last characters in the column. The quotes that enclose "Sheldon" and "Physics" are dropped because of their positions.

Within a collection value, the COLLECTIONENCLOSE parameter is like ENCLOSED BY for individual elements of the collection.

Changing the null indicator (NULL)

By default, an empty string ('') for a column value means NULL. You can specify a different ASCII value in the range E'\001' to E'\177' inclusive (any ASCII character except NUL : E'\000') as the NULL indicator. You cannot use the same character for both the DELIMITER and NULL options.

A column containing one or more whitespace characters is not NULL unless the sequence of whitespace exactly matches the NULL string.

A NULL is case-insensitive and must be the only value between the data field delimiters. For example, if the null string is NULL and the delimiter is the default vertical bar (|):

|NULL| indicates a null value.

| NULL | does not indicate a null value.

When you use the COPY statement in a script, you must substitute a double-backslash for each null string that includes a backslash. For example, the scripts used to load the example database contain:

```
COPY ... NULL E'\\n' ...
```

Changing the null indicator for collection values (COLLECTIONNULLELEMENT)

The NULL option specifies the value to be treated as null for a column value. For a column with a collection type (ARRAY or SET), a separate option specifies how to interpret null *elements* . By default, "null" indicates a null value. An empty value, meaning two consecutive element delimiters, does *not* indicate null:

```
=> COPY researchers FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 17|Howard|["nasa-143",,"nasa-6262"]|[10000,1650,15367]
>> 19|Leonard|["us-177",null,"us-6327"]|[16200,64000,26500]
>> \.

=> SELECT * FROM researchers;
 id |  name   |          grants            |       values
----+---------+----------------------------+--------------------
 17 | Howard  | ["nasa-143","","nasa-6262"] | [10000,1650,15367]
 19 | Leonard | ["us-177",null,"us-6327"]   | [16200,64000,26500]
(2 rows)
```

Use COLLECTIONNULLELEMENT to specify a different value, as in the following example.

```
=> COPY researchers from STDIN COLLECTIONNULLELEMENT 'x';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 42|Sheldon|[x,"us-1672"]|[x,165000]
>> \.

=> SELECT * FROM researchers;
 id |  name   |          grants            |       values
----+---------+----------------------------+--------------------
 17 | Howard  | ["nasa-143","","nasa-6262"] | [10000,1650,15367]
 19 | Leonard | ["us-177",null,"us-6327"]   | [16200,64000,26500]
 42 | Sheldon | [null, "us-1672"]           | [null,165000]
(3 rows)
```

Filling missing columns (TRAILING NULLCOLS)

By default, COPY fails if the input does not contain enough columns. Use the TRAILING NULLCOLS option to instead insert NULL values for any columns that lack data. This option cannot be used with columns that have a NOT NULL constraint.

The following example demonstrates use of this option.

```
=> CREATE TABLE z (a INT, b INT, c INT );

--- insert with enough data:
=> INSERT INTO z VALUES (1, 2, 3);

=> SELECT * FROM z;
 a | b | c
---+---+---
 1 | 2 | 3
(1 row)

--- insert deficient data:
=> COPY z FROM STDIN TRAILING NULLCOLS;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 4 | 5 | 6
>> 7 | 8
>> \.

=> SELECT * FROM z;
 a | b | c
---+---+---
 1 | 2 | 3
 4 | 5 | 6
 7 | 8 |
(3 rows)
```

Changing the escape character (ESCAPE AS, NO ESCAPE)

You can specify an escape character, which enables any special characters to be treated as part of the data. For example, if an element from a CSV file should contain a comma, you can indicate that by pre-pending the escape character to the comma in the data. The default escape character is a backslash (\ ).

To change the default to a different character, use the ESCAPE AS option. You can set the escape character to be any ASCII value in the range E'\001' to E'\177' inclusive.

If you do not want any escape character and want to prevent any characters from being interpreted as escape sequences, use the NO ESCAPE option.

ESCAPE AS and NO ESCAPE can be set at both the column and global levels.

Changing the end-of-line character (RECORD TERMINATOR)

To specify the literal character string that indicates the end of a data file record, use the RECORD TERMINATOR parameter, followed by the string to use. If you do not specify a value, then Vertica attempts to determine the correct line ending, accepting either just a linefeed (E'\n') common on UNIX systems, or a carriage return and linefeed (E'\r\n') common on Windows platforms.

For example, if your file contains comma-separated values terminated by line feeds that you want to maintain, use the RECORD TERMINATOR option to specify an alternative value:

```
=>  COPY mytable FROM STDIN DELIMITER ',' RECORD TERMINATOR E'\n';
```

To specify the RECORD TERMINATOR as non-printing characters, use either the extended string syntax or Unicode string literals. The following table lists some common record terminator characters. See String Literals for an explanation of the literal string formats.

Extended String Syntax	Unicode Literal String	Description	ASCII Decimal
E'\b'	U&'\0008'	Backspace	8
E'\t'	U&'\0009'	Horizontal tab	9
E'\n'	U&'\000a'	Linefeed	10
E'\f'	U&'\000c'	Formfeed	12

E'\r'	U&'\000d'	Carriage return	13
E'\\'	U&'\005c'	Backslash	92

If you use the RECORD TERMINATOR option to specify a custom value, be sure the input file matches the value. Otherwise, you may get inconsistent data loads.

> **Note**
> The record terminator cannot be the same as DELIMITER , NULL , ESCAPE , or ENCLOSED BY .

If using JDBC, Vertica recommends that you use the following value for RECORD TERMINATOR :

```
System.getProperty("line.separator")
```

Binary (native) data

You can load binary data using the NATIVE parser option, except with COPY LOCAL , which does not support this option. Since binary-format data does not require the use and processing of delimiters, it precludes the need to convert integers, dates, and timestamps from text to their native storage format, and improves load performance over delimited data. All binary-format files must adhere to the formatting specifications described in Appendix: creating native binary format files .

Native binary format data files are typically larger than their delimited text format counterparts, so compress the data before loading it. The NATIVE parser does not support concatenated compressed binary files. You can load native (binary) format files when developing plug-ins to ETL applications.

There is no copy format to load binary data byte-for-byte because the column and record separators in the data would have to be escaped. Binary data type values are padded and translated on input, and also in the functions, operators, and casts supported.

Loading hexadecimal, octal, and bitstring data

You can use the formats hexadecimal, octal, and bitstring only to load binary columns. To specify these column formats, use the COPY statement's FORMAT options:

- Hexadecimal
- Octal
- Bitstring

The following examples illustrate how to use the FORMAT option.

1. Create a table:
```
=> CREATE TABLE t(oct VARBINARY(5),
   hex VARBINARY(5),
   bitstring VARBINARY(5) );
```
2. Create the projection:
```
=> CREATE PROJECTION t_p(oct, hex, bitstring) AS SELECT * FROM t;
```
3. Use a COPY statement with the STDIN clause, specifying each of the formats:
```
=> COPY t (oct FORMAT 'octal', hex FORMAT 'hex',
        bitstring FORMAT 'bitstring')
  FROM STDIN DELIMITER ',';
```
4. Enter the data to load, ending the statement with a backslash () and a period (.) on a separate line:
```
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 141142143144145,0x6162636465,0110000101100010011000110110010001100101
>> \.
```
5. Use a select query on table t to view the input values results:

```
=> SELECT * FROM t;
 oct   | hex   | bitstring
-------+-------+-----------
abcde  | abcde | abcde
(1 row)
```

COPY uses the same default format to load binary data, as used to input binary data. Since the backslash character ('\') is the default escape character, you must escape octal input values. For example, enter the byte '\141' as '\\141' .

> **Note**
> If you enter an escape character followed by an invalid octal digit or an escape character being escaped, COPY returns an error.

On input, COPY translates string data as follows:

- Uses the HEX_TO_BINARY function to translate from hexadecimal representation to binary.
- Uses the BITSTRING_TO_BINARY function to translate from bitstring representation to binary.

Both functions take a VARCHAR argument and return a VARBINARY value.

You can also use the escape character to represent the (decimal) byte 92 by escaping it twice; for example, '\\\\' . Note that vsql inputs the escaped backslash as four backslashes. Equivalent inputs are hex value '0x5c' and octal value '\134' (134 = 1 x 8^2 + 3 x 8^1 + 4 x 8^0 = 92).

You can load a delimiter value if you escape it with a backslash. For example, given delimiter '|' , '\\001\|\\002' is loaded as { 1,124,2 }, which can also be represented in octal format as '\\001\\174\\002' .

If you insert a value with more bytes than fit into the target column, COPY returns an error. For example, if column c1 is VARBINARY(1) :

```
=> INSERT INTO t (c1) values ('ab');   ERROR: 2-byte value too long for type Varbinary(1)
```

If you implicitly or explicitly cast a value with more bytes than fit the target data type, COPY silently truncates the data. For example:

```
=> SELECT 'abcd'::binary(2);
 binary
--------
 ab
(1 row)
```

Hexadecimal data

The optional '0x' prefix indicates that a value is hexadecimal, not decimal, although not all hexadecimal values use A-F; for example, 5396. COPY ignores the 0x prefix when loading the input data.

If there are an odd number of characters in the hexadecimal value, the first character is treated as the low nibble of the first (furthest to the left) byte.

Octal data

Loading octal format data requires that each byte be represented by a three-digit octal code. The first digit must be in the range [0,3] and the second and third digits must both be in the range [0,7].

If the length of an octal value is not a multiple of three, or if one of the three digits is not in the proper range, the value is invalid and COPY rejects the row in which the value appears. If you supply an invalid octal value, COPY returns an error. For example:

```
=> SELECT '\\000\\387'::binary(8);
ERROR: invalid input syntax for type binary
```

Rows that contain binary values with invalid octal representations are also rejected. For example, COPY rejects '\\008' because '\\ 008' is not a valid octal number.

BitString data

Loading bitstring data requires that each character must be zero (0) or one (1), in multiples of eight characters. If the bitstring value is not a multiple of eight characters, COPY treats the first *n* characters as the low bits of the first byte (furthest to the left), where *n* is the remainder of the value's length, divided by eight.

Examples

The following example shows VARBINARY HEX_TO_BINARY(VARCHAR) and VARCHAR TO_HEX(VARBINARY) usage.

1. Create table t and and its projection with binary columns:

```
=> CREATE TABLE t (c BINARY(1));
=> CREATE PROJECTION t_p (c) AS SELECT c FROM t;
```

2. Insert minimum and maximum byte values, including an IP address represented as a character string:

```
=> INSERT INTO t values(HEX_TO_BINARY('0x00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFF'));
=> INSERT INTO t values (V6_ATON('2001:DB8::8:800:200C:417A'));
```

   Use the TO_HEX function to format binary values in hexadecimal on output:

```
=> SELECT TO_HEX(c) FROM t;
 to_hex
--------
 00
 ff
 20
(3 rows)
```

See also

- Binary data types (BINARY and VARBINARY)
- Formatting functions
- ASCII

Native varchar data

Use the NATIVE VARCHAR parser option when the raw data consists primarily of CHAR or VARCHAR data. COPY performs the conversion to the actual table data types on the database server. This parser option is not supported with COPY LOCAL .

Using NATIVE VARCHAR does not provide the same efficiency as NATIVE . However, NATIVE VARCHAR precludes the need to use delimiters or to escape special characters, such as quotes, which can make working with client applications easier.

> **Note**
> NATIVE VARCHAR does not support concatenated compressed files.

Batch data inserts performed through the Vertica ODBC and JDBC drivers automatically use the NATIVE VARCHAR format.

Fixed-width format data

Use the FIXEDWIDTH parser option to bulk load fixed-width data. You must specify the COLSIZES option values to specify the number of bytes for each column. The definition of the table you are loading (COPY table f (x, y, z)) determines the number of COLSIZES values to declare.

To load fixed-width data, use the COLSIZES option to specify the number of bytes for each input column. If any records do not have values, COPY inserts one or more null characters to equal the specified number of bytes. The last record in a fixed-width data file must include a record terminator to determine the end of the load data.

The following COPY options are not supported:

- DELIMITER
- ENCLOSED BY
- ESCAPE AS
- TRAILING NULLCOLS

Using nulls in fixed-width data

The default NULL string for a fixed-width load cannot be an empty string, and instead, consists of all spaces. The number of spaces depends on the column width declared with the COLSIZES (integer, [,...]) option.

For fixed-width loads, the NULL definition depends on whether you specify NULL at the column or statement level:

- *Statement level* : NULL must be defined as a single-character. The default (or custom) NULL character is repeated for the entire width of the column.
- *Column level* : NULL must be defined as a string whose length matches the column width.

For fixed-width loads, if the input data column has fewer values than the specified column size, COPY inserts NULL characters. The number of NULLs must match the declared column width. If you specify a NULL string at the column level, COPY matches the string with the column width.

> **Note**
> To turn off NULLs, use the NULL AS option and specify NULL AS '' .

Defining a null character (statement level)

1. Create a two-column table (fw):

```
=> CREATE TABLE fw(co int, ci int);
CREATE TABLE
```

2. Copy the table, specifying null as 'N' , and enter some data:

```
=> COPY fw FROM STDIN FIXEDWIDTH colsizes(2,2) null AS 'N' NO COMMIT;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> NN12
>> 23NN
>> NNNN
>> nnnn
>> \.
```

3. Select all (*) from the table:

```
=> SELECT * FROM fw;
 co | ci
----+----
    | 12
 23 |
(2 rows)
```

Defining a custom record terminator

To define a record terminator other than the COPY default when loading fixed-width data, take these steps:

1. Create table fw with two columns, co and ci :

```
=> CREATE TABLE fw(co int, ci int);
CREATE TABLE
```

2. Copy table fw , specifying two 2-byte column sizes, and specifying a comma (,) as the record terminator:

```
=> COPY fw FROM STDIN FIXEDWIDTH colsizes(2,2) RECORD TERMINATOR ',';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1234,1444,6666
>> \.
```

3. Query all data in table fw :

```
=> SELECT * FROM fw;
 co | ci
----+----
 12 | 34
 14 | 44
(2 rows)
```

The SELECT output indicates only two values. COPY rejected the third value (6666) because it was not followed by a comma (,) record terminator. Fixed-width data requires a trailing record terminator only if you explicitly specify a record terminator explicitly.

Copying fixed-width data

Use COPY FIXEDWIDTH COLSIZES (n [,...) to load files into a Vertica database. By default, all spaces are NULLs. For example:

```
=> CREATE TABLE mytest(co int, ci int);
=> CREATE PROJECTION mytest_p1 AS SELECT * FROM mytest SEGMENTED BY HASH(co) ALL NODES;
=> COPY mytest(co,ci) FROM STDIN FIXEDWIDTH colsizes(6,4) NO COMMIT;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> \.
=> SELECT * FROM mytest ORDER BY co;
 co | ci
----+----
(0 rows)
```

Skipping content in fixed-width data

The COPY statement has two options to skip input data. The SKIP BYTES option is only for fixed-width data loads:

SKIP BYTES *num-bytes*	Skips the specified number of bytes from the input data.
SKIP *num-records*	Skips the specified number of records.

The following example uses SKIP BYTES to skip 11 bytes when loading a fixed-width table with two columns (4 and 6 bytes):

1. Copy a table using SKIP BYTES :

```
=> COPY fw FROM STDIN FIXEDWIDTH colsizes (4,6) SKIP BYTES 11;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 2222666666
>> 1111999999
>> 1632641282
>> \.
```

2. Query all data in table fw :

```
=> SELECT * FROM fw ORDER BY co;
 co  |   ci
------+--------
 1111 | 999999
 1632 | 641282
(2 rows)
```

The output confirms that COPY skipped the first 11 bytes of loaded data.

The following example uses SKIP when loading a fixed-width (4,6) table:

1. Copy a table, using SKIP to skip two records of input data:

```
=> COPY fw FROM STDIN FIXEDWIDTH colsizes (4,6) SKIP 2;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 2222666666
>> 1111999999
>> 1632641282
>> 3333888888
>> \.
```

2. Query all data in table fw :

```
=> SELECT * FROM fw ORDER BY co;
 co  |   ci
------+--------
 1632 | 641282
 3333 | 888888
(2 rows)
```

The output confirms that COPY skipped the first two records of load data.

Trimming characters in fixed-width data loads

Use the TRIM option to trim a character. TRIM accepts a single-byte character, which is trimmed at the beginning and end of the data. For fixed-width data loads, when you specify a TRIM character, COPY first checks to see if the row is NULL. If the row is not null, COPY trims the character(s). The next example instructs COPY to trim the character A, and shows the results:

1. Copy table fw , specifying TRIM character A :

```
=> COPY fw FROM STDIN FIXEDWIDTH colsizes(4,6) TRIM 'A';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> A22A444444
>> A22AA4444A
>> \.
```

2. Query all data in table fw :

```
=> SELECT * FROM fw ORDER BY co;
 co |   ci
----+--------
 22 |   4444
 22 | 444444
(2 rows)
```

Using padding in fixed-width data loads

By default, the padding character is ' ' (a single space). The padding behavior for fixed-width data loads is similar to how a space is treated in other formats, differing by data type as follows:

Data type	Padding
Integer	Leading and trailing spaces
Bool	Leading and trailing spaces
Float	Leading and trailing spaces
[var]Binary	None, all characters are significant.
[Var]Char	Trailing spaces if string is too large
DateInterval , Time , Timestamp , TimeTZ	None, all characters are significant. COPY uses an internal algorithm to parse these data types.
Date (formatted)	Use the COPY FORMAT option string to match the expected column length.
Numerics	Leading and trailing spaces

ORC data

The ORC (Optimized Row Columnar) format is a column-oriented file format. Vertica has a parser for this format that can take advantage of the columnar layout.

In the COPY statement, specify the parser as follows. Note that this is not the usual PARSER *parser-name* syntax; COPY supports ORC directly:

```
=> COPY tableName FROM path ORC[(...)];
```

The parser takes several optional parameters; see the ORC reference page.

Be aware that if you load from multiple files in the same COPY statement, and any of them is aborted, the entire load aborts. This behavior differs from that for delimited files, where the COPY statement loads what it can and ignores the rest.

Supported data types

Vertica can natively read columns of all Hive primitive data types. For a complete list, see HIVE Data Types (specifically the Numeric, Date/Time, String, and Misc lists). Vertica can also load UUIDs and complex types (arrays and structs in any combination).

The data types you specify for COPY or CREATE EXTERNAL TABLE AS COPY must exactly match the types in the data, though Vertica permits implicit casting among compatible numeric types.

Timestamps and time zones

To correctly report timestamps, Vertica must know what time zone the data was written in. Older versions of the ORC format do not record the time zone.

Hive version 1.2.0 and later records the writer time zone in the stripe footer. Vertica uses that time zone to make sure the timestamp values read into the database match the ones written in the source file. For ORC files that are missing this time zone information, Vertica assumes the values were written in the local time zone and logs an ORC_FILE_INFO event in the QUERY_EVENTS system table. Check for events of this type after your first query to verify that timestamps are being handled as you expected.

Parquet data

Parquet is a column-oriented file format. Vertica has a parser that can take advantage of the columnar layout.

In the COPY statement, specify the parser as follows. Note that this is not the usual PARSER *parser-name* syntax; COPY supports Parquet directly:

```
=> COPY tableName FROM path PARQUET[(...)];
```

The parser takes several optional parameters; see the PARQUET reference page.

Be aware that if you load from multiple files in the same COPY statement, and any of them is aborted, the entire load aborts. This behavior differs from that for delimited files, where the COPY statement loads what it can and ignores the rest.

Schema matching

By default, the Parquet parser uses strong schema matching. This means that the load must consume all columns in the Parquet data in the order they occur in the data. You can, instead, use loose schema matching, which allows you to select the columns you want and ignore the rest. Loose schema matching depends on the names of the columns in the data rather than their order, so the column names in your table must match those in the data. Types must match or be coercible. For more information on how to use loose schema matching, see Loose Schema Matching on the PARQUET reference page.

You can use loose schema matching for columns of primitive types and one-dimensional arrays of primitive types. For other complex types, use flexible complex types instead. See Flexible complex types .

Use the do_soft_schema_match_by_name parameter to specify loose schema matching. In the following example, the Parquet data contains more columns than those used in the table.

```
=> CREATE EXTERNAL TABLE restaurants(name VARCHAR, cuisine VARCHAR)
   AS COPY FROM '/data/rest*.parquet'
   PARQUET(do_soft_schema_match_by_name='True');

=> SELECT * from restaurant;
      name        | cuisine
------------------+----------
 Bob's pizzeria    | Italian
 Bakersfield Tacos | Mexican
(2 rows)
```

Metadata caching

Parquet files include metadata that Vertica uses when loading data. To avoid repeatedly fetching this data, particularly from remote sources or where API calls incur financial costs, Vertica caches this metadata on each participating node during the planning phase for use during the execution phase.

Vertica uses TEMP storage for the cache, and only if TEMP storage is on the local file system.

You can limit the size of the cache by setting the ParquetMetadataCacheSizeMB configuration parameter. The default is 4GB.

Supported data types

Vertica can natively read columns of all Hive primitive data types. For a complete list, see HIVE Data Types (specifically the Numeric, Date/Time, String, and Misc lists). Vertica can also load UUIDs and complex types (arrays and structs in any combination).

The data types you specify for COPY or CREATE EXTERNAL TABLE AS COPY must exactly match the types in the data, though Vertica permits implicit casting among compatible numeric types.

For the Parquet format only, you can use flexible complex types instead of fully specifying the schema for complex types. See Flexible complex types .

Timestamps and time zones

To correctly report timestamps, Vertica must know what time zone the data was written in. Hive does not record the writer time zone. Vertica assumes timestamp values were written in the local time zone and reports a warning at query time.

Hive provides an option, when writing Parquet files, to record timestamps in the local time zone. If you are using Parquet files that record times in this way, set the UseLocalTzForParquetTimestampConversion configuration parameter to 0 to disable the conversion done by Vertica. (See General parameters .)

JSON data

Use FJSONPARSER to load data in JSON format.

The schema for JSON data is the set of property names in the property/value pairs. When you load JSON data into a columnar table or materialized columns in a Flex table, the property names in the data must match the column names in the table. You do not need to load all of the columns in the data.

The JSON parser can load data into columns of any scalar type, strongly-typed complex type , or flexible complex type . A flexible complex type means you do not fully specify the schema for that column. You define these columns in the table as LONG VARBINARY , and you can use Flex functions to extract values from them.

In the COPY statement, use the PARSER parameter to specify the JSON parser as in the following example:

```
=> CREATE EXTERNAL TABLE customers(id INT, address VARCHAR, transactions ARRAY[INT,10])
   AS COPY FROM 'cust.json' PARSER FJSONPARSER();
```

This parser has several optional parameters, some of which are specific to use with Flex tables and flexible complex types.

Before loading JSON data, consider using a tool such as JSONLint to verify that the data is valid.

If you load JSON data into a Flex table, Vertica loads all data into the __raw__ (VMap) column, including complex types found in the data. You can use Flex functions to extract values.

Strongly-typed complex types

JSON data can contain arrays, structs, and combinations of the two. You can load this data either as flexible (VMap) columns or with strong typing. Strong typing allows you to query values directly, without having to use functions to unpack a VMap column.

Use the ARRAY and ROW types in the table definition as usual:

```
=> CREATE EXTERNAL TABLE rest
  (name VARCHAR, cuisine VARCHAR,
   location_city ARRAY[VARCHAR(80),50],
   menu ARRAY[ ROW(item VARCHAR(80), price FLOAT), 100 ]
  )
 AS COPY FROM :restdata PARSER FJSONPARSER();

=> SELECT name, location_city, menu FROM rest;
      name        |       location_city        |                                  menu
-------------------+----------------------------+-------------------------------------------------------------------------------
 Bob's pizzeria    | ["Cambridge","Pittsburgh"] | [{"item":"cheese pizza","price":8.25},{"item":"spinach pizza","price":10.5}]
 Bakersfield Tacos | ["Pittsburgh"]             | [{"item":"veggie taco","price":9.95},{"item":"steak taco","price":10.95}]
(2 rows)
```

When loading JSON data into a table with strong typing for complex types, Vertica ignores the parser's flatten_maps and flatten_arrays parameters.

Strong and flexible complex types

An advantage of strong typing is the easier (and more efficient) access in queries. A disadvantage is that additional values found in the data but not included in the column definition are ignored. If the menu struct in this data includes more attributes, such as calories, they are not loaded because the definition of the column only specified item and price. The following example uses flexible complex types to reveal the extra attributes:

```
=> CREATE EXTERNAL TABLE rest
  (name VARCHAR, cuisine VARCHAR,
   location_city LONG VARBINARY, menu LONG VARBINARY)
 AS COPY FROM :restdata
 PARSER FJSONPARSER(flatten_maps=false);

=> SELECT name, MAPTOSTRING(location_city) as location_city, MAPTOSTRING(menu) AS menu FROM rest;
       name        |       location_city      |                    menu
-------------------+--------------------------+-----------------------------------------------
 Bob's pizzeria    | {
    "0": "Cambridge",
    "1": "Pittsburgh"
} | {
    "0": {
        "calories": "1200",
        "item": "cheese pizza",
        "price": "8.25"
    },
    "1": {
        "calories": "900",
        "item": "spinach pizza",
        "price": "10.50"
    }
}
 Bakersfield Tacos | {
    "0": "Pittsburgh"
}                  | {
    "0": {
        "item": "veggie taco",
        "price": "9.95",
        "vegetarian": "true"
    },
    "1": {
        "item": "steak taco",
        "price": "10.95"
    }
}
(2 rows)
```

Unmatched fields

The JSON parser loads all fields from the data that are part of the table definition. If the data contains other fields, the parser produces a warning and logs the new fields in a system table. You can review the logged events and decide whether to modify your table definition. For details, see Schema evolution .

Loading from a specific start point

You need not load an entire JSON file. You can use the start_point parameter to load data beginning at a specific key, rather than at the beginning of a file. Data is parsed from after the start_point key until the end of the file, or to the end of the first start_point 's value. The parser ignores any subsequent instance of the start_point , even if that key appears multiple times in the input file. If the input data contains only one copy of the start_point key, and that value is a list of JSON elements, the parser loads each element in the list as a row.

If a start_point value occurs more than once in your JSON data, you can use the start_point_occurrence integer parameter to specify the occurrence at which to start parsing.

This example uses the following JSON data, saved to a file named alphanums.json :

```
{ "A": { "B": { "C": [ { "d": 1, "e": 2, "f": 3 }, { "g": 4, "h": 5, "i": 6 },
{ "j": 7, "k": 8, "l": 9 } ] } } }
```

Loading this data into a flex table produces the following results:

```
=> CREATE FLEX TABLE start_json;
CREATE TABLE

=> COPY start_json FROM '/home/dbadmin/data/alphanums.json' PARSER FJSONPARSER();
 Rows Loaded
-------------
           1
(1 row)

=> SELECT maptostring(__raw__) FROM start_json;
                        maptostring
-------------------------------------------------------------------------------
 {
   "A.B.C" : {
      "0.d" : "1",
      "0.e" : "2",
      "0.f" : "3",
      "1.g" : "4",
      "1.h" : "5",
      "1.i" : "6",
      "2.j" : "7",
      "2.k" : "8",
      "2.l" : "9"
   }
}

(1 row)
```

The following load specifies a start point:

```
=> TRUNCATE TABLE start_json;
TRUNCATE TABLE

=>  COPY start_json FROM '/home/dbadmin/data/alphanums.json' PARSER FJSONPARSER(start_point='B');
 Rows Loaded
-------------
           1
(1 row)

=> SELECT maptostring(__raw__) FROM start_json;
                         maptostring
-------------------------------------------------------------------------------
 {
   "C" : {
      "0.d" : "1",
      "0.e" : "2",
      "0.f" : "3",
      "1.g" : "4",
      "1.h" : "5",
      "1.i" : "6",
      "2.j" : "7",
      "2.k" : "8",
      "2.l" : "9"
   }
}
(1 row)
```

Dealing with invalid JSON records

If your JSON data contains syntax errors, your load can fail due to invalid records. You can use the RECORD_TERMINATOR option in the COPY statement to skip these invalid records if your JSON records are consistently delimited by a character like a line break. Setting a record terminator allows the parser to skip over invalid records and continue parsing the rest of the data.

If your records are not consistently marked by a character, you can use the ERROR TOLERANCE option. ERROR TOLERANCE skips entire source files with invalid JSON records, while RECORD_TERMINATOR skips individual malformed JSON records. You can use the two options together.

The following example uses invalid records:

```
=> => CREATE FLEX TABLE fruits();
CREATE TABLE

=> COPY fruits FROM STDIN PARSER FJSONPARSER(RECORD_TERMINATOR=E'\n');
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself
>> {"name": "orange", "type": "fruit", "color": "orange", "rating": 5 }
>> {"name": "apple", "type": "fruit", "color": "green" }
>> {"name": "blueberry", "type": "fruit", "color": "blue", "rating": 10 }
>> "type": "fruit", "rating": 7 }
>> {"name": "banana", "type" : "fruit", "color": "yellow", "rating": 3 }
>> \.
```

View the flex table using MAPTOSTRING to confirm that the invalid record was skipped while the rest of the records were successfully loaded:

```
=> SELECT MAPTOSTRING(__raw__) FROM fruits;
maptostring
-----------------------------------------------------------------------------------------
{
"color" : "orange",
"name" : "orange",
"rating" : "5",
"type" : "fruit"
}
{
"color" : "green",
"name" : "apple",
"type" : "fruit"
}
{
"color" : "blue",
"name" : "blueberry",
"rating" : "10",
"type" : "fruit"
}
{
"color" : "yellow",
"name" : "banana",
"rating" : "3",
"type" : "fruit"
}
(4 rows)
```

Rejecting data on materialized column type errors

By default, if FJSONPARSER cannot coerce a data value to a type that matches the column definition, it sets the value to NULL. You can choose to instead reject these values using the reject_on_materialized_type_error parameter. If this parameter is true, COPY rejects such rows and reports an error.

If the column is a strongly-typed complex type , as opposed to a flexible complex type , then a type mismatch anywhere in the complex type causes the entire column to be treated as a mismatch. The parser does not partially load complex types; if any ROW field or ARRAY element cannot be coerced, it loads NULL for the column.

The following example attempts to load invalid data. Note that the invalid row is missing from the query results:

```
=> CREATE TABLE test(one VARCHAR, two INT);
CREATE TABLE

=> COPY test FROM stdin
   PARSER FJSONPARSER(reject_on_materialized_type_error=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"one": 1, "two": 2}
>> {"one": "one", "two": "two"}
>> {"one": "one", "two": 2}
>> \.

=> SELECT one, two FROM test;
 one | two
-----+-----
 1   |   2
 one |   2
(2 rows)
```

Rejecting or omitting empty keys in flex tables

Valid JSON files can include empty key and value pairs. By default, for a Flex table, FJSONPARSER loads them, as in the following example:

```
=> CREATE FLEX TABLE fruits();
CREATE TABLE

=> COPY fruits FROM STDIN PARSER FJSONPARSER();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"name": "orange", "rating": 5}
>> {"name": "apple", "rating" : 10}
>> {"": "banana", "rating" : 2}
>> \.

=> SELECT MAPTOSTRING(__raw__) FROM fruits;
              maptostring
----------------------------------------
 {
       "name": "orange",
       "rating": "5"
}
 {
       "name": "apple",
       "rating": "10"
}
 {
       "": "banana",
       "rating": "2"
}
(3 rows)
```

To omit fields with empty keys, use the omit_empty_keys parameter:

```
=> COPY fruits FROM STDIN PARSER FJSONPARSER(omit_empty_keys=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"name": "apple", "rating": 5}
>> {"": "missing", "rating" : 1}
>> {"name": "", "rating" : 3}
>> \.
=> SELECT MAPTOSTRING(__raw__) FROM fruits;
            maptostring
---------------------------------------
 {
       "name": "apple",
       "rating": "5"
}
 {
       "rating": "1"
}
 {
       "name": "",
       "rating": "3"
}
(3 rows)
```

Note that the second value, with the missing name, still loads with the other (non-empty) field. To instead reject the row entirely, use the reject_on_empty_key parameter:

```
=> COPY fruits FROM STDIN PARSER FJSONPARSER(reject_on_empty_key=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"name" : "apple", "rating" : 5}
>> {"" : "missing", "rating" : 1}
>> {"name" : "", "rating" : 3}
>> \.
=> SELECT MAPTOSTRING(__raw__) FROM fruits;
            maptostring
---------------------------------------
 {
       "name": "apple",
       "rating": "5"
}
 {
       "name": "",
       "rating": "3"
}
(2 rows)
```

Avro data

Use FAVROPARSER to load Avro data files. This parser supports both columnar and Flex tables.

A column can be of any scalar type, a strongly-typed complex type , or a flexible complex type . A flexible complex type means you do not fully specify the schema for that column. You define these columns in the table as LONG VARBINARY, and you can use Flex functions to extract values from them.

The following requirements apply:

- Avro files must be encoded in the Avro binary serialization encoding format, described in the Apache Avro standard . The parser also supports Snappy and deflate compression.
- The Avro file must have its related schema in the file being loaded; FAVROPARSER does not support Avro files with separate schema files.

In the COPY statement, use the PARSER parameter to specify the Avro parser as in the following example:

```
=> COPY weather FROM '/home/dbadmin/data/weather.avro' PARSER FAVROPARSER()
```

This parser has several optional parameters, some of which are specific to use with Flex tables and flexible complex types.

Avro schemas and columnar tables

Avro includes the schema with the data. When you load Avro data into a columnar table, the column names in the schema in the data must match the column names in the table. You do not need to load all of the columns in the data.

For example, the following Avro schema uses the Avro record type to represent a user profile:

```
{
  "type": "record",
  "name": "Profile",
  "fields" : [
     {"name": "UserName", "type": "string"},
     {"name": "Email", "type": "string"},
     {"name": "Address", "type": "string"}
  ]
}
```

To successfully load the data into a columnar table with this schema, each target column name must match the "name" value in the schema. In the following example, the profiles table does not load values corresponding to the schema's Email field because the target column is named EmailAddr :

```
=> COPY profiles FROM '/home/dbadmin/data/user_profile.avro' PARSER FAVROPARSER();

=> SELECT * FROM profiles;
    UserName   |    EmailAddr     |      Address
---------------+------------------+--------------------
   dbadmin     |                  |   123 Main St.
```

Strongly-typed complex types

Avro data can contain arrays, structs, and combinations of the two. You can read this data either as flexible (VMap) columns or with strong typing. Strong typing allows you to query values directly, without having to use functions to unpack a VMap column.

Use the ARRAY and ROW types in the table definition as usual:

```
=> CREATE EXTERNAL TABLE rest
  (name VARCHAR, cuisine VARCHAR,
   location_city ARRAY[VARCHAR(80)],
   menu ARRAY[ ROW(item VARCHAR(80), price FLOAT) ]
  )
AS COPY FROM :avro_file PARSER FAVROPARSER();
```

You can use strong typing in both external tables and native tables.

An alternative to strong typing for complex types is flexible complex types .

Unmatched fields

The Avro parser loads all fields from the data that are part of the table definition. If the data contains other fields, the parser produces a warning and logs the new fields in a system table. You can review the logged events and decide whether to modify your table definition. For details, see Schema evolution .

Rejecting data on materialized column type errors

By default, if FAVROPARSER cannot coerce a data value to a type that matches the column definition, it sets the value to NULL. You can choose to instead reject these values using the reject_on_materialized_type_error parameter. If this parameter is true, COPY rejects such rows and reports an error.

If the column is a strongly-typed complex type , as opposed to a flexible complex type , then a type mismatch anywhere in the complex type causes the entire column to be treated as a mismatch. The parser does not partially load complex types.

If a flex table has a materialized column, the data being loaded must be coercible to that column's type. For example, if a materialized column is declared as a FLOAT and you try to load a VARCHAR value for that key, FAVROPARSER rejects the data row.

See also

- Manually consume data from Kafka

Matches from regular expressions

You can load flex or columnar tables with the matched results of a regular expression, using the fregexparser . This section describes some examples of using the options that the flex parsers support.

Sample regular expression

These examples use the following regular expression, which searches information that includes the timestamp , date , thread_name , and thread_id strings.

> **Caution**
> For display purposes, this sample regular expression adds new line characters to split long lines of text. To use this expression in a query, first copy and edit the example to remove any new line characters.

This example expression loads any thread_id hex value, regardless of whether it has a 0x prefix, (<thread_id>(?:0x)?[0-9a-f]+).

```
'^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
(?<thread_name>[A-Za-z ]+):(?<thread_id>(?:0x)?[0-9a-f]+)
-?(?<transaction_id>[0-9a-f])?(?:[(?<component>\w+)]
\<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @[?(?<enode>\w+)]?: )
?(?<text>.*)'
```

Using regular expression matches for a flex table

You can load the results from a regular expression into a flex table, using the fregexparser . For a complete example of doing so, see FREGEXPARSER .

Using fregexparser for columnar tables

This section illustrates how to load the results of a regular expression used with a sample log file for a Vertica database. By using an external table definition, the section presents an example of using fregexparser to load data into a columnar table. Using a flex table parser for a columnar tables gives you the capability to mix data loads in one table. For example, you can load the results of a regular expression in one session, and JSON data in another.

The following basic examples illustrate this usage.

1. Create a columnar table, vlog , with the following columns:

```
=> CREATE TABLE vlog (
   "text"            varchar(2322),
   thread_id         varchar(28),
   thread_name       varchar(44),
   "time"            varchar(46),
   component         varchar(30),
   level             varchar(20),
   transaction_id    varchar(32),
   elevel            varchar(20),
   enode             varchar(34)
);
```

2. Use COPY to load parts of a log file using the sample regular expression presented above, with the fregexparser . Be sure to remove any line characters from this expression example before trying it yourself:

```
=> COPY v_log FROM '/home/dbadmin/data/flex/vertica.log' PARSER
FRegexParser(pattern=
'^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
 (?<thread_name>[A-Za-z ]+):(?<thread_id>(?:0x)?[0-9a-f]+)
-?(?<transaction_id>[0-9a-f])?(?:[(?<component>\w+)]
\<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @[?(?<enode>\w+)]?: )
?(?<text>.*)'


) rejected data as table fregex_reject;
```

3. Query the time column:

```
=> SELECT time FROM flogs limit 10;
          time
-------------------------
 2014-04-02 04:02:02.613
 2014-04-02 04:02:02.613
 2014-04-02 04:02:02.614
 2014-04-02 04:02:51.008
 2014-04-02 04:02:51.010
 2014-04-02 04:02:51.012
 2014-04-02 04:02:51.012
 2014-04-02 04:02:51.013
 2014-04-02 04:02:51.014
 2014-04-02 04:02:51.017
(10 rows)
```

Using external tables with fregexparser

By creating an external columnar table for your Vertica log file, querying the table will return updated log information. The following basic example illustrate this usage.

1. Create a columnar table, vertica_log , using the AS COPY clause and fregexparser to load matched results from the regular expression. For illustrative purposes, this regular expression has new line characters to split long text lines. Remove any line returns before testing with this expression:

```
=> CREATE EXTERNAL TABLE public.vertica_log
(
    "text" varchar(2322),
    thread_id varchar(28),
    thread_name varchar(44),
    "time" varchar(46),
    component varchar(30),
    level varchar(20),
    transaction_id varchar(32),
    elevel varchar(20),
    enode varchar(34)
)
AS COPY
FROM '/home/dbadmin/data/vertica.log'
PARSER FRegexParser(pattern=
'^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
 (?<thread_name>[A-Za-z ]+):(?<thread_id>(?:0x)?[0-9a-f]+)
-?(?<transaction_id>[0-9a-f])?(?:[(?<component>\w+)]
\<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @[?(?<enode>\w+)]?: )
?(?<text>.*)'



);
```

2. Query from the external table to get updated results:

```
=> SELECT component, thread_id, time FROM vertica_log limit 10;
 component | thread_id  |          time
-----------+------------+-------------------------
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
 Init      | 0x16321430 | 2014-04-02 04:02:02.613
           | 0x16321430 | 2014-04-02 04:02:02.614
           | 0x16321430 | 2014-04-02 04:02:02.614
           | 0x16321430 | 2014-04-02 04:02:02.614
(10 rows)
```

Common event format (CEF) data

Use the flex parser fcefparser to load OpenText ArcSight or other Common Event Format (CEF) log file data into columnar and flexible tables.

When you use the parser to load arbitrary CEF-format files, it interprets key names in the data as virtual columns in your flex table. After loading, you can query your CEF data directly, regardless of which set of keys exist in each row. You can also use the associated flex table data and map functions to manage CEF data access.

Create a flex table and load CEF data

This section uses a sample set of CEF data. All IP addresses have been purposely changed to be inaccurate, and Return characters added for illustration.

To use this sample data, copy the following text and remove all Return characters. Save the file as CEF_sample.cef , which is the name used throughout these examples.

```
CEF:0|ArcSight|ArcSight|6.0.3.6664.0|agent:030|Agent [test] type [testalertng] started|Low|
eventId=1 mrt=1396328238973 categorySignificance=/Normal categoryBehavior=/Execute/Start
categoryDeviceGroup=/Application catdt=Security Mangement categoryOutcome=/Success
categoryObject=/Host/Application/Service art=1396328241038 cat=/Agent/Started
deviceSeverity=Warning rt=1396328238937 fileType=Agent
cs2=<Resource ID\="3DxKlG0UBABCAA0cXXAZIwA\=\="/> c6a4=fe80:0:0:0:495d:cc3c:db1a:de71
cs2Label=Configuration Resource c6a4Label=Agent
IPv6 Address ahost=SKEELES10 agt=888.99.100.1 agentZoneURI=/All Zones/ArcSight
System/Private Address Space
Zones/RFC1918: 888.99.0.0-888.200.255.255 av=6.0.3.6664.0 atz=Australia/Sydney
aid=3DxKlG0UBABCAA0cXXAZIwA\=\= at=testalertng dvchost=SKEELES10 dvc=888.99.100.1
deviceZoneURI=/All Zones/ArcSight System/Private Address Space Zones/RFC1918:
888.99.0.0-888.200.255.255 dtz=Australia/Sydney _cefVer=0.1
```

1. Create a flex table logs :

```
=> CREATE FLEX TABLE logs();
CREATE TABLE
```

2. Load the sample CEF file, using the flex parser fcefparser :

```
=> COPY logs FROM '/home/dbadmin/data/CEF_sample.cef' PARSER fcefparser();
 Rows Loaded
-------------
           1
(1 row)
```

3. Use the maptostring() function to see the contents of the logs flex table:

```
=> SELECT maptostring(__raw__) FROM logs;
                                maptostring
-------------------------------------------------------------------------------------
 {
  "_cefver" : "0.1",
  "agentzoneuri" : "/All Zones/ArcSight System/Private Address
   Space Zones/RFC1918: 888.99.0.0-888.200.255.255",
  "agt" : "888.99.100.1",
  "ahost" : "SKEELES10",
  "aid" : "3DxKlG0UBABCAA0cXXAZIwA==",
  "art" : "1396328241038",
  "at" : "testalertng",
  "atz" : "Australia/Sydney",
  "av" : "6.0.3.6664.0",
  "c6a4" : "fe80:0:0:0:495d:cc3c:db1a:de71",
  "c6a4label" : "Agent IPv6 Address",
  "cat" : "/Agent/Started",
  "catdt" : "Security Mangement",
  "categorybehavior" : "/Execute/Start",
  "categorydevicegroup" : "/Application",
  "categoryobject" : "/Host/Application/Service",
  "categoryoutcome" : "/Success",
  "categorysignificance" : "/Normal",
  "cs2" : "<Resource ID=\"3DxKlG0UBABCAA0cXXAZIwA==\"/>",
  "cs2label" : "Configuration Resource",
  "deviceproduct" : "ArcSight",
  "deviceseverity" : "Warning",
  "devicevendor" : "ArcSight",
  "deviceversion" : "6.0.3.6664.0",
  "devicezoneuri" : "/All Zones/ArcSight System/Private Address Space
   Zones/RFC1918: 888.99.0.0-888.200.255.255",
  "dtz" : "Australia/Sydney",
  "dvc" : "888.99.100.1",
  "dvchost" : "SKEELES10",
  "eventid" : "1",
  "filetype" : "Agent",
  "mrt" : "1396328238973",
  "name" : "Agent [test] type [testalertng] started",
  "rt" : "1396328238937",
  "severity" : "Low",
  "signatureid" : "agent:030",
  "version" : "0"
}

(1 row)
```

Create a columnar table and load CEF data

This example lets you compare the flex table for CEF data with a columnar table. You do so by creating a new table and load the same CEF_sample.cef file used in the preceding flex table example.

1. Create a columnar table, col_logs , defining the prefix names that are hard coded in fcefparser :

```
=> CREATE TABLE col_logs(version INT,
 devicevendor VARCHAR,
 deviceproduct VARCHAR,
 deviceversion VARCHAR,
 signatureid VARCHAR,
 name VARCHAR,
 severity VARCHAR);
CREATE TABLE
```

2. Load the sample file into col_logs , as you did for the flex table:

```
=> COPY col_logs FROM '/home/dbadmin/data/CEF_sample.cef' PARSER fcefparser();
 Rows Loaded
-------------
           1
(1 row)
```

3. Query the table. You can find the identical information in the flex table output.

```
=> \x
Expanded display is on.
VMart=> SELECT * FROM col_logs;
-[ RECORD 1 ]-+----------------------------------------
version       | 0
devicevendor  | ArcSight
deviceproduct | ArcSight
deviceversion | 6.0.3.6664.0
signatureid   | agent:030
name          | Agent [test] type [testalertng] started
severity      | Low
```

Compute keys and build a flex table view

In this example, you use a flex helper function to compute keys and build a view for the logs flex table.

1. Use the compute_flextable_keys_and_build_view function to compute keys and populate a view generated from the logs flex table:

```
=> SELECT compute_flextable_keys_and_build_view('logs');
                          compute_flextable_keys_and_build_view
-------------------------------------------------------------------------------------------
 Please see public.logs_keys for updated keys
The view public.logs_view is ready for querying
(1 row)
```

2. Query the logs_keys table to see what the function computed from the sample CEF data:

```
=> SELECT * FROM logs_keys;
      key_name       | frequency | data_type_guess
---------------------+-----------+-----------------
 c6a4                |         1 | varchar(60)
 c6a4label           |         1 | varchar(36)
 categoryobject      |         1 | varchar(50)
 categoryoutcome     |         1 | varchar(20)
 categorysignificance |         1 | varchar(20)
 cs2                 |         1 | varchar(84)
 cs2label            |         1 | varchar(44)
 deviceproduct       |         1 | varchar(20)
 deviceversion       |         1 | varchar(24)
 devicezoneuri       |         1 | varchar(180)
 dvchost             |         1 | varchar(20)
 version             |         1 | varchar(20)
 ahost               |         1 | varchar(20)
 art                 |         1 | varchar(26)
 at                  |         1 | varchar(22)
 cat                 |         1 | varchar(28)
 catdt               |         1 | varchar(36)
 devicevendor        |         1 | varchar(20)
 dtz                 |         1 | varchar(32)
 dvc                 |         1 | varchar(24)
 filetype            |         1 | varchar(20)
 mrt                 |         1 | varchar(26)
 _cefver             |         1 | varchar(20)
 agentzoneuri        |         1 | varchar(180)
 agt                 |         1 | varchar(24)
 aid                 |         1 | varchar(50)
 atz                 |         1 | varchar(32)
 av                  |         1 | varchar(24)
 categorybehavior    |         1 | varchar(28)
 categorydevicegroup |         1 | varchar(24)
 deviceseverity      |         1 | varchar(20)
 eventid             |         1 | varchar(20)
 name                |         1 | varchar(78)
 rt                  |         1 | varchar(26)
 severity            |         1 | varchar(20)
 signatureid         |         1 | varchar(20)
(36 rows)
```

3. Query several columns from the logs_view :

```
=> \x
Expanded display is on.
VMart=> select version, devicevendor, deviceversion, name, severity, signatureid
  from logs_view;
-[ RECORD 1 ]-+---------------------------------------
version       | 0
devicevendor  | ArcSight
deviceversion | 6.0.3.6664.0
name          | Agent [test] type [testalertng] started
severity      | Low
signatureid   | agent:030
```

Use the fcefparser delimiter parameter

In this example, you use the fcefparser delimiter parameter to query events located in California, New Mexico, and Arizona.

1. Create a new columnar table, CEFData3 :

```
=> CREATE TABLE CEFData3(eventId INT, location VARCHAR(20));
CREATE TABLE
```

2. Using the delimiter=',' parameter, load some CEF data into the table:

```
=> COPY CEFData3 FROM stdin PARSER fcefparser(delimiter=',');
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> eventId=1,location=California
>> eventId=2,location=New Mexico
>> eventId=3,location=Arizona
>> \.
```

3. Query the table:

```
=> SELECT eventId, location FROM CEFData3;
 eventId |  location
---------+------------
       1 | California
       2 | New Mexico
       3 | Arizona
(3 rows)
```

Complex types

Tables can have columns of complex types, including nested complex types. You can use the ROW (struct), ARRAY , and SET types in native and external tables, including flex tables. Sets are limited to one-dimensional collections of scalar types. A limited MAP type is available for external tables, but you can use ARRAY and ROW to express a map instead. Selected parsers support loading data with complex types.

You can define a column for heterogeneous combinations of the ARRAY and ROW types: a struct containing array fields or an array of structs. These types can be nested up to the maximum nesting depth of 100.

Restrictions for native tables

Complex types used in native tables have some restrictions, in addition to the restrictions for individual types listed on their reference pages:

- A native table must have at least one column that is a primitive type or a native array (one-dimensional array of a primitive type). If a flex table has real columns, it must also have at least one column satisfying this restriction.
- Complex type columns cannot be used in ORDER BY or PARTITION BY clauses nor as FILLER columns.
- Complex type columns cannot have constraints .
- Expressions returning complex types cannot be used as projection columns, and projections cannot be segmented or ordered by columns of complex types.
- Tables with columns of complex types cannot use DEFAULT and SET USING.

See CREATE TABLE and ALTER TABLE for additional restrictions.

Deriving a table definition from the data

You can use the INFER_TABLE_DDL function to inspect Parquet, ORC, JSON, or Avro data and produce a starting point for a table definition. This function returns a CREATE TABLE statement, which might require further editing. For columns where the function could not infer the data type, the function labels the type as unknown and emits a warning. For VARCHAR and VARBINARY columns, you might need to adjust the length. Always review the statement the function returns, but especially for tables with many columns, using this function can save time and effort.

Parquet, ORC, and Avro files include schema information, but JSON files do not. For JSON, the function inspects the raw data to produce one or more candidate table definitions. In the following example, two input files differ in the structure of the menu column:

```
=> SELECT INFER_TABLE_DDL ('/data/*.json'
    USING PARAMETERS table_name='restaurants', format='json',
max_files=3, max_candidates=3);
WARNING 0:  This generated statement contains one or more float types which might lose precision
WARNING 0:  This generated statement contains one or more varchar/varbinary types which default to length 80

                INFER_TABLE_DDL
-------------------------------------------------------------------------------
 Candidate matched 1/2 of total files(s):
  create table "restaurants"(
    "cuisine" varchar,
    "location_city" Array[varchar],
    "menu" Array[Row(
      "item" varchar,
      "price" float
    )],
    "name" varchar
  );
Candidate matched 1/2 of total files(s):
  create table "restaurants"(
    "cuisine" varchar,
    "location_city" Array[varchar],
    "menu" Array[Row(
      "items" Array[Row(
        "item" varchar,
        "price" numeric
      )],
      "time" varchar
    )],
    "name" varchar
  );

(1 row)
```

Alternative to strong typing

Though you can fully specify a column representing any combination of ROW and ARRAY types, there might be cases where you prefer a more flexible approach. If the data contains a struct with hundreds of fields, only a few of which you need, you might prefer to extract just those few at query time instead of defining all of the fields. Similarly, if the data structure is likely to change, you might prefer to defer fully specifying the complex types. You can use flexible columns as an alternative to fully specifying the structure of a complex column. This is the same approach used for flex tables, where all data is initially loaded into a single binary column and materialized from there as needed. See Flexible complex types for more information about using this approach.

In this section

- Structs
- Arrays
- Flexible complex types
- System tables for complex types

Structs

Columns can contain structs, which store property-value pairs. For example, a struct representing an address could have strings for the street address and city/state and an integer for the postal code:

```
{ "street":"150 Cambridgepark Dr.",
  "city":"Cambridge MA",
  "postalcode":02140}
```

Struct fields can be primitive types or other complex types.

Use the ROW expression to define a struct column. In the following example, the data has columns for customer name, address, and account number, and the address is a struct in the data. The types you declare in Vertica must be compatible with the types in the data you load into them.

```
=> CREATE TABLE customers (
   name VARCHAR,
   address ROW(street VARCHAR, city VARCHAR, zipcode INT),
   accountID INT);
```

Within the ROW, you specify the fields and their data types using the same syntax as for columns. Vertica treats the ROW as a single column for purposes of queries.

Structs can contain other structs. In the following example, employees have various personal information, including an address which is itself a struct.

```
=> CREATE TABLE employees(
   employeeID INT,
   personal ROW(
     name VARCHAR,
     address ROW(street VARCHAR, city VARCHAR, zipcode INT),
     taxID INT),
   department VARCHAR);
```

Structs can contain arrays of primitive types, arrays, or structs.

```
=> CREATE TABLE customers(
 name VARCHAR,
 contact ROW(
   street VARCHAR,
   city VARCHAR,
   zipcode INT,
   email ARRAY[VARCHAR]
 ),
 accountid INT );
```

When defining an external table with Parquet or ORC data, Vertica requires the definition of the table to match the schema of the data. For example, with the data used in the previous employees example, the following definition is an error:

```
=> CREATE EXTERNAL TABLE employees(
   employeeID INT,
   personal ROW(
     name VARCHAR,
     address ROW(street VARCHAR, city VARCHAR),
     zipcode INT,
     taxID INT),
   department VARCHAR)
 AS COPY FROM '...' PARQUET;
ERROR 9151: Datatype mismatch [...]
```

The data contains an address struct with three fields (street, city, zipcode), so the external table must also use a ROW with three fields. Changing the ROW to have two fields and promoting one of the fields to the parent ROW is a mismatch. Each ROW must match and, if structs are nested in the data, the complete structure must match.

For native tables, you can specify which columns to load from the data, so you do not need to account for all of them. For the columns you load, the definition of the table must match the schema in the data file. Some parsers report fields found in that data that are not part of the table definition.

Handling nulls

If a struct exists but a field value is null, Vertica assigns NULL as its value in the ROW. A struct where all fields are null is treated as a ROW with null fields. If the struct itself is null, Vertica reads the ROW as NULL.

Queries

See Rows (structs) .

Restrictions

ROW columns have several restrictions:

- Maximum nesting depth is 100.
- Vertica tables support up to 9800 columns and fields. The ROW itself is not counted, only its fields.
- ROW columns cannot use any constraints (such as NOT NULL) or defaults.

- ROW fields cannot be auto_increment or setof.
- ROW definition must include at least one field.
- Row is a reserved keyword within a ROW definition, but is permitted as the name of a table or column.
- Tables containing ROW columns cannot also contain IDENTITY, default, SET USING, or named sequence columns.

Arrays

Columns can contain arrays, which store ordered lists of elements of the same type. For example, an address column could use an array of strings to store multiple addresses that an individual might have, such as ['668 SW New Lane', '518 Main Ave', '7040 Campfire Dr'] .

There are two types of arrays:

- Native array: a one-dimensional array of a primitive type.
- Non-native array: all other supported arrays, including arrays that contain other arrays (multi-dimensional arrays) or structs (ROWs). Non-native arrays have some usage restrictions .

Use the ARRAY type to define an array column, specifying the type of its elements (a primitive type, a ROW (struct), or an array):

```
=> CREATE TABLE orders
  (orderkey    INT,
   custkey     INT,
   prodkey     ARRAY[VARCHAR(10)],
   orderprices ARRAY[DECIMAL(12,2)],
   orderdate   DATE
  );
```

If an array is multi-dimensional, represent it as an array containing an array:

```
ARRAY[ARRAY[FLOAT]]
```

Queries

See Arrays and sets (collections) .

Restrictions

- Native arrays support only data of primitive types, for example, int, UUID, and so on.
- Array dimensionality is enforced. A column cannot contain arrays of varying dimensions. For example, a column that contains a three-dimensional array can only contain other three-dimensional arrays; it cannot simultaneously include a one-dimensional array. However, the arrays in a column can vary in size, where one array can contain four elements while another contains ten.
- Array bounds, if specified, are enforced for all operations that load or alter data. Unbounded arrays may have as many elements as will fit in the allotted binary size.
- An array has a maximum binary size. If this size is not set when the array is defined, a default value is used.
- Arrays do not support LONG types (like LONG VARBINARY or LONG VARCHAR) or user-defined types (like Geometry).

Flexible complex types

When defining tables, you can use strongly-typed complex types to fully describe any combination of structs and arrays. However, there are times when you might prefer not to:

- If the data contains a struct with a very large number of fields, and in your queries you will need only a few of them, you can avoid having to enumerate the rest in the table DDL. Further, a deeply-nested set of structs could exceed the nesting limit for the table if fully specified.
- If the data schema is still evolving, you can delay finalizing strongly-typed DDL.
- If you anticipate the introduction of new fields in the data, you can use flexible types to discover them. A table with strong typing, on the other hand, would silently ignore those values. For an example of using flexible types to discover new fields, see Strong and Flexible Typing .

Flexible types are a way to store complex or unstructured data as a binary blob in one column, in a way that allows access to individual elements of that data. This is the same approach that Vertica uses with flex tables , which support loading unstructured or semi-structured data. In a flex table, all data from a source is loaded into a single VMap column named __raw__ . From this column you can materialize other columns, such as a specific field in JSON data, or use special lookup functions in queries to read values directly out of the __raw__ column.

Vertica uses a similar approach with complex types. You can describe the types fully using the ROW and ARRAY types in your table definition, or you can instead treat a complex type as a flexible type and not fully describe it. Each complex type that you choose to treat this way becomes its own flex-style column. You are not limited to a single column containing all data as in flex tables; instead, you can treat any complex type column, no matter how deeply it nests other types, as one flex-like column.

Defining flexible columns

To use a flexible complex type, declare the column as LONG VARBINARY . You might also need to set other parameters in the parser, as described in the parser documentation.

Consider a Parquet file with a restaurants table and the following columns:

- name: varchar
- cuisine type: varchar
- location (cities): array[varchar]
- menu: array of structs, each struct having an item name and a price

This data contains two complex columns, location (an array) and menu (an array of structs). The following example defines both columns as flexible columns by using LONG VARBINARY :

```
=> CREATE EXTERNAL TABLE restaurants(name VARCHAR, cuisine VARCHAR,
      location_city LONG VARBINARY, menu LONG VARBINARY)
   AS COPY FROM '/data/rest*.parquet'
   PARQUET(allow_long_varbinary_match_complex_type='True');
```

The allow_long_varbinary_match_complex_type parameter is specific to the Parquet parser. It is required if you define any column as a flexible type. Without this parameter, Vertica tries to match the LONG VARBINARY declaration in the table to a VARBINARY column in the Parquet file, finds a complex type instead, and reports a data-type mismatch.

You need not treat all complex columns as flexible types. The following definition is also valid:

```
=> CREATE EXTERNAL TABLE restaurants(
          name VARCHAR, cuisine VARCHAR,
          location_city ARRAY[VARCHAR,50],
          menu LONG VARBINARY)
   AS COPY FROM '/data/rest*.parquet'
   PARQUET(allow_long_varbinary_match_complex_type='True');
```

For many common data formats, you can use the INFER_TABLE_DDL function to derive a table definition from a data file. This function uses strong typing for complex types in almost all cases.

Querying flexible columns

Flexible columns are stored as LONG VARBINARY , so selecting them directly produces unhelpful results. Instead, use the flex mapping functions to extract values from these columns. The MAPTOSTRING function translates the complex type to JSON, as shown in the following example:

```
=> SELECT name, location_city, MAPTOSTRING(menu) AS menu FROM restaurants;
      name       |      location_city       |                  menu
-----------------+--------+-----------------+---------------------------------------
 Bob's pizzeria    | ["Cambridge","Pittsburgh"] | {
   "0": {
      "item": "cheese pizza",
      "price": "$8.25"
   },
   "1": {
      "item": "spinach pizza",
      "price": "$10.50"
   }
}
 Bakersfield Tacos | ["Pittsburgh"]             | {
   "0": {
      "item": "veggie taco",
      "price": "$9.95"
   },
   "1": {
      "item": "steak taco",
      "price": "$10.95"
   }
}
(2 rows)
```

The menu column is an array of structs. Notice that the output is a set of key/value pairs, with the key being the array index. Bob's Pizzeria has two items on its menu, and each value is a struct. The first item ("0") is a struct with an "item" value of "cheese pizza" and a "price" of "$8.25".

You can use keys to access specific values. The following example selects the first menu item from each restaurant. Note that all keys are strings, even array indexes:

```
=> SELECT name, location_city, menu['0']['item'] AS item, menu['0']['price'] AS price FROM restaurants;
      name       |      location_city       |     item     | price
-----------------+--------------------------+--------------+-------
 Bob's pizzeria  | ["Cambridge","Pittsburgh"] | cheese pizza | $8.25
 Bakersfield Tacos | ["Pittsburgh"]           | veggie taco  | $9.95
(2 rows)
```

Instead of accessing specific indexes, you can use the MAPITEMS function in a subquery to explode a flexible type, as in the following example:

```
=>  SELECT name, location_city, menu_items['item'], menu_items['price']
   FROM (SELECT mapitems(menu, name, location_city) OVER(PARTITION BEST)
       AS (indexes, menu_items, name, location_city)
   FROM restaurants) explode_menu;
     name       |      location_city       |  menu_items   | menu_items
-----------------+--------------------------+---------------+------------
 Bob's pizzeria  | ["Cambridge","Pittsburgh"] | cheese pizza  | $8.25
 Bob's pizzeria  | ["Cambridge","Pittsburgh"] | spinach pizza | $10.50
 Bakersfield Tacos | ["Pittsburgh"]           | veggie taco   | $9.95
 Bakersfield Tacos | ["Pittsburgh"]           | steak taco    | $10.95
(4 rows)
```

For a complete list of flex mapping functions, see Flex data functions .

JSON and Avro flexible types

The parsers for JSON and Avro support both flexible and strong types for complex types. When using flexible complex types or loading into a flex table, use the flatten_maps and flatten_arrays parameters to control how the parser handles complex data. These parsers ignore these parameters for strongly-typed complex types.

The following example demonstrates the use of flexible complex types. Consider a JSON file containing the following data:

```
{
   "name" : "Bob's pizzeria",
   "cuisine" : "Italian",
   "location_city" : ["Cambridge", "Pittsburgh"],
   "menu" : [{"item" : "cheese pizza", "price" : "$8.25"},
            {"item" : "spinach pizza", "price" : "$10.50"}]
}
{
   "name" : "Bakersfield Tacos",
   "cuisine" : "Mexican",
   "location_city" : ["Pittsburgh"],
   "menu" : [{"item" : "veggie taco", "price" : "$9.95"},
            {"item" : "steak taco", "price" : "$10.95"}]
}
```

Create a table, using LONG VARBINARY for the flexible complex types, and load data specifying these parameters:

```
=> CREATE TABLE restaurant(name VARCHAR, cuisine VARCHAR,
       location_city LONG VARBINARY, menu LONG VARBINARY);

=> COPY restaurant FROM '/data/restaurant.json'
  PARSER FJSONPARSER(flatten_maps=false, flatten_arrays=false);
```

You can use Flex functions and direct access (through indexes) to return readable values:

```
=> SELECT MAPTOSTRING(location_city), MAPTOSTRING(menu) FROM restaurant;
                maptostring                |                maptostring
-------------------------------------------+-------------------------------------------
 {
    "0": "Cambridge",
    "1": "Pittsburgh"
} | {
    "0": {
        "item": "cheese pizza",
        "price": "$8.25"
    },
    "1": {
        "item": "spinach pizza",
        "price": "$10.50"
    }
}
 {
    "0": "Pittsburgh"
}                          | {
    "0": {
        "item": "veggie taco",
        "price": "$9.95"
    },
    "1": {
        "item": "steak taco",
        "price": "$10.95"
    }
}
(2 rows)

=> SELECT menu['0']['item'] FROM restaurant;
     menu
--------------
 cheese pizza
 veggie taco
(2 rows)
```

The COPY statement shown in this example sets flatten_maps to false. Without that change, the keys for the complex columns would not work as expected, because record and array keys would be "flattened" at the top level. Querying menu['0']['item'] would produce no results. Instead, query flattened values as in the following example:

```
=> SELECT menu['0.item'] FROM restaurant;
     menu
--------------
 veggie taco
 cheese pizza
(2 rows)
```

Flattening directives apply to the entire COPY statement. You cannot flatten some columns and not others, or prevent flattening values in a complex column that is itself within a flattened flex table. Because flexible complex types and strongly-typed complex types require different values for flattening, you cannot combine strong and flexible complex types in the same load operation.

System tables for complex types

Information about all complex types is recorded in the COMPLEX_TYPES system table. You must have read permission for the external table that uses a type to see its entries in this system table. Complex types are not shown in the TYPES system table.

For ROW types, each row in COMPLEX_TYPES represents one field of one ROW. The field name is the name used in the table definition if present, or a generated name beginning with _field otherwise. Each row also includes the (generated) name of its containing type, a string beginning with _ct_. ("CT" stands for "complex type".)

The following example defines one external table and then shows the types in COMPLEX_TYPES:

```
=> CREATE EXTERNAL TABLE warehouse(
   name VARCHAR, id_map MAP<INT,VARCHAR>,
   data row(record INT, total FLOAT, description VARCHAR(100)),
   prices ARRAY[INT], comment VARCHAR(200), sales_total FLOAT, storeID INT)
 AS COPY FROM ... PARQUET;

=> SELECT type_id,type_kind,type_name,field_id,field_name,field_type_name,field_position
   FROM COMPLEX_TYPES ORDER BY type_id,field_name;

     type_id      | type_kind |       type_name       | field_id | field_name  | field_type_name | field_position
------------------+-----------+-----------------------+----------+-------------+-----------------+----------------
 45035996274278280 | Map       | _ct_45035996274278280 |        6 | key         | int             |              0
 45035996274278280 | Map       | _ct_45035996274278280 |        9 | value       | varchar(80)     |              1
 45035996274278282 | Row       | _ct_45035996274278282 |        9 | description | varchar(80)     |              2
 45035996274278282 | Row       | _ct_45035996274278282 |        6 | record      | int             |              0
 45035996274278282 | Row       | _ct_45035996274278282 |        7 | total       | float           |              1
 45035996274278284 | Array     | _ct_45035996274278284 |        6 |             | int             |              0
(6 rows)
```

This table shows the fields for the two ROW types defined in the table. When a ROW contains another ROW, as is the case here with the nested address field, the field_type_name column uses the generated name of the contained ROW. The same number, minus the leading " *ct* ", serves as the field_id.

Schema evolution

When you load data, Vertica must be able to match the columns or fields in the data to the columns defined in the table. Typically, you use strong typing to specify the table columns and parsers reject data that does not match the table definition. Some parsers cause the entire load to fail if the data does not match, and some load what they can and produce warnings about unhandled data.

For long-lived databases or more volatile domains, new columns and fields can appear in data, disrupting existing ETL pipelines. Some parsers provide ways to handle differences between the data and the table definition.

Parquet

By default, the Parquet parser uses strong schema matching. This means that all columns in the Parquet data must be loaded in the same order as in the data. Unmatched columns produce load errors:

```
=> CREATE TABLE orders_summary(orderid INT, accountid INT);

=> COPY orders_summary FROM '/data/orders.parquet' PARQUET;
ERROR 9135:  Attempt to load 2 columns from a parquet source [/data/orders.parquet] that has 3 columns
```

You can pick out only the specific columns you want using loose schema matching. Use the do_soft_schema_match_by_name parameter in the PARQUET parser. With loose schema matching, columns in the data are matched to columns in the table by their names. Columns in the data that are not part of the table definition are ignored:

```
=> COPY orders_summary FROM '/data/orders.parquet'
   PARQUET(do_soft_schema_match_by_name='true');
 Rows Loaded
-------------
           3
(1 row)
```

Alternatively, you can use ALTER TABLE to add new columns to the table definition and continue to use strong schema matching.

JSON and Avro

By default, the JSON and Avro parsers skip fields in the data that are not part of the table definition and proceed with the rest of the load. This behavior is in contrast to the Parquet and ORC parsers, which by default require that a table consume all of the columns in the data. This means that the JSON and Avro parsers are more flexible in the face of variability in the data, but also that you must monitor your loads more closely for new fields.

When creating tables that will use JSON or Avro data, a good first step is to infer a table definition from a sample data file using INFER_TABLE_DDL . You can use the results to define a table. If later data files add more fields, the parser emits a warning and logs information about the new fields in the UDX_EVENTS system table. You can use this information to decide whether to alter the table definition or ignore the new fields.

If you are only interested in certain fields and don't care about new fields found in data loads, you can use the suppress_warnings parameter on the JSON or Avro parser to ignore them. If you suppress warnings, you can separately check for new fields from time to time with INFER_TABLE_DDL or by loading sample data with warnings enabled and without committing (COPY...NO COMMIT).

For a detailed discussion of this workflow, see Schema changes in Data exploration .

Handling Non-UTF-8 input

Vertica supports loading data files in the Unicode UTF-8 format. You can load ASCII data, which is UTF-8 compatible. Character sets like ISO 8859-1 (Latin1) are incompatible with UTF-8 and are not directly supported.

If you have data that does not meet the UTF-8 standard, you can modify the data during the load or you can transform the data files before loading.

Checking data format

Before loading data from text files, you can use several Linux tools to ensure that your data is in UTF-8 format. The file command reports the encoding of any text files. For example:

```
$ file Date_Dimension.tbl
Date_Dimension.tbl: ASCII text
```

The file command could indicate ASCII text even though the file contains multibyte characters.

To check for multibyte characters in an ASCII file, use the wc command. For example:

```
$ wc Date_Dimension.tbl
  1828   5484 221822 Date_Dimension.tbl
```

If the wc command returns an error such as Invalid or incomplete multibyte or wide character , the data file is using an incompatible character set.

This example shows two files that are not UTF-8 data files:

```
$ file data*
data1.txt: Little-endian UTF-16 Unicode text
data2.txt: ISO-8859 text
```

The results indicate that neither of the files is in UTF-8 format.

Converting data while loading

You can remove or replace non-UTF-8 characters in text data during the load. The MAKEUTF8 function removes such characters by default, or you can specify a replacement string.

The following example shows how to use this function during a load. The original data is loaded into the orig_name column, and the transformed data is loaded into the name column. Typically you would use a FILLER column for the original value instead of adding the column to the table definition; this example adds the column to show the differences side by side.

```
=> CREATE TABLE people (orig_name VARCHAR, name VARCHAR);
CREATE TABLE

=> COPY people (orig_name, name AS MAKEUTF8(orig_name)) FROM ...;
 Rows Loaded
-------------
           8
(1 row)

=> SELECT * FROM people;
orig_name |  name
----------+--------
 Dáithí    | Dith
 Fíona     | Fona
 Móirín    | Mirn
 Róisín    | Risn
 Séamus    | Samus
 Séan      | San
 Tiarnán   | Tiarnn
 Áine      | ine
(8 rows)
```

For general information about transforming data, see Transforming data during loads .

Converting files before loading data

To convert files before loading them into Vertica, use the iconv UNIX command. For example, to convert the data2.txt file from the previous example, use the iconv command as follows:

```
$ iconv -f ISO88599 -t utf-8 data2.txt > data2-utf8.txt
```

See the man pages for file and iconv for more information.

Checking UTF-8 compliance after loading data

After loading data, use the ISUTF8 function to verify that all of the string-based data in the table is in UTF-8 format. For example, if you loaded data into a table named people that has a VARCHAR column named name , you can use this statement to verify that all of the strings are UTF-8 encoded:

```
=> SELECT name FROM people WHERE NOT ISUTF8(name);
```

If all of the strings are in UTF-8 format, the query should not return any rows.

Transforming data during loads

To promote a consistent database and reduce the need for scripts to transform data at the source, you can transform data with an expression as part of loading. Transforming data while loading lets you compute values to insert into a target column, either from other columns or from values in the data that you load as FILLER columns (see Deriving Table Columns From Data File Columns). You can transform data to be loaded into columns of scalar types and native arrays, but not other complex types .

For example, you might have text data that is not compatible with UTF-8, the encoding that Vertica expects. You can use the MAKEUTF8 function during load to remove or replace non-UTF-8 characters, as illustrated in Converting Data While Loading . Or you might want to extract fields for day, month, and year from a single input date.

When transforming data during a load, you load the data into a column normally, and then use that column in an expression to populate another column. The COPY statement must always contain at least one parsed column, which can be a FILLER column. You can intersperse parsed and computed columns in a COPY statement.

The following example extracts day, month, and year columns from a single input date column:

```
=> CREATE TABLE purchases
   (id INT, year VARCHAR(10), month VARCHAR(10), day VARCHAR(10), ts TIMESTAMP);

=> COPY purchases (id, year AS TO_CHAR(ts,'YYYY'),
                month AS TO_CHAR(ts,'MM'), day AS TO_CHAR(ts, 'DD'),
                ts FORMAT 'YYYY-MM-DD') FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1943|2021-03-29
>> 1256|2021-03-30
>> 1512|2021-03-31
>> \.

=> SELECT * FROM purchases;
  id  | year | month | day |         ts
------+------+-------+-----+---------------------
 1256 | 2021 | 03    | 30  | 2021-03-30 00:00:00
 1512 | 2021 | 03    | 31  | 2021-03-31 00:00:00
 1943 | 2021 | 03    | 29  | 2021-03-29 00:00:00
(3 rows)
```

The input data has two columns, id and ts (timestamp). The COPY statement specifies the format of the timestamp column using the FORMAT option. The TO_CHAR function uses that format information to extract the year , month , and day columns.

Using expressions in COPY statements

The expression in a COPY statement can be as simple as a single column, or more complex, such as a case statement for multiple columns. An expression can specify multiple columns, and multiple expressions can refer to the same parsed column. You can use expressions for columns of all supported data types.

COPY expressions can use many SQL functions, operators, constants, NULLs, and comments, including these functions:

- Date/time
- Formatting functions
- String
- Null-handling
- System information

Requirements and restrictions:

- COPY expressions cannot use SQL meta-functions, analytic functions, aggregate functions, or computed columns.
- For computed columns, you must list all parsed columns in the COPY statement expression. Do not specify FORMAT or RAW in the source data for a computed column.
- The return data type of the expression must be coercible to that of the target column. Parsed column parameters are also coerced to match the expression.

Handling expression errors

Errors in expressions within your COPY statement are SQL errors. As such, they are handled differently from parse errors. When a parse error occurs, COPY rejects the row and adds it to the rejected data file or table. COPY also adds the reason for a rejected row to the exceptions file or the rejected data table. For example, COPY parsing does not implicitly cast data types. If a type mismatch occurs between the data being loaded and a column type (such as attempting to load a text value into a FLOAT column), COPY rejects the row and continues processing.

If an error occurs in an expression in your COPY statement, then by default the entire load fails. For example, if your COPY statement uses a function expression, and a syntax error exists in that expression, the entire load is rolled back. All SQL errors, including ones caused by rolling back the COPY, are stored in the Vertica log file. However, unlike parse rejections and exception messages, SQL expression errors are brief and may require further research.

You can have COPY treat errors in transformation expressions like parse errors. Rejected rows are added to the same file or table, and exceptions are added to the same exceptions file or table. To enable this behavior, set the CopyFaultTolerantExpressions configuration parameter to 1. (See General parameters .)

Loading data with expression rejections is potentially slower than loading with the same number of parse rejections. Enable expression rejections if your data has a few bad rows, to allow the rest of the data to be loaded. If you are concerned about the time it takes to complete a load with many bad rows, use the REJECTMAX parameter to set a limit. If COPY finds more than REJECTMAX bad rows, it aborts and rolls back the load.

See Handling messy data for more information about managing rejected data.

Deriving table columns from data file columns

When loading data, your source data might contain one or more columns that do not exist in the target table. Or, the source and target tables have matched columns, but you want to omit one or more source columns from the target table.

Use the FILLER parameter to identify a column of source data that COPY can ignore or use to compute new values that are loaded into the target table. The following requirements apply:

- Define the FILLER parameter data type so it is compatible with the source data. For example, be sure to define a VARCHAR in the target table so its length can contain all source data; otherwise, data might be truncated. You can specify multiple filler columns by using the FILLER parameter more than once in the COPY statement.
  Important
  If the source field's data type is VARCHAR, be sure to set the VARCHAR length to ensure that the combined length of all FILLER source fields does not exceed the target column's defined length; otherwise, the COPY command might return with an error.

- The name of the filler column must not match the name of any column in the target table.

In the following example, the table has columns for first name, last name, and full name, but the data being loaded contains columns for first, middle, and last names. The COPY statement reads all of the source data but only loads the source columns for first and last names. It constructs the data for the full name by concatenating each of the source data columns, including the middle name. The middle name is read as a FILLER column so it can be used in the concatenation, but is ignored otherwise. (There is no table column for middle name.)

```
=> CREATE TABLE names(first VARCHAR(20), last VARCHAR(20), full VARCHAR(60));
CREATE TABLE
=> COPY names(first,
            middle FILLER VARCHAR(20),
            last,
            full AS first||' '||middle||' '||last)
     FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> Marc|Gregory|Smith
>> Sue|Lucia|Temp
>> Jon|Pete|Hamilton
>> \.
=> SELECT * from names;
 first |   last   |        full
-------+----------+--------------------
 Jon   | Hamilton | Jon Pete Hamilton
 Marc  | Smith    | Marc Gregory Smith
 Sue   | Temp     | Sue Lucia Temp
(3 rows)
```

Distributing a load

Vertica can divide the work of loading data among multiple database nodes, taking advantage of parallelism to speed up the operation. How this is done depends on where the data is and what types of parallelism the parsers support.

Vertica can be most effective in distributing a load when the data to be loaded is found in shared storage available to all nodes. Sometimes, however, data is available only on specific nodes, which you must specify.

Types of load parallelism

Vertica supports several types of parallelism. All built-in parsers support distributed load, and some parsers support apportioned load and cooperative parse. The reference pages for individual parsers include the kinds of load parallelism they support.

Load parallelism is enabled by default but can be disabled. The configuration parameters named in this section are described in General parameters .

Distributed load

Vertica distributes files in a multi-file load to several nodes to load in parallel, instead of loading all of them on a single node. Vertica automatically distributes a load if the files are accessible to all nodes and you do not restrict participating nodes.

Apportioned load and cooperative parse both require an input that can be divided at record boundaries. The difference is that cooperative parse does a sequential scan to find record boundaries, while apportioned load first jumps (seeks) to a given position and then scans. Some formats, like generic XML, do not support seeking.

Apportioned load

In an apportioned load, Vertica divides a single large file or other single source into segments (portions), which it assigns to several nodes to load in parallel. Apportioned load divides the load at planning time, based on available nodes and cores on each node.

To use apportioned load, you must ensure that the source is reachable by all participating database nodes. You typically use apportioned load with distributed file systems.

Apportioned load is enabled by default for parsers that support it. To disable it, set the EnableApportionLoad configuration parameter to 0.

Cooperative parse

By default, Vertica parses a data source in a single thread on one database node. If a parser supports cooperative parse, the node instead uses multiple threads to parallelize the parse. Cooperative parse divides a load at execution time, based on how threads are scheduled.

Cooperative parse is enabled by default for parsers that support it. To disable it, set the EnableCooperativeParse configuration parameter to 0.

Loading on specific nodes

You can indicate which node or nodes should parse an input path by using any of the following:

- A node name: ON *node*
- A set of nodes: ON *nodeset* (see Specifying Distributed File Loads)
- ON ANY NODE (default for HDFS and cloud paths)
- ON EACH NODE

Using the ON ANY NODE clause indicates that the source path is available on all of the nodes. If you specify this clause, COPY parses the files from any node in the cluster. If the path contains a glob, COPY expands the glob on the initiator node. ON ANY NODE is the default for all file systems except Linux.

Using the ON *nodeset* clause indicates that the source file is on all named nodes. If you specify this clause, COPY opens the file and parses it from any node in the set. Be sure that the source file you specify is available and accessible on each applicable cluster node.

ON *node* and ON ANY NODE load each file once, choosing one node to perform the load. If, instead, you have different data on each node and you want to load all of it, and the path is the same, use ON EACH NODE :

```
=> COPY myTable FROM '/local_export/*.dat' ON EACH NODE;
```

If the path is not valid on all nodes, COPY loads the valid paths and produces a warning. If the path is a shared location, COPY loads it only once as for ON ANY NODE .

Use ON EACH NODE when you want to load from the same path on every node and the files are different. For example, if you have machine-specific data, such as system event logs, or if an operation wrote data on each node individually, you can use this clause to load all of it.

If the data to be loaded is on a client, use COPY FROM LOCAL instead of specifying nodes. All local files are loaded and parsed serially with each COPY statement, so you cannot perform parallel loads with the LOCAL option.

Specifying distributed file loads

You can direct individual files in a multi-file load to specific nodes:

```
=> COPY t FROM '/data/file1.dat' ON v_vmart_node0001, '/data/file2.dat' ON v_vmart_node0002;
```

You can use globbing (wildcard expansion) to specify a group of files with the ON ANY NODE option:

```
=> COPY t FROM '/data/*.dat' ON ANY NODE;
```

You can limit the nodes that participate in an apportioned load. Doing so is useful if you need to balance several concurrent loads. Vertica apportions each load individually; it does not account for other loads that might be in progress on those nodes. You can, therefore, potentially speed up your loads by managing apportioning yourself:

```
=> COPY t FROM '/data/big1.dat' ON (v_vmart_node0001, v_vmart_node0002, v_vmart_node0003),
     '/data/big2.dat' ON (v_vmart_node0004, v_vmart_node0005);
```

You can specify a compression type for each path. However, because file compression is a filter, you cannot use apportioned load for a compressed file.

Specifying distributed loads with sources

You can also apportion loads using COPY WITH SOURCE. You can create sources and parsers with the user-defined load (UDL) API. If both the source and parser support apportioned load, and EnableApportionLoad is set, then Vertica attempts to divide the load among nodes.

The following example shows a load that you could apportion:

```
=> COPY t WITH SOURCE MySource() PARSER MyParser();
```

The built-in delimited parser supports apportioning, so you can use it with a user-defined source:

```
=> COPY t WITH SOURCE MySource();
```

Number of load streams

Although the number of files you can load is not restricted, the optimal number of load streams depends on several factors, including:

- Number of nodes
- Physical and logical schemas
- Host processors
- Memory
- Disk space

Using too many load streams can deplete or reduce system memory required for optimal query processing. See Best practices for managing workload resources for advice on configuring load streams.

Using transactions to stage a load

By default, COPY automatically commits itself and other current transactions except when loading temporary tables or querying external tables. You can override this behavior by qualifying the COPY statement with the NO COMMIT option. When you specify NO COMMIT , Vertica does not commit the transaction until you explicitly issue a COMMIT statement.

You can use COPY...NO COMMIT in two ways:

- Execute multiple COPY commands as a single transaction.
- Check data for constraint violations before committing the load.

Combine multiple COPY statements in the same transaction

When you combine multiple COPY...NO COMMIT statements in the same transaction, Vertica can consolidate the data for all operations into fewer ROS containers, and thereby perform more efficiently.

For example, the following set of COPY...NO COMMIT statements performs several copy statements sequentially, and then commits them all. In this way, all of the copied data is either committed or rolled back as a single transaction.

```
COPY... NO COMMIT;
COPY... NO COMMIT;
COPY... NO COMMIT;
COPY X FROM LOCAL NO COMMIT;
COMMIT;
```

> **Tip**
> Be sure to commit or roll back any previous DML operations before you use COPY...NO COMMIT . Otherwise, COPY...NO COMMIT is liable to include earlier operations that are still in progress, such as INSERT , in its own transaction. In this case, the previous operation and copy operation are combined as a single transaction and committed together.

Check constraint violations

If constraints are not enforced in the target table, COPY does not check for constraint violations when it loads data. To troubleshoot loaded data for constraint violations, use COPY...NO COMMIT with ANALYZE_CONSTRAINTS . Doing so enables you detect constraint violations before you commit the load operation and, if necessary, roll back the operation. For details, see Detecting constraint violations .

Handling messy data

Loading data with COPY has two main phases, parsing and loading. During parsing, if COPY encounters errors it rejects the faulty data and continues loading data. Rejected data is created whenever COPY cannot parse a row of data. Following are some parser errors that can cause a rejected row:

- Unsupported parser options
- Incorrect data types for the table into which data is being loaded, including incorrect data types for members of collections
- Malformed context for the parser in use
- Missing delimiters

Optionally, COPY can reject data and continue loading when transforming data during the load phase. This behavior is controlled by a configuration parameter. By default, COPY aborts a load if it encounters errors during the loading phase.

Several optional parameters let you determine how strictly COPY handles rejections. For example, you can have COPY fail when it rejects a single row, or allow a specific number of rejections before the load fails. This section presents the parameters to determine how COPY handles rejected data.

Save rejected rows (REJECTED DATA and EXCEPTIONS)

The COPY statement automatically saves a copy of each rejected row in a rejections file. COPY also saves a corresponding explanation of what caused the rejection in an exceptions file. By default, Vertica saves both files in a database catalog subdirectory, called CopyErrorLogs , as shown in this example:

```
v_mart_node003_catalog\CopyErrorLogs\trans-STDIN-copy-from-rejected-data.1
v_mart_node003_catalog\CopyErrorLogs\trans-STDIN-copy-from-exceptions.1
```

You can optionally save COPY rejections and exceptions in one of two other ways:

- Use the REJECTED DATA and EXCEPTIONS options to save both outputs to locations of your choice. REJECTED DATA records rejected rows, while EXCEPTIONS records a description of why each row was rejected. If a path value is an existing directory or ends in '/', or the load includes multiple sources, files are written in that directory. (COPY creates the directory if it does not exist.) If a path value is a file, COPY uses it as a file prefix if multiple files are written.
- Use the REJECTED DATA AS TABLE option. This option writes both the rejected data and the exception descriptions to the same table. For more information, see Saving rejected data to a table .

> **Note**
> Vertica recommends saving rejected data to a table. However, saving to a table excludes saving to a default or specific rejected data file.

If you save rejected data to a table, the table files are stored in the data subdirectory. For example, in a VMart database installation, rejected data table records are stored in the RejectionTableData directory as follows:

```
=> cd v_mart_node003_data\RejectionTableData\
=> ls
TABLE_REJECTED_RECORDS_"bg"_mytest01.example.-25441:0x6361_45035996273805099_1.1
TABLE_REJECTED_RECORDS_"bg"_mytest01.example.-25441:0x6361_45035996273805113_2.2
.
.
.
TABLE_REJECTED_RECORDS_"delimr"_mytest01.example.-5958:0x3d47_45035996273815749_1.1
TABLE_REJECTED_RECORDS_"delimr"_mytest01.example.-5958:0x3d47_45035996273815749_1.2
```

COPY LOCAL rejected data

For COPY LOCAL operations, if you use REJECTED DATA or EXCEPTIONS with a file path, the files are written on the client. If you want rejections to be available on all nodes, use REJECTED DATA AS TABLE instead of REJECTED DATA .

Enforce truncating or rejecting rows (ENFORCELENGTH)

When parsing CHAR, VARCHAR, BINARY, or VARBINARY data, rows may exceed the target table length. By default, COPY truncates such rows without rejecting them. Use the ENFORCELENGTH option to instead reject rows that exceed the target table.

For example, loading 'abc' into a table column specified as VARCHAR(2) results in COPY truncating the value to 'ab' and loading it. Loading the same row with the ENFORCELENGTH option causes COPY to reject the row.

> **Note**
> Vertica supports NATIVE and NATIVE VARCHAR values up to 65K. If any value exceeds this limit, COPY rejects the row, even when ENFORCELENGTH is not in use.

Specify a maximum number of rejections (REJECTMAX)

The REJECTMAX option specifies the maximum number of logical records that can be rejected before a load fails. A rejected row consists of the data that could not be parsed (or optionally transformed) into the corresponding data type during a bulk load. Rejected data does not indicate referential constraints. For information about using constraints, and the option of enforcing constraints during bulk loading, see Constraints .

When the number of rejected records becomes equal to the REJECTMAX value, the load fails. If you do not specify a value for REJECTMAX , or if the value is 0, COPY allows an unlimited number of exceptions to occur.

If you allow COPY to reject rows and proceed when it encounters transformation errors, consider using REJECTMAX to limit the impact. See Handling Transformation Errors .

Handling transformation errors

By default, COPY aborts a load if it encounters errors when performing transformations. This is the default because rejecting transformation errors is potentially more expensive than rejecting parse errors. Sometimes, however, you would prefer to load the data anyway and reject the problematic rows, the way it does for parse errors.

To have COPY treat errors in transformation expressions like parse errors, set the CopyFaultTolerantExpressions configuration parameter to 1. (See General parameters .) Rows that are rejected during transformation, in the expression-evaluation phase of a data load, are written to the same destination as rows rejected during parsing. Use REJECTED DATA or REJECTED DATA AS TABLE to specify the output location.

You might want to enable transformation rejections if your data contains a few bad rows. By enabling these rejections, you can load the majority of your data and proceed. Vertica recommends using REJECTMAX when enabling transformation rejections.

If your data contains many bad values, then the performance for loading the good rows could be worse than with parser errors.

Abort data loads for any error (ABORT ON ERROR)

Using the ABORT ON ERROR option is the most restrictive way to load data, because no exceptions or rejections are allowed. A COPY operation stops if any row is rejected. No data is loaded and Vertica rolls back the command.

If you use ABORT ON ERROR as part of a CREATE EXTERNAL TABLE statement, the option is used whenever a query references the external table. The offending error is saved in the COPY exceptions or rejected data file.

Understanding row rejections and rollback errors

Depending on the type of error that COPY encounters, Vertica does one of the following:

- Rejects the offending row and loads other rows into a table
- Rolls back the entire COPY statement without loading any data

> **Note**
> If you specify ABORT ON ERROR with the COPY statement, the load automatically rolls back if COPY cannot parse any row.

COPY cannot parse rows that contain any of the following:

- Incompatible data types
- Missing fields
- Missing delimiters

COPY rolls back a load if it encounters any of these conditions:

- Server-side errors, such as lack of memory
- Primary key or foreign key constraint violations
- Loading NULL data into a NOT NULL column
- Transformation errors (by default)

This example illustrates what happens when Vertica cannot coerce a row to the requested data type. For example, in the following COPY statement, "a::INT + b::INT" is a SQL expression in which a and b are derived values:

```
=> CREATE TABLE t (i INT);
=> COPY t (a FILLER VARCHAR, b FILLER VARCHAR, i AS a::INT + b::INT)
   FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> cat|dog
>> \.
```

Vertica cannot parse the row to the requested data type and rejects the row:

```
ERROR 2827:  Could not convert "cat" from column "*FILLER*".a to an int8
```

If a resolved to 'cat' and b to 'dog' , the next expression 'cat'::INT + 'dog'::INT would return an expression evaluator error:

```
=> SELECT 'cat'::INT + 'dog'::INT;
ERROR 3681:  Invalid input syntax for integer: "cat"
```

The following COPY statement would also roll back because Vertica cannot parse the row to the requested data type:

```
=> COPY t (a FILLER VARCHAR, i AS a::INT) FROM STDIN;
```

In the following COPY statement, Vertica rejects only the offending row without rolling back the statement. Instead of evaluating the 'cat' row as a VARCHAR type, COPY parses 'cat' directly as an INTEGER.

```
=> COPY t (a FILLER INT, i AS a) FROM STDIN;
```

In the following example, transformation errors are rejected instead of aborting the load.

```
=> ALTER DATABASE DEFAULT SET CopyFaultTolerantExpressions = 1;
ALTER DATABASE

=> CREATE TABLE sales (price INTEGER);
CREATE TABLE

=> COPY sales (f FILLER VARCHAR, price AS f::INT)
   FROM STDIN REJECTED DATA AS TABLE sales_rej;

=> COPY sales FROM STDIN REJECTED DATA AS TABLE sales_rej;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1
>>2a
>>3
>>\.

=> SELECT * FROM sales;
 price
-------
     1
     3
(2 rows)

=> SELECT rejected_data, rejected_reason FROM sales_rej;
 rejected_data |                rejected_reason
---------------+-----------------------------------------------
 Tuple (2a)    | ERROR 2827:  Could not convert "2a" from column "*FILLER*".f to an int8
(1 row)
```

See also

- Cast Failures

In this section

- Saving load rejections (REJECTED DATA)
- Saving rejected data to a table

- Saving load exceptions (EXCEPTIONS)
- COPY rejected data and exception files
- COPY LOCAL rejection and exception files

Saving load rejections (REJECTED DATA)

COPY load rejections are data rows that did not load due to a parser exception or, optionally, transformation error. By default, if you do not specify a rejected data file, COPY saves rejected data files to this location:

```
catalog_dir/CopyErrorLogs/target_table-source-copy-from-rejected-data.``*``n``*
```

catalog_dir /	The database catalog files directory, for example: /home/dbadmin/VMart/v_vmart_node0001_catalog
target_table	The table into which data was loaded (*target_table*).
source	The source of the load data, which can be STDIN, or a file name, such as baseball.csv .
copy-from-rejected-data. *n*	The default name for a rejected data file, followed by *n* suffix, indicating the number of files, such as .1 , .2, .3 . For example, this default file name indicates file 3 after loading from STDIN: fw-STDIN-copy-from-rejected-data.3 .

Saving rejected data to the default location, or to a location of your choice, lets you review the file contents, resolve problems, and reload the data from the rejected data files. Saving rejected data to a table, lets you query the table to see rejected data rows and the reasons (exceptions) why the rows could not be parsed. Vertica recommends saving rejected data to a table.

Multiple rejected data files

Unless a load is very small (< 10MB), COPY creates more than one file to hold rejected rows. Several factors determine how many files COPY creates for rejected data. Here are some of the factors:

- Number of sources being loaded
- Total number of rejected rows
- Size of the source file (or files)
- Cooperative parsing and number of threads being used
- UDLs that support apportioned loads
- For your own COPY parser, the number of objects returned from prepareUDSources()

Naming conventions for rejected files

You can specify one or more rejected data files with the files you are loading. Use the REJECTED DATA parameter to specify a file location and name, and separate consecutive rejected data file names with a comma (,). Do not use the ON ANY NODE option because it is applicable only to load files.

If you specify one or more files, and COPY requires multiple files for rejected data, COPY uses the rejected data file names you supply as a prefix, and appends a numeric suffix to each rejected data file. For example, if you specify the name my_rejects for the REJECTED_DATA parameter, and the file you are loading is large enough (> 10MB), several files such as the following will exist:

- my_rejects-1
- my_rejects-2
- my_rejects-3

COPY uses cooperative parsing by default, having the nodes parse a specific part of the file contents. Depending on the file or portion size, each thread generates at least one rejected data file per source file or portion, and returns load results to the initiator node. The file suffix is a thread index when COPY uses multiple threads (.1, .2, .3, and so on).

The maximum number of rejected data files cannot be greater than the number of sources being loaded, per thread to parse any portion. The resource pool determines the maximum number of threads. For cooperative parse, use all available threads.

If you use COPY with a UDL that supports apportioned load, the file suffix is an offset value. UDL's that support apportioned loading render cooperative parsing unnecessary. For apportioned loads, COPY creates at least one rejected file per data portion, and more files depending on the size of the load and number of rejected rows.

For all data loads except COPY LOCAL , COPY behaves as follows:

No rejected data file specified...	Rejected data file specified...
For a single data file (**pathToData** or STDIN), COPY stores one or more rejected data files in the default location.	For one data file, COPY interprets the rejected data path as a file, and stores all rejected data at the location. If more than one files is required from parallel processing, COPY appends a numeric suffix. If the path is not a file, COPY returns an error.
For multiple source files, COPY stores all rejected data in separate files in the default directory, using the source file as a file name prefix, as noted.	For multiple source files, COPY interprets the rejected path as a directory. COPY stores all information in separate files, one for each source. If path is not a directory, COPY returns an error. COPY accepts only one path per node. For example, if you specify the rejected data path as my_rejected_data, COPY creates a directory of that name on each node. If you provide more than one rejected data path, COPY returns an error.
Rejected data files are returned to the initiator node.	Rejected data files are not shipped to the initiator node.

Maximum length of file names

Loading multiple input files in one statement requires specifying full path names for each file. Keep in mind that long input file names, combined with rejected data file names, can exceed the operating system's maximum length (typically 255 characters). To work around file names that exceed the maximum length, use a path for the rejected data file that differs from the default path—for example, \tmp\<shorter-file-name> .

Saving rejected data to a table

Use the REJECTED DATA parameter with the AS TABLE clause to specify a table in which to save rejected data. Saving rejected data to a file is mutually exclusive with using the AS TABLE clause.

When you use the AS TABLE clause, Vertica creates a new table if one does not exist, or appends to an existing table. If no parsing rejections occur during a load, the table exists but is empty. The next time you load data, Vertica inserts any rejected rows to the existing table.

The load rejection tables are a special type of table with the following capabilities and limitations:

- Support SELECT statements
- Can use DROP TABLE
- Cannot be created outside of a COPY statement
- Do not support DML and DDL activities
- Are not K-safe

To make the data in a rejected table K-safe, you can do one of the following:

- Write a CREATE TABLE..AS statement, such as this example:

```
=> CREATE TABLE new_table AS SELECT * FROM rejected_table;
```

- Create a table to store rejected records, and run INSERT..SELECT operations into the new table

Using COPY NO COMMIT

If the COPY statement includes options NO COMMIT and REJECTED DATA AS TABLE , and the *reject-table* does not already exist, Vertica Analytic Database saves the rejected data table as a LOCAL TEMP table and returns a message that a LOCAL TEMP table is being created.

Rejected-data tables are useful for Extract-Load-Transform workflows, where you will likely use temporary tables more frequently. The rejected-data tables let you quickly load data and identify which records failed to load. If you load data into a temporary table that you created using the ON COMMIT DELETE clause, the COPY operation will not commit.

Location of rejected data table records

When you save rejected records to a table, using the REJECTED DATA AS TABLE *table_name* option, the data for the table is saved in a database data subdirectory, RejectionTableData . For example, for a VMart database, table data files reside here:

```
/home/dbadmin/VMart/v_vmart_node0001_data/RejectionTableData
```

Rejected data tables include both rejected data and the reason for the rejection (exceptions), along with other data columns, described next. Vertica suggests that you periodically drop any rejected data tables that you no longer require.

Querying a rejected data table

When you specify a rejected data table when loading data with COPY , you can query that table for information about rejected data after the load operation is complete. For example:

1. Create the loader table:

```
=> CREATE TABLE loader(a INT)
CREATE TABLE
```

2. Use COPY to load values, saving rejected data to a table, loader_rejects :

```
=> COPY loader FROM STDIN REJECTED DATA AS TABLE loader_rejects;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1
>> 2
>> 3
>> a
>> \.
```

3. Query the loader table after loading data:

```
=> SELECT * FROM loader;
 x
---
 1
 2
 3
(3 rows)
```

4. Query the loader_rejects table to see its column rows:

```
=> SELECT * FROM loader_rejects;
-[ RECORD 1 ]------------+---------------------------------------------
node_name                | v_vmart_node0001
file_name                | STDIN
session_id               | v_vmart_node0001.example.-24016:0x3439
transaction_id           | 45035996274080923
statement_id             | 1
batch_number             | 0
row_number               | 4
rejected_data            | a
rejected_data_orig_length | 1
rejected_reason          | Invalid integer format 'a' for column 1 (x)
```

The rejected data table has the following columns:

Column	Data Type	Description
node_name	VARCHAR	The name of the Vertica node on which the input load file was located.
file_name	VARCHAR	The name of the file being loaded, which applies if you loaded a file (as opposed to using STDIN).
session_id	VARCHAR	The session ID number in which the COPY statement occurred.
transaction_id	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL.

statement_id	INTEGER	The unique identification number of the statement within the transaction that included the rejected data. **Tip** You can use the session_id , transaction_id , and statement_id columns to create joins with many system tables. For example, if you join against the QUERY_REQUESTS table using those three columns, the QUERY_REQUESTS.REQUEST column contains the actual COPY statement (as a string) used to load this data.
batch_number	INTEGER	INTERNAL USE. Represents which batch (chunk) the data comes from.
row_number	INTEGER	The rejected row number from the input file, or -1 if it could not be determined. The value can be -1 when using cooperative parse. Each parse operation resets the row number, so in an apportioned load, there can be several entries with the same row number but different rows.
rejected_data	LONG VARCHAR	The data that was not loaded.
rejected_data_orig_length	INTEGER	The length of the rejected data.
rejected_reason	VARCHAR	The error that caused the rejected row. This column returns the same message that exists in a load exceptions file when you do not save to a table.

Exporting the rejected records table

You can export the contents of the column rejected_data to a file to capture only the data rejected during the first COPY statement. Then, correct the data in the file, save it, and load the updated file.

To export rejected records:

1. Create a sample table:
```
=> CREATE TABLE t (i int);
CREATE TABLE
```
2. Copy data directly into the table, using a table to store rejected data:
```
=> COPY t FROM STDIN REJECTED DATA AS TABLE t_rejects;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1
>> 2
>> 3
>> 4
>> a
>> b
>> c
>> \.
```
3. Show only tuples and set the output format:
```
=> \t
Showing only tuples.
=> \a
Output format is unaligned.
```
4. Output to a file:

```
=> \o rejected.txt
=> select rejected_data from t_rejects;
=> \o
```

5. Use the cat command on the saved file:

```
=> \! cat rejected.txt
a
b
c
```

After a file exists, you can fix load errors and use the corrected file as load input to the COPY statement.

Saving load exceptions (EXCEPTIONS)

COPY exceptions consist of informational messages describing why a row of data could not be parsed. The EXCEPTIONS option lets you specify a file to which COPY writes exceptions. If you omit this option, COPY saves exception files to the following path: *catalog-dir* /CopyErrorLogs/ *tablename* - *sourcefilename* -copy-from-exceptions , where:

- *catalog-dir* is the directory holding the database catalog files
- *table* is the name of the table being loaded into
- *sourcefile* is the name of the file being loaded

> **Note**
> REJECTED DATA AS TABLE is mutually exclusive with EXCEPTIONS .

The file produced by the EXCEPTIONS option indicates the line number and the reason for each exception.

If copying from STDIN, the source file name is STDIN .

You can specify rejected data and exceptions files for individual files in a data load. Separate rejected data and exception file names with commas in the COPY statement.

You must specify a filename in the path to load multiple input files. Keep in mind that long table names combined with long data file names can exceed the operating system's maximum length (typically 255 characters). To work around file names exceeding the maximum length, use a path for the exceptions file that differs from the default path; for example, /tmp/<shorter-file-name> .

If you specify an EXCEPTIONS path:

- For one data file, the path must be a file, and COPY stores all information in this file.
- For multiple data files, the path must be a directory. COPY creates one file in this directory for each data file.
- Exceptions files are not stored on the initiator node.
- You can specify only one path per node.

If you do not specify the EXCEPTIONS path, COPY stores exception files in the default directory.

COPY rejected data and exception files

When executing a COPY statement, and parallel processing is ON (the default setting), COPY creates separate threads to process load files. Typically, the number of threads depends on the number of node cores in the system. Each node processes part of the load data. If the load succeeds overall, any parser rejections that occur during load processing are written to that node's specific rejected data and exceptions files. If the load fails, the rejected data file contents can be incomplete, or empty. If you do not specify a file name explicitly, COPY uses a default name and location for rejected data files. See the next topic for specifying your own rejected data and exception files.

Both rejected data and exceptions files are saved and stored on a per-node basis. This example uses multiple files as COPY inputs. Since the statement does not include either the REJECTED DATA or EXCEPTIONS parameters, rejected data and exceptions files are written to the default location, the database catalog subdirectory, CopyErrorLogs , on each node:

```
\set dir `pwd`/data/ \set remote_dir /vertica/test_dev/tmp_ms/
\set file1 '':dir'C1_large_tbl.dat'''
\set file2 '':dir'C2_large_tbl.dat'''
\set file3 '':remote_dir'C3_large_tbl.dat'''
\set file4 '':remote_dir'C4_large_tbl.dat'''
=>COPY large_tbl FROM :file1 ON site01,:file2 ON site01,
            :file3 ON site02,
            :file4 ON site02
            DELIMITER '|';
```

Specifying rejected data and exceptions files

The optional COPY REJECTED DATA and EXCEPTIONS parameters 'path' element lets you specify a non-default path in which to store the files.

If *path* resolves to a storage location, and the user invoking COPY is not a superuser, these are the required permissions:

- The storage location must have been created (or altered) with the USER option (see CREATE LOCATION and ALTER_LOCATION_USE)
- The user must already have been granted READ access to the storage location where the file(s) exist, as described in GRANT (storage location)

Both parameters also have an optional ON *nodename* clause that uses the specified path:

```
...[ EXCEPTIONS 'path' [ ON nodename ] [, ...] ]...[ REJECTED DATA 'path' [ ON nodename ] [, ...] ]
```

While *'path'* specifies the location of the rejected data and exceptions files (with their corresponding parameters), the optional ON *nodename* clause moves any existing rejected data and exception files on the node to the specified path on the same node.

Saving rejected data and exceptions files to a single server

The COPY statement does not have a facility to merge exception and rejected data files after COPY processing is complete. To see the contents of exception and rejected data files requires accessing each node's specific files.

> **Note**
> To save all exceptions and rejected data files on a network host, be sure to give each node's files unique names, so that different cluster nodes do not overwrite other nodes' files. For instance, if you set up a server with two directories (/vertica/exceptions and /vertica/rejections), specify file names for each Vertica cluster node to identify each node, such as node01_exceptions.txt and node02_exceptions.txt . This way, each cluster node's files are easily distinguishable in the exceptions and rejections directories.

Using VSQL variables for rejected data and exceptions files

This example uses vsql variables to specify the path and file names to use with the exceptions and rejected data parameters (except_s1 and reject_s1). The COPY statement specifies a single input file (large_tbl) on the initiator node:

```
\set dir `pwd`/data/ \set file1 '':dir'C1_large_tbl.dat'''
\set except_s1 '':dir'exceptions'''
\set reject_s1 '':dir'rejections'''

COPY large_tbl FROM :file1 ON site01 DELIMITER '|'
REJECTED DATA :reject_s1 ON site01
EXCEPTIONS :except_s1 ON site01;
```

This example uses variables to specify exception and rejected date files (except_s2 and reject_s2) on a remote node. The COPY statement consists of a single input file on a remote node (site02):

```
\set remote_dir /vertica/test_dev/tmp_ms/\set except_s2 '':remote_dir'exceptions'''
\set reject_s2 '':remote_dir'rejections'''

COPY large_tbl FROM :file1 ON site02 DELIMITER '|'
REJECTED DATA :reject_s2 ON site02
EXCEPTIONS :except_s2 ON site02;
```

This example uses variables to specify that the exception and rejected data files are on a remote node (indicated by :remote_dir). The inputs to the COPY statement consist of multiple data files on two nodes (site01 and site02). The exceptions and rejected data options use the ON *nodename* clause with the variables to indicate where the files reside (site01 and site02):

```
\set dir `pwd`/data/ \set remote_dir /vertica/test_dev/tmp_ms/
\set except_s1 '''':dir''''
\set reject_s1 '''':dir''''
\set except_s2 '''':remote_dir''''
\set reject_s2 '''':remote_dir''''
COPY large_tbl FROM :file1 ON site01,
        :file2 ON site01,
        :file3 ON site02,
        :file4 ON site02
        DELIMITER '|'
        REJECTED DATA :reject_s1 ON site01, :reject_s2 ON site02
        EXCEPTIONS :except_s1 ON site01, :except_s2 ON site02;
```

COPY LOCAL rejection and exception files

Invoking COPY LOCAL (or COPY LOCAL FROM STDIN) does not automatically create rejected data and exceptions files. This behavior differs from using COPY, which saves both files automatically, regardless of whether you use the optional REJECTED DATA and EXCEPTIONS parameters to specify either file explicitly.

Use the REJECTED DATA and EXCEPTIONS parameters with COPY LOCAL and COPY LOCAL FROM STDIN to save the corresponding output files on the client. If you do *not* use these options, rejected data parsing events (and the exceptions that describe them) are not retained, even if they occur.

You can load multiple input files using COPY LOCAL (or COPY LOCAL FROM STDIN). If you also use the REJECTED DATA and EXCEPTIONS options, the statement writes rejected rows and exceptions and to separate files. The respective files contain all rejected rows and corresponding exceptions, respectively, regardless of how many input files were loaded.

If COPY LOCAL does not reject any rows, it does not create either file.

> **Note**
> Because COPY LOCAL (and COPY LOCAL FROM STDIN) must write any rejected rows and exceptions to the client, you cannot use the [ON nodename] clause with either the rejected data or exceptions options.

Specifying rejected data and exceptions files

To save any rejected data and their exceptions to files:

1. In the COPY LOCAL (and COPY LOCAL FROM STDIN) statement, use the REJECTED DATA 'path' and the EXCEPTIONS 'path' parameters, respectively.
2. Specify two different file names for the two options. You cannot use one file for both the REJECTED DATA and the EXCEPTIONS .
3. When you invoke COPY LOCAL or COPY LOCAL FROM STDIN, the files you specify need not pre-exist. If they do, COPY LOCAL must be able to overwrite them.

You can specify the path and file names with vsql variables:

```
\set rejected ../except_reject/copyLocal.rejected
\set exceptions ../except_reject/copyLocal.exceptions
```

> **Note**
> Using COPY LOCAL does not support storing rejected data in a table, as you can when using the COPY statement.

When you use the COPY LOCAL or COPY LOCAL FROM STDIN statement, specify the variable names for the files with their corresponding parameters:

```
=> COPY large_tbl FROM LOCAL rejected data :rejected exceptions :exceptions;
=> COPY large_tbl FROM LOCAL STDIN rejected data :rejected exceptions :exceptions;
```

Monitoring COPY loads and metrics

You can check COPY loads using:

- Vertica functions

- LOAD_STREAMS system table
- LOAD_SOURCES system table

More generally, the EXECUTION_ENGINE_PROFILES system table records information about query events, including loads.

Using Vertica functions

Two meta-functions return COPY metrics for the number of accepted or rejected rows from the most recent COPY statement run in the current session:

1. To get the number of accepted rows, use the GET_NUM_ACCEPTED_ROWS function:

```
=> select get_num_accepted_rows();
 get_num_accepted_rows
-----------------------
                    11
(1 row)
```

2. To check the number of rejected rows, use the GET_NUM_REJECTED_ROWS function:

```
=> select get_num_rejected_rows();
 get_num_rejected_rows
-----------------------
                     0
(1 row)
```

> **Note**
> GET_NUM_ACCEPTED_ROWS and GET_NUM_REJECTED_ROWS support loads from STDIN, COPY LOCAL from a Vertica client, or a single file on the initiator. You cannot use these functions for multi-node loads.

Using the CURRENT_LOAD_SOURCE function

You can include the CURRENT_LOAD_SOURCE function as a part of the COPY statement. Doing so allows you to insert into a column the input file name or value computed by this function.

To insert the file names into a column from multiple source files:

```
=> COPY t (c1, c2, c3 as CURRENT_LOAD_SOURCE()) FROM '/home/load_file_1' ON  exampledb_node02,
'/home/load_file_2' ON exampledb_node03 DELIMITER ',';
```

Using the LOAD_STREAMS system table

Vertica includes a set of system tables that include monitoring information. The LOAD_STREAMS system table includes information about load stream metrics from COPY and COPY FROM VERTICA statements. Thus, you can query table values to get COPY metrics.

Vertica maintains system table metrics until they reach a designated size quota (in kilobytes). This quota is set through internal processes, which you cannot set or view directly.

Labeling copy streams

COPY can include the STREAM NAME parameter to label its load stream so it is easy to identify in the LOAD_STREAMS system table. For example:

```
=> COPY mytable FROM myfile DELIMITER '|' STREAM NAME 'My stream name';
```

Load stream metrics

The following LOAD_STREAMS columns show on the status of a load as it progresses:

Column name	Value...
ACCEPTED_ROW_COUNT	Increases during parsing, up to the maximum number of rows in the input file.

PARSE_COMPLETE_PERCENT	Remains zero (0) until all named pipes return an EOF. While COPY awaits an EOF from multiple pipes, it can appear to be hung. However, before canceling the COPY statement, check your system CPU and disk accesses to determine if any activity is in progress. In a typical load, the PARSE_COMPLETE_PERCENT value can either increase slowly or jump quickly to 100%, if you are loading from named pipes or STDIN.
SORT_COMPLETE_PERCENT	Remains at 0 when loading from named pipes or STDIN. After PARSE_COMPLETE_PERCENT reaches 100 percent, SORT_COMPLETE_PERCENT increases to 100 percent.

Depending on the data sizes, a significant lag can occur between the time PARSE_COMPLETE_PERCENT reaches 100 percent and the time SORT_COMPLETE_PERCENT begins to increase.

This example queries load stream data from the LOAD_STREAMS system table:

```
=> \pset expanded
Expanded display is on.
=> SELECT stream_name, table_name, load_start, accepted_row_count,
   rejected_row_count, read_bytes, unsorted_row_count, sorted_row_count,
   sort_complete_percent FROM load_streams;
-[ RECORD 1 ]----------+---------------------------
stream_name            | fact-13
table_name             | fact
load_start             | 2010-12-28 15:07:41.132053
accepted_row_count     | 900
rejected_row_count     | 100
read_bytes             | 11975
input_file_size_bytes  | 0
parse_complete_percent | 0
unsorted_row_count     | 3600
sorted_row_count       | 3600
sort_complete_percent  | 100
```

Using the LOAD_SOURCES system table

The LOAD_STREAMS table shows the total number of rows that were loaded or rejected. Grouping this information by source can help you determine from where data is coming. The LOAD_SOURCES system table includes some of the same data as LOAD_STREAMS does but adds this source-specific information. If apportioning is enabled, LOAD_SOURCES also provides information about how loads are apportioned among nodes.

You can use this table to identify causes of differing query results. For example, you can use the following statement to create an external table based on globs:

```
=> CREATE EXTERNAL TABLE tt AS COPY WITH SOURCE AWS(dir = 'foo', file = '*');
```

If you select from this table, Vertica loads data from every file in the foo directory and creates one row in the LOAD_SOURCES table for each file. Suppose you later repeat the query and see different results. You could look at the LOAD_SOURCES table and discover that—between the two queries—somebody added another file to the foo directory. Because each file is recorded in LOAD_SOURCES, you can see the new file that explains the changed query results.

If you are using many data sources. you might prefer to disable this reporting. To disable reporting, set the LoadSourceStatisticsLimit configuration parameter to 0. This parameter sets the upper bound on the number of sources profiled by LOAD_SOURCES per load. The default value is 256.

Using load scripts

You can write and run a load script for the COPY statement using a simple text-delimited file format. For information about other load formats see Data formats . Vertica recommends that you load the smaller tables before the largest tables. To check data formats before loading, see Handling Non-UTF-8 input .

Using absolute paths in a load script

Unless you are using the COPY FROM LOCAL statement, using COPY on a remote client requires an absolute path for a data file. You cannot use relative paths on a remote client. For a load script, you can use vsql variables to specify the locations of data files relative to your Linux working directory.

To use vsql variables to specify data file locations:

1. Create a vsql variable containing your Linux current directory.
```
\set t_pwd `pwd`
```
2. Create another vsql variable that uses a path relative to the Linux current directory variable for a specific data file.
```
\set input_file '\'':t_pwd'/Date_Dimension.tbl\''
```
3. Use the second variable in the COPY statement:
```
=> COPY Date_Dimension FROM :input_file DELIMITER '|';
```
4. Repeat steps 2 and 3 to load all data files.

> **Note**
> COPY FROM LOCAL does not require an absolute path for data files. You can use paths that are relative to the client's directory system.

Running a load script

You can run a load script on any host, as long as the data files are on that host.

1. Change your Linux working directory to the location of the data files.
```
$ cd /opt/vertica/doc/retail_example_database
```
2. Run the Administration Tools.
```
$ /opt/vertica/bin/admintools
```
3. Connect to the database.
4. Run the load script.

Troubleshooting data loads

You might encounter the following issues when loading data. For issues specific to external tables, see Troubleshooting external tables .

Cannot load data because all columns are complex types

A native table must contain at least one column that is either a scalar type or a native array (one-dimensional array of scalar types). If the data you wish to load consists entirely of complex types, you must add a scalar column to the table before loading the data. You do not need to populate the scalar column.

First, create a native table with the columns in the data plus one scalar column, and then load the data into that table as shown in the following example:

```
-- Native table with extra scalar column:
=> CREATE TABLE customers(person ROW(name VARCHAR, card VARCHAR), tempval int);
CREATE TABLE

-- Load complex data, ignoring scalar column:
=> COPY customers(person) FROM :custdata PARQUET;
 Rows Loaded
-------------
           3
(1 row)
```

Load fails because of disk quota

COPY can fail if the volume of data being loaded causes a table or schema to exceed a disk quota:

```
ERROR 0: Disk Quota for the Table Level Exceeded
HINT: Delete data and PURGE or increase disk quota at the table level
```

By default, Vertica tables and schemas do not have a disk quota; they are limited only by the capacity of your hardware and license. However, a superuser can set quotas, as described in Disk quotas .

If you encounter a failure due to quota, you cannot simply use DELETE to reduce consumption. DELETE does not free space, because deleted data is still preserved in the storage containers. To reclaim space you must purge deleted data; see Removing table data .

Reads from Parquet files report unexpected data-type mismatches

If a Parquet file contains a column of type STRING but the column in Vertica is of a different type, such as INTEGER, you might see an unclear error message. In this case Vertica reports the column in the Parquet file as BYTE_ARRAY, as shown in the following example:

```
ERROR 7247:  Datatype mismatch: column 2 in the parquet_cpp source
[/tmp/nation.0.parquet] has type BYTE_ARRAY, expected int
```

This behavior is specific to Parquet files; with an ORC file the type is correctly reported as STRING. The problem occurs because Parquet does not natively support the STRING type and uses BYTE_ARRAY for strings instead. Because the Parquet file reports its type as BYTE_ARRAY, Vertica has no way to determine if the type is actually a BYTE_ARRAY or a STRING.

Error 7087: wrong number of columns

When loading ORC or Parquet data, you might see an error stating that you have the wrong number of columns:

```
=> CREATE TABLE nation (nationkey bigint, name varchar(500),
        regionkey bigint, comment varchar(500));
  CREATE TABLE

=> COPY nation from :orc_dir ORC;
  ERROR 7087: Attempt to load 4 columns from an orc source
  [/tmp/orc_glob/test.orc] that has 9 columns
```

When you load data from ORC or Parquet files, your table must consume all of the data in the file, or this error results. To avoid this problem, add the missing columns to your table definition.

For Parquet data, time zones in timestamp values are not correct

Reading timestamps from a Parquet file in Vertica might result in different values, based on the local time zone. This issue occurs because the Parquet format does not support the SQL TIMESTAMP data type. If you define the column in your table with the TIMESTAMP data type, Vertica interprets timestamps read from Parquet files as values in the local time zone. This same behavior occurs in Hive. When this situation occurs, Vertica produces a warning at query time such as the following:

```
WARNING 0:  SQL TIMESTAMPTZ is more appropriate for Parquet TIMESTAMP
because values are stored in UTC
```

When creating the table in Vertica, you can avoid this issue by using the TIMESTAMPTZ data type instead of TIMESTAMP.

Time zones can also be incorrect in ORC data, but the reason is different.

For ORC data, time zones in timestamp values are not correct

Vertica and Hive both use the Apache ORC library to interact with ORC data. The behavior of this library changed with Hive version 1.2.0, so timestamp representation depends on what version was used to write the data.

When writing timestamps, the ORC library now records the time zone in the stripe footer. Vertica looks for this value and applies it when loading timestamps. If the file was written with an older version of the library, the time zone is missing from the file.

If the file does not contain a time zone, Vertica uses the local time zone and logs an ORC_FILE_INFO event in the QUERY_EVENTS system table.

The first time you query a new ORC data source, you should query this table to look for missing time zone information:

```
=> SELECT event_category, event_type, event_description, operator_name, event_details, COUNT(event_type)
   AS COUNT FROM QUERY_EVENTS WHERE event_type ILIKE 'ORC_FILE_INFO'
   GROUP BY event_category, event_type, event_description, operator_name, event_details
   ORDER BY event_details;
event_category | event_type | event_description | operator_name | event_details | count
---------------+--------------+-----------------------------------------------------+--------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------------
EXECUTION | ORC_FILE_INFO | ORC file does not have writer timezone information | OrcParser | Timestamp values in the ORC source [data/sales_stats.orc] will be computed using local timezone | 2

(1 row)
```

Time zones can also be incorrect in Parquet data, but the reason is different.

Some date and timestamp values are wrong by several days

When Hive writes ORC or Parquet files, it converts dates before 1583 from the Gregorian calendar to the Julian calendar. Vertica does not perform this conversion. If your file contains dates before this time, values in Hive and the corresponding values in Vertica can differ by up to ten days. This difference applies to both DATE and TIMESTAMP values.

Working with external data

An alternative to importing data into Vertica is to query it in place. Querying external data instead of importing it can be advantageous in some cases:

- If you want to explore data, such as in a data lake, before selecting data to load into Vertica.
- If you are one of several consumers sharing the same data, for example in a data lake, then reading it in place eliminates concerns about whether query results are up to date. There's only one copy, so all consumers see the same data.
- If your data changes rapidly but you do not want to stream it into Vertica, you can instead query the latest updates automatically.
- If you have a very large volume of data and do not want to increase your license capacity.
- If you have lower-priority data in Vertica that you still want to be able to query.

To query external data, you must describe your data as an *external table* . Like native tables, external tables have table definitions and can be queried. Unlike native tables, external tables have no catalog and Vertica loads selected data from the external source as needed. For some formats, the query planner can take advantage of partitions and sorting in the data, so querying an external table does not mean you load all of the data at query time. (For more information about native tables, see Working with native tables .)

There is one special type of external data not covered in this section. If you are reading data from Hadoop, and specifically from a Hive data warehouse, then instead of defining your own external tables you can read the schema information from Hive. For more information, see Using the HCatalog Connector .

In this section

- How external tables differ from native tables
- Creating external tables
- Partitioned file paths
- Querying external tables
- Monitoring external tables
- Troubleshooting external tables

How external tables differ from native tables

You can use external tables in the same ways you use Vertica native tables. Because the data is external to the database, however, there are some differences in how external tables operate.

Data

The data for an external table can reside anywhere, so long as all database nodes can access it. S3, HDFS, and NFS mount points are common places to find external data. Naturally, querying external data can incur some latency compared to querying locally-stored ROS data, but Vertica has optimizations that can reduce the impact. For example, Vertica can take advantage of node and rack locality for HDFS data.

Because the data is external, Vertica loads external data each time you query it. Vertica is optimized to reduce the volume of read data, including predicate pushdown and partition pruning for formats that support partitioning. The ORC and Parquet formats support these optimizations.

Because the data is read at query time, you must ensure that your users have and retain permission to read the data in its original location. Depending on where the data is stored, you might need to take additional steps to manage access, such as creating AWS IAM roles on S3.

Because the data is not stored in Vertica, external tables do not use superprojections and buddy projections.

Resource consumption

External tables add very little to the Vertica catalog, which reduces the resources that queries consume. Because the data is not stored in Vertica, external tables are not affected by the Tuple Mover and do not cause ROS pushback. Vertica uses a small amount of memory when reading external table data, because the table contents are not part of your database and are parsed each time the external table is used.

Backup and restore

Because the data in external tables is managed outside of Vertica, only the external table definitions, not the data files, are included in database backups. Arrange for a separate backup process for your external table data.

DML support

External tables allow you to read external data. They do not allow you to modify it. Some DML operations are therefore not available for external tables, including:

- DELETE FROM
- INSERT INTO
- SELECT...FOR UPDATE

Sequences and IDENTITY columns

The COPY statement definition for external tables can include IDENTITY columns and sequences. Whenever a select statement queries the external table, sequences and IDENTITY columns are re-evaluated. This results in changing the external table column values, even if the underlying external table data remains the same.

Creating external tables

To create an external table, combine a table definition with a copy statement using the CREATE EXTERNAL TABLE AS COPY statement. CREATE EXTERNAL TABLE AS COPY uses a subset of parameters from CREATE TABLE and COPY.

You define your table columns as you would for a Vertica native table using CREATE TABLE . You also specify a COPY FROM clause to describe how to read the data, as you would for loading data. How you specify the FROM path depends on where the file is located and the data format. See Specifying where to load data from and Data formats .

As with native tables, you can use the INFER_TABLE_DDL function to derive column definitions from data files in supported formats.

The following example defines an external table for delimited data stored in HDFS:

```
=> CREATE EXTERNAL TABLE sales (itemID INT, date DATE, price FLOAT)
   AS COPY FROM 'hdfs:///data/ext1.csv' DELIMITER ',';
```

The following example uses data in the ORC format that is stored in S3. The data has two partition columns. For more information about partitions, see Partitioned file paths .

```
=> CREATE EXTERNAL TABLE records (id int, name varchar(50), created date, region varchar(50))
  AS COPY FROM 's3://datalake/sales/*/*/*'
  PARTITION COLUMNS created, region;
```

The following example shows how you can read from all Parquet files in a local directory, with no partitions and no globs:

```
=> CREATE EXTERNAL TABLE sales (itemID INT, date DATE, price FLOAT)
   AS COPY FROM '/data/sales/*.parquet' PARQUET;
```

When you create an external table, data is not added to the database and no projections are created. Instead, Vertica performs a syntactic check of the CREATE EXTERNAL TABLE AS COPY statement and stores the table name and COPY statement definition in the catalog. Each time a SELECT query references an external table, Vertica parses and executes the stored COPY statement to obtain the referenced data. Any problems in the table definition, such as incorrect column types, can be discovered only by querying the table.

Successfully returning data from an external table requires that the COPY definition be correct, and that other dependencies, such as files, nodes, and other resources are accessible and available at query time. If the table definition uses globs (wildcards), and files are added or deleted, the data in the external table can change between queries.

ORC and Parquet formats

When defining an external table for ORC or Parquet data, you must define all of the data columns in the file. If you omit data columns, queries using the table abort with an error.

If you load from multiple ORC or Parquet files in the same COPY statement, and any of them is aborted, the entire load aborts. This behavior differs from that for delimited files, where the COPY statement loads what it can and ignores the rest.

Iceberg format

You can create an external table for data stored by Apache Iceberg using CREATE EXTERNAL TABLE ICEBERG . An Iceberg table consists of data files and metadata describing the schema. Unlike other external tables, an Iceberg external table need not specify column definitions (DDL). The information is read from Iceberg metadata at query time. For certain data types you can adjust column definitions when creating the table, for example to specify VARCHAR sizes.

Special considerations for external tables

If the maximum length of a column is smaller than the actual data, such as a VARCHAR that is too short, Vertica truncates the data and logs the event.

You can see unexpected query results if constraints on columns cause values to be rejected:

- If you specify a NOT NULL column constraint and the data contains null values, those rows are rejected.
- If you use ENFORCELENGTH, values that are too long are rejected rather than being truncated.
- When reading ORC data, if you declare a scalar precision and some data does not fit, that row is rejected. For example, if you specify a column as Decimal(6,5), a value of 123.456 is rejected.

One way to know if column constraints have caused data to be rejected is if COUNT on a column returns a different value than COUNT(*).

The JSON and Avro parsers produce warnings when the data contains columns or fields that are not part of the table definition. External tables load data at query time, so these warnings appear for each query. You can use the parser's suppress_warnings parameter to prevent these warnings in external tables. See the parser reference pages for an example.

When using the COPY parameter ON ANY NODE , confirm that the source file definition is identical on all nodes. Specifying different external files can produce inconsistent results.

You can take advantage of partitioning to limit the amount of data that Vertica reads. For more information about using partitioned data, see Partitioned file paths .

Canceling a CREATE EXTERNAL TABLE AS COPY statement can cause unpredictable results. If you realize after beginning the operation that your table definition is incorrect (for example, you inadvertently specify the wrong external location), wait for the query to complete. When the external table exists, use DROP TABLE to remove its definition.

> **Tip**
> When working with a new external data source, consider setting REJECTMAX to 1 to make problems in the data apparent. Testing in this way allows you to discover problems in the data before running production queries against it.

After you create an external table, analyze its row count to improve query performance. See Improving Query Performance for External Tables .

Required permissions

In addition to having permission in Vertica, users must have read access to the external data.

- For data on the local disk this access is governed by local file permissions.
- For data in HDFS, access might be governed by Kerberos authentication. See Accessing kerberized HDFS data .
- For data on S3, you need access through an AWS IAM role. See S3 object store .

For data in GCS, you must enable S3 compatibility before reading data. See Google Cloud Storage (GCS) object store .

By default, you must also be a database superuser to access external tables through a SELECT statement.

In most cases, to allow users without superuser access to query external tables, an administrator must create a USER storage location and grant those users read access to the location. See CREATE LOCATION and GRANT (storage location) . This location must be a parent of the path used in the COPY statement when creating the external table. This requirement does not apply to external tables stored in HDFS. The following example shows granting access to a user named Bob to any external table whose data is located under /tmp (including in subdirectories to any depth):

```
=> CREATE LOCATION '/tmp' ALL NODES USAGE 'user';
=> GRANT ALL ON LOCATION '/tmp' to Bob;
```

Organizing external table data

If the data you store in external tables changes regularly (for instance, each month in the case of storing recent historical data), you can use partitioning in combination with wildcards (globs) to make parsing the stored COPY statement definition more dynamic. For instance, if you store monthly data on an NFS mount, you could organize monthly files within a top-level directory for a calendar year, such as:

```
/year=2018/month=01/
```

You can then read the year and month values from the directory names in the COPY statement:

```
=> CREATE EXTERNAL TABLE archive (...) AS COPY FROM '/nfs_name/*/*/*' PARTITION COLUMNS year, month;
```

Whenever a Vertica query references the external table archive , and Vertica parses the COPY statement, all stored data in the top-level monthly directory is accessible to the query. If the query filters by year or month, such as in a WHERE clause, Vertica skips irrelevant directories when evaluating the glob. See Partitioned file paths for more information.

Validating table definitions

When you create an external table, Vertica validates the syntax of the CREATE EXTERNAL TABLE AS COPY FROM statement. For example, if you omit a required keyword in the statement, creating the external table fails:

```
=> CREATE EXTERNAL TABLE ext (ts timestamp, d varchar)
   AS COPY '/home/dbadmin/designer.log';
ERROR 2778:  COPY requires a data source; either a FROM clause or a WITH SOURCE for a user-defined source
```

Checking other components of the COPY definition, such as path statements and node availability, does not occur until a SELECT query references the external table.

To validate an external table definition, run a SELECT query that references the external table. Check that the returned query data is what you expect. If the query does not return data correctly, check the COPY exception and rejected data log files.

Because the COPY definition determines what occurs when you query an external table, COPY statement errors can reveal underlying problems. For more information about COPY exceptions and rejections, see Handling messy data .

Viewing external table definitions

When you create an external table, Vertica stores the COPY definition statement in the table_definition column of the TABLES system table.

To list all tables, use a SELECT * query, as shown:

```
=> SELECT * FROM TABLES WHERE table_definition <> '';
```

Use a query such as the following to list the external table definitions:

```
=> SELECT table_name, table_definition FROM TABLES;
 table_name |                          table_definition
------------+-----------------------------------------------------------------------
 t1         | COPY           FROM 'TMPDIR/external_table.dat'  DELIMITER ','
 t1_copy    | COPY           FROM 'TMPDIR/external_table.dat'  DELIMITER ','
 t2         | COPY FROM 'TMPDIR/external_table2.dat' DELIMITER ','
(3 rows)
```

Partitioned file paths

Data files are sometimes partitioned in the file system using the directory structure. Partitioning allows you to move values out of the raw data, where they have to be included for each row, and into the directory structure. Partitioning can improve query performance by allowing entire directories to be skipped. Partitioning can also save disk space if partition values are not repeated in the data.

Partitioning by date is one common use:

```
/data/created=2016-11-01/*
/data/created=2016-11-02/*
/data/created=2016-11-03/*
/data/...
```

The files in the globs do not contain a created column because this information is expressed through the file system. Vertica can read the partitioned values (dates, in this example) into a table column (created , in this example).

Data can be partitioned by more than one value:

```
/data/created=2016-11-01/region=northeast/*
/data/created=2016-11-01/region=central/*
/data/created=2016-11-01/region=southeast/*
/data/created=2016-11-01/...
/data/created=2016-11-02/region=northeast/*
/data/created=2016-11-02/region=central/*
/data/created=2016-11-02/region=southeast/*
/data/created=2016-11-02/...
/data/created=2016-11-03/...
/data/...
```

If data is partitioned by more than one value, the partitions must appear in the same order in all directory paths in the glob. In this example, creation date comes before region and must do so consistently.

COPY syntax

To read values from partitioned files, use wildcards (globs) in the COPY path and the PARTITION COLUMNS option:

```
=> CREATE EXTERNAL TABLE records (id int, name varchar(50), created date, region varchar(50))
  AS COPY FROM 'webhdfs:////data/*/*/*'
  PARTITION COLUMNS created, region;
```

The path includes one wildcard (*) for each level of directory partitioning and then one more for the files. The number of wildcards must be at least one more than the number of partitioned columns. Data files must include at least one real column; you cannot represent data entirely through directory structure.

The first part of each partition directory name must match the column name in the table definition. COPY parses the string after the = for the values. Empty values, for example a directory named created= , are treated as null values. For backward compatibility, a value of __HIVE_DEFAULT_PARTITION__ also means null.

Values that cannot be coerced to the correct types are rejected in the same way that non-coercible values in data are rejected.

Query execution

When executing queries with predicates, Vertica skips subdirectories that do not satisfy the predicate. This process is called *partition pruning* and it can significantly improve query performance. The following example reads only the partitions for the specified region, for all dates. Although the data is also partitioned by date, the query does not restrict the date.

```
=> SELECT * FROM records WHERE region='northeast';
```

To verify that Vertica is pruning partitions, look in the explain plan for a message similar to the following:

```
files with unmatched partition have been pruned
```

Partitions on object stores

When reading partitioned data from a file system, COPY can prune unneeded directories at each level, skipping entire directories of files. Object stores (S3, GCS, and Azure) do not have directories, so COPY cannot prune paths in the same way.

By default, when querying data on an object store, COPY first fetches a list of all object names. It then prunes unneeded ones before fetching the objects themselves. This approach is called flat listing. Accessing object stores can have high latency, and flat listing maximizes the number of results returned per request. Flat listing retrieves more object names than are needed but does so in as few calls to the object store as possible. In most cases this strategy is an acceptable tradeoff.

If the number of partitions at each level is large, or if there are many layers of partitioning, fetching all of the object names can have a noticeable performance cost. In this situation, you can set the ObjectStoreGlobStrategy configuration parameter to have COPY use hierarchical listing instead of flat listing. With hierarchical listing, COPY fetches information about objects one level at a time, so that COPY can prune unproductive paths earlier.

Consider data that is partitioned by date and then by region:

```
/data/created=2016-11-01/region=northeast/*
/data/created=2016-11-01/region=central/*
/data/created=2016-11-01/region=southeast/*
/data/created=2016-11-01/...
/data/created=2016-11-02/region=northeast/*
/data/created=2016-11-02/region=central/*
/data/created=2016-11-02/region=southeast/*
/data/created=2016-11-02/...
/data/created=2016-11-03/...
/data/...
```

Suppose a query needs only created=2016-11-02 and region=central . With the flat strategy, COPY retrieves the names of all objects in all dates and all regions before pruning most of them. With the hierarchical strategy, COPY first fetches the names of all of the created values, without expanding the paths further. In a second call, it then fetches the names of all of the region values below only that date, ignoring other paths. Finally, it fetches the names of the objects only below that path.

Creating partition structures

To create partitioned file structures, you can use Hive or the file exporters . For information about using Hive, see Hive primer for Vertica integration .

You can create partitions for columns of any simple data type. As a best practice, however, you should avoid partitioning columns with BOOLEAN, FLOAT, and NUMERIC types.

Querying external tables

After you create an external table, you can query it as you would query any other table. Suppose you have created the following external tables:

```
=> CREATE EXTERNAL TABLE catalog (id INT, description VARCHAR, category VARCHAR)
   AS COPY FROM 'hdfs:///dat/catalog.csv' DELIMITER ',';
CREATE TABLE
=> CREATE EXTERNAL TABLE inventory(storeID INT, prodID INT, quantity INT)
   AS COPY FROM 'hdfs:///dat/inventory.csv' DELIMITER ',';
CREATE TABLE
```

You can now write queries against these tables, such as the following:

```
=> SELECT * FROM catalog;
 id |     description      |  category
----+----------------------+-------------
 10 | 24in monitor         | computers
 11 | 27in monitor         | computers
 12 | 24in IPS monitor     | computers
 20 | 1TB USB drive        | computers
 21 | 2TB USB drive        | computers
 22 | 32GB USB thumb drive | computers
 30 | 40in LED TV          | electronics
 31 | 50in LED TV          | electronics
 32 | 60in plasma TV       | electronics
(9 rows)

=> SELECT * FROM inventory;
 storeID | prodID | quantity
---------+--------+----------
     502 |     10 |       17
     502 |     11 |        2
     517 |     10 |        1
     517 |     12 |        2
     517 |     12 |        4
     542 |     10 |        3
     542 |     11 |       11
     542 |     12 |        1
(8 rows)

=> SELECT inventory.storeID,catalog.description,inventory.quantity
   FROM inventory JOIN catalog ON inventory.prodID = catalog.id;
 storeID |   description    | quantity
---------+------------------+----------
     502 | 24in monitor     |       17
     517 | 24in monitor     |        1
     542 | 24in monitor     |        3
     502 | 27in monitor     |        2
     542 | 27in monitor     |       11
     517 | 24in IPS monitor |        2
     517 | 24in IPS monitor |        4
     542 | 24in IPS monitor |        1
(8 rows)
```

One important difference between external tables and Vertica native tables is that querying an external table reads the external data every time. (See How external tables differ from native tables .) Specifically, each time a select query references the external table, Vertica parses the COPY statement definition again to access the data. Certain errors in either your table definition or your data do not become apparent until you run a query, so test your external tables before deploying them in a production environment.

Handling errors

Querying external table data with an incorrect COPY FROM statement definition can potentially result in many rejected rows. To limit the number of rejections, Vertica sets the maximum number of retained rejections with the ExternalTablesExceptionsLimit configuration parameter. The default value is 100. Setting the ExternalTablesExceptionsLimit to –1 removes the limit, but is not recommended.

If COPY errors reach the maximum number of rejections, the external table query continues, but COPY generates a warning in the vertica.log file and does not report subsequent rejected rows.

Using the ExternalTablesExceptionsLimit configuration parameter differs from using the COPY statement REJECTMAX parameter to set a low rejection threshold. The REJECTMAX value controls how many rejected rows to permit before causing the load to fail. If COPY encounters a number of rejected rows equal to or greater than REJECTMAX , COPY aborts execution instead of logging a warning in vertica.log .

Improving query performance for external tables

Queries that include joins perform better if the smaller table is the inner one. For native tables, the query optimizer uses cardinality to choose the inner table. For external tables, the query optimizer uses the row count if available.

After you create an external table, use ANALYZE_EXTERNAL_ROW_COUNT to collect this information. Calling this function is potentially expensive because it has to materialize one column of the table to be able to count the rows, so do this analysis when your database is not busy with critical queries. (This is why Vertica does not perform this operation automatically when you create the table.)

The query optimizer uses the results of your most-recent call to this function when planning queries. If the volume of data changes significantly, therefore, you should run it again to provide updated statistics. A difference of a few percent does not matter, but if your data volume grows by 20% or more, you should repeat this operation when able.

If your data is partitioned, Vertica automatically prunes partitions that cannot affect query results, causing less data to be loaded.

For ORC and Parquet data written with Hive version 0.14 and later, Vertica automatically uses predicate pushdown to further improve query performance. Predicate pushdown moves parts of the query execution closer to the data, reducing the amount of data that must be read from disk or across the network. ORC files written with earlier versions of Hive might not contain the statistics required to perform this optimization. When executing a query against a file that lacks these statistics, Vertica logs an EXTERNAL_PREDICATE_PUSHDOWN_NOT_SUPPORTED event in the QUERY_EVENTS system table. If you are seeing performance problems with your queries, check this table for these events.

Using external tables with user-defined load (UDL) functions

You can use external tables in conjunction with UDL functions that you create. For more information about using UDLs, see User Defined Load (UDL) .

Monitoring external tables

Vertica records information about external tables in system tables. You can use these tables to track your external data and queries against it.

The TABLES system table contains data about all tables, both native and external. The TABLE_DEFINITION column is specific to external tables. You can query this column to see all external data sources currently in use, as in the following example:

```
=> SELECT table_name, create_time, table_definition FROM tables WHERE table_definition != '';
table_name   |          create_time          |                    table_definition
---------------+-------------------------------+-------------------------------------------------------------
customers_orc | 2018-03-21 11:07:30.159442-04 | COPY from '/home/dbadmin/sample_orc_files/0*' ORC
miscprod      | 2018-06-26 17:40:04.012121-04 | copy from '/home/dbadmin/data/prod.csv'
students      | 2018-06-26 17:46:50.695024-04 | copy from '/home/dbadmin/students.csv'
numbers       | 2018-06-26 17:53:52.407441-04 | copy from '/home/dbadmin/tt.dat'
catalog       | 2018-06-26 18:12:28.598519-04 | copy from '/home/dbadmin/data/prod.csv' delimiter ','
inventory     | 2018-06-26 18:13:06.951802-04 | copy from '/home/dbadmin/data/stores.csv' delimiter ','
test          | 2018-06-27 16:31:39.170866-04 | copy from '/home/dbadmin/data/stores.csv' delimiter ','
               (7 rows)
```

The EXTERNAL_TABLE_DETAILS table provides more details, including file sizes. Vertica computes the values in this table at query time, which is potentially expensive, so consider restricting the query by schema or table.

```
=> SELECT table_name, source_format, total_file_size_bytes FROM external_table_details;
table_name   | source_format | total_file_size_bytes
---------------+---------------+-----------------------
customers_orc | ORC           |            619080883
miscprod      | DELIMITED     |                 254
students      | DELIMITED     |                 763
numbers       | DELIMITED     |                  30
catalog       | DELIMITED     |                 254
inventory     | DELIMITED     |                  74
test          | DELIMITED     |                 74
(7 rows)
```

If the size of an external table changes significantly over time, you should rerun ANALYZE_EXTERNAL_ROW_COUNT() to gather updated statistics. See Improving Query Performance for External Tables .

The LOAD_SOURCES table shows information for loads currently in progress. This table does not record information about loads of ORC or Parquet data.

Troubleshooting external tables

You might encounter the following issues when creating or querying external tables. For general data-load troubleshooting, see Troubleshooting data loads .

File not found or permission denied

If a query against an external table produces a file or permission error, ensure that the user executing the query has the necessary permissions in both Vertica and the file system. See the permissions section in Creating external tables .

Error 7226: cannot find partition column

When querying external tables backed by partitioned data, you might see an error message stating that a partition column is missing:

```
ERROR 7226: Cannot find partition column [region] in parquet source
[/data/table_int/int_original/000000_0]
```

If you create an external table and then change the partition structure, for example by renaming a column, you must then re-create the external table. If you see this error, update your table to match the partitioning on disk.

For more information about partition structure, see Partitioned file paths).

Error 6766: is a directory

When querying data, you might see an error message stating that an input file is a directory:

```
ERROR 6766: Error reading from orc parser input stream
[/tmp/orc_glob/more_nations]: Is a directory
```

This error occurs if the glob in the table's COPY FROM clause matches an empty directory. This error occurs only for files in the Linux file system; empty directories in HDFS are ignored.

To correct the error, make the glob more specific. Instead of *, for example, use *.orc.

Data analysis

This guide explains how to query and analyze data in your Vertica database.

In this section

- Queries
- Query optimization
- Views
- Flattened tables
- SQL analytics
- Machine learning for predictive analytics
- Geospatial analytics
- Time series analytics
- Data aggregation

Queries

Queries are database operations that retrieve data from one or more tables or views. In Vertica, the top-level SELECT statement is the query, and a query nested within another SQL statement is called a subquery.

Vertica is designed to run the same SQL standard queries that run on other databases. However, there are some differences between Vertica queries and queries used in other relational database management systems.

The Vertica transaction model is different from the SQL standard in a way that has a profound effect on query performance. You can:

- Run a query on a static backup of the database from any specific date and time. Doing so avoids holding locks or blocking other database operations.
- Use a subset of the standard SQL isolation levels and access modes (read/write or read-only) for a user session .

In Vertica, the primary structure of a SQL query is its statement. Each statement ends with a semicolon, and you can write multiple queries separated by semicolons; for example:

```
=> CREATE TABLE t1( ..., date_col date NOT NULL, ...);
=> CREATE TABLE t2( ..., state VARCHAR NOT NULL, ...);
```

In this section

- Historical queries
- Temporary tables
- SQL queries

- Arrays and sets (collections)
- Rows (structs)
- Subqueries
- Joins

Historical queries

Vertica can execute historical queries, which execute on a snapshot of the database taken at a specific timestamp or epoch. Historical queries can be used to evaluate and possibly recover data that was deleted but has not yet been purged .

You specify a historical query by qualifying the SELECT statement with an AT *epoch* clause, where *epoch* is one of the following:

- EPOCH LATEST : Return data up to but not including the current epoch. The result set includes data from the latest committed DML transaction.
- EPOCH *integer* : Return data up to and including the *integer* -specified epoch.
- TIME ' *timestamp* ' : Return data from the *timestamp* -specified epoch.

> **Note**
> These options are ignored if used to query temporary or external tables.

See Epochs for additional information about how Vertica uses epochs.

Historical queries return data only from the specified epoch. Because they do not return the latest data, historical queries hold no locks or blocking write operations.

Query results are private to the transaction and valid only for the length of the transaction. Query execution is the same regardless of the transaction isolation level.

Restrictions

- The specified epoch, or epoch of the specified timestamp, cannot be less than the Ancient History Mark epoch.
- Vertica does not support running historical queries on temporary tables.

Important

Any changes to a table schema are reflected across all epochs. For example, if you add a column to a table and specify a default value for it, all historical queries on that table display the new column and its default value.

Temporary tables

You can use the CREATE TEMPORARY TABLE statement to implement certain queries using multiple steps:

1. Create one or more temporary tables.
2. Execute queries and store the result sets in the temporary tables.
3. Execute the main query using the temporary tables as if they were a normal part of the logical schema .

See CREATE TEMPORARY TABLE in the SQL Reference Manual for details.

SQL queries

All DML (Data Manipulation Language) statements can contain queries. This section introduces some of the query types in Vertica, with additional details in later sections.

> **Note**
> Many of the examples in this chapter use the VMart schema .

Simple queries

Simple queries contain a query against one table. Minimal effort is required to process the following query, which looks for product keys and SKU numbers in the product table:

```
=> SELECT product_key, sku_number FROM public.product_dimension;
product_key  | sku_number
-------------+-----------
43           | SKU-#129
87           | SKU-#250
42           | SKU-#125
49           | SKU-#154
37           | SKU-#107
36           | SKU-#106
86           | SKU-#248
41           | SKU-#121
88           | SKU-#257
40           | SKU-#120
(10 rows)
```

Tables can contain arrays. You can select the entire array column, an index into it, or the results of a function applied to the array. For more information, see Arrays and sets (collections) .

Joins

Joins use a relational operator that combines information from two or more tables. The query's ON clause specifies how tables are combined, such as by matching foreign keys to primary keys. In the following example, the query requests the names of stores with transactions greater than 70 by joining the store key ID from the store schema's sales fact and sales tables:

```
=> SELECT store_name, COUNT(*) FROM store.store_sales_fact
   JOIN store.store_dimension ON store.store_sales_fact.store_key = store.store_dimension.store_key
   GROUP BY store_name HAVING COUNT(*) > 70 ORDER BY store_name;

 store_name | count
------------+-------
 Store49    |    72
 Store83    |    78
(2 rows)
```

For more detailed information, see Joins . See also the Multicolumn subqueries section in Subquery examples .

Cross joins

Also known as the Cartesian product, a cross join is the result of joining every record in one table with every record in another table. A cross join occurs when there is no join key between tables to restrict records. The following query, for example, returns all instances of vendor and store names in the vendor and store tables:

```
=> SELECT vendor_name, store_name FROM public.vendor_dimension
   CROSS JOIN store.store_dimension;
vendor_name         | store_name
--------------------+------------
Deal Warehouse      | Store41
Deal Warehouse      | Store12
Deal Warehouse      | Store46
Deal Warehouse      | Store50
Deal Warehouse      | Store15
Deal Warehouse      | Store48
Deal Warehouse      | Store39
Sundry Wholesale    | Store41
Sundry Wholesale    | Store12
Sundry Wholesale    | Store46
Sundry Wholesale    | Store50
Sundry Wholesale    | Store15
Sundry Wholesale    | Store48
Sundry Wholesale    | Store39
Market Discounters  | Store41
Market Discounters  | Store12
Market Discounters  | Store46
Market Discounters  | Store50
Market Discounters  | Store15
Market Discounters  | Store48
Market Discounters  | Store39
Market Suppliers    | Store41
Market Suppliers    | Store12
Market Suppliers    | Store46
Market Suppliers    | Store50
Market Suppliers    | Store15
Market Suppliers    | Store48
Market Suppliers    | Store39
...                 | ...
(4000 rows)
```

This example's output is truncated because this particular cross join returned several thousand rows. See also Cross joins .

Subqueries

A subquery is a query nested within another query. In the following example, we want a list of all products containing the highest fat content. The inner query (subquery) returns the product containing the highest fat content among all food products to the outer query block (containing query). The outer query then uses that information to return the names of the products containing the highest fat content.

```
=> SELECT product_description, fat_content FROM public.product_dimension
  WHERE fat_content IN
    (SELECT MAX(fat_content) FROM public.product_dimension
     WHERE category_description = 'Food' AND department_description = 'Bakery')
  LIMIT 10;
       product_description         | fat_content
-----------------------------------+-------------
 Brand #59110 hotdog buns          |          90
 Brand #58107 english muffins      |          90
 Brand #57135 english muffins      |          90
 Brand #54870 cinnamon buns        |          90
 Brand #53690 english muffins      |          90
 Brand #53096 bagels               |          90
 Brand #50678 chocolate chip cookies |          90
 Brand #49269 wheat bread          |          90
 Brand #47156 coffee cake          |          90
 Brand #43844 corn muffins         |          90
(10 rows)
```

For more information, see Subqueries .

Sorting queries

Use the ORDER BY clause to order the rows that a query returns.

Special note about query results

You could get different results running certain queries on one machine or another for the following reasons:

- Partitioning on a FLOAT type could return nondeterministic results because of the precision, especially when the numbers are close to one another, such as results from the RADIANS() function, which has a very small range of output.
  To get deterministic results, use NUMERIC if you must partition by data that is not an INTEGER type.
- Most analytics (with analytic aggregations, such as MIN()/MAX()/SUM()/COUNT()/AVG() as exceptions) rely on a unique order of input data to get deterministic result. If the analytic window-order clause cannot resolve ties in the data, results could be different each time you run the query. For example, in the following query, the analytic ORDER BY does not include the first column in the query, promotion_key . So for a tie of AVG(RADIANS(cost_dollar_amount)), product_version , the same promotion_key could have different positions within the analytic partition, resulting in a different NTILE() number. Thus, DISTINCT could also have a different result:

```
=> SELECT COUNT(*) FROM
    (SELECT DISTINCT SIN(FLOOR(MAX(store.store_sales_fact.promotion_key))),
  NTILE(79) OVER(PARTITION BY AVG (RADIANS
    (store.store_sales_fact.cost_dollar_amount ))
  ORDER BY store.store_sales_fact.product_version)
  FROM store.store_sales_fact
  GROUP BY store.store_sales_fact.product_version,
       store.store_sales_fact.sales_dollar_amount ) AS store;
 count
-------
  1425
(1 row)
```

  If you add MAX(promotion_key) to analytic ORDER BY , the results are the same on any machine:

```
=> SELECT COUNT(*) FROM (SELECT DISTINCT MAX(store.store_sales_fact.promotion_key),
   NTILE(79) OVER(PARTITION BY MAX(store.store_sales_fact.cost_dollar_amount)
  ORDER BY store.store_sales_fact.product_version,
  MAX(store.store_sales_fact.promotion_key))
  FROM store.store_sales_fact
  GROUP BY store.store_sales_fact.product_version,
    store.store_sales_fact.sales_dollar_amount) AS store;
```

Arrays and sets (collections)

Tables can include collections (arrays or sets). An ARRAY is an ordered collection of elements that allows duplicate values, and a SET is an unordered collection of unique values.

Consider an orders table with columns for product keys, customer keys, order prices, and order date, with some containing arrays. A basic query in Vertica results in the following:

```
=> SELECT * FROM orders LIMIT 5;
 orderkey | custkey |        prodkey         |         orderprices          | orderdate
----------+---------+------------------------+------------------------------+------------
    19626 |      91 | ["P1262","P68","P101"] | ["192.59","49.99","137.49"] | 2021-03-14
    25646 |     716 | ["P997","P31","P101"]  | ["91.39","29.99","147.49"]  | 2021-03-14
    25647 |     716 | ["P12"]                | ["8.99"]                    | 2021-03-14
    19743 |     161 | ["P68","P101"]         | ["49.99","137.49"]          | 2021-03-15
    19888 |     241 | ["P1262","P101"]       | ["197.59","142.49"]         | 2021-03-15
(5 rows)
```

As shown in this example, array values are returned in JSON format . Set values are also returned in JSON array format.

You can access elements of nested arrays (multi-dimensional arrays) with multiple indexes:

```
=> SELECT host, pingtimes FROM network_tests;
 host |                pingtimes
------+-----------------------------------------------------
 eng1 | [[24.24,25.27,27.16,24.97], [23.97,25.01,28.12,29.50]]
 eng2 | [[27.12,27.91,28.11,26.95], [29.01,28.99,30.11,31.56]]
 qa1  | [[23.15,25.11,24.63,23.91], [22.85,22.86,23.91,31.52]]
(3 rows)

=> SELECT pingtimes[0] FROM network_tests;
     pingtimes
-------------------------
[24.24,25.27,27.16,24.97]
[27.12,27.91,28.11,26.95]
[23.15,25.11,24.63,23.91]
(3 rows)

=> SELECT pingtimes[0][0] FROM network_tests;
 pingtimes
-----------
24.24
27.12
23.15
(3 rows)
```

Vertica supports several functions to manipulate arrays and sets.

Consider the orders table, which has an array of product keys for all items purchased in a single order. You can use the APPLY_COUNT_ELEMENTS function to find out how many items each order contains (excluding null values):

```
=> SELECT APPLY_COUNT_ELEMENTS(prodkey) FROM orders LIMIT 5;
apply_count_elements
--------------------
3
2
2
3
1
(5 rows)
```

Vertica also supports aggregate functions for collections. Consider a column with an array of prices for items purchased in a single order. You can use the APPLY_SUM function to find the total amount spent for each order:

```
=> SELECT APPLY_SUM(orderprices) FROM orders LIMIT 5;
apply_sum
-----------
380.07
187.48
340.08
268.87
  8.99
(5 rows)
```

Most of the array functions operate only on one-dimensional arrays. To use them with multi-dimensional arrays, first dereference one dimension:

```
=> SELECT APPLY_MAX(pingtimes[0]) FROM network_tests;
 apply_max
-----------
 27.16
 28.11
 25.11
(3 rows)
```

See Collection functions for a comprehensive list of functions.

You can include both column names and literal values in queries. The following example returns the product keys for orders where the number of items in each order is three or more:

```
=> SELECT prodkey FROM orders WHERE APPLY_COUNT_ELEMENTS(prodkey)>2;
     prodkey
------------------------
 ["P1262","P68","P101"]
 ["P997","P31","P101"]
(2 rows)
```

You can use aggregate functions in a WHERE clause:

```
=> SELECT custkey, cust_custname, cust_email, orderkey, prodkey, orderprices FROM orders
 JOIN cust ON custkey = cust_custkey
 WHERE APPLY_SUM(orderprices)>150 ;
custkey|  cust_custname   |        cust_email         |   orderkey   |                 prodkey                 |        orderprices
-------+------------------+---------------------------+--------------+-----------------------------------========---+---------------------------
342799 | "Ananya Patel"   | "ananyapatel98@gmail.com" | "113-341987" | ["MG-7190","VA-4028","EH-1247","MS-7018"] | [60.00,67.00,22.00,14.99]
342845 | "Molly Benton"   | "molly_benton@gmail.com"  | "111-952000" | ["ID-2586","IC-9010","MH-2401","JC-1905"] | [22.00,35.00,90.00,12.00]
342989 | "Natasha Abbasi" | "natsabbasi@live.com"     | "111-685238" | ["HP-4024"]                               | [650.00]
342176 | "Jose Martinez"  | "jmartinez@hotmail.com"   | "113-672238" | ["HP-4768","IC-9010"]                     | [899.00,60.00]
342845 | "Molly Benton"   | "molly_benton@gmail.com"  | "113-864153" | ["AE-7064","VA-4028","GW-1808"]           | [72.00,99.00,185.00]
(5 rows)
```

Element data types

Collections support elements of any scalar type, arrays, or structs (ROW). In the following version of the orders table, an array of ROW elements contains information about all shipments for an order:

```
=> CREATE TABLE orders(
  orderid INT,
  accountid INT,
  shipments ARRAY[
    ROW(
      shipid INT,
      address ROW(
        street VARCHAR,
        city VARCHAR,
        zip INT
        ),
      shipdate DATE
    )
  ]
);
```

Some orders consist of more than one shipment. Line breaks have been inserted into the following output for legibility:

```
=> SELECT * FROM orders;
 orderid | accountid |                                                
```

```
=> SELECT orderid, shipments[0].shipdate AS ship1, shipments[1].shipdate AS ship2 FROM orders;
 orderid |   ship1    |   ship2
---------+------------+------------
   99123 | 2020-11-05 | 2020-11-06
   99149 | 2020-11-06 |
   99162 | 2020-11-04 | 2020-11-11
(3 rows)
```

This example selects specific array indices. To access all entries, use EXPLODE . To search or filter elements, see Searching and Filtering .

Some data formats have a map type, which is a set of key/value pairs. Vertica does not directly support querying maps, but you can define a map column as an array of structs and query that. In the following example, the prods column in the data is a map:

```
=> CREATE EXTERNAL TABLE orders
 (orderkey INT,
  custkey INT,
  prods ARRAY[ROW(key VARCHAR(10), value DECIMAL(12,2))],
  orderdate DATE
 ) AS COPY FROM '...' PARQUET;

=> SELECT orderkey, prods FROM orders;
 orderkey |                                        prods
----------+-------------------------------------------------------------------------------------------------
    19626 | [{"key":"P68","value":"49.99"},{"key":"P1262","value":"192.59"},{"key":"P101","value":"137.49"}]
    25646 | [{"key":"P997","value":"91.39"},{"key":"P101","value":"147.49"},{"key":"P31","value":"29.99"}]
    25647 | [{"key":"P12","value":"8.99"}]
    19743 | [{"key":"P68","value":"49.99"},{"key":"P101","value":"137.49"}]
    19888 | [{"key":"P1262","value":"197.59"},{"key":"P101","value":"142.49"}]
(5 rows)
```

You cannot use complex columns in CREATE TABLE AS SELECT (CTAS). This restriction applies for the entire column or for field selection within it.

Ordering and grouping

You can use Comparison operators with collections of scalar values. Null collections are ordered last. Otherwise, collections are compared element by element until there is a mismatch, and then they are ordered based on the non-matching elements. If all elements are equal up to the length of the shorter one, then the shorter one is ordered first.

You can use collections in the ORDER BY and GROUP BY clauses of queries. The following example shows ordering query results by an array column:

```
=> SELECT * FROM employees
   ORDER BY grant_values;
 id | department |          grants          |  grant_values
----+------------+--------------------------+----------------
 36 | Astronomy  | ["US-7376","DARPA-1567"] | [5000,4000]
 36 | Physics    | ["US-7376","DARPA-1567"] | [10000,25000]
 33 | Physics    | ["US-7376"]              | [30000]
 42 | Physics    | ["US-7376","DARPA-1567"] | [65000,135000]
(4 rows)
```

The following example queries the same table using GROUP BY :

```
=> SELECT department, grants, SUM(apply_sum(grant_values))
   FROM employees
   GROUP BY grants, department;
 department |          grants          |  SUM
------------+--------------------------+--------
 Physics    | ["US-7376","DARPA-1567"] | 235000
 Astronomy  | ["US-7376","DARPA-1567"] |   9000
 Physics    | ["US-7376"]              |  30000
(3 rows)
```

See the "Functions and Operators" section on the ARRAY reference page for information on how Vertica orders collections. (The same information is also on the SET reference page.)

Null semantics

Null semantics for collections are consistent with normal columns in most regards. See NULL sort order for more information on null-handling.

The null-safe equality operator (<=>) behaves differently from equality (=) when the collection is null rather than empty. Comparing a collection to NULL strictly returns null:

```
=> SELECT ARRAY[1,3] = NULL;
?column?
----------

(1 row)

=> SELECT ARRAY[1,3] <=> NULL;
 ?column?
----------
 f
(1 row)
```

In the following example, the grants column in the table is null for employee 99:

```
=> SELECT grants = NULL FROM employees WHERE id=99;
 ?column?
----------

(1 row)

=> SELECT grants <=> NULL FROM employees WHERE id=99;
 ?column?
----------
 t
(1 row)
```

Empty collections are not null and behave as expected:

```
=> SELECT ARRAY[]::ARRAY[INT] = ARRAY[]::ARRAY[INT];
 ?column?
----------
 t
(1 row)
```

Collections are compared element by element. If a comparison depends on a null element, the result is unknown (null), not false. For example, ARRAY[1,2,null]=ARRAY[1,2,null] and ARRAY[1,2,null]=ARRAY[1,2,3] both return null, but ARRAY[1,2,null]=ARRAY[1,4,null] returns false because the second elements do not match.

Out-of-bound indexes into collections return NULL:

```
=> SELECT prodkey[2] FROM orders LIMIT 4;
prodkey
---------

"EH-1247"
"MH-2401"

(4 rows)
```

The results of the query return NULL for two out of four rows, the first and the fourth, because the specified index is greater than the size of those arrays.

Casting

When the data type of an expression value is unambiguous, it is implicitly coerced to match the expected data type. However, there can be ambiguity about the data type of an expression. Write an explicit cast to avoid the default:

```
=> SELECT APPLY_SUM(ARRAY['1','2','3']);
ERROR 5595:  Invalid argument type VarcharArray1D in function apply_sum

=> SELECT APPLY_SUM(ARRAY['1','2','3']::ARRAY[INT]);
 apply_sum
-----------
         6
(1 row)
```

You can cast arrays or sets of one scalar type to arrays or sets of other (compatible) types, following the same rules as for casting scalar values. Casting a collection casts each element of that collection. Casting an array to a set also removes any duplicates.

You can cast arrays (but not sets) with elements that are arrays or structs (or combinations):

```
=> SELECT shipments::ARRAY[ROW(id INT,addr ROW(VARCHAR,VARCHAR,INT),shipped DATE)]
FROM orders;
                    shipments
-------------------------------------------------------------------------------
 [{"id":1,"addr":{"street":"911 San Marcos St","city":"Austin","zip":73344},"shipped":"2020-11-05"},
  {"id":2,"addr":{"street":"100 main St Apt 4B","city":"Pasadena","zip":91001},"shipped":"2020-11-06"}]
 [{"id":3,"addr":{"street":"100 main St Apt 4B","city":"Pasadena","zip":91001},"shipped":"2020-11-06"}]
 [{"id":4,"addr":{"street":"100 main St Apt 4B","city":"Pasadena","zip":91001},"shipped":"2020-11-04"},
  {"id":5,"addr":{"street":"100 Main St Apt 4A","city":"Pasadena","zip":91001},"shipped":"2020-11-11"}]
(3 rows)
```

You can change the bound of an array or set by casting. When casting to a bounded native array, inputs that are too long are truncated. When casting to a multi-dimensional array, if the new bounds are too small for the data the cast fails:

```
=> SELECT ARRAY[1,2,3]::ARRAY[VARCHAR,2];
   array
-----------
 ["1","2"]
(1 row)

=> SELECT ARRAY[ARRAY[1,2,3],ARRAY[4,5,6]]::ARRAY[ARRAY[VARCHAR,2],2];
ERROR 9227:  Output array isn't big enough
DETAIL:  Type limit is 4 elements, but value has 6 elements
```

If you cast to a bounded multi-dimensional array, you must specify the bounds at all levels.

An array or set with a single null element must be explicitly cast because no type can be inferred.

See Data type coercion for more information on casting for data types.

Exploding and imploding array columns

To simplify access to elements, you can use the EXPLODE and UNNEST functions. These functions take one or more array columns from a table and expand them, producing one row per element. You can use EXPLODE and UNNEST when you need to perform aggregate operations across all elements of all arrays. You can also use EXPLODE when you need to operate on individual elements. See Manipulating Elements .

EXPLODE and UNNEST differ in their output:

- EXPLODE returns two columns for each array, one for the element index and one for the value at that position. If the function explodes a single array, these columns are named position and value by default. If the function explodes two or more arrays, the columns for each array are named pos_ *column-name* and val_ *column-name* .
- UNNEST returns only the values. For a single array, the output column is named value . For multiple arrays, each output column is named val_ *column-name* .

EXPLODE and UNNEST also differ in their inputs. UNNEST accepts only array arguments and expands all of them. EXPLODE can accept other arguments and passes them through, expanding only as many arrays as requested (default 1).

Consider an orders table with the following contents:

```
=> SELECT orderkey, custkey, prodkey, orderprices, email_addrs
   FROM orders LIMIT 5;
  orderkey  | custkey |                  prodkey                  |          orderprices          |                                                  email_addrs
------------+---------+-------------------------------------------+-------------------------------+-------------------------------------------------------------------------------------------------
----
 113-341987 |  342799 | ["MG-7190 ","VA-4028 ","EH-1247 ","MS-7018 "] | ["60.00","67.00","22.00","14.99"] |
["bob@example,com","robert.jones@example.com"]
 111-952000 |  342845 | ["ID-2586 ","IC-9010 ","MH-2401 ","JC-1905 "] | ["22.00","35.00",null,"12.00"]    | ["br92@cs.example.edu"]
 111-345634 |  342536 | ["RS-0731 ","SJ-2021 "]                       | ["50.00",null]                    | [null]
 113-965086 |  342176 | ["GW-1808 "]                                  | ["108.00"]                        | ["joe.smith@example.com"]
 111-335121 |  342321 | ["TF-3556 "]                                  | ["50.00"]                         | ["789123@example-
isp.com","alexjohnson@example.com","monica@eng.example.com","sara@johnson.example.name",null]
(5 rows)
```

The following query explodes the order prices for a single customer. The other two columns are passed through and are repeated for each returned row:

```
=> SELECT EXPLODE(orderprices, custkey, email_addrs
                USING PARAMETERS skip_partitioning=true)
          AS (position, orderprices, custkey, email_addrs)
   FROM orders WHERE custkey='342845' ORDER BY orderprices;
 position | orderprices | custkey |         email_addrs
----------+-------------+---------+-----------------------------
        2 |             |  342845 | ["br92@cs.example.edu",null]
        3 |       12.00 |  342845 | ["br92@cs.example.edu",null]
        0 |       22.00 |  342845 | ["br92@cs.example.edu",null]
        1 |       35.00 |  342845 | ["br92@cs.example.edu",null]
(4 rows)
```

The previous example uses the skip_partitioning parameter. Instead of setting it for each call to EXPLODE, you can set it as a session parameter. EXPLODE is part of the ComplexTypesLib UDx library. The following example returns the same results:

```
=> ALTER SESSION SET UDPARAMETER FOR ComplexTypesLib skip_partitioning=true;

=> SELECT EXPLODE(orderprices, custkey, email_addrs)
          AS (position, orderprices, custkey, email_addrs)
   FROM orders WHERE custkey='342845' ORDER BY orderprices;
```

You can explode more than one column by specifying the explode_count parameter:

```
=> SELECT EXPLODE(orderkey, prodkey, orderprices
                  USING PARAMETERS explode_count=2, skip_partitioning=true)
          AS (orderkey,pk_idx,pk_val,ord_idx,ord_val)
   FROM orders
   WHERE orderkey='113-341987';
  orderkey  | pk_idx |  pk_val  | ord_idx | ord_val
------------+--------+----------+---------+---------
 113-341987 |      0 | MG-7190  |       0 |   60.00
 113-341987 |      0 | MG-7190  |       1 |   67.00
 113-341987 |      0 | MG-7190  |       2 |   22.00
 113-341987 |      0 | MG-7190  |       3 |   14.99
 113-341987 |      1 | VA-4028  |       0 |   60.00
 113-341987 |      1 | VA-4028  |       1 |   67.00
 113-341987 |      1 | VA-4028  |       2 |   22.00
 113-341987 |      1 | VA-4028  |       3 |   14.99
 113-341987 |      2 | EH-1247  |       0 |   60.00
 113-341987 |      2 | EH-1247  |       1 |   67.00
 113-341987 |      2 | EH-1247  |       2 |   22.00
 113-341987 |      2 | EH-1247  |       3 |   14.99
 113-341987 |      3 | MS-7018  |       0 |   60.00
 113-341987 |      3 | MS-7018  |       1 |   67.00
 113-341987 |      3 | MS-7018  |       2 |   22.00
 113-341987 |      3 | MS-7018  |       3 |   14.99
(16 rows)
```

If you do not need the element positions, you can use UNNEST:

```
=> SELECT orderkey, UNNEST(prodkey, orderprices)
   FROM orders WHERE orderkey='113-341987';
  orderkey  | val_prodkey | val_orderprices
------------+-------------+-----------------
 113-341987 | MG-7190     |           60.00
 113-341987 | MG-7190     |           67.00
 113-341987 | MG-7190     |           22.00
 113-341987 | MG-7190     |           14.99
 113-341987 | VA-4028     |           60.00
 113-341987 | VA-4028     |           67.00
 113-341987 | VA-4028     |           22.00
 113-341987 | VA-4028     |           14.99
 113-341987 | EH-1247     |           60.00
 113-341987 | EH-1247     |           67.00
 113-341987 | EH-1247     |           22.00
 113-341987 | EH-1247     |           14.99
 113-341987 | MS-7018     |           60.00
 113-341987 | MS-7018     |           67.00
 113-341987 | MS-7018     |           22.00
 113-341987 | MS-7018     |           14.99
(16 rows)
```

The following example uses a multi-dimensional array:

```
=> SELECT name, pingtimes FROM network_tests;
 name |                    pingtimes
------+-------------------------------------------------------
 eng1 | [[24.24,25.27,27.16,24.97],[23.97,25.01,28.12,29.5]]
 eng2 | [[27.12,27.91,28.11,26.95],[29.01,28.99,30.11,31.56]]
 qa1  | [[23.15,25.11,24.63,23.91],[22.85,22.86,23.91,31.52]]
(3 rows)

=> SELECT EXPLODE(name, pingtimes USING PARAMETERS explode_count=1) OVER()
   FROM network_tests;
 name | position |          value
------+----------+--------------------------
 eng1 |        0 | [24.24,25.27,27.16,24.97]
 eng1 |        1 | [23.97,25.01,28.12,29.5]
 eng2 |        0 | [27.12,27.91,28.11,26.95]
 eng2 |        1 | [29.01,28.99,30.11,31.56]
 qa1  |        0 | [23.15,25.11,24.63,23.91]
 qa1  |        1 | [22.85,22.86,23.91,31.52]
(6 rows)
```

You can rewrite the previous query as follows to produce the same results:

```
=> SELECT name, EXPLODE(pingtimes USING PARAMETERS skip_partitioning=true)
   FROM network_tests;
```

The IMPLODE function is the inverse of EXPLODE and UNNEST: it takes a column and produces an array containing the column's values. You can use WITHIN GROUP ORDER BY to control the order of elements in the imploded array. Combined with GROUP BY, IMPLODE can be used to reverse an EXPLODE operation.

If the output array would be too large for the column, IMPLODE returns an error. To avoid this, you can set the allow_truncate parameter to omit some elements from the results. Truncation never applies to individual elements; for example, the function does not shorten strings.

Searching and filtering

You can search array elements without having to first explode the array using the following functions:

- CONTAINS : tests whether an array contains an element
- ARRAY_FIND : returns the position of the first matching element
- FILTER : returns an array containing only matching elements from an input array

You can use CONTAINS and ARRAY_FIND to search for specific elements:

```
=> SELECT CONTAINS(email, 'frank@example.com') FROM people;
 CONTAINS
----------
 f
 t
 f
 f
(4 rows)
```

Suppose, instead of finding a particular address, you want to find all of the people who use an example.com email address. Instead of specifying a literal value, you can supply a lambda function to test elements. A lambda function has the following syntax:

```
argument -> expression
```

The lambda function takes the place of the second argument:

```
=> SELECT CONTAINS(email, e -> REGEXP_LIKE(e,'example.com','i')) FROM people;
 CONTAINS
----------
 f
 t
 f
 t
(4 rows)
```

The FILTER function tests array elements, like ARRAY_FIND and CONTAINS, but then returns an array containing only the elements that match the filter:

```
=> SELECT name, email FROM people;
     name      |                      email
---------------+-------------------------------------------------
 Elaine Jackson | ["ejackson@somewhere.org","elaine@jackson.com"]
 Frank Adams    | ["frank@example.com"]
 Lee Jones      | ["lee.jones@somewhere.org"]
 M Smith        | ["msmith@EXAMPLE.COM","ms@MSMITH.COM"]
(4 rows)

=> SELECT name, FILTER(email, e -> NOT REGEXP_LIKE(e,'example.com','i')) AS 'real_email'
   FROM people;
     name      |                 real_email
---------------+-------------------------------------------------
 Elaine Jackson | ["ejackson@somewhere.org","elaine@jackson.com"]
 Frank Adams    | []
 Lee Jones      | ["lee.jones@somewhere.org"]
 M Smith        | ["ms@MSMITH.COM"]
(4 rows)
```

To filter out entire rows without real email addresses, test the array length after applying the filter:

```
=> SELECT name, FILTER(email, e -> NOT REGEXP_LIKE(e,'example.com','i')) AS 'real_email'
   FROM people
   WHERE ARRAY_LENGTH(real_email) > 0;
     name      |                 real_email
---------------+-------------------------------------------------
 Elaine Jackson | ["ejackson@somewhere.org","elaine@jackson.com"]
 Lee Jones      | ["lee.jones@somewhere.org"]
 M Smith        | ["ms@MSMITH.COM"]
(3 rows)
```

The lambda function has an optional second argument, which is the element index. For an example that uses the index argument, see Lambda functions .

Manipulating elements

You can filter array elements using the FILTER function as explained in Searching and Filtering . Filtering does not allow for operations on the filtered elements. For this case, you can use FILTER, EXPLODE, and IMPLODE together.

Consider a table of people, where each row has a person's name and an array of email addresses, some invalid:

```
=> SELECT * FROM people;
 id |     name      |                      email
----+---------------+-------------------------------------------------
 56 | Elaine Jackson | ["ejackson@somewhere.org","elaine@jackson.com"]
 61 | Frank Adams    | ["frank@example.com"]
 87 | Lee Jones      | ["lee.jones@somewhere.org"]
 91 | M Smith        | ["msmith@EXAMPLE.COM","ms@MSMITH.COM"]
(4 rows)
```

Email addresses are case-insensitive but VARCHAR values are not. The following example filters out invalid email addresses and normalizes the remaining ones by converting them to lowercase. The order of operations for each row is:

1. Filter each array to remove addresses using example.com , partitioning by name.
2. Explode the filtered array.
3. Convert each element to lowercase.
4. Implode the lowercase elements, grouping by name.

```
=> WITH exploded AS
(SELECT EXPLODE(FILTER(email, e -> NOT REGEXP_LIKE(e, 'example.com', 'i')), name)
       OVER (PARTITION BEST) AS (pos, addr, name)
       FROM people)
SELECT name, IMPLODE(LOWER(addr)) AS email
FROM exploded GROUP BY name;
     name      |                    email
---------------+-----------------------------------------------
 Elaine Jackson | ["ejackson@somewhere.org","elaine@jackson.com"]
 Lee Jones      | ["lee.jones@somewhere.org"]
 M Smith        | ["ms@msmith.com"]
(3 rows)
```

Because the second row in the original table did not have any remaining email addresses after the filter step, there was nothing to partition by. Therefore, the row does not appear in the results at all.

Rows (structs)

Tables can include columns of the ROW data type. A ROW, sometimes called a struct, is a set of typed property-value pairs.

Consider a table of customers with columns for name, address, and an ID. The address is a ROW with fields for the elements of an address (street, city, and postal code). As shown in this example, ROW values are returned in JSON format :

```
=> SELECT * FROM customers ORDER BY accountID;
       name        |                          address                          | accountID
-------------------+---------------------------------------------------------------------------+-----------
 Missy Cooper      | {"street":"911 San Marcos St","city":"Austin","zipcode":73344}    |        17
 Sheldon Cooper    | {"street":"100 Main St Apt 4B","city":"Pasadena","zipcode":91001} |       139
 Leonard Hofstadter | {"street":"100 Main St Apt 4A","city":"Pasadena","zipcode":91001} |       142
 Leslie Winkle     | {"street":"23 Fifth Ave Apt 8C","city":"Pasadena","zipcode":91001} |       198
 Raj Koothrappali  | {"street":null,"city":"Pasadena","zipcode":91001}                 |       294
 Stuart Bloom      |                                                                   |       482
(6 rows)
```

Most values are cast to UTF-8 strings, as shown for street and city here. Integers and booleans are cast to JSON Numerics and thus not quoted.

Use dot notation (*column* . *field*) to access individual fields:

```
=> SELECT address.city FROM customers;
  city
----------
 Pasadena
 Pasadena
 Pasadena
 Pasadena
 Austin

(6 rows)
```

In the following example, the contact information in the customers table has an email field, which is an array of addresses:

```
=> SELECT name, contact.email FROM customers;
       name        |                    email
-------------------+-----------------------------------------------
 Missy Cooper      | ["missy@mit.edu","mcooper@cern.gov"]
 Sheldon Cooper    | ["shelly@meemaw.name","cooper@caltech.edu"]
 Leonard Hofstadter | ["hofstadter@caltech.edu"]
 Leslie Winkle     | []
 Raj Koothrappali  | ["raj@available.com"]
 Stuart Bloom      |
(6 rows)
```

You can use ROW columns or specific fields to restrict queries, as in the following example:

```
=> SELECT address FROM customers WHERE address.city ='Pasadena';
                          address
-------------------------------------------------------------------
 {"street":"100 Main St Apt 4B","city":"Pasadena","zipcode":91001}
 {"street":"100 Main St Apt 4A","city":"Pasadena","zipcode":91001}
 {"street":"23 Fifth Ave Apt 8C","city":"Pasadena","zipcode":91001}
 {"street":null,"city":"Pasadena","zipcode":91001}
(4 rows)
```

You can use the ROW syntax to specify literal values, such as the address in the WHERE clause in the following example:

```
=> SELECT name,address FROM customers
   WHERE address = ROW('100 Main St Apt 4A','Pasadena',91001);
       name        |                           address
-------------------+-------------------------------------------------------------------
 Leonard Hofstadter | {"street":"100 Main St Apt 4A","city":"Pasadena","zipcode":91001}
(1 row)
```

You can join on field values as you would from any other column:

```
=> SELECT accountID,department from customers JOIN employees
   ON customers.name=employees.personal.name;
 accountID | department
-----------+------------
       139 | Physics
       142 | Physics
       294 | Astronomy
```

You can join on full structs. The following example joins the addresses in the employees and customers tables:

```
=> SELECT employees.personal.name,customers.accountID FROM employees
JOIN customers ON employees.personal.address=customers.address;
       name        | accountID
-------------------+-----------
 Sheldon Cooper    |       139
 Leonard Hofstadter |       142
(2 rows)
```

You can cast structs, optionally specifying new field names:

```
=> SELECT contact::ROW(str VARCHAR, city VARCHAR, zip VARCHAR, email ARRAY[VARCHAR,
20]) FROM customers;
                                        contact

-------------------------------------------------------------------------------
---------------------------------
 {"str":"911 San Marcos St","city":"Austin","zip":"73344","email":["missy@mit.ed
u","mcooper@cern.gov"]}
 {"str":"100 Main St Apt 4B","city":"Pasadena","zip":"91001","email":["shelly@me
emaw.name","cooper@caltech.edu"]}
 {"str":"100 Main St Apt 4A","city":"Pasadena","zip":"91001","email":["hofstadte
r@caltech.edu"]}
 {"str":"23 Fifth Ave Apt 8C","city":"Pasadena","zip":"91001","email":[]}
 {"str":null,"city":"Pasadena","zip":"91001","email":["raj@available.com"]}

(6 rows)
```

You can use structs in views and in subqueries, as in the following example:

```
=> CREATE VIEW neighbors (num_neighbors, area(city, zipcode))
AS SELECT count(*), ROW(address.city, address.zipcode)
FROM customers GROUP BY address.city, address.zipcode;
CREATE VIEW

=> SELECT employees.personal.name, neighbors.area FROM neighbors, employees
WHERE employees.personal.address.zipcode=neighbors.area.zipcode AND neighbors.nu
m_neighbors > 1;
       name       |               area
------------------+-----------------------------------
 Sheldon Cooper   | {"city":"Pasadena","zipcode":91001}
 Leonard Hofstadter | {"city":"Pasadena","zipcode":91001}
(2 rows)
```

If a reference is ambiguous, Vertica prefers column names over field names.

You can use many operators and predicates with ROW columns, including JOIN, GROUP BY, ORDER BY, IS [NOT] NULL, and comparison operations in nullable filters. Some operators do not logically apply to structured data and are not supported. See the ROW reference page for a complete list.

Subqueries

A subquery is a SELECT statement embedded within another SELECT statement. The embedded subquery is often referenced as the query's inner statement, while the containing query is typically referenced as the query's statement, or outer query block. A subquery returns data that the outer query uses as a condition to determine what data to retrieve. There is no limit to the number of nested subqueries you can create.

Like any query, a subquery returns records from a table that might consist of a single column and record, a single column with multiple records, or multiple columns and records. Subqueries can be noncorrelated or correlated . You can also use them to update or delete table records, based on values in other database tables.

In this section

- Subqueries used in search conditions
- Subqueries in the SELECT list
- Noncorrelated and correlated subqueries
- Flattening FROM clause subqueries
- Subqueries in UPDATE and DELETE statements
- Subquery examples
- Subquery restrictions

Subqueries used in search conditions

Subqueries are used as search conditions in order to filter results. They specify the conditions for the rows returned from the containing query's select-list, a query expression, or the subquery itself. The operation evaluates to TRUE, FALSE, or UNKNOWN (NULL).

Syntax

```
search-condition {
   [ { AND | OR | NOT } {  predicate | ( search-condition ) } ]
  }[,... ]
 predicate
    { expression comparison-operator expression
       | string-expression [ NOT ] { LIKE | ILIKE | LIKEB | ILIKEB } string-expression
       | expression IS [ NOT ] NULL
       | expression [ NOT ] IN ( subquery | expression[,... ] )
       | expression comparison-operator [ ANY | SOME ] ( subquery )
       | expression comparison-operator ALL ( subquery )
       | expression OR ( subquery )
       | [ NOT ] EXISTS ( subquery )
       | [ NOT ] IN ( subquery )
    }
```

Arguments

search-condition	Specifies the search conditions for the rows returned from one of the following: • Containing query's select-list • Query expression • Subquery If the subquery is used with an UPDATE or DELETE statement, UPDATE specifies the rows to update and DELETE specifies the rows to delete.
{ AND | OR | NOT }	Logical operators: • AND : Combines two conditions and evaluates to TRUE when both of the conditions are TRUE. • OR : Combines two conditions and evaluates to TRUE when either condition is TRUE. • NOT : Negates the Boolean expression specified by the predicate.
predicate	An expression that returns TRUE, FALSE, or UNKNOWN (NULL).
expression	A column name, constant, function, or scalar subquery, or combination of column names, constants, and functions connected by operators or subqueries.
comparison-operator	An operator that tests conditions between two expressions, one of the following: • < : less than • > : greater than • <= : less than or equal • >= : greater than or equal • = : equal; returns UNKNOWN if either expression does • <=> : Like the = operator, but returns TRUE (instead of UNKNOWN) if both expressions evaluate to UNKNOWN, and FALSE (instead of UNKNOWN) if one expression evaluates to UNKNOWN. • <> : not equal • != : not equal
string-expression	A character string with optional wildcard (*) characters.
[NOT] { LIKE | ILIKE | LIKEB | ILIKEB }	Indicates that the character string following the predicate is to be used (or not used) for pattern matching.

IS [NOT] NULL	Searches for values that are null or are not null.
ALL	Used with a comparison operator and a subquery. Returns TRUE for the left-hand predicate if all values returned by the subquery satisfy the comparison operation, or FALSE if not all values satisfy the comparison or if the subquery returns no rows to the outer query block.
ANY | SOME	ANY and SOME are synonyms and are used with a comparison operator and a subquery. Either returns TRUE for the left-hand predicate if any value returned by the subquery satisfies the comparison operation, or FALSE if no values in the subquery satisfy the comparison or if the subquery returns no rows to the outer query block. Otherwise, the expression is UNKNOWN.
[NOT] EXISTS	Used with a subquery to test for the existence of records that the subquery returns.
[NOT] IN	Searches for an expression on the basis of an expression's exclusion or inclusion from a list. The list of values is enclosed in parentheses and can be a subquery or a set of constants.

Expressions as subqueries

Subqueries that return a single value (unlike a list of values returned by IN subqueries) can generally be used anywhere an expression is allowed in SQL: a column name, constant, function, scalar subquery, or a combination of column names, constants, and functions connected by operators or subqueries.

For example:

```
=> SELECT c1 FROM t1 WHERE c1 = ANY (SELECT c1 FROM t2) ORDER BY c1;
=> SELECT c1 FROM t1 WHERE COALESCE((t1.c1 > ANY (SELECT c1 FROM t2)), TRUE);
=> SELECT c1 FROM t1 GROUP BY c1 HAVING
   COALESCE((t1.c1 <> ALL (SELECT c1 FROM t2)), TRUE);
```

Multi-column expressions are also supported:

```
=> SELECT c1 FROM t1 WHERE (t1.c1, t1.c2) = ALL (SELECT c1, c2 FROM t2);
=> SELECT c1 FROM t1 WHERE (t1.c1, t1.c2) <> ANY (SELECT c1, c2 FROM t2);
```

Vertica returns an error on queries where more than one row would be returned by any subquery used as an expression:

```
=> SELECT c1 FROM t1 WHERE c1 = (SELECT c1 FROM t2) ORDER BY c1;
 ERROR:  more than one row returned by a subquery used as an expression
```

See also

- Subquery restrictions

Subqueries in the SELECT list

Subqueries can occur in the select list of the containing query. The results from the following statement are ordered by the first column (customer_name). You could also write ORDER BY 2 and specify that the results be ordered by the select-list subquery.

```
=> SELECT c.customer_name, (SELECT AVG(annual_income) FROM customer_dimension
   WHERE deal_size = c.deal_size) AVG_SAL_DEAL FROM customer_dimension c
    ORDER BY 1;
 customer_name | AVG_SAL_DEAL
---------------+--------------
 Goldstar      |       603429
 Metatech      |       628086
 Metadata      |       666728
 Foodstar      |       695962
 Verihope      |       715683
 Veridata      |       868252
 Bettercare    |       879156
 Foodgen       |       958954
 Virtacom      |       991551
 Inicorp       |      1098835
...
```

Notes

- Scalar subqueries in the select-list return a single row/column value. These subqueries use Boolean comparison operators: =, >, <, <>, <=, >=. If the query is correlated, it returns NULL if the correlation results in 0 rows. If the query returns more than one row, the query errors out at run time and Vertica displays an error message that the scalar subquery must only return 1 row.
- Subquery expressions such as [NOT] IN, [NOT] EXISTS, ANY/SOME, or ALL always return a single Boolean value that evaluates to TRUE, FALSE, or UNKNOWN; the subquery itself can have many rows. Most of these queries can be correlated or noncorrelated.

  > **Note**
  > ALL subqueries cannot be correlated.

- Subqueries in the ORDER BY and GROUP BY clauses are supported; for example, the following statement says to order by the first column, which is the select-list subquery:

  ```
  => SELECT (SELECT MAX(x) FROM t2 WHERE y=t1.b) FROM t1 ORDER BY 1;
  ```

See also

- Subquery restrictions

Noncorrelated and correlated subqueries

Subqueries can be categorized into two types:

- A *noncorrelated* subquery obtains its results independently of its containing (outer) statement.
- A *correlated* subquery requires values from its outer query in order to execute.

Noncorrelated subqueries

A noncorrelated subquery executes independently of the outer query. The subquery executes first, and then passes its results to the outer query, For example:

```
=> SELECT name, street, city, state FROM addresses WHERE state IN (SELECT state FROM states);
```

Vertica executes this query as follows:

1. Executes the subquery SELECT state FROM states (in bold).
2. Passes the subquery results to the outer query.

A query's WHERE and HAVING clauses can specify noncorrelated subqueries if the subquery resolves to a single row, as shown below:

In WHERE clause

```
=> SELECT COUNT(*) FROM SubQ1 WHERE SubQ1.a = (SELECT y from SubQ2);
```

In HAVING clause

```
=> SELECT COUNT(*) FROM SubQ1 GROUP BY SubQ1.a HAVING SubQ1.a = (SubQ1.a & (SELECT y from SubQ2)
```

Correlated subqueries

A correlated subquery typically obtains values from its outer query before it executes. When the subquery returns, it passes its results to the outer query. Correlated subqueries generally conform to the following format:

```
SELECT outer-column[,...] FROM t1 outer
  WHERE outer-column comparison-operator
    (SELECT sq-column[,...] FROM t2 sq
      WHERE sq.expr = outer.expr);
```

> **Note**
> You can use an outer join to obtain the same effect as a correlated subquery.

In the following example, the subquery needs values from the addresses.state column in the outer query:

```
=> SELECT name, street, city, state FROM addresses
   WHERE EXISTS (SELECT * FROM states WHERE states.state = addresses.state);
```

Vertica executes this query as follows:

1. Extracts and evaluates each addresses.state value in the outer subquery records.
2. Using the EXISTS predicate, checks addresses in the inner (correlated) subquery.
3. Stops processing when it finds the first match.

When Vertica executes this query, it translates the full query into a JOIN WITH SIPS.

Flattening FROM clause subqueries

FROM clause subqueries are always evaluated before their containing query. In some cases, the optimizer *flattens* FROM clause subqueries so the query can execute more efficiently.

For example, in order to create a query plan for the following statement, the Vertica query optimizer evaluates all records in table t1 before it evaluates the records in table t0 :

```
=> SELECT * FROM (SELECT a, MAX(a) AS max FROM (SELECT * FROM t1) AS t0 GROUP BY a);
```

Given the previous query, the optimizer can internally flatten it as follows:

```
=> SELECT * FROM (SELECT a, MAX(a) FROM t1 GROUP BY a) AS t0;
```

Both queries return the same results, but the flattened query runs more quickly.

Flattening views

When a query's FROM clause specifies a view , the optimizer expands the view by replacing it with the query that the view encapsulates. If the view contains subqueries that are eligible for flattening, the optimizer produces a query plan that flattens those subqueries.

Flattening restrictions

The optimizer cannot create a flattened query plan if a subquery or view contains one of the following elements:

- Aggregate function
- Analytic function
- Outer join (left, right or full)
- GROUP BY , ORDER BY , or HAVING clause
- DISTINCT keyword
- LIMIT or OFFSET clause
- UNION , EXCEPT , or INTERSECT clause
- EXISTS subquery

Examples

If a predicate applies to a view or subquery, the flattening operation can allow for optimizations by evaluating the predicates before the flattening takes place. Two examples follow.

View flattening

In this example, view v1 is defined as follows:

```
=> CREATE VIEW v1 AS SELECT * FROM a;
```

The following query specifies this view:

```
=> SELECT * FROM v1 JOIN b ON x=y WHERE x > 10;
```

Without flattening, the optimizer evaluates the query as follows:

1. Evalutes the subquery.
2. Applies the predicate WHERE x > 10 .

In contrast, the optimizer can create a flattened query plan by applying the predicate before evaluating the subquery. This reduces the optimizer's work because it returns only the records WHERE x > 10 to the containing query.

Vertica internally transforms the previous query as follows:

```
=> SELECT * FROM (SELECT * FROM a) AS t1 JOIN b ON x=y WHERE x > 10;
```

The optimizer then flattens the query:

```
=> SELECT * FROM a JOIN b ON x=y WHERE x > 10;
```

Subquery flattening

The following example shows how Vertica transforms FROM clause subqueries within a WHERE clause IN subquery. Given the following query:

```
=> SELECT * FROM a
  WHERE b IN (SELECT b FROM (SELECT * FROM t2)) AS D WHERE x=1;
```

The optimizer flattens it as follows:

```
=> SELECT * FROM a
  WHERE b IN (SELECT b FROM t2) AS D WHERE x=1;
```

See also

Subquery restrictions

Subqueries in UPDATE and DELETE statements

You can nest subqueries within UPDATE and DELETE statements.

UPDATE subqueries

You can update records in one table according to values in others, by nesting a subquery within an UPDATE statement. The example below illustrates this through a couple of noncorrelated subqueries . You can reproduce this example with the following tables:

```
=> CREATE TABLE addresses(cust_id INTEGER, address VARCHAR(2000));
CREATE TABLE
dbadmin=> INSERT INTO addresses VALUES(20,'Lincoln Street'),(30,'Booth Hill Road'),(30,'Beach Avenue'),(40,'Mt. Vernon Street'),(50,'Hillside Avenue');
 OUTPUT
--------
      5
(1 row)

=> CREATE TABLE new_addresses(new_cust_id integer, new_address Boolean DEFAULT 'T');
CREATE TABLE
dbadmin=> INSERT INTO new_addresses VALUES (20),(30),(80);
 OUTPUT
--------
      3
(1 row)

=> INSERT INTO new_addresses VALUES (60,'F');
 OUTPUT
--------
      1

=> COMMIT;
COMMIT
```

Queries on these tables return the following results:

```
=> SELECT * FROM addresses;
 cust_id |      address
---------+--------------------
      20 | Lincoln Street
      30 | Beach Avenue
      30 | Booth Hill Road
      40 | Mt. Vernon Street
      50 | Hillside Avenue
(5 rows)

=> SELECT * FROM new_addresses;
 new_cust_id | new_address
-------------+-------------
          20 | t
          30 | t
          80 | t
          60 | f
(4 rows)
```

1. The following UPDATE statement uses a noncorrelated subquery to join new_addresses and addresses records on customer IDs. UPDATE sets the value 'New Address' in the joined addresses records. The statement output indicates that three rows were updated:

   ```
   => UPDATE addresses SET address='New Address'
     WHERE cust_id IN (SELECT new_cust_id FROM new_addresses WHERE new_address='T');
    OUTPUT
   --------
   3
   (1 row)
   ```

2. Query the addresses table to see the changes for matching customer ID 20 and 30. Addresses for customer ID 40 and 50 are not updated:

```
=> SELECT * FROM addresses;
 cust_id |      address
---------+-------------------
      40 | Mt. Vernon Street
      50 | Hillside Avenue
      20 | New Address
      30 | New Address
      30 | New Address
(5 rows)

=>COMMIT;
COMMIT
```

DELETE subqueries

You can delete records in one table based according to values in others by nesting a subquery within a DELETE statement.

For example, you want to remove records from new_addresses that were used earlier to update records in addresses . The following DELETE statement uses a noncorrelated subquery to join new_addresses and addresses records on customer IDs. It then deletes the joined records from table new_addresses :

```
=> DELETE FROM new_addresses
   WHERE new_cust_id IN (SELECT cust_id FROM addresses WHERE address='New Address');
 OUTPUT
--------
      2
(1 row)
=> COMMIT;
COMMIT
```

Querying new_addresses confirms that the records were deleted:

```
=> SELECT * FROM new_addresses;
 new_cust_id | new_address
-------------+-------------
          60 | f
          80 | t
(2 rows)
```

Subquery examples

This topic illustrates some of the subqueries you can write. The examples use the VMart example database.

Single-row subqueries

Single-row subqueries are used with single-row comparison operators (=, >=, <=, <>, and <=>) and return exactly one row.

For example, the following query retrieves the name and hire date of the oldest employee in the Vmart database:

```
=> SELECT employee_key, employee_first_name, employee_last_name, hire_date
   FROM employee_dimension
   WHERE hire_date = (SELECT MIN(hire_date) FROM employee_dimension);
 employee_key | employee_first_name | employee_last_name | hire_date
--------------+---------------------+--------------------+------------
         2292 | Mary                | Bauer              | 1956-01-11
(1 row)
```

Multiple-row subqueries

Multiple-row subqueries return multiple records.

For example, the following IN clause subquery returns the names of the employees making the highest salary in each of the six regions:

```
=> SELECT employee_first_name, employee_last_name, annual_salary, employee_region
    FROM employee_dimension WHERE annual_salary IN
    (SELECT MAX(annual_salary) FROM employee_dimension GROUP BY employee_region)
  ORDER BY annual_salary DESC;
 employee_first_name | employee_last_name | annual_salary |  employee_region
---------------------+--------------------+---------------+-------------------
 Alexandra           | Sanchez            |        992363 | West
 Mark                | Vogel              |        983634 | South
 Tiffany             | Vu                 |        977716 | SouthWest
 Barbara             | Lewis              |        957949 | MidWest
 Sally               | Gauthier           |        927335 | East
 Wendy               | Nielson            |        777037 | NorthWest
(6 rows)
```

Multicolumn subqueries

Multicolumn subqueries return one or more columns. Sometimes a subquery's result set is evaluated in the containing query in column-to-column and row-to-row comparisons.

> **Note**
> Multicolumn subqueries can use the <>, !=, and = operators but not the <, >, <=, >= operators.

You can substitute some multicolumn subqueries with a join, with the reverse being true as well. For example, the following two queries ask for the sales transactions of all products sold online to customers located in Massachusetts and return the same result set. The only difference is the first query is written as a join and the second is written as a subquery.

Join query:

```
=> SELECT *
   FROM online_sales.online_sales_fact
   INNER JOIN public.customer_dimension
   USING (customer_key)
   WHERE customer_state = 'MA';
```

Subquery:

```
=> SELECT *
  FROM online_sales.online_sales_fact
  WHERE customer_key IN
    (SELECT customer_key
     FROM public.customer_dimension
     WHERE customer_state = 'MA');
```

The following query returns all employees in each region whose salary is above the average:

```
=> SELECT e.employee_first_name, e.employee_last_name, e.annual_salary,
     e.employee_region, s.average
   FROM employee_dimension e,
     (SELECT employee_region, AVG(annual_salary) AS average
      FROM employee_dimension GROUP BY employee_region) AS s
   WHERE  e.employee_region = s.employee_region AND e.annual_salary > s.average
   ORDER BY annual_salary DESC;
 employee_first_name | employee_last_name | annual_salary | employee_region |     average
---------------------+--------------------+---------------+-----------------+-----------------
 Doug                | Overstreet         |        995533 | East            |  61192.786013986
 Matt                | Gauthier           |        988807 | South           | 57337.8638902996
 Lauren              | Nguyen             |        968625 | West            | 56848.4274914089
 Jack                | Campbell           |        963914 | West            | 56848.4274914089
 William             | Martin             |        943477 | NorthWest       | 58928.2276119403
 Luigi               | Campbell           |        939255 | MidWest         | 59614.9170454545
 Sarah               | Brown              |        901619 | South           | 57337.8638902996
 Craig               | Goldberg           |        895836 | East            |  61192.786013986
 Sam                 | Vu                 |        889841 | MidWest         | 59614.9170454545
 Luigi               | Sanchez            |        885078 | MidWest         | 59614.9170454545
 Michael             | Weaver             |        882685 | South           | 57337.8638902996
 Doug                | Pavlov             |        881443 | SouthWest       | 57187.2510548523
 Ruth                | McNulty            |        874897 | East            |  61192.786013986
 Luigi               | Dobisz             |        868213 | West            | 56848.4274914089
 Laura               | Lang               |        865829 | East            |  61192.786013986
 ...
```

You can also use the EXCEPT , INTERSECT , and UNION [ALL] keywords in FROM, WHERE, and HAVING clauses.

The following subquery returns information about all Connecticut-based customers who bought items through either stores or online sales channel and whose purchases amounted to more than 500 dollars:

```
=> SELECT DISTINCT customer_key, customer_name FROM public.customer_dimension
   WHERE customer_key IN (SELECT customer_key FROM store.store_sales_fact
     WHERE sales_dollar_amount > 500
     UNION ALL
     SELECT customer_key FROM online_sales.online_sales_fact
     WHERE sales_dollar_amount > 500)
   AND customer_state = 'CT';
 customer_key |  customer_name
--------------+------------------
          200 | Carla Y. Kramer
          733 | Mary Z. Vogel
          931 | Lauren X. Roy
         1533 | James C. Vu
         2948 | Infocare
         4909 | Matt Z. Winkler
         5311 | John Z. Goldberg
         5520 | Laura M. Martin
         5623 | Daniel R. Kramer
         6759 | Daniel Q. Nguyen
 ...
```

HAVING clause subqueries

A HAVING clause is used in conjunction with the GROUP BY clause to filter the select-list records that a GROUP BY returns. HAVING clause subqueries must use Boolean comparison operators: =, >, <, <>, <=, >= and take the following form:

```
SELECT <column, ...>
FROM <table>
GROUP BY <expression>
HAVING <expression>
 (SELECT <column, ...>
  FROM <table>
  HAVING <expression>);
```

For example, the following statement uses the VMart database and returns the number of customers who purchased lowfat products. Note that the GROUP BY clause is required because the query uses an aggregate (COUNT).

```
=> SELECT s.product_key, COUNT(s.customer_key) FROM store.store_sales_fact s
  GROUP BY s.product_key HAVING s.product_key IN
    (SELECT product_key FROM product_dimension WHERE diet_type = 'Low Fat');
```

The subquery first returns the product keys for all low-fat products, and the outer query then counts the total number of customers who purchased those products.

```
product_key | count
------------+-------
        15 |     2
        41 |     1
        66 |     1
       106 |     1
       118 |     1
       169 |     1
       181 |     2
       184 |     2
       186 |     2
       211 |     1
       229 |     1
       267 |     1
       289 |     1
       334 |     2
       336 |     1
(15 rows)
```

Subquery restrictions

The following restrictions apply to Vertica subqueries:

- Subqueries are not allowed in the defining query of a CREATE PROJECTION statement.
- Subqueries can be used in the SELECT list, but GROUP BY or aggregate functions are not allowed in the query if the subquery is not part of the GROUP BY clause in the containing query. For example, the following two statement returns an error message:

  ```
  => SELECT y, (SELECT MAX(a) FROM t1) FROM t2 GROUP BY y;
    ERROR:  subqueries in the SELECT or ORDER BY are not supported if the
    subquery is not part of the GROUP BY
  => SELECT MAX(y), (SELECT MAX(a) FROM t1) FROM t2;
    ERROR:  subqueries in the SELECT or ORDER BY are not supported if the
    query has aggregates and the subquery is not part of the GROUP BY
  ```

- Subqueries are supported within UPDATE statements with the following exceptions:
  - You cannot use SET column = { *expression* } to specify a subquery.
  - The table specified in the UPDATE list cannot also appear in the FROM clause (no self joins).
- FROM clause subqueries require an alias but tables do not. If the table has no alias, the query must refer its columns as *table-name* . *column-name* . However, column names that are unique among all tables in the query do not need to be qualified by their table name.
- If the ORDER BY clause is inside a FROM clause subquery, rather than in the containing query, the query is liable to return unexpected sort results. This occurs because Vertica data comes from multiple nodes, so sort order cannot be guaranteed unless the outer query block specifies an ORDER BY clause. This behavior complies with the SQL standard, but it might differ from other databases.
- Multicolumn subqueries cannot use the <, >, <=, >= comparison operators. They can use <>, !=, and = operators.
- WHERE and HAVING clause subqueries must use Boolean comparison operators: =, >, <, <>, <=, >=. Those subqueries can be noncorrelated and correlated.

- [NOT] IN and ANY subqueries nested in another expression are not supported if any of the column values are NULL. In the following statement, for example, if column x from either table t1 or t2 contains a NULL value, Vertica returns a run-time error:

```
=> SELECT * FROM t1 WHERE (x IN (SELECT x FROM t2)) IS FALSE;
ERROR:  NULL value found in a column used by a subquery
```

- Vertica returns an error message during subquery run time on scalar subqueries that return more than one row.
- Aggregates and GROUP BY clauses are allowed in subqueries, as long as those subqueries are not correlated.
- Correlated expressions under ALL and [NOT] IN are not supported.
- Correlated expressions under OR are not supported.
- Multiple correlations are allowed only for subqueries that are joined with an equality (=) predicate. However, IN / NOT IN , EXISTS / NOT EXISTS predicates within correlated subqueries are not allowed:

```
=> SELECT t2.x, t2.y, t2.z FROM t2 WHERE t2.z NOT IN
     (SELECT t1.z FROM t1 WHERE t1.x = t2.x);
 ERROR: Correlated subquery with NOT IN is not supported
```

- Up to one level of correlated subqueries is allowed in the WHERE clause if the subquery references columns in the immediate outer query block. For example, the following query is not supported because the t2.x = t3.x subquery can only refer to table t1 in the outer query, making it a correlated expression because t3.x is two levels out:

```
=> SELECT t3.x, t3.y, t3.z FROM t3 WHERE t3.z IN (
    SELECT t1.z FROM t1 WHERE EXISTS (
      SELECT 'x' FROM t2 WHERE t2.x = t3.x) AND t1.x = t3.x);
ERROR:  More than one level correlated subqueries are not supported
```

  The query is supported if it is rewritten as follows:

```
=> SELECT t3.x, t3.y, t3.z FROM t3 WHERE t3.z IN
     (SELECT t1.z FROM t1 WHERE EXISTS
       (SELECT 'x' FROM t2 WHERE t2.x = t1.x)
  AND t1.x = t3.x);
```

Joins

Queries can combine records from multiple tables, or multiple instances of the same table. A query that combines records from one or more tables is called a join. Joins are allowed in SELECT statements and subqueries.

Supported join types

Vertica supports the following join types:

- Inner (including natural, cross) joins
- Left, right, and full outer joins
- Optimizations for equality and range joins predicates

Vertica does not support nested loop joins.

Join algorithms

Vertica's query optimizer implements joins with either the hash join or merge join algorithm. For details, see Hash joins versus merge joins .

In this section

- Join syntax
- Join conditions vs. filter conditions
- Inner joins
- Outer joins
- Controlling join inputs
- Range joins
- Event series joins

Join syntax

Vertica supports the ANSI SQL-92 standard for joining tables, as follows:

```
table-reference [join-type] JOIN table-reference [ ON join-predicate ]
```

where *join-type* can be one of the following:

- INNER (default)
- LEFT [OUTER]
- RIGHT [OUTER]
- FULL [OUTER]
- NATURAL
- CROSS

For example:

```
=> SELECT * FROM T1 INNER JOIN T2 ON T1.id = T2.id;
```

> **Note**
> The ON *join-predicate* clause is invalid for NATURAL and CROSS joins, required for all other join types.

Alternative syntax options

Vertica also supports two older join syntax conventions:

Join specified by WHERE clause join predicate

INNER JOIN is equivalent to a query that specifies its join predicate in a WHERE clause. For example, this example and the previous one return equivalent results. They both specify an inner join between tables T1 and T2 on columns T1.id and T2.id , respectively.

```
=> SELECT * FROM T1, T2 WHERE T1.id = T2.id;
```

JOIN USING clause

You can join two tables on identically named columns with a JOIN USING clause. For example:

```
=> SELECT * FROM T1 JOIN T2 USING(id);
```

By default, a join that is specified by JOIN USING is always an inner join.

> **Note**
> JOIN USING joins the two tables by combining the two join columns into one. Therefore, the two join column data types must be the same or compatible—for example, FLOAT and INTEGER—regardless of the actual data in the joined columns.

Benefits of SQL-92 join syntax

Vertica recommends that you use SQL-92 join syntax for several reasons:

- SQL-92 outer join syntax is portable across databases; the older syntax was not consistent between databases.
- SQL-92 syntax provides greater control over whether predicates are evaluated during or after outer joins. This was also not consistent between databases when using the older syntax.
- SQL-92 syntax eliminates ambiguity in the order of evaluating the joins, in cases where more than two tables are joined with outer joins.

Join conditions vs. filter conditions

If you do not use the SQL-92 syntax, join conditions (predicates that are evaluated during the join) are difficult to distinguish from filter conditions (predicates that are evaluated after the join), and in some cases cannot be expressed at all. With SQL-92, join conditions and filter conditions are separated into two different clauses, the ON clause and the WHERE clause, respectively, making queries easier to understand.

- **The ON clause** contains relational operators (for example, <, <=, >, >=, <>, =, <=>) or other predicates that specify which records from the left and right input relations to combine, such as by matching foreign keys to primary keys. ON can be used with inner, left outer, right outer, and full outer joins. Cross joins and union joins do not use an ON clause.
  Inner joins return all pairings of rows from the left and right relations for which the ON clause evaluates to TRUE. In a left join, all rows from the left relation in the join are present in the result; any row of the left relation that does not match any rows in the right relation is still present in the result but with nulls in any columns taken from the right relation. Similarly, a right join preserves all rows from the right relation, and a full join retains all rows from both relations.
- **The WHERE clause** is evaluated after the join is performed. It filters records returned by the FROM clause, eliminating any records that do not satisfy the WHERE clause condition.

Vertica automatically converts outer joins to inner joins in cases where it is correct to do so, allowing the optimizer to choose among a wider set of query plans and leading to better performance.

Inner joins

An inner join combines records from two tables based on a join predicate and requires that each record in the first table has a matching record in the second table. Thus, inner joins return only records from both joined tables that satisfy the join condition. Records that contain no matches are excluded from the result set.

Inner joins take the following form:

```
SELECT column-list FROM left-join-table
  [INNER] JOIN right-join-table ON join-predicate
```

If you omit the INNER keyword, Vertica assumes an inner join. Inner joins are commutative and associative. You can specify tables in any order without changing the results.

Example

The following example specifies an inner join between tables store.store_dimension and public.employee_dimension whose records have matching values in columns store_region and employee_region , respectively:

```
=> SELECT s.store_region, SUM(e.vacation_days) TotalVacationDays
   FROM public.employee_dimension e
   JOIN store.store_dimension s ON s.store_region=e.employee_region
   GROUP BY s.store_region ORDER BY TotalVacationDays;
```

This join can also be expressed as follows:

```
=> SELECT s.store_region, SUM(e.vacation_days) TotalVacationDays
    FROM public.employee_dimension e, store.store_dimension s
    WHERE s.store_region=e.employee_region
    GROUP BY s.store_region ORDER BY TotalVacationDays;
```

Both queries return the same result set:

```
 store_region | TotalVacationDays
--------------+-------------------
 NorthWest    |             23280
 SouthWest    |            367250
 MidWest      |            925938
 South        |           1280468
 East         |           1952854
 West         |           2849976
(6 rows)
```

If the join's inner table store.store_dimension has any rows with store_region values that do not match employee_region values in table public.employee_dimension , those rows are excluded from the result set. To include that row, you can specify an outer join .

In this section

- Equi-joins and non equi-joins
- Natural joins
- Cross joins

Equi-joins and non equi-joins

Vertica supports any arbitrary join expression with both matching and non-matching column values. For example:

```
SELECT * FROM fact JOIN dim ON fact.x = dim.x;
SELECT * FROM fact JOIN dim ON fact.x > dim.y;
SELECT * FROM fact JOIN dim ON fact.x <= dim.y;
SELECT * FROM fact JOIN dim ON fact.x <> dim.y;
SELECT * FROM fact JOIN dim ON fact.x <=> dim.y;
```

Note

> Operators = and <=> generally run the fastest.

Equi-joins are based on equality (matching column values). This equality is indicated with an equal sign (=), which functions as the comparison operator in the ON clause using SQL-92 syntax or the WHERE clause using older join syntax.

The first example below uses SQL-92 syntax and the ON clause to join the online sales table with the call center table using the call center key; the query then returns the sale date key that equals the value 156:

```
=> SELECT sale_date_key, cc_open_date FROM online_sales.online_sales_fact
   INNER JOIN online_sales.call_center_dimension
   ON (online_sales.online_sales_fact.call_center_key =
    online_sales.call_center_dimension.call_center_key
   AND sale_date_key = 156);
 sale_date_key | cc_open_date
---------------+--------------
           156 | 2005-08-12
(1 row)
```

The second example uses older join syntax and the WHERE clause to join the same tables to get the same results:

```
=> SELECT sale_date_key, cc_open_date
    FROM online_sales.online_sales_fact, online_sales.call_center_dimension
   WHERE online_sales.online_sales_fact.call_center_key =
     online_sales.call_center_dimension.call_center_key
   AND sale_date_key = 156;
 sale_date_key | cc_open_date
---------------+--------------
           156 | 2005-08-12
(1 row)
```

Vertica also permits tables with compound (multiple-column) primary and foreign keys. For example, to create a pair of tables with multi-column keys:

```
=> CREATE TABLE dimension(pk1 INTEGER NOT NULL, pk2 INTEGER NOT NULL);=> ALTER TABLE dimension ADD PRIMARY KEY (pk1, pk2);
=> CREATE TABLE fact (fk1 INTEGER NOT NULL, fk2 INTEGER NOT NULL);
=> ALTER TABLE fact ADD FOREIGN KEY (fk1, fk2) REFERENCES dimension (pk1, pk2);
```

To join tables using compound keys, you must connect two join predicates with a Boolean AND operator. For example:

```
=> SELECT * FROM fact f JOIN dimension d ON f.fk1 = d.pk1 AND f.fk2 = d.pk2;
```

You can write queries with expressions that contain the <=> operator for NULL=NULL joins.

```
=> SELECT * FROM fact JOIN dim ON fact.x <=> dim.y;
```

The <=> operator performs an equality comparison like the = operator, but it returns true, instead of NULL , if both operands are NULL , and false, instead of NULL , if one operand is NULL .

```
=> SELECT 1 <=> 1, NULL <=> NULL, 1 <=> NULL;
 ?column? | ?column? | ?column?
----------+----------+----------
 t        | t        | f
(1 row)
```

Compare the <=> operator to the = operator:

```
=> SELECT 1 = 1, NULL = NULL, 1 = NULL;
 ?column? | ?column? | ?column?
----------+----------+----------
 t        |          |
(1 row)
```

> **Note**

> Writing NULL=NULL joins on primary key/foreign key combinations is not an optimal choice because PK/FK columns are usually defined as NOT NULL .

When composing joins, it helps to know in advance which columns contain null values. An employee's hire date, for example, would not be a good choice because it is unlikely hire date would be omitted. An hourly rate column, however, might work if some employees are paid hourly and some are salaried. If you are unsure about the value of columns in a given table and want to check, type the command:

```
=> SELECT COUNT(*) FROM tablename WHERE columnname IS NULL;
```

Natural joins

A natural join is just a join with an implicit join predicate. Natural joins can be inner, left outer, right outer, or full outer joins and take the following form:

```
SELECT column-list FROM left-join-table
NATURAL [ INNER | LEFT OUTER | RIGHT OUTER | FULL OUTER ] JOIN right-join-table
```

Natural joins are, by default, natural inner joins; however, there can also be natural left/right/full outer joins. The primary difference between an inner and natural join is that inner joins have an explicit join condition, whereas the natural join's conditions are formed by matching all pairs of columns in the tables that have the same name and compatible data types, making natural joins equi-joins because join condition are equal between common columns. (If the data types are incompatible, Vertica returns an error.)

> **Note**
> The Data type coercion chart lists the data types that can be cast to other data types. If one data type can be cast to the other, those two data types are compatible.

The following query is a simple natural join between tables T1 and T2 when the T2 column val is greater than 5:

```
=> SELECT * FROM T1 NATURAL JOIN T2 WHERE T2.val > 5;
```

The store_sales_fact table and the product_dimension table have two columns that share the same name and data type: product_key and product_version . The following example creates a natural join between those two tables at their shared columns:

```
=> SELECT product_description, sales_quantity FROM store.store_sales_fact
  NATURAL JOIN public.product_dimension;
```

The following three queries return the same result expressed as a basic query, an inner join, and a natural join. at the table expressions are equivalent only if the common attribute in the store_sales_fact table and the store_dimension table is store_key . If both tables have a column named store_key , then the natural join would also have a store_sales_fact.store_key = store_dimension.store_key join condition. Since the results are the same in all three instances, they are shown in the first (basic) query only:

```
=> SELECT store_name FROM store.store_sales_fact, store.store_dimension
  WHERE store.store_sales_fact.store_key = store.store_dimension.store_key
  AND store.store_dimension.store_state = 'MA' ORDER BY store_name;
 store_name
------------
 Store11
 Store128
 Store178
 Store66
 Store8
 Store90
(6 rows)
```

The query written as an inner join:

```
=> SELECT store_name FROM store.store_sales_fact
  INNER JOIN store.store_dimension
  ON (store.store_sales_fact.store_key = store.store_dimension.store_key)
  WHERE store.store_dimension.store_state = 'MA' ORDER BY store_name;
```

In the case of the natural join, the join predicate appears implicitly by comparing all of the columns in both tables that are joined by the same column name. The result set contains only one column representing the pair of equally-named columns.

```
=> SELECT store_name FROM store.store_sales_fact
  NATURAL JOIN store.store_dimension
  WHERE store.store_dimension.store_state = 'MA' ORDER BY store_name;
```

Cross joins

Cross joins are the simplest joins to write, but they are not usually the fastest to run because they consist of all possible combinations of two tables' records. Cross joins contain no join condition and return what is known as a Cartesian product, where the number of rows in the result set is equal to the number of rows in the first table multiplied by the number of rows in the second table.

The following query returns all possible combinations from the promotion table and the store sales table:

```
=> SELECT * FROM promotion_dimension CROSS JOIN store.store_sales_fact;
```

Because this example returns over 600 million records, many cross join results can be extremely large and difficult to manage. Cross joins can be useful, however, such as when you want to return a single-row result set.

> **Tip**
> Filter out unwanted records in a cross with WHERE clause join predicates:
>
> ```
> => SELECT * FROM promotion_dimension p   CROSS JOIN store.store_sales_fact f
>   WHERE p.promotion_key LIKE f.promotion_key;
> ```

Implicit versus explicit joins

Vertica recommends that you do not write implicit cross joins (comma-separated tables in the FROM clause). These queries can imply accidental omission of a join predicate.

The following query implicitly cross joins tables promotion_dimension **and** store.store_sales_fact :

```
=> SELECT * FROM promotion_dimension, store.store_sales_fact;
```

It is better practice to express this cross join explicitly, as follows:

```
=> SELECT * FROM promotion_dimension CROSS JOIN store.store_sales_fact;
```

Examples

The following example creates two small tables and their superprojections and then runs a cross join on the tables:

```
=> CREATE TABLE employee(employee_id INT, employee_fname VARCHAR(50));
=> CREATE TABLE department(dept_id INT, dept_name VARCHAR(50));
=> INSERT INTO employee VALUES (1, 'Andrew');
=> INSERT INTO employee VALUES (2, 'Priya');
=> INSERT INTO employee VALUES (3, 'Michelle');
=> INSERT INTO department VALUES (1, 'Engineering');
=> INSERT INTO department VALUES (2, 'QA');
=> SELECT * FROM employee CROSS JOIN department;
```

In the result set, the cross join retrieves records from the first table and then creates a new row for every row in the 2nd table. It then does the same for the next record in the first table, and so on.

```
 employee_id | employee_name | dept_id | dept_name
 ------------+---------------+---------+-----------
           1 | Andrew        |       1 |  Engineering
           2 | Priya         |       1 |  Engineering
           3 | Michelle      |       1 |  Engineering
           1 | Andrew        |       2 |  QA
           2 | Priya         |       2 |  QA
           3 | Michelle      |       2 |  QA
(6 rows)
```

Outer joins

Outer joins extend the functionality of inner joins by letting you preserve rows of one or both tables that do not have matching rows in the non-preserved table. Outer joins take the following form:

```
SELECT column-list FROM left-join-table
[ LEFT | RIGHT | FULL ] OUTER JOIN right-join-table ON join-predicate
```

> **Note**
> Omitting the keyword OUTER from your statements does not affect results of left and right joins. LEFT OUTER JOIN and LEFT JOIN perform the same operation and return the same results.

Left outer joins

A left outer join returns a complete set of records from the left-joined (preserved) table T1 , with matched records, where available, in the right-joined (non-preserved) table T2 . Where Vertica finds no match, it extends the right side column (T2) with null values.

```
=> SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.x = T2.x;
```

To exclude the non-matched values from T2, write the same left outer join, but filter out the records you don't want from the right side by using a WHERE clause:

```
=> SELECT * FROM T1 LEFT OUTER JOIN T2
  ON T1.x = T2.x WHERE T2.x IS NOT NULL;
```

The following example uses a left outer join to enrich telephone call detail records with an incomplete numbers dimension. It then filters out results that are known not to be from Massachusetts:

```
=> SELECT COUNT(*) FROM calls LEFT OUTER JOIN numbers
  ON calls.to_phone = numbers.phone WHERE NVL(numbers.state, '') <> 'MA';
```

Right outer joins

A right outer join returns a complete set of records from the right-joined (preserved) table, as well as matched values from the left-joined (non-preserved) table. If Vertica finds no matching records from the left-joined table (T1), NULL values appears in the T1 column for any records with no matching values in T1 . A right join is, therefore, similar to a left join, except that the treatment of the tables is reversed.

```
=> SELECT * FROM T1 RIGHT OUTER JOIN T2 ON T1.x = T2.x;
```

The above query is equivalent to the following query, where T1 RIGHT OUTER JOIN T2 = T2 LEFT OUTER JOIN T1 .

```
=> SELECT * FROM T2 LEFT OUTER JOIN T1 ON T2.x = T1.x;
```

The following example identifies customers who have *not* placed an order:

```
=> SELECT customers.customer_id FROM orders RIGHT OUTER JOIN customers
  ON orders.customer_id = customers.customer_id
  GROUP BY customers.customer_id HAVING COUNT(orders.customer_id) = 0;
```

Full outer joins

A full outer join returns results for both left and right outer joins. The joined table contains all records from both tables, including nulls (missing matches) from either side of the join. This is useful if you want to see, for example, each employee who is assigned to a particular department and each department that has an employee, but you also want to see all the employees who are not assigned to a particular department, as well as any department that has no employees:

```
=> SELECT employee_last_name, hire_date FROM  employee_dimension emp
  FULL OUTER JOIN department dept ON emp.employee_key = dept.department_key;
```

Notes

Vertica also supports joins where the outer (preserved) table or subquery is replicated on more than one node and the inner (non-preserved) table or subquery is segmented across more than one node. For example, in the following query, the fact table, which is almost always segmented, appears on the non-preserved side of the join, and it is allowed:

```
=> SELECT sales_dollar_amount, transaction_type, customer_name
    FROM store.store_sales_fact f RIGHT JOIN customer_dimension d
   ON f.customer_key = d.customer_key;
 sales_dollar_amount | transaction_type | customer_name
---------------------+------------------+---------------
                 252 | purchase         | Inistar
                 363 | purchase         | Inistar
                 510 | purchase         | Inistar
                -276 | return           | Foodcorp
                 252 | purchase         | Foodcorp
                 195 | purchase         | Foodcorp
                 290 | purchase         | Foodcorp
                 222 | purchase         | Foodcorp
                     |                  | Foodgen
                     |                  | Goldcare
(10 rows)
```

Controlling join inputs

By default, the optimizer uses its own internal logic to determine whether to join one table to another as an inner or outer input. Occasionally, the optimizer might choose the larger table as the inner input to a join. Doing so can incur performance and concurrency issues.

If the configuration parameter configuration parameter EnableForceOuter is set to 1, you can control join inputs for specific tables through ALTER TABLE..FORCE OUTER . The FORCE OUTER option modifies a table's force_outer setting in the system table TABLES . When implementing a join, Vertica compares the force_outer settings of the participating tables:

- If table settings are unequal, Vertica uses them to set the join inputs:
  - A table with a low force_outer setting relative to other tables is joined to them as an inner input.
  - A table with a high force_outer setting relative to other tables is joined to them as an outer input.
- If all table settings are equal, Vertica ignores them and assembles the join on its own.

The force_outer column is initially set to 5 for all newly-defined tables. You can use ALTER TABLE..FORCE OUTER to reset force_outer to a value equal to or greater than 0. For example, you might change the force_outer settings of tables abc and xyz to 3 and 8, respectively:

```
=> ALTER TABLE abc FORCE OUTER 3;
=> ALTER TABLE xyz FORCE OUTER 8;
```

Given these settings, the optimizer joins abc as the inner input to any table with a force_outer value greater than 3. The optimizer joins xyz as the outer input to any table with a force_outer value less than 8.

Projection inheritance

When you query a projection directly, it inherits the force_outer setting of its anchor table. The query then uses this setting when joined to another projection.

Enabling forced join inputs

The configuration parameter EnableForceOuter determines whether Vertica uses a table's force_outer value to implement a join. By default, this parameter is set to 0, and forced join inputs are disabled. You can enable forced join inputs at session and database scopes, through ALTER SESSION and ALTER DATABASE , respectively:

```
=> ALTER SESSION SET EnableForceOuter = { 0 | 1 };
=> ALTER DATABASE db-name SET EnableForceOuter =  { 0 | 1 };
```

If EnableForceOuter is set to 0, ALTER TABLE..FORCE OUTER statements return with this warning:

> **Note**
>
> ```
> WARNING 0: Set configuration parameter EnableForceOuter for the current session or the database in order to use
> force_outer value
> ```

Viewing forced join inputs

EXPLAIN -generated query plans indicate whether the configuration parameter EnableForceOuter is on. A join query might include tables whose force_outer settings are less or greater than the default value of 5. In this case, the query plan includes a Force outer level field for the relevant join inputs.

For example, the following query joins tables store.store_sales and public.products , where both tables have the same force_outer setting (5). EnableForceOuter is on, as indicated in the generated query plan:

```
=> EXPLAIN SELECT s.store_key, p.product_description, s.sales_quantity, s.sale_date
  FROM store.store_sales s JOIN public.products p ON s.product_key=p.product_key
  WHERE s.sale_date='2014-12-01' ORDER BY s.store_key, s.sale_date;

 EnableForceOuter is on
 Access Path:
 +-SORT [Cost: 7K, Rows: 100K (NO STATISTICS)] (PATH ID: 1)
 |  Order: sales.store_key ASC, sales.sale_date ASC
 |  Execute on: All Nodes
 | +---> JOIN HASH [Cost: 5K, Rows: 100K (NO STATISTICS)] (PATH ID: 2) Outer (BROADCAST)(LOCAL ROUND ROBIN)
 | |      Join Cond: (sales.product_key = products.product_key)
 | |      Execute on: All Nodes
 | | +-- Outer -> STORAGE ACCESS for sales [Cost: 2K, Rows: 100K (NO STATISTICS)] (PATH ID: 3)
 | | |      Projection: store.store_sales_b0
 | | |      Materialize: sales.sale_date, sales.store_key, sales.product_key, sales.sales_quantity
 | | |      Filter: (sales.sale_date = '2014-12-01'::date)
 | | |      Execute on: All Nodes
 | | +-- Inner -> STORAGE ACCESS for products [Cost: 177, Rows: 60K (NO STATISTICS)] (PATH ID: 4)
 | | |      Projection: public.products_b0
 | | |      Materialize: products.product_key, products.product_description
 | | |      Execute on: All Nodes
```

The following ALTER TABLE statement resets the force_outer setting of public.products to 1:

```
=> ALTER TABLE public.products FORCE OUTER 1;
ALTER TABLE
```

The regenerated query plan for the same join now includes a Force outer level field and specifies public.products as the inner input:

```
=> EXPLAIN SELECT s.store_key, p.product_description, s.sales_quantity, s.sale_date
  FROM store.store_sales s JOIN public.products p ON s.product_key=p.product_key
  WHERE s.sale_date='2014-12-01' ORDER BY s.store_key, s.sale_date;

 EnableForceOuter is on
 Access Path:
 +-SORT [Cost: 7K, Rows: 100K (NO STATISTICS)] (PATH ID: 1)
 |  Order: sales.store_key ASC, sales.sale_date ASC
 |  Execute on: All Nodes
 | +---> JOIN HASH [Cost: 5K, Rows: 100K (NO STATISTICS)] (PATH ID: 2) Outer (BROADCAST)(LOCAL ROUND ROBIN)
 | |      Join Cond: (sales.product_key = products.product_key)
 | |      Execute on: All Nodes
 | | +-- Outer -> STORAGE ACCESS for sales [Cost: 2K, Rows: 100K (NO STATISTICS)] (PATH ID: 3)
 | | |      Projection: store.store_sales_b0
 | | |      Materialize: sales.sale_date, sales.store_key, sales.product_key, sales.sales_quantity
 | | |      Filter: (sales.sale_date = '2014-12-01'::date)
 | | |      Execute on: All Nodes
 | | +-- Inner -> STORAGE ACCESS for products [Cost: 177, Rows: 60K (NO STATISTICS)] (PATH ID: 4)
 | | |      Projection: public.products_b0
 | | |      Force outer level: 1
 | | |      Materialize: products.product_key, products.product_description
 | | |      Execute on: All Nodes
```

If you change the force_outer setting of public.products to 8, Vertica creates a different query plan that specifies public.products as the outer input:

```
=> ALTER TABLE public.products FORCE OUTER 8;
ALTER TABLE

=> EXPLAIN SELECT s.store_key, p.product_description, s.sales_quantity, s.sale_date
   FROM store.store_sales s JOIN public.products p ON s.product_key=p.product_key
   WHERE s.sale_date='2014-12-01' ORDER BY s.store_key, s.sale_date;


EnableForceOuter is on
 Access Path:
 +-SORT [Cost: 7K, Rows: 100K (NO STATISTICS)] (PATH ID: 1)
 |  Order: sales.store_key ASC, sales.sale_date ASC
 |  Execute on: All Nodes
 | +---> JOIN HASH [Cost: 5K, Rows: 100K (NO STATISTICS)] (PATH ID: 2) Inner (BROADCAST)
 | |      Join Cond: (sales.product_key = products.product_key)
 | |      Materialize at Output: products.product_description
 | |      Execute on: All Nodes
 | | +-- Outer -> STORAGE ACCESS for products [Cost: 20, Rows: 60K (NO STATISTICS)] (PATH ID: 3)
 | | |      Projection: public.products_b0
 | | |      Force outer level: 8
 | | |      Materialize: products.product_key
 | | |      Execute on: All Nodes
 | | |      Runtime Filter: (SIP1(HashJoin): products.product_key)
 | | +-- Inner -> STORAGE ACCESS for sales [Cost: 2K, Rows: 100K (NO STATISTICS)] (PATH ID: 4)
 | | |      Projection: store.store_sales_b0
 | | |      Materialize: sales.sale_date, sales.store_key, sales.product_key, sales.sales_quantity
 | | |      Filter: (sales.sale_date = '2014-12-01'::date)
 | | |      Execute on: All Nodes
```

Restrictions

Vertica ignores force_outer settings when it performs the following operations:

- Outer joins: Vertica generally respects OUTER JOIN clauses regardless of the force_outer settings of the joined tables.
- MERGE statement joins.
- Queries that include the SYNTACTIC_JOIN hint.
- Half-join queries such as SEMI JOIN .
- Joins to subqueries, where the subquery is always processed as having a force_outer setting of 5 regardless of the force_outer settings of the tables that are joined in the subquery. This setting determines a subquery's designation as inner or outer input relative to other join inputs. If two subqueries are joined, the optimizer determines which one is the inner input, and which one the outer.

Range joins

Vertica provides performance optimizations for <, <=, >, >=, and BETWEEN predicates in join ON clauses. These optimizations are particularly useful when a column from one table is restricted to be in a range specified by two columns of another table.

Key ranges

Multiple, consecutive key values can map to the same dimension values. Consider, for example, a table of IPv4 addresses and their owners. Because large subnets (ranges) of IP addresses can belong to the same owner, this dimension can be represented as:

```
=> CREATE TABLE ip_owners(
    ip_start INTEGER,
    ip_end INTEGER,
    owner_id INTEGER);
=> CREATE TABLE clicks(
    ip_owners INTEGER,
    dest_ip INTEGER);
```

A query that associates a click stream with its destination can use a join similar to the following, which takes advantage of range optimization:

```
=> SELECT owner_id, COUNT(*) FROM clicks JOIN ip_owners
  ON clicks.dest_ip BETWEEN ip_start AND ip_end
  GROUP BY owner_id;
```

Requirements

Operators <, <=, >, >=, or BETWEEN must appear as top-level conjunctive predicates for range join optimization to be effective, as shown in the following examples:

`BETWEEN` as the only predicate:

```
=> SELECT COUNT(*) FROM fact JOIN dim
   ON fact.point BETWEEN dim.start AND dim.end;
```

Comparison operators as top-level predicates (within `AND`):

```
=> SELECT COUNT(*) FROM fact JOIN dim
   ON fact.point > dim.start AND fact.point < dim.end;
```

`BETWEEN` as a top-level predicate (within `AND`):

```
=> SELECT COUNT(*) FROM fact JOIN dim
  ON (fact.point BETWEEN dim.start AND dim.end) AND fact.c <> dim.c;
```

Query not optimized because `OR` is top-level predicate (disjunctive):

```
=> SELECT COUNT(*) FROM fact JOIN dim
  ON (fact.point BETWEEN dim.start AND dim.end) OR dim.end IS NULL;
```

Notes

- Expressions are optimized in range join queries in many cases.
- If range columns can have NULL values indicating that they are open-ended, it is possible to use range join optimizations by replacing nulls with very large or very small values:

  ```
  => SELECT COUNT(*) FROM fact JOIN dim
    ON fact.point BETWEEN NVL(dim.start, -1) AND NVL(dim.end, 1000000000000);
  ```

- If there is more than one set of ranging predicates in the same ON clause, the order in which the predicates are specified might impact the effectiveness of the optimization:

  ```
  => SELECT COUNT(*) FROM fact JOIN dim ON fact.point1 BETWEEN dim.start1 AND dim.end1
    AND fact.point2 BETWEEN dim.start2 AND dim.end2;
  ```

  The optimizer chooses the first range to optimize, so write your queries so that the range you most want optimized appears first in the statement.
- The use of the range join optimization is not directly affected by any characteristics of the physical schema; no schema tuning is required to benefit from the optimization.
- The range join optimization can be applied to joins without any other predicates, and to HASH or MERGE joins.
- To determine if an optimization is in use, search for RANGE in the EXPLAIN plan.

Event series joins

An event series join is a Vertica SQL extension that enables the analysis of two series when their measurement intervals don't align precisely, such as with mismatched timestamps. You can compare values from the two series directly, rather than having to normalize the series to the same measurement interval.

Event series joins are an extension of Outer joins , but instead of padding the non-preserved side with NULL values when there is no match, the event series join pads the non-preserved side values that it interpolates from either the previous or next value, whichever is specified in the query.

The difference in how you write a regular join versus an event series join is the INTERPOLATE predicate, which is used in the ON clause. For example, the following two statements show the differences, which are shown in greater detail in Writing event series joins .

Examples

Regular full outer join

```
SELECT * FROM hTicks h FULL OUTER JOIN aTicks a
ON (h.time = a.time);
```

Event series join

```
SELECT * FROM hTicks h FULL OUTER JOIN aTicks a
ON (h.time INTERPOLATE PREVIOUS VALUE a.time);
```

Similar to regular joins, an event series join has inner and outer join modes, which are described in the topics that follow.

For full syntax, including notes and restrictions, see INTERPOLATE

In this section

- Sample schema for event series joins examples
- Writing event series joins

Sample schema for event series joins examples

If you don't plan to run the queries and just want to look at the examples, you can skip this topic and move straight to Writing event series joins .

Schema of hTicks and aTicks tables

The examples that follow use the following hTicks and aTicks tables schemas:

```
CREATE TABLE hTicks (
  stock VARCHAR(20),
  time TIME,
  price NUMERIC(8,2)
);
CREATE TABLE aTicks (
  stock VARCHAR(20),
  time TIME,
  price NUMERIC(8,2)
);
```

Although TIMESTAMP is more commonly used for the event series column, the examples in this topic use TIME to keep the output simple.

```
INSERT INTO hTicks VALUES ('HPQ', '12:00', 50.00);
INSERT INTO hTicks VALUES ('HPQ', '12:01', 51.00);
INSERT INTO hTicks VALUES ('HPQ', '12:05', 51.00);
INSERT INTO hTicks VALUES ('HPQ', '12:06', 52.00);
INSERT INTO aTicks VALUES ('ACME', '12:00', 340.00);
INSERT INTO aTicks VALUES ('ACME', '12:03', 340.10);
INSERT INTO aTicks VALUES ('ACME', '12:05', 340.20);
INSERT INTO aTicks VALUES ('ACME', '12:05', 333.80);
COMMIT;
```

Output of the two tables:

hTicks		**aTicks**

=> SELECT * FROM hTicks;

There are no entry records between 12:02–12:04:

```
 stock |   time   | price
-------+----------+-------
 HPQ   | 12:00:00 | 50.00
 HPQ   | 12:01:00 | 51.00
 HPQ   | 12:05:00 | 51.00
 HPQ   | 12:06:00 | 52.00
(4 rows)
```

=> SELECT * FROM aTicks;

There are no entry records at 12:01, 12:02 and at 12:04:

```
stock |   time   | price
-------+----------+--------
 ACME  | 12:00:00 | 340.00
 ACME  | 12:03:00 | 340.10
 ACME  | 12:05:00 | 340.20
 ACME  | 12:05:00 | 333.80
(4 rows)
```

Example query showing gaps

A full outer join shows the gaps in the timestamps:

```
=> SELECT * FROM hTicks h FULL OUTER JOIN aTicks a ON h.time = a.time;
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 |       |          |
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
 HPQ   | 12:06:00 | 52.00 |       |          |
       |          |       | ACME  | 12:03:00 | 340.10
(6 rows)
```

Schema of bid and asks tables

The examples that follow use the following hTicks and aTicks tables.

```
CREATE TABLE bid(stock VARCHAR(20), time TIME, price NUMERIC(8,2));
CREATE TABLE ask(stock VARCHAR(20), time TIME, price NUMERIC(8,2));
INSERT INTO bid VALUES ('HPQ', '12:00', 100.10);
INSERT INTO bid VALUES ('HPQ', '12:01', 100.00);
INSERT INTO bid VALUES ('ACME', '12:00', 80.00);
INSERT INTO bid VALUES ('ACME', '12:03', 79.80);
INSERT INTO bid VALUES ('ACME', '12:05', 79.90);
INSERT INTO ask VALUES ('HPQ', '12:01', 101.00);
INSERT INTO ask VALUES ('ACME', '12:00', 80.00);
INSERT INTO ask VALUES ('ACME', '12:02', 75.00);
COMMIT;
```

Output of the two tables:

bid	**ask**

=> SELECT * FROM bid;

There are no entry records for stocks HPQ and ACME at 12:02 and at 12:04:

```
 stock |   time   | price
-------+----------+--------
 HPQ   | 12:00:00 | 100.10
 HPQ   | 12:01:00 | 100.00
 ACME  | 12:00:00 |  80.00
 ACME  | 12:03:00 |  79.80
 ACME  | 12:05:00 |  79.90
(5 rows)
```

=> SELECT * FROM ask;

There are no entry records for stock HPQ at 12:00 and none for ACMEat 12:01:

```
 stock |   time   | price
-------+----------+--------
 HPQ   | 12:01:00 | 101.00
 ACME  | 12:00:00 |  80.00
 ACME  | 12:02:00 |  75.00
(3 rows)
```

Example query showing gaps

A full outer join shows the gaps in the timestamps:

```
=> SELECT * FROM bid b FULL OUTER JOIN ask a ON b.time = a.time;
 stock |   time   | price  | stock |   time   | price
-------+----------+--------+-------+----------+--------
 HPQ   | 12:00:00 | 100.10 | ACME  | 12:00:00 |  80.00
 HPQ   | 12:01:00 | 100.00 | HPQ   | 12:01:00 | 101.00
 ACME  | 12:00:00 |  80.00 | ACME  | 12:00:00 |  80.00
 ACME  | 12:03:00 |  79.80 |       |          |
 ACME  | 12:05:00 |  79.90 |       |          |
       |          |        | ACME  | 12:02:00 |  75.00
(6 rows)
```

Writing event series joins

The examples in this topic contains mismatches between timestamps—just as you'd find in real life situations; for example, there could be a period of inactivity on stocks where no trade occurs, which can present challenges when you want to compare two stocks whose timestamps don't match.

The hTicks and aTicks tables

As described in the example ticks schema , tables, hTicks is missing input rows for 12:02, 12:03, and 12:04, and aTicks is missing inputs at 12:01, 12:02, and 12:04.

hTicks

```
=> SELECT * FROM hTicks;
 stock |   time   | price
-------+----------+-------
 HPQ   | 12:00:00 | 50.00
 HPQ   | 12:01:00 | 51.00
 HPQ   | 12:05:00 | 51.00
 HPQ   | 12:06:00 | 52.00
(4 rows)
```

aTicks

```
=> SELECT * FROM aTicks;
 stock |   time   | price
-------+----------+--------
 ACME  | 12:00:00 | 340.00
 ACME  | 12:03:00 | 340.10
 ACME  | 12:05:00 | 340.20
 ACME  | 12:05:00 | 333.80
(4 rows)
```

Querying event series data with full outer joins

Using a traditional full outer join, this query finds a match between tables hTicks and aTicks at 12:00 and 12:05 and pads the missing data points with NULL values.

```
=> SELECT * FROM hTicks h FULL OUTER JOIN aTicks a ON (h.time = a.time);
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 |       |          |
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
 HPQ   | 12:06:00 | 52.00 |       |          |
       |          |       | ACME  | 12:03:00 | 340.10
(6 rows)
```

To replace the gaps with interpolated values for those missing data points, use the INTERPOLATE predicate to create an event series join. The join condition is restricted to the ON clause, which evaluates the equality predicate on the timestamp columns from the two input tables. In other words, for each row in outer table hTicks, the ON clause predicates are evaluated for each combination of each row in the inner table aTicks.

Simply rewrite the full outer join query to use the INTERPOLATE predicate with either the required PREVIOUS VALUE or NEXT VALUE keywords. Note that a full outer join on event series data is the most common scenario for event series data, where you keep all rows from both tables.

```
=> SELECT * FROM hTicks h FULL OUTER JOIN aTicks a
   ON (h.time INTERPOLATE PREVIOUS VALUE a.time);
```

Vertica interpolates the missing values (which appear as NULL in the full outer join) using that table's previous or next value, whichever is specified. This example shows INTERPOLATE PREVIOUS. Notice how in the second row, blank cells have been filled using values interpolated from the previous row:

```
=> SELECT * FROM hTicks h FULL OUTER JOIN aTicks a ON (h.time = a.time);
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 | ACME  | 12:03:00 | 340.10
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 52.00 | ACME  | 12:05:00 | 340.20
 HPQ   | 12:06:00 | 52.00 | ACME  | 12:05:00 | 340.20
(6 rows)
```

> **Note**
> The output ordering above is different from the regular full outer join because in the event series join, interpolation occurs independently for each stock (hTicks and aTicks), where the data is partitioned and sorted based on the equality predicate. This means that interpolation occurs within, not across, partitions.

If you review the regular full outer join output, you can see that both tables have a match in the time column at 12:00 and 12:05, but at 12:01, there is no entry record for ACME. So the operation interpolates a value for ACME (ACME,12:00,340) based on the previous value in the aTicks table.

Querying event series data with left outer joins

You can also use left and right outer joins. You might, for example, decide you want to preserve only hTicks values. So you'd write a left outer join:

```
=> SELECT * FROM hTicks h LEFT OUTER JOIN aTicks a
   ON (h.time INTERPOLATE PREVIOUS VALUE a.time);
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
 HPQ   | 12:06:00 | 52.00 | ACME  | 12:05:00 | 340.20
(5 rows)
```

Here's what the same data looks like using a traditional left outer join:

```
=> SELECT * FROM hTicks h LEFT OUTER JOIN aTicks a ON h.time = a.time;
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 |       |          |
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
 HPQ   | 12:06:00 | 52.00 |       |          |
(5 rows)
```

Note that a right outer join has the same behavior with the preserved table reversed.

Querying event series data with inner joins

Note that INNER event series joins behave the same way as normal ANSI SQL-99 joins, where all gaps are omitted. Thus, there is nothing to interpolate, and the following two queries are equivalent and return the same result set:

A regular inner join:

```
=> SELECT * FROM HTicks h JOIN aTicks a
   ON (h.time INTERPOLATE PREVIOUS VALUE a.time);
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
(3 rows)
```

An event series inner join:

```
=> SELECT * FROM hTicks h INNER JOIN aTicks a ON (h.time = a.time);
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
(3 rows)
```

The bid and ask tables

Using the example schema for the bid and ask tables, write a full outer join to interpolate the missing data points:

```
=> SELECT * FROM bid b FULL OUTER JOIN ask a
   ON (b.stock = a.stock AND b.time INTERPOLATE PREVIOUS VALUE a.time);
```

In the below output, the first row for stock HPQ shows nulls because there is no entry record for HPQ before 12:01.

```
 stock |   time   | price  | stock |   time   | price
-------+----------+--------+-------+----------+--------
 ACME  | 12:00:00 |  80.00 | ACME  | 12:00:00 |  80.00
 ACME  | 12:00:00 |  80.00 | ACME  | 12:02:00 |  75.00
 ACME  | 12:03:00 |  79.80 | ACME  | 12:02:00 |  75.00
 ACME  | 12:05:00 |  79.90 | ACME  | 12:02:00 |  75.00
 HPQ   | 12:00:00 | 100.10 |       |          |
 HPQ   | 12:01:00 | 100.00 | HPQ   | 12:01:00 | 101.00
(6 rows)
```

Note also that the same row (ACME,12:02,75) from the ask table appears three times. The first appearance is because no matching rows are present in the bid table for the row in ask , so Vertica interpolates the missing value using the ACME value at 12:02 (75.00). The second appearance occurs because the row in bid (ACME,12:05,79.9) has no matches in ask . The row from ask that contains (ACME,12:02,75) is the closest row; thus, it is used to interpolate the values.

If you write a regular full outer join, you can see where the mismatched timestamps occur:

```
=> SELECT * FROM bid b FULL OUTER JOIN ask a ON (b.time = a.time);
 stock |   time   | price  | stock |   time   | price
-------+----------+--------+-------+----------+--------
 ACME  | 12:00:00 |  80.00 | ACME  | 12:00:00 |  80.00
 ACME  | 12:03:00 |  79.80 |       |          |
 ACME  | 12:05:00 |  79.90 |       |          |
 HPQ   | 12:00:00 | 100.10 | ACME  | 12:00:00 |  80.00
 HPQ   | 12:01:00 | 100.00 | HPQ   | 12:01:00 | 101.00
       |          |        | ACME  | 12:02:00 |  75.00
(6 rows)
```

Query optimization

When you submit a query to Vertica for processing, the Vertica query optimizer automatically chooses a set of operations to compute the requested result. These operations together are called a *query plan* . The choice of operations can significantly affect how many resources are needed to compute query results, and overall run-time performance. Optimal performance depends in great part on the projections that are available for a given query.

This section describes the different operations that the optimizer uses and how you can facilitate optimizer performance.

> **Note**
> Database response time depends on many factors. These include type and size of the application query, database design, data size and data types stored, available computational power, and network bandwidth. Adding nodes to a database cluster does not necessarily improve the system response time for every query, especially if response times are already short, or are not hardware bound.

In this section

- Initial process for improving query performance
- Column encoding
- Projections for queries with predicates
- GROUP BY queries
- DISTINCT in a SELECT query list
- JOIN queries
- ORDER BY queries
- Analytic functions
- LIMIT queries
- INSERT-SELECT operations
- DELETE and UPDATE queries
- Data collector table queries

Initial process for improving query performance

To optimize query performance, begin by performing the following tasks:

1. Run Database Designer.
2. Check query events proactively.
3. Review the query plan.

Run Database Designer

Database Designer creates a physical schema for your database that provides optimal query performance. The first time you run Database Designer, you should create a comprehensive design that includes relevant sample queries and data on which to base the design. If you develop performance issues later, consider loading additional queries that you run frequently and then rerunning Database Designer to create an incremental design.

When you run Database Designer, choose the option, Update Statistics. The Vertica query optimizer uses statistics about the data to create a query plan. Statistics help the optimizer determine:

- Multiple eligible projections to answer the query
- The best order in which to perform joins
- Data distribution algorithms, such as broadcast and resegmentation

If your statistics become out of date, run the Vertica function ANALYZE_STATISTICS function to update statistics for a schema, table, or columns. For more information, see Collecting database statistics .

Check query events proactively

The QUERY_EVENTS system table returns information on query planning, optimization, and execution events.

The EVENT_TYPE column provides various event types:

- Some event types are informational .
- Others you should review for possible corrective action .
- Several are most important to address .

Review the query plan

A *query plan* is a sequence of step-like paths that the Vertica query optimizer selects to access or alter information in your Vertica database. There are two ways to get information about the query plan:

- Run the EXPLAIN command. Each step (path) represents a single operation that the optimizer uses for its execution strategy.
- Query the QUERY_PLAN_PROFILES system table. This table provides detailed execution status for currently running queries. Output from the QUERY_PLAN_PROFILES table shows the real-time flow of data and the time and resources consumed for each path in each query plan.

See also

- QUERY_EVENTS

Column encoding

You can potentially make queries faster by changing column encoding. Encoding reduces the on-disk size of your data so the amount of I/O required for queries is reduced, resulting in faster execution times. Make sure all columns and projections included in the query use the correct data encoding. To do this, take the following steps:

1. Run Database Designer to create an incremental design. Database Designer implements the optimum encoding and projection design.
2. After creating the incremental design, update statistics using the ANALYZE_STATISTICS function.
3. Run EXPLAIN with one or more of the queries you submitted to the design to make sure it is using the new projections.

Alternatively, run DESIGNER_DESIGN_PROJECTION_ENCODINGS to re-evaluate the current encoding and update it if necessary.

In this section

- Improving column compression
- Using run length encoding

Improving column compression

If you see slow performance or a large storage footprint with your FLOAT data, evaluate the data and your business needs to see if it can be contained in a NUMERIC column with a precision of 18 digits or less. Converting a FLOAT column to a NUMERIC column can improve data compression, reduce the on-disk size of your database, and improve performance of queries on that column.

When you define a NUMERIC data type, you specify the precision and the scale; NUMERIC data are exact representations of data. FLOAT data types represent variable precision and approximate values; they take up more space in the database.

Converting FLOAT columns to NUMERIC columns is most effective when:

- NUMERIC precision is 18 digits or less. Performance of NUMERIC data is fine-tuned for the common case of 18 digits of precision. Vertica recommends converting FLOAT columns to NUMERIC columns only if they require precision of 18 digits or less.
- FLOAT precision is bounded, and the values will all fall within a specified precision for a NUMERIC column. One example is monetary values like product prices or financial transaction amounts. For example, a column defined as NUMERIC(11,2) can accommodate prices from 0 to a few million dollars and can store cents, and compresses more efficiently than a FLOAT column.

If you try to load a value into a NUMERIC column that exceeds the specified precision, Vertica returns an error and does not load the data. If you assign a value with more decimal digits than the specified scale, the value is rounded to match the specified scale and stored in that column.

See also

Numeric data types

Using run length encoding

When you run Database Designer, you can choose to optimize for loads, which minimizes database footprint. In this case, Database Designer applies encodings to columns to maximize query performance. Encoding options include run length encoding (RLE), which replaces sequences (runs) of identical values in a column with a set of pairs, where each pair represents the number of contiguous occurrences for a given value: (*occurrences* , *value*).

RLE is generally applicable to a column with low-cardinality, and where identical values are contiguous—typically, because table data is sorted on that column. For example, a customer profile table typically includes a gender column that contains values of F and M only. Sorting on gender ensures runs of F or M values that can be expressed as a set of two pairs: (*occurrences* , F) and (*occurrences* , M). So, given 8,147 occurrences of F and 7,956 occurrences of M, and a projection that is sorted primarily on gender, Vertica can apply RLE and store these values as a single set of two pairs: (8147, F) and (7956, M). Doing so reduces this projection's footprint and improves query performance.

Projections for queries with predicates

If your query contains one or more predicates, you can modify the projections to improve the query's performance, as described in the following two examples.

Queries that use date ranges

This example shows how to encode data using RLE and change the projection sort order to improve the performance of a query that retrieves all data within a given date range.

Suppose you have a query that looks like this:

```
=> SELECT * FROM trades
  WHERE trade_date BETWEEN '2016-11-01' AND '2016-12-01';
```

To optimize this query, determine whether all of the projections can perform the SELECT operation in a timely manner. Run SELECT COUNT(*) statement for each projection, specifying the date range, and note the response time. For example:

```
=> SELECT COUNT(*) FROM [ projection_name ]
  WHERE trade_date BETWEEN '2016-11-01' AND '2016-12-01;
```

If one or more of the queries is slow, check the uniqueness of the trade_date column and determine if it needs to be in the projection's ORDER BY clause and/or can be encoded using RLE. RLE replaces sequences of the same data values within a column by a pair that represents the value and a count. For best results, order the columns in the projection from lowest cardinality to highest cardinality, and use RLE to encode the data in low-cardinality columns.

> **Note**
> For an example of using sorting and RLE, see Choosing sort order: best practices .

If the number of unique columns is unsorted, or if the average number of repeated rows is less than 10, trade_date is too close to being unique and cannot be encoded using RLE. In this case, add a new column to minimize the search scope.

The following example adds a new column trade_year :

1. Determine if the new column trade_year returns a manageable result set. The following query returns the data grouped by trade_year :
   ```
   => SELECT DATE_TRUNC('trade_year', trade_date), COUNT(*)
     FROM trades
     GROUP BY DATE_TRUNC('trade_year',trade_date);
   ```
2. Assuming that trade_year = 2007 is near 8k, add a column for trade_year to the trades table. The SELECT statement then becomes:
   ```
   => SELECT * FROM trades
     WHERE trade_year = 2007
     AND trade_date BETWEEN '2016-11-01' AND '2016-12-01';
   ```
   As a result, you have a projection that is sorted on trade_year , which can be encoded using RLE.

Queries for tables with a high-cardinality primary key

This example demonstrates how you can modify the projection to improve the performance of queries that select data from a table with a high-cardinality primary key.

Suppose you have the following query:

```
=> SELECT FROM [table]
  WHERE pk IN (12345, 12346, 12347,...);
```

Because the primary key is a high-cardinality column, Vertica has to search a large amount of data.

To optimize the schema for this query, create a new column named buckets and assign it the value of the primary key divided by 10000. In this example, buckets=(int) pk/10000 . Use the buckets column to limit the search scope as follows:

```
=> SELECT FROM [table]
  WHERE buckets IN (1,...)
  AND pk IN (12345, 12346, 12347,...);
```

Creating a lower cardinality column and adding it to the query limits the search scope and improves the query performance. In addition, if you create a projection where buckets is first in the sort order, the query may run even faster.

GROUP BY queries

The following sections include several examples that show how you can design your projections to optimize the performance of your GROUP BY queries.

In this section

- GROUP BY implementation options
- Avoiding resegmentation during GROUP BY optimization with projection design

GROUP BY implementation options

Vertica implements a query GROUP BY clause with one of these algorithms: GROUPBY PIPELINED or GROUPBY HASH. Both algorithms return the same results. Performance of both is generally similar for queries that return a small number of distinct groups—typically a thousand per node .

You can use EXPLAIN to determine which algorithm the query optimizer chooses for a given query. The following conditions generally determine which algorithm is chosen:

- GROUPBY PIPELINED requires all GROUP BY data to be specified in the projection's ORDER BY clause. For details, see GROUPBY PIPELINED Requirements below.
  Because GROUPBY PIPELINED only needs to retain in memory the current group data, this algorithm generally requires less memory and executes faster than GROUPBY HASH. Performance improvements are especially notable for queries that aggregate large numbers of distinct groups.
- GROUPBY HASH is used for any query that does not comply with GROUPBY PIPELINED sort order requirements. In this case, Vertica must build a hash table on GROUP BY columns before it can start grouping the data.

GROUPBY PIPELINED requirements

You can enable use of the GROUPBY PIPELINED algorithm by ensuring that the query and one of its projections comply with GROUPBY PIPELINED requirements. The following conditions apply to GROUPBY PIPELINED. If none of them is true for the query, then Vertica uses GROUPBY HASH.

All examples that follow assume this schema:

```
CREATE TABLE sortopt (
   a INT NOT NULL,
   b INT NOT NULL,
   c INT,
   d INT
);
CREATE PROJECTION sortopt_p (
  a_proj,
  b_proj,
  c_proj,
  d_proj )
AS SELECT * FROM sortopt
ORDER BY a,b,c
UNSEGMENTED ALL NODES;
INSERT INTO sortopt VALUES(5,2,13,84);
INSERT INTO sortopt VALUES(14,22,8,115);
INSERT INTO sortopt VALUES(79,9,401,33);
```

Condition 1

All GROUP BY columns are also included in the projection ORDER BY clause. For example:

GROUP BY columns	GROUPBY algorithm	Reason chosen

a a,b b,a a,b,c c,a,b	GROUPBY PIPELINED	Columns a , b , and c are included in the projection sort columns.
a,b,c,d	GROUPBY HASH	Column d is not part of the projection sort columns.

Condition 2

If the query's GROUP BY clause has fewer columns than the projection's ORDER BY clause, the GROUP BY columns must:

- Be a subset of ORDER BY columns that are contiguous.
- Include the first ORDER BY column.

For example:

GROUP BY columns	**GROUPBY algorithm**	**Reason chosen**
a a,b b,a	GROUPBY PIPELINED	All GROUP BY columns are a subset of contiguous columns in the projection's ORDER BY clause {a,b,c} , and include column a .
a,c	GROUPBY HASH	GROUP BY columns {a,c} are not contiguous in the projection ORDER BY clause {a,b,c} .
b,c	GROUPBY HASH	GROUP BY columns {b,c } do not include the projection's first ORDER BY column a .

Condition 3

If a query's GROUP BY columns do not appear first in the projection's ORDER BY clause, then any early-appearing projection sort columns that are missing in the query's GROUP BY clause must be present as single-column constant equality predicates in the query's WHERE clause.

For example:

Query	**GROUPBY algorithm**	**Reason chosen**
SELECT SUM(a) FROM sortopt WHERE a = 10 GROUP BY b	GROUPBY PIPELINED	All columns preceding b in the projection sort order appear as constant equality predicates.
SELECT SUM(a) FROM sortopt WHERE a = 10 GROUP BY a, b	GROUPBY PIPELINED	Grouping column a is redundant but has no effect on algorithm selection.
SELECT SUM(a) FROM sortopt WHERE a = 10 GROUP BY b, c	GROUPBY PIPELINED	All columns preceding b and c in the projection sort order appear as constant equality predicates.
SELECT SUM(a) FROM sortopt WHERE a = 10 GROUP BY c, b	GROUPBY PIPELINED	All columns preceding b and c in the projection sort order appear as constant equality predicates.

`SELECT SUM(a) FROM sortopt WHERE a = 10 GROUP BY c`	GROUPBY HASH	All columns preceding c in the projection sort order do not appear as constant equality predicates.

Controlling GROUPBY algorithm choice

It is generally best to allow Vertica to determine which GROUP BY algorithm is best suited for a given query. Occasionally, you might want to use one algorithm over another. In such cases, you can qualify the GROUP BY clause with a GBYTYPE hint:

```
GROUP BY /*+ GBYTYPE( HASH | PIPE ) */
```

For example, given the following query, the query optimizer uses the GROUPBY PIPELINED algorithm:

```
=> EXPLAIN SELECT SUM(a) FROM sortopt GROUP BY a,b;
 ------------------------------
 QUERY PLAN DESCRIPTION:
 ------------------------------

 EXPLAIN SELECT SUM(a) FROM sortopt GROUP BY a,b;

 Access Path:
 +-GROUPBY PIPELINED [Cost: 11, Rows: 3 (NO STATISTICS)] (PATH ID: 1)
 |  Aggregates: sum(sortopt.a)
 |  Group By: sortopt.a, sortopt.b

...
```

You can use the GBYTYPE hint to force the query optimizer to use the GROUPBY HASH algorithm instead:

```
=> EXPLAIN SELECT SUM(a) FROM sortopt GROUP BY /*+GBYTYPE(HASH) */ a,b;
 ------------------------------
 QUERY PLAN DESCRIPTION:
 ------------------------------

 EXPLAIN SELECT SUM(a) FROM sortopt GROUP BY /*+GBYTYPE(HASH) */ a,b;

 Access Path:
 +-GROUPBY HASH (LOCAL RESEGMENT GROUPS) [Cost: 11, Rows: 3 (NO STATISTICS)] (PATH ID: 1)
 |  Aggregates: sum(sortopt.a)
 |  Group By: sortopt.a, sortopt.b

...
```

The GBYTYPE hint can specify a PIPE (GROUPBY PIPELINED algorithm) argument only if the query and one of its projections comply with GROUPBY PIPELINED requirements. Otherwise, Vertica issues a warning and uses GROUPBY HASH.

For example, the following query cannot use the GROUPBY PIPELINED algorithm, as the GROUP BY columns {b,c } do not include the projection's first ORDER BY column a :

```
=> SELECT SUM(a) FROM sortopt GROUP BY /*+GBYTYPE(PIPE) */ b,c;
WARNING 7765:  Cannot apply Group By Pipe algorithm. Proceeding with Group By Hash and hint will be ignored
 SUM
-----
  79
  14
   5
(3 rows)
```

Avoiding resegmentation during GROUP BY optimization with projection design

To compute the correct result of a query that contains a GROUP BY clause, Vertica must ensure that all rows with the same value in the GROUP BY expressions end up at the same node for final computation. If the projection design already guarantees the data is segmented by the GROUP BY columns, no resegmentation is required at run time.

To avoid resegmentation, the GROUP BY clause must contain all the segmentation columns of the projection, but it can also contain other columns.

When your query includes a GROUP BY clause and joins, if the join depends on the results of the GROUP BY , as in the following example, Vertica performs the GROUP BY first:

```
=> EXPLAIN SELECT * FROM (SELECT b from foo GROUP BY b) AS F, foo WHERE foo.a = F.b;
Access Path:
+-JOIN MERGEJOIN(inputs presorted) [Cost: 649, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
|  Join Cond: (foo.a = F.b)
|  Materialize at Output: foo.b
|  Execute on: All Nodes
| +-- Outer -> STORAGE ACCESS for foo [Cost: 202, Rows: 10K (NO STATISTICS)] (PATH ID: 2)
| |      Projection: public.foo_super
| |      Materialize: foo.a
| |      Execute on: All Nodes
| |      Runtime Filter: (SIP1(MergeJoin): foo.a)
| +-- Inner -> SELECT [Cost: 245, Rows: 10K (NO STATISTICS)] (PATH ID: 3)
| |      Execute on: All Nodes
| | +---> GROUPBY HASH (SORT OUTPUT) (GLOBAL RESEGMENT GROUPS) (LOCAL RESEGMENT GROUPS) [Cost: 245, Rows: 10K (NO
STATISTICS)] (PATH ID:
4)
| | |      Group By: foo.b
| | |      Execute on: All Nodes
| | | +---> STORAGE ACCESS for foo [Cost: 202, Rows: 10K (NO STATISTICS)] (PATH ID: 5)
| | | |      Projection: public.foo_super
| | | |      Materialize: foo.b
| | | |      Execute on: All Nodes
```

If the result of the join operation is the input to the GROUP BY clause, Vertica performs the join first, as in the following example. The segmentation of those intermediate results may not be consistent with the GROUP BY clause in your query, resulted in resegmentation at run time.

```
=> EXPLAIN SELECT * FROM foo AS F, foo WHERE foo.a = F.b GROUP BY 1,2,3,4;
Access Path:
+-GROUPBY HASH (LOCAL RESEGMENT GROUPS) [Cost: 869, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
|  Group By: F.a, F.b, foo.a, foo.b
|  Execute on: All Nodes
| +---> JOIN HASH [Cost: 853, Rows: 10K (NO STATISTICS)] (PATH ID: 2) Outer (RESEGMENT)(LOCAL ROUND ROBIN)
| |      Join Cond: (foo.a = F.b)
| |      Execute on: All Nodes
| | +-- Outer -> STORAGE ACCESS for F [Cost: 403, Rows: 10K (NO STATISTICS)] (PUSHED GROUPING) (PATH ID: 3)
| | |      Projection: public.foo_super
| | |      Materialize: F.a, F.b
| | |      Execute on: All Nodes
| | +-- Inner -> STORAGE ACCESS for foo [Cost: 403, Rows: 10K (NO STATISTICS)] (PATH ID: 4)
| | |      Projection: public.foo_super
| | |      Materialize: foo.a, foo.b
| | |      Execute on: All Nodes
```

If your query does not include joins, the GROUP BY clauses are processed using the existing database projections.

Examples

Assume the following projection:

```
CREATE PROJECTION ... SEGMENTED BY HASH(a,b) ALL NODES
```

The following table explains whether or not resegmentation occurs at run time and why.

GROUP BY a	Requires resegmentation at run time. The query does not contain all the projection segmentation columns.

GROUP BY a, b	Does not require resegmentation at run time. The GROUP BY clause contains all the projection segmentation columns.
GROUP BY a, b, c	Does not require resegmentation at run time. The GROUP BY clause contains all the projection segmentation columns.
GROUP BY a+1, b	Requires resegmentation at run time because of the expression on column a .

To determine if resegmentation will occurs during your GROUP BY query, look at the EXPLAIN -generated query plan.

For example, the following plan uses GROUPBY PIPELINED sort optimization and requires resegmentation to perform the GROUP BY calculation:

+-GROUPBY PIPELINED (RESEGMENT GROUPS) [Cost: 194, Rows: 10K (NO STATISTICS)] (PATH ID: 1)

The following plan uses GROUPBY PIPELINED sort optimization, but does not require resegmentation:

+-GROUPBY PIPELINED [Cost: 459, Rows: 10K (NO STATISTICS)] (PATH ID: 1)

DISTINCT in a SELECT query list

This section describes how to optimize queries that have the DISTINCT keyword in their SELECT list. The techniques for optimizing DISTINCT queries are similar to the techniques for optimizing GROUP BY queries because when processing queries that use DISTINCT , the Vertica optimizer rewrites the query as a GROUP BY query.

The following sections below this page describe specific situations:

- Query has no aggregates in SELECT list
- COUNT (DISTINCT) and other DISTINCT aggregates
- Approximate count distinct functions
- Single DISTINCT aggregates
- Multiple DISTINCT aggregates

Examples in these sections use the following table:

```
=> CREATE TABLE table1 (
   a INT,
   b INT,
   c INT
);
```

In this section

- Query has no aggregates in SELECT list
- COUNT (DISTINCT) and other DISTINCT aggregates
- Approximate count distinct functions
- Single DISTINCT aggregates
- Multiple DISTINCT aggregates

Query has no aggregates in SELECT list

If your query has no aggregates in the SELECT list, internally, Vertica treats the query as if it uses GROUP BY instead.

For example, you can rewrite the following query:

```
SELECT DISTINCT a, b, c FROM table1;
```

as:

```
SELECT a, b, c FROM table1 GROUP BY a, b, c;
```

For fastest execution, apply the optimization techniques for GROUP BY queries described in GROUP BY queries .

COUNT (DISTINCT) and other DISTINCT aggregates

Computing a DISTINCT aggregate generally requires more work than other aggregates. Also, a query that uses a single DISTINCT aggregate consumes fewer resources than a query with multiple DISTINCT aggregates.

> **Tip**
> Vertica executes queries with multiple distinct aggregates more efficiently when all distinct aggregate columns have a similar number of distinct values.

Examples

The following query returns the number of distinct values in a column:

```
=> SELECT COUNT (DISTINCT date_key) FROM date_dimension;

 COUNT
-------
  1826
(1 row)
```

This example returns the number of distinct return values from an expression:

```
=> SELECT COUNT (DISTINCT date_key + product_key) FROM inventory_fact;

 COUNT
-------
 21560
(1 row)
```

You can create an equivalent query using the LIMIT keyword to restrict the number of rows returned:

```
=> SELECT COUNT(date_key + product_key) FROM inventory_fact GROUP BY date_key LIMIT 10;

 COUNT
-------
   173
    31
   321
   113
   286
    84
   244
   238
   145
   202
(10 rows)
```

The following query uses GROUP BY to count distinct values within groups:

```
=> SELECT product_key, COUNT (DISTINCT date_key) FROM inventory_fact
   GROUP BY product_key LIMIT 10;

 product_key | count
-------------+-------
           1 |    12
           2 |    18
           3 |    13
           4 |    17
           5 |    11
           6 |    14
           7 |    13
           8 |    17
           9 |    15
          10 |    12
(10 rows)
```

The following query returns the number of distinct products and the total inventory within each date key:

```
=> SELECT date_key, COUNT (DISTINCT product_key), SUM(qty_in_stock) FROM inventory_fact
   GROUP BY date_key LIMIT 10;

 date_key | count |  sum
----------+-------+--------
        1 |   173 |  88953
        2 |    31 |  16315
        3 |   318 | 156003
        4 |   113 |  53341
        5 |   285 | 148380
        6 |    84 |  42421
        7 |   241 | 119315
        8 |   238 | 122380
        9 |   142 |  70151
       10 |   202 |  95274
(10 rows)
```

This query selects each distinct product_key value and then counts the number of distinct date_key values for all records with the specific product_key value. It also counts the number of distinct warehouse_key values in all records with the specific product_key value:

```
=> SELECT product_key, COUNT (DISTINCT date_key), COUNT (DISTINCT warehouse_key) FROM inventory_fact
   GROUP BY product_key LIMIT 15;

 product_key | count | count
-------------+-------+-------
           1 |    12 |    12
           2 |    18 |    18
           3 |    13 |    12
           4 |    17 |    18
           5 |    11 |     9
           6 |    14 |    13
           7 |    13 |    13
           8 |    17 |    15
           9 |    15 |    14
          10 |    12 |    12
          11 |    11 |    11
          12 |    13 |    12
          13 |     9 |     7
          14 |    13 |    13
          15 |    18 |    17
(15 rows)
```

This query selects each distinct product_key value, counts the number of distinct date_key and warehouse_key values for all records with the specific product_key value, and then sums all qty_in_stock values in records with the specific product_key value. It then returns the number of product_version values in records with the specific product_key value:

```
=> SELECT product_key, COUNT (DISTINCT date_key),
     COUNT (DISTINCT warehouse_key),
     SUM (qty_in_stock),
     COUNT (product_version)
     FROM inventory_fact GROUP BY product_key LIMIT 15;

 product_key | count | count |  sum   | count
-------------+-------+-------+-------+-------
           1 |    12 |    12 |  5530 |    12
           2 |    18 |    18 |  9605 |    18
           3 |    13 |    12 |  8404 |    13
           4 |    17 |    18 | 10006 |    18
           5 |    11 |     9 |  4794 |    11
           6 |    14 |    13 |  7359 |    14
           7 |    13 |    13 |  7828 |    13
           8 |    17 |    15 |  9074 |    17
           9 |    15 |    14 |  7032 |    15
          10 |    12 |    12 |  5359 |    12
          11 |    11 |    11 |  6049 |    11
          12 |    13 |    12 |  6075 |    13
          13 |     9 |     7 |  3470 |     9
          14 |    13 |    13 |  5125 |    13
          15 |    18 |    17 |  9277 |    18
(15 rows)
```

Approximate count distinct functions

The aggregate function COUNT(DISTINCT) computes the exact number of distinct values in a data set. COUNT(DISTINCT) performs well when it executes with the GROUPBY PIPELINED algorithm.

An aggregate COUNT operation performs well on a data set when the following conditions are true:

- One of the target table's projections has an ORDER BY clause that facilitates sorted aggregation.
- The number of distinct values is fairly small.
- Hashed aggregation is required to execute the query.

Alternatively, consider using the APPROXIMATE_COUNT_DISTINCT function instead of COUNT(DISTINCT) when the following conditions are true:

- You have a large data set and you do not require an exact count of distinct values.
- The performance of COUNT(DISTINCT) on a given data set is insufficient.
- You calculate several distinct counts in the same query.
- The plan for COUNT(DISTINCT) uses hashed aggregation.

The expected value that APPROXIMATE_COUNT_DISTINCT returns is equal to COUNT(DISTINCT), with an error that is lognormally distributed with standard deviation *s* . You can control the standard deviation by setting the function's optional error tolerance argument—by default, 1.25 percent.

Other APPROXIMATE_COUNT_DISTINCT functions

Vertica supports two other functions that you can use together, instead of APPROXIMATE_COUNT_DISTINCT: APPROXIMATE_COUNT_DISTINCT_SYNOPSIS and APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS . Use these functions when the following conditions are true:

- You have a large data set and you don't require an exact count of distinct values.
- The performance of COUNT(DISTINCT) on a given data set is insufficient.
- You want to pre-compute the distinct counts and later combine them in different ways.

Use the two functions together as follows:

1. Pass APPROXIMATE_COUNT_DISTINCT_SYNOPSIS the data set and a normally distributed confidence interval. The function returns a subset of the data, as a binary *synopsis* object*.*
2. Pass the synopsis to the APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS function, which then performs an approximate count distinct on the synopsis.

You also use APPROXIMATE_COUNT_DISTINCT_SYNOPSIS_MERGE , which merges multiple synopses into one synopsis. With this function, you can continually update a "master" synopsis by merging in one or more synopses that cover more recent, shorter periods of time.

Example

The following example shows how to use APPROXIMATE_COUNT_DISTINCT functions to keep an approximate running count of users who click on a given web page within a given time span.

1. Create the pviews table to store data about website visits—time of visit, web page visited, and visitor:

```
=> CREATE TABLE pviews(
visit_time TIMESTAMP NOT NULL,
page_id INTEGER NOT NULL,
user_id INTEGER NOT NULL)
ORDER BY page_id, visit_time
SEGMENTED BY HASH(user_id) ALL NODES KSAFE
PARTITION BY visit_time::DATE GROUP BY CALENDAR_HIERARCHY_DAY(visit_time::DATE, 2, 2);
```

   pviews is segmented by hashing user_id data, so all visits by a given user are stored on the same segment, on the same node. This prevents inefficient cross-node transfer of data, when later we do a COUNT (DISTINCT user_id).
   The table also uses hierarchical partitioning on time of visit to optimize the ROS storage. Doing so improves performance when filtering data by time.
2. Load data into pviews :

```
=> INSERT INTO pviews VALUES
    ('2022-02-01 10:00:02',1002,1),
    ('2022-02-01 10:00:03',1002,2),
    ('2022-02-01 10:00:04',1002,1),
    ('2022-02-01 10:00:05',1002,3),
    ('2022-02-01 10:00:01',1000,1),
    ('2022-02-01 10:00:06',1002,1),
    ('2022-02-01 10:00:07',1002,3),
    ('2022-02-01 10:00:08',1002,1),
    ('2022-02-01 10:00:09',1002,3),
    ('2022-02-01 10:00:12',1002,2),
    ('2022-02-02 10:00:01',1000,1),
    ('2022-02-02 10:00:02',1002,4),
    ('2022-02-02 10:00:03',1002,2),
    ('2022-02-02 10:00:04',1002,1),
    ('2022-02-02 10:00:05',1002,3),
    ('2022-02-02 10:00:06',1002,4),
    ('2022-02-02 10:00:07',1002,3),
    ('2022-02-02 10:00:08',1002,4),
    ('2022-02-02 10:00:09',1002,3),
    ('2022-02-02 10:00:12',1002,2),
    ('2022-03-02 10:00:01',1000,1),
    ('2022-03-02 10:00:02',1002,1),
    ('2022-03-02 10:00:03',1002,2),
    ('2022-03-02 10:00:04',1002,1),
    ('2022-03-02 10:00:05',1002,3),
    ('2022-03-02 10:00:06',1002,4),
    ('2022-03-02 10:00:07',1002,3),
    ('2022-03-02 10:00:08',1002,6),
    ('2022-03-02 10:00:09',1002,5),
    ('2022-03-02 10:00:12',1002,2),
    ('2022-03-02 11:00:01',1000,5),
    ('2022-03-02 11:00:02',1002,6),
    ('2022-03-02 11:00:03',1002,7),
    ('2022-03-02 11:00:04',1002,4),
    ('2022-03-02 11:00:05',1002,1),
    ('2022-03-02 11:00:06',1002,6),
    ('2022-03-02 11:00:07',1002,8),
    ('2022-03-02 11:00:08',1002,6),
    ('2022-03-02 11:00:09',1002,7),
    ('2022-03-02 11:00:12',1002,1),
    ('2022-03-03 10:00:01',1000,1),
    ('2022-03-03 10:00:02',1002,2),
    ('2022-03-03 10:00:03',1002,4),
    ('2022-03-03 10:00:04',1002,1),
    ('2022-03-03 10:00:05',1002,2),
    ('2022-03-03 10:00:06',1002,6),
    ('2022-03-03 10:00:07',1002,9),
    ('2022-03-03 10:00:08',1002,10),
    ('2022-03-03 10:00:09',1002,7),
    ('2022-03-03 10:00:12',1002,1);
 OUTPUT
--------
    50
(1 row)

=> COMMIT;
COMMIT
```

3. Create the pview_summary table by querying pviews with CREATE TABLE...AS SELECT . Each row of this table summarizes data selected from pviews for a given date:

- partial_visit_count stores the number of rows (website visits) in pviews with that date.
- daily_users_acdp uses APPROXIMATE_COUNT_DISTINCT_SYNOPSIS to construct a synopsis that approximates the number of distinct users (user_id) who visited that website on that date.

```
=> CREATE TABLE pview_summary AS SELECT
     visit_time::DATE "date",
     COUNT(*) partial_visit_count,
     APPROXIMATE_COUNT_DISTINCT_SYNOPSIS(user_id) AS daily_users_acdp
   FROM pviews GROUP BY 1;
CREATE TABLE
=> ALTER TABLE pview_summary ALTER COLUMN "date" SET NOT NULL;
```

4. Update the pview_summary table so it is partitioned like pviews . The REORGANIZE keyword forces immediate repartitioning of the table data:

```
=> ALTER TABLE pview_summary
     PARTITION BY "date"
     GROUP BY CALENDAR_HIERARCHY_DAY("date", 2, 2) REORGANIZE;
vsql:/home/ale/acd_ex4.sql:93: NOTICE 8364:  The new partitioning scheme will produce partitions in 2 physical storage containers per projection
vsql:/home/ale/acd_ex4.sql:93: NOTICE 4785:  Started background repartition table task
ALTER TABLE
```

5. Use CREATE TABLE..LIKE to create two ETL tables, pviews_etl and pview_summary_etl with the same DDL as pviews and pview_summary , respectively. These tables serve to process incoming data:

```
=> CREATE TABLE pviews_etl LIKE pviews INCLUDING PROJECTIONS;
CREATE TABLE
=> CREATE TABLE pview_summary_etl LIKE pview_summary INCLUDING PROJECTIONS;
CREATE TABLE
```

6. Load new data into pviews_etl :

```
=> INSERT INTO pviews_etl VALUES
     ('2022-03-03 11:00:01',1000,8),
     ('2022-03-03 11:00:02',1002,9),
     ('2022-03-03 11:00:03',1002,1),
     ('2022-03-03 11:00:04',1002,11),
     ('2022-03-03 11:00:05',1002,10),
     ('2022-03-03 11:00:06',1002,12),
     ('2022-03-03 11:00:07',1002,3),
     ('2022-03-03 11:00:08',1002,10),
     ('2022-03-03 11:00:09',1002,1),
     ('2022-03-03 11:00:12',1002,1);
 OUTPUT
--------
     10
(1 row)

=> COMMIT;
COMMIT
```

7. Summarize the new data in pview_summary_etl :

```
=> INSERT INTO pview_summary_etl SELECT
     visit_time::DATE visit_date,
     COUNT(*) partial_visit_count,
     APPROXIMATE_COUNT_DISTINCT_SYNOPSIS(user_id) AS daily_users_acdp
    FROM pviews_etl GROUP BY visit_date;
 OUTPUT
--------
     1
(1 row)
```

8. Append the pviews_etl data to pviews with COPY_PARTITIONS_TO_TABLE :

```
=> SELECT COPY_PARTITIONS_TO_TABLE('pviews_etl', '01-01-0000'::DATE, '01-01-9999'::DATE, 'pviews');
            COPY_PARTITIONS_TO_TABLE
-----------------------------------------------------
 1 distinct partition values copied at epoch 1403.

(1 row)

=> SELECT COPY_PARTITIONS_TO_TABLE('pview_summary_etl', '01-01-0000'::DATE, '01-01-9999'::DATE, 'pview_summary');
            COPY_PARTITIONS_TO_TABLE
-----------------------------------------------------
 1 distinct partition values copied at epoch 1404.

(1 row)
```

9. Create views and distinct (approximate) views by day for all data, including the partition that was just copied from pviews_etl :

```
=> SELECT
    "date" visit_date,
    SUM(partial_visit_count) visit_count,
    APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS(daily_users_acdp) AS daily_users_acd
  FROM pview_summary GROUP BY visit_date ORDER BY visit_date;
 visit_date | visit_count | daily_users_acd
------------+-------------+-----------------
 2022-02-01 |          10 |               3
 2022-02-02 |          10 |               4
 2022-03-02 |          20 |               8
 2022-03-03 |          20 |              11
(4 rows)
```

10. Create views and distinct (approximate) views by month:

```
=> SELECT
    DATE_TRUNC('MONTH', "date")::DATE "month",
    SUM(partial_visit_count) visit_count,
    APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS(daily_users_acdp) AS monthly_users_acd
  FROM pview_summary GROUP BY month ORDER BY month;
  month     | visit_count | monthly_users_acd
------------+-------------+-------------------
 2022-02-01 |          20 |                 4
 2022-03-01 |          40 |                12
(2 rows)
```

11. Merge daily synopses into monthly synopses:

```
=> CREATE TABLE pview_monthly_summary AS SELECT
    DATE_TRUNC('MONTH', "date")::DATE "month",
    SUM(partial_visit_count) partial_visit_count,
    APPROXIMATE_COUNT_DISTINCT_SYNOPSIS_MERGE(daily_users_acdp) AS monthly_users_acdp
  FROM pview_summary GROUP BY month ORDER BY month;
CREATE TABLE
```

12. Create views and distinct views by month, generated from the merged synopses:

```
=> SELECT
    month,
    SUM(partial_visit_count) monthly_visit_count,
    APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS(monthly_users_acdp) AS monthly_users_acd
  FROM pview_monthly_summary GROUP BY month ORDER BY month;
  month     | monthly_visit_count | monthly_users_acd
------------+---------------------+-------------------
 2019-02-01 |                  20 |                 4
 2019-03-01 |                  40 |                12
(2 rows)
```

13. You can use the monthly summary to produce a yearly summary. This approach is likely to be faster than using a daily summary if a lot of data needs to be processed:

```
=> SELECT
    DATE_TRUNC('YEAR', "month")::DATE "year",
    SUM(partial_visit_count) yearly_visit_count,
    APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS(monthly_users_acdp) AS yearly_users_acd
  FROM pview_monthly_summary GROUP BY year ORDER BY year;
   year    | yearly_visit_count | yearly_users_acd
------------+--------------------+------------------
 2022-01-01 |                 60 |               12
(1 row)
```

14. Drop the ETL tables:

```
=> DROP TABLE IF EXISTS pviews_etl, pview_summary_etl;
DROP TABLE
```

See also

- APPROXIMATE_COUNT_DISTINCT
- APPROXIMATE_COUNT_DISTINCT_SYNOPSIS
- APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS
- APPROXIMATE_COUNT_DISTINCT_SYNOPSIS_MERGE
- COUNT [aggregate]

Single DISTINCT aggregates

Vertica computes a DISTINCT aggregate by first removing all duplicate values of the aggregate's argument to find the distinct values. Then it computes the aggregate.

For example, you can rewrite the following query:

```
SELECT a, b, COUNT(DISTINCT c) AS dcnt FROM table1 GROUP BY a, b;
```

as:

```
SELECT a, b, COUNT(dcnt) FROM
  (SELECT a, b, c AS dcnt FROM table1 GROUP BY a, b, c)
GROUP BY a, b;
```

For fastest execution, apply the optimization techniques for GROUP BY queries.

Multiple DISTINCT aggregates

If your query has multiple DISTINCT aggregates, there is no straightforward SQL rewrite that can compute them. The following query cannot easily be rewritten for improved performance:

```
SELECT a, COUNT(DISTINCT b), COUNT(DISTINCT c) AS dcnt FROM table1 GROUP BY a;
```

For a query with multiple DISTINCT aggregates, there is no projection design that can avoid using GROUPBY HASH and resegmenting the data. To improve performance of this query, make sure that it has large amounts of memory available. For more information about memory allocation for queries, see Resource manager .

JOIN queries

In general, you can optimize execution of queries that join multiple tables in several ways:

- Create projections for the joined tables that are sorted on join predicate columns . This facilitates use of the merge join algorithm, which generally joins tables more efficiently than the hash join algorithm.
- Create projections that are identically segmented on the join keys .

Other best practices

Vertica also executes joins more efficiently if the following conditions are true:

- Query construction enables the query optimizer to create a plan where the larger table is defined as the outer input.
- The columns on each side of the equality predicate are from the same table. For example in the following query, the left and right sides of the equality predicate include only columns from tables T and X, respectively:

```
=> SELECT * FROM T JOIN X ON T.a + T.b = X.x1 - X.x2;
```

Conversely, the following query incurs more work to process, because the right side of the predicate includes columns from both tables T and X:

```
=> SELECT * FROM T JOIN X WHERE T.a = X.x1 + T.b
```

In this section

- Hash joins versus merge joins
- Identical segmentation
- Joining variable length string data

Hash joins versus merge joins

The Vertica optimizer implements a join with one of the following algorithms:

- **Merge join** is used when projections of the joined tables are sorted on the join columns. Merge joins are faster and uses less memory than hash joins.
- **Hash join** is used when projections of the joined tables are not already sorted on the join columns. In this case, the optimizer builds an in-memory hash table on the inner table's join column. The optimizer then scans the outer table for matches to the hash table, and joins data from the two tables accordingly. The cost of performing a hash join is low if the entire hash table can fit in memory. Cost rises significantly if the hash table must be written to disk.

The optimizer automatically chooses the most appropriate algorithm to execute a query, given the projections that are available.

Facilitating merge joins

To facilitate a merge join, create projections for the joined tables that are sorted on the join predicate columns. The join predicate columns should be the first columns in the ORDER BY clause.

For example, tables first and second are defined as follows, with projections first_p1 and second_p1 , respectively. The projections are sorted on data_first and data_second :

```
CREATE TABLE first ( id INT, data_first INT );
CREATE PROJECTION first_p1 AS SELECT * FROM first ORDER BY data_first;

CREATE TABLE second ( id INT, data_second INT );
CREATE PROJECTION second_p1 AS SELECT * FROM second ORDER BY data_second;
```

When you join these tables on unsorted columns first.id and second.id , Vertica uses the hash join algorithm:

```
EXPLAIN SELECT first.data_first, second.data_second FROM first JOIN second ON first.id = second.id;

Access Path:
+-JOIN HASH [Cost: 752, Rows: 300K] (PATH ID: 1) Inner (BROADCAST)
```

You can facilitate execution of this query with the merge join algorithm by creating projections first_p 2 and second_p2 , which are sorted on join columns first_p2.id and second_p2.id , respectively:

```
CREATE PROJECTION first_p2 AS SELECT id, data_first FROM first ORDER BY id SEGMENTED BY hash(id, data_first) ALL NODES;
CREATE PROJECTION second_p2 AS SELECT id, data_second FROM second ORDER BY id SEGMENTED BY hash(id, data_second) ALL NODES;
```

If the query joins significant amounts of data, the query optimizer uses the merge algorithm:

```
EXPLAIN SELECT first.data_first, second.data_second FROM first JOIN second ON first.id = second.id;

Access Path:
+-JOIN MERGEJOIN(inputs presorted) [Cost: 731, Rows: 300K] (PATH ID: 1) Inner (BROADCAST)
```

You can also facilitate a merge join by using subqueries to pre-sort the join predicate columns. For example:

```
SELECT first.id, first.data_first, second.data_second FROM
 (SELECT * FROM first ORDER BY id ) first JOIN (SELECT * FROM second ORDER BY id) second ON first.id = second.id;
```

Identical segmentation

To improve query performance when you join multiple tables, create projections that are identically segmented on the join keys. Identically-segmented projections allow the joins to occur locally on each node, thereby helping to reduce data movement across the network during query processing.

To determine if projections are identically-segmented on the query join keys, create a query plan with EXPLAIN . If the query plan contains RESEGMENT or BROADCAST , the projections are not identically segmented.

The Vertica optimizer chooses a projection to supply rows for each table in a query. If the projections to be joined are segmented, the optimizer evaluates their segmentation against the query join expressions. It thereby determines whether the rows are placed on each node so it can join them without fetching data from another node.

Join conditions for identically segmented projections

A projection p is segmented on join columns if all column references in p 's segmentation expression are a subset of the columns in the join expression.

The following conditions must be true for two segmented projections p1 of table t1 and p2 of table t2 to participate in a join of t1 to t2 :

- The join condition must have the following form:
  ```
  t1.j1 = t2.j1 AND t1.j2 = t2.j2 AND ... t1.jN = t2.jN
  ```
- The join columns must share the same base data type. For example:
  - If t1.j1 is an INTEGER, t2.j1 can be an INTEGER but it cannot be a FLOAT.
  - If t1.j1 is a CHAR(10), t2.j1 can be any CHAR or VARCHAR (for example, CHAR(10), VARCHAR(10), VARCHAR(20)), but t2.j1 cannot be an INTEGER.
- If p1 is segmented by an expression on columns { t1.s1, t1.s2, ... t1.sN }, each segmentation column t1.sX must be in the join column set { t1.jX }.
- If p2 is segmented by an expression on columns { t2.s1, t2.s2, ... t2.sN }, each segmentation column t2.sX must be in the join column set { t2.jX }.
- The segmentation expressions of p1 and p2 must be structurally equivalent. For example:
  - If p1 is SEGMENTED BY hash(t1.x) and p2 is SEGMENTED BY hash(t2.x) , p1 and p2 are identically segmented.
  - If p1 is SEGMENTED BY hash(t1.x) and p2 is SEGMENTED BY hash(t2.x + 1) , p1 and p2 are not identically segmented.
- p1 and p2 must have the same segment count.
- The assignment of segments to nodes must match. For example, if p1 and p2 use an OFFSET clause, their offsets must match.
- If Vertica finds projections for t1 and t2 that are not identically segmented, the data is redistributed across the network during query run time, as necessary.

> **Tip**
> If you create custom designs, try to use segmented projections for joins whenever possible.

Examples

The following statements create two tables and specify to create identical segments:

```
=> CREATE TABLE t1 (id INT, x1 INT, y1 INT) SEGMENTED BY HASH(id, x1) ALL NODES;
=> CREATE TABLE t2 (id INT, x1 INT, y1 INT) SEGMENTED BY HASH(id, x1) ALL NODES;
```

Given this design, the join conditions in the following queries can leverage identical segmentation:

```
=> SELECT * FROM t1 JOIN t2 ON t1.id = t2.id;
=> SELECT * FROM t1 JOIN t2 ON t1.id = t2.id AND t1.x1 = t2.x1;
```

Conversely, the join conditions in the following queries require resegmentation:

```
=> SELECT * FROM t1 JOIN t2 ON t1.x1 = t2.x1;
=> SELECT * FROM t1 JOIN t2 ON t1.id = t2.x1;
```

See also

- Partitioning and segmentation
- CREATE PROJECTION

Joining variable length string data

When you join tables on VARCHAR columns, Vertica calculates how much storage space it requires to buffer join column data. It does so by formatting the column data in one of two ways:

- Uses the join column metadata to size column data to a fixed length and buffer accordingly. For example, given a column that is defined as VARCHAR(1000) , Vertica always buffers 1000 characters.
- Uses the actual length of join column data, so buffer size varies for each join. For example, given a join on strings Xi, John, and Amrita, Vertica buffers only as much storage as it needs for each join—in this case, 2, 4, and 6 bytes, respectively.

The second approach can improve join query performance. It can also reduce memory consumption, which helps prevent join spills and minimize how often memory is borrowed from the resource manager. In general, these benefits are especially marked in cases where the defined size of a join column significantly exceeds the average length of its data.

Setting and verifying variable length formatting

You can control how Vertica implements joins at the session or database levels, through configuration parameter JoinDefaultTupleFormat , or for individual queries, through the JFMT hint. Vertica supports variable length formatting for all joins except merge and event series joins.

Use EXPLAIN VERBOSE to verify whether a given query uses variable character formatting, by checking for these flags:

- JF_EE_VARIABLE_FORMAT
- JF_EE_FIXED_FORMAT

ORDER BY queries

You can improve the performance of queries that contain only ORDER BY clauses if the columns in a projection's ORDER BY clause are the same as the columns in the query.

If you define the projection sort order in the CREATE PROJECTION statement, the Vertica query optimizer does not have to sort projection data before performing certain ORDER BY queries.

The following table, sortopt , contains the columns a , b , c , and d . Projection sortopt_p specifies to order on columns a , b , and c .

```
CREATE TABLE sortopt (
   a INT NOT NULL,
   b INT NOT NULL,
   c INT,
   d INT
);
CREATE PROJECTION sortopt_p (
  a_proj,
  b_proj,
  c_proj,
  d_proj )
AS SELECT * FROM sortopt
ORDER BY a,b,c
UNSEGMENTED ALL NODES;
INSERT INTO sortopt VALUES(5,2,13,84);
INSERT INTO sortopt VALUES(14,22,8,115);
INSERT INTO sortopt VALUES(79,9,401,33);
```

Based on this sort order, if a SELECT * FROM sortopt query contains one of the following ORDER BY clauses, the query does not have to resort the projection:

- ORDER BY a
- ORDER BY a, b
- ORDER BY a, b, c

For example, Vertica does not have to resort the projection in the following query because the sort order includes columns specified in the CREATE PROJECTION..ORDER BY a, b, c clause, which mirrors the query's ORDER BY a, b, c clause:

```
=> SELECT * FROM sortopt ORDER BY a, b, c;
 a  | b  |  c  |  d
----+----+-----+-----
  5 |  2 |  13 |  84
 14 | 22 |   8 | 115
 79 |  9 | 401 |  33
(3 rows)
```

If you include column d in the query, Vertica must re-sort the projection data because column d was not defined in the CREATE PROJECTION..ORDER BY clause. Therefore, the ORDER BY d query won't benefit from any sort optimization.

You cannot specify an ASC or DESC clause in the CREATE PROJECTION statement's ORDER BY clause. Vertica always uses an ascending sort order in physical storage, so if your query specifies descending order for any of its columns, the query still causes Vertica to re-sort the projection data. For example, the following query requires Vertica to sort the results:

```
=> SELECT * FROM sortopt ORDER BY a DESC, b, c;
 a  | b  |  c  |  d
----+----+-----+-----
 79 |  9 | 401 |  33
 14 | 22 |   8 | 115
  5 |  2 |  13 |  84
(3 rows)
```

See also

CREATE PROJECTION

Analytic functions

The following sections describe how to optimize SQL-99 analytic functions that Vertica supports.

In this section

- Empty OVER clauses
- NULL sort order
- Runtime sorting of NULL values in analytic functions

Empty OVER clauses

The OVER() clause does not require a windowing clause. If your query uses an analytic function like SUM(x) and you specify an empty OVER() clause, the analytic function is used as a reporting function, where the entire input is treated as a single partition; the aggregate returns the same aggregated value for each row of the result set. The query executes on a single node, potentially resulting in poor performance.

If you add a PARTITION BY clause to the OVER() clause, the query executes on multiple nodes, improving its performance.

NULL sort order

By default, projection column values are stored in ascending order, but placement of NULL values depends on a column's data type.

NULL placement differences with ORDER BY clauses

The analytic OVER(*window-order-clause*) and the SQL ORDER BY clause have slightly different semantics:

OVER(ORDER BY ...)

The analytic window order clause uses the ASC or DESC sort order to determine NULLS FIRST or NULLS LAST placement for analytic function results. NULL values are placed as follows:

- ASC , NULLS LAST — NULL values appear at the end of the sorted result.
- DESC , NULLS FIRST — NULL values appear at the beginning of the sorted result.

(SQL) ORDER BY

The SQL and Vertica ORDER BY clauses produce different results. The SQL ORDER BY clause specifies only ascending or descending sort order. The Vertica ORDER BY clause determines NULL placement based on the column data type:

- NUMERIC, INTEGER, DATE, TIME, TIMESTAMP, and INTERVAL columns: NULLS FIRST (NULL values appear at the beginning of a sorted projection.)
- FLOAT, STRING, and BOOLEAN columns: NULLS LAST (NULL values appear at the end of a sorted projection.)

NULL sort options

If you do not care about NULL placement in queries that involve analytic computations, or if you know that columns do not contain any NULL values, specify NULLS AUTO —irrespective of data type. Vertica chooses the placement that gives the fastest performance, as in the following query. Otherwise, specify NULLS FIRST or NULLS LAST .

```
=> SELECT x, RANK() OVER (ORDER BY x NULLS AUTO) FROM t;
```

You can carefully formulate queries so Vertica can avoid sorting the data and increase query performance, as illustrated by the following example. Vertica sorts inputs from table t on column x , as specified in the OVER(ORDER BY) clause, and then evaluates RANK() :

```
=> CREATE TABLE t (
   x FLOAT,
   y FLOAT );
=> CREATE PROJECTION t_p (x, y) AS SELECT * FROM t
  ORDER BY x, y UNSEGMENTED ALL NODES;
=> SELECT x, RANK() OVER (ORDER BY x) FROM t;
```

In the preceding SELECT statement, Vertica eliminates the ORDER BY clause and executes the query quickly because column x is a FLOAT data type. As a result, the projection sort order matches the analytic default ordering (ASC + NULLS LAST). Vertica can also avoid having to sort the data when the underlying projection is already sorted.

However, if column x is an INTEGER data type, Vertica must sort the data because the projection sort order for INTEGER data types (ASC + NULLS FIRST) does not match the default analytic ordering (ASC + NULLS LAST). To help Vertica eliminate the sort, specify the placement of NULLs to match the default ordering:

```
=> SELECT x, RANK() OVER (ORDER BY x NULLS FIRST) FROM t;
```

If column x is a STRING, the following query eliminates the sort:

```
=> SELECT x, RANK() OVER (ORDER BY x NULLS LAST) FROM t;
```

If you omit NULLS LAST in the preceding query, Ver eliminates the sort because ASC + NULLS LAST is the default sort specification for both the analytic ORDER BY clause and for string-related columns in Vertica.

See also

- Runtime sorting of NULL values in analytic functions
- SQL analytics

Runtime sorting of NULL values in analytic functions

By carefully writing queries or creating your design (or both), you can help the Vertica query optimizer skip sorting all columns in a table when performing an analytic function, which can improve query performance.

To minimize Vertica's need to sort projections during query execution, redefine the employee table and specify that NULL values are not allowed in the sort fields:

```
=> DROP TABLE employee CASCADE;
=> CREATE TABLE employee
  (empno INT,
   deptno INT NOT NULL,
   sal INT NOT NULL);
CREATE TABLE
=> CREATE PROJECTION employee_p AS
  SELECT * FROM employee
  ORDER BY deptno, sal;
CREATE PROJECTION
=> INSERT INTO employee VALUES(101,10,50000);
=> INSERT INTO employee VALUES(103,10,43000);
=> INSERT INTO employee VALUES(104,10,45000);
=> INSERT INTO employee VALUES(105,20,97000);
=> INSERT INTO employee VALUES(108,20,33000);
=> INSERT INTO employee VALUES(109,20,51000);
=> COMMIT;
COMMIT
```

```
=> SELECT * FROM employee;
 empno | deptno |  sal
-------+--------+-------
   101 |     10 | 50000
   103 |     10 | 43000
   104 |     10 | 45000
   105 |     20 | 97000
   108 |     20 | 33000
   109 |     20 | 51000
(6 rows)
=> SELECT deptno, sal, empno, RANK() OVER
    (PARTITION BY deptno ORDER BY sal)
  FROM employee;
 deptno |  sal  | empno | ?column?
--------+-------+-------+----------
     10 | 43000 |   103 |        1
     10 | 45000 |   104 |        2
     10 | 50000 |   101 |        3
     20 | 33000 |   108 |        1
     20 | 51000 |   109 |        2
     20 | 97000 |   105 |        3
(6 rows)
```

> **Tip**
> If you do not care about NULL placement in queries that involve analytic computations, or if you know that columns contain no NULL values, specify NULLS AUTO in your queries . Vertica attempts to choose the placement that gives the fastest performance. Otherwise, specify NULLS FIRST or NULLS LAST .

LIMIT queries

A query can include a LIMIT clause to limit its result set in two ways:

- Return a subset of rows from the entire result set.
- Set window partitions on the result set and limit the number of rows in each window.

Limiting the query result set

Queries that use the LIMIT clause with ORDER BY return a specific subset of rows from the queried dataset. Vertica processes these queries efficiently using *Top-K optimization* , which is a database query ranking process. Top-K optimization avoids sorting (and potentially writing to disk) an entire data set to find a small number of rows. This can significantly improve query performance.

For example, the following query returns the first 20 rows of data in table customer_dimension , as ordered by number_of_employees :

```
=> SELECT store_region, store_city||', '||store_state location, store_name, number_of_employees
    FROM store.store_dimension ORDER BY number_of_employees DESC LIMIT 20;
 store_region |       location       | store_name | number_of_employees
--------------+----------------------+------------+---------------------
 East         | Nashville, TN        | Store141   |                  50
 East         | Manchester, NH       | Store225   |                  50
 East         | Portsmouth, VA       | Store169   |                  50
 SouthWest    | Fort Collins, CO     | Store116   |                  50
 SouthWest    | Phoenix, AZ          | Store232   |                  50
 South        | Savannah, GA         | Store201   |                  50
 South        | Carrollton, TX       | Store8     |                  50
 West         | Rancho Cucamonga, CA | Store102   |                  50
 MidWest      | Lansing, MI          | Store105   |                  50
 West         | Provo, UT            | Store73    |                  50
 East         | Washington, DC       | Store180   |                  49
 MidWest      | Sioux Falls, SD      | Store45    |                  49
 NorthWest    | Seattle, WA          | Store241   |                  49
 SouthWest    | Las Vegas, NV        | Store104   |                  49
 West         | El Monte, CA         | Store100   |                  49
 SouthWest    | Fort Collins, CO     | Store20    |                  49
 East         | Lowell, MA           | Store57    |                  48
 SouthWest    | Arvada, CO           | Store188   |                  48
 MidWest      | Joliet, IL           | Store82    |                  48
 West         | Berkeley, CA         | Store248   |                  48
(20 rows)
```

Important
If a LIMIT clause omits ORDER BY , results can be nondeterministic.

Limiting window partitioning results

You can use LIMIT to set window partitioning on query results, and limit the number of rows that are returned in each window:

```
SELECT ... FROM dataset LIMIT num-rows OVER ( PARTITION BY column-expr-x, ORDER BY column-expr-y [ASC | DESC] )
```

where querying *dataset* returns *num-rows* rows in each *column-expr-x* partition with the highest or lowest values of *column-expr-y* .

For example, the following statement queries table store.store_dimension and includes a LIMIT clause that specifies window partitioning. In this case, Vertica partitions the result set by store_region , where each partition window displays for one region the two stores with the fewest employees:

```
=> SELECT store_region, store_city||', '||store_state location, store_name, number_of_employees FROM store.store_dimension
    LIMIT 2 OVER (PARTITION BY store_region ORDER BY number_of_employees ASC);
 store_region |       location       | store_name | number_of_employees
--------------+----------------------+------------+---------------------
 West         | Norwalk, CA          | Store43    |                  10
 West         | Lancaster, CA        | Store95    |                  11
 East         | Stamford, CT         | Store219   |                  12
 East         | New York, NY         | Store122   |                  12
 SouthWest    | North Las Vegas, NV  | Store170   |                  10
 SouthWest    | Phoenix, AZ          | Store228   |                  11
 NorthWest    | Bellevue, WA         | Store200   |                  19
 NorthWest    | Portland, OR         | Store39    |                  22
 MidWest      | South Bend, IN       | Store134   |                  10
 MidWest      | Evansville, IN       | Store30    |                  11
 South        | Mesquite, TX         | Store124   |                  10
 South        | Beaumont, TX         | Store226   |                  11
(12 rows)
```

INSERT-SELECT operations

There are several ways to optimize an INSERT-SELECT query that has the following format:

```
INSERT /*+direct*/ INTO destination SELECT * FROM source;
```

In this section

- Matching sort orders
- Identical segmentation

Matching sort orders

When performing INSERT - SELECT operations, to avoid the sort phase of the INSERT , make sure that the sort order for the SELECT query matches the projection sort order of the target table.

For example, on a single-node database:

```
=> CREATE TABLE source (col1 INT, col2 INT, col3 INT);
=> CREATE PROJECTION source_p (col1, col2, col3)
    AS SELECT col1, col2, col3 FROM source
    ORDER BY col1, col2, col3
    SEGMENTED BY HASH(col3)
    ALL NODES;
=> CREATE TABLE destination (col1 INT, col2 INT, col3 INT);
=> CREATE PROJECTION destination_p (col1, col2, col3)
    AS SELECT col1, col2, col3 FROM destination
    ORDER BY col1, col2, col3
    SEGMENTED BY HASH(col3)
    ALL NODES;
```

The following INSERT does not require a sort because the query result has the column order of the projection:

```
=> INSERT /*+direct*/ INTO destination SELECT * FROM source;
```

The following INSERT requires a sort because the order of the columns in the SELECT statement does not match the projection order:

```
=> INSERT /*+direct*/ INTO destination SELECT col1, col3, col2 FROM source;
```

The following INSERT does not require a sort. The order of the columns doesn't match, but the explicit ORDER BY causes the output to be sorted by c1 , c3 , c2 in Vertica:

```
=> INSERT /*+direct*/ INTO destination SELECT col1, col3, col2 FROM source
     GROUP BY col1, col3, col2
     ORDER BY col1, col2, col3 ;
```

Identical segmentation

When performing an INSERT-SELECT operation from a segmented source table to a segmented destination table, segment both projections on the same column to avoid resegmenting the data, as in the following example:

```
CREATE TABLE source (col1 INT, col2 INT, col3 INT);
CREATE PROJECTION source_p (col1, col2, col3) AS
  SELECT col1, col2, col3 FROM source
  SEGMENTED BY HASH(col3) ALL NODES;
CREATE TABLE destination (col1 INT, col2 INT, col3 INT);
CREATE PROJECTION destination_p (col1, col2, col3) AS
   SELECT col1, col2, col3 FROM destination
   SEGMENTED BY HASH(col3) ALL NODES;
INSERT /*+direct*/ INTO destination SELECT * FROM source;
```

DELETE and UPDATE queries

Vertica is optimized for query-intensive workloads, so DELETE and UPDATE queries might not achieve the same level of performance as other queries. DELETE and UPDATE operations must update all projections, so these operations can be no faster than the slowest projection. For details, see Optimizing DELETE and UPDATE .

Data collector table queries

The Vertica Data Collector extends system table functionality by gathering and retaining information about your database cluster. The Data Collector makes this information available in system tables.

Vertica Analytic Database stores Data Collection data in the Data Collector directory under the Vertica or catalog path. Use Data Collector information to query the past state of system tables and extract aggregate information.

In general, queries on Data Collector tables are more efficient when they include only the columns that contain the desired data. Queries are also more efficient when they:

- Avoid resegmentation
- Use time predicates

Avoiding resegmentation

You can avoid resegmentation when you join the following DC tables on session_id or transaction_id , because all data is local:

- dc_session_starts
- dc_session_ends
- dc_requests_issued
- dc_requests_completed

Resegmentation is not required when a query includes the node_name column. For example:

```
=> SELECT dri.transaction_id, dri.request, drc.processed_row_count
   FROM dc_requests_issued dri
   JOIN dc_requests_completed drc
   USING (node_name, session_id, request_id)
   WHERE dri.time between 'April 7,2015'::timestamptz and 'April 8,2015'::timestamptz
   AND drc.time between 'April 7,2015'::timestamptz and 'April 8,2015'::timestamptz;
```

This query runs efficiently because:

- The initiator node writes only to dc_requests_issued and dc_requests_completed .
- Columns session_id and node_name are correlated.

Using time predicates

Use non-volatile functions and TIMESTAMP for the time range predicates. Vertica Analytic Database optimizes SQL performance for DC tables that use the time predicate.

Each DC table has a time column. Use this column to enter the time range as the query predicate.

For example, this query returns data for dates between September 1 and September 10: select * from dc_foo where time > 'Sept 1, 2015::timestamptz and time < 'Sept 10 2015':: timestamptz; You can change the minimum and maximum time values to adjust the time for which you want to retrieve data.

You must use non-volatile functions as time predicates. Volatile functions cause queries to run inefficiently. This example returns all queries that started and ended on April 7, 2015. However, the query runs at less than optimal performance because trunc and timestamp are volatile:

```
=> SELECT dri.transaction_id, dri.request, drc.processed_row_count
   FROM dc_requests_issued dri
   LEFT JOIN dc_requests_completed drc
   USING (session_id, request_id)
   WHERE trunc(dri.time, 'DDD') > 'April 7,2015'::timestamp
   AND trunc(drc.time, 'DDD') < 'April 8,2015'::timestamp;
```

Views

A view is a stored query that encapsulates one or more SELECT statements. Views dynamically access and compute data from the database at execution time. A view is read-only, and can reference any combination of tables, temporary tables, and other views.

You can use views to achieve the following goals:

- Hide the complexity of SELECT statements from users for support or security purposes. For example, you can create a view that exposes only the data users need from various tables, while withholding sensitive data from the same tables.
- Encapsulate details about table structures, which might change over time, behind a consistent user interface.

Unlike projections, views are not materialized—that is, they do not store data on disk. Thus, the following restrictions apply:

- Vertica does not need to refresh view data when the underlying table data changes. However, a view does incur overhead to access and compute data.
- Views do not support inserts, deletes, or updates.

In this section

- Creating views
- Using views
- View execution
- Managing views

Creating views

You can create two types of views:

- CREATE VIEW creates a view that persists across all sessions until it is explicitly dropped with DROP VIEW
- CREATE LOCAL TEMPORARY VIEW creates a view that is accessible only during the current Vertica session, and only to its creator. The view is automatically dropped when the current session ends.

After you create a view, you cannot change its definition. You can replace it with another view of the same name; or you can delete and redefine it.

Permissions

To create a view, a non-superuser must have the following privileges:

Privilege	Objects
CREATE	Schema where the view is created
DROP	The view (only required if you specify an existing view in CREATE OR REPLACE VIEW)
SELECT	Tables and views referenced by the view query
USAGE	All schemas that contain tables and views referenced by the view query

For information about enabling users to access views, see View Access Permissions .

Using views

Views can be used in the FROM clause of any SQL query or subquery. At execution, Vertica internally substitutes the name of the view used in the query with the actual query used in the view definition.

CREATE VIEW example

The following CREATE VIEW statement creates the view myview , which sums all individual incomes of customers listed in the store.store_sales_fact table, and groups results by state:

```
=> CREATE VIEW myview AS
  SELECT SUM(annual_income), customer_state FROM public.customer_dimension
    WHERE customer_key IN (SELECT customer_key FROM store.store_sales_fact)
    GROUP BY customer_state
    ORDER BY customer_state ASC;
```

You can use this view to find all combined salaries greater than $2 billion:

```
=> SELECT * FROM myview where sum > 2000000000 ORDER BY sum DESC;
    SUM     | customer_state
------------+----------------
 29253817091 | CA
 14215397659 | TX
  5225333668 | MI
  4907216137 | CO
  4581840709 | IL
  3769455689 | CT
  3330524215 | FL
  3310667307 | IN
  2832710696 | TN
  2806150503 | PA
  2793284639 | MA
  2723441590 | AZ
  2642551509 | UT
  2128169759 | NV
(14 rows)
```

Enabling view access

You can query any view that you create. To enable other non-superusers to access a view, you must:

- Have SELECT...WITH GRANT OPTION privileges on the view's base table
- Grant users USAGE privileges on the view schema
- Grant users SELECT privileges to the view itself

The following example grants user2 access to view schema1.view1 :

```
=> GRANT USAGE ON schema schema1 TO user2;
=> GRANT SELECT ON schema1.view1 TO user2;
```

Important

If the view references an external table, you must also grant USAGE privileges to the external table's schema. So, if schema1.view1 references external table schema2.extTable1 , you must also grant user2 USAGE privileges to schema2 :

```
=> GRANT USAGE on schema schema2 to user2;
```

For more information see GRANT (view) .

View execution

When Vertica processes a query that contains a view, it treats the view as a subquery. Vertica executes the query by expanding it to include the query in the view definition. For example, Vertica expands the query on the view myview shown in Using Views , to include the query that the view encapsulates, as follows:

```
=> SELECT * FROM
  (SELECT SUM(annual_income), customer_state FROM public.customer_dimension
   WHERE customer_key IN
     (SELECT customer_key FROM store.store_sales_fact)
   GROUP BY customer_state
   ORDER BY customer_state ASC)
   AS ship where sum > 2000000000;
```

View optimization

If you query a view and your query only includes columns from a subset of the tables that are joined in that view, Vertica executes that query by expanding it to include only those tables. This optimization requires one of the following conditions to be true:

- Join columns are foreign and primary keys.
- The join is a left or right outer join on columns with unique values.

View sort order

When processing a query on a view, Vertica considers the ORDER BY clause only in the outermost query. If the view definition includes an ORDER BY clause, Vertica ignores it. Thus, in order to sort the results returned by a view, you must specify the ORDER BY clause in the outermost query:

```
=> SELECT * FROM view-name ORDER BY view-column;
```

> **Note**
> One exception applies: Vertica sorts view data when the view includes a LIMIT clause. In this case, Vertica must sort the data before it can process the LIMIT clause.

For example, the following view definition contains an ORDER BY clause inside a FROM subquery:

```
=> CREATE VIEW myview AS SELECT SUM(annual_income), customer_state FROM public.customer_dimension
  WHERE customer_key IN
    (SELECT customer_key FROM store.store_sales_fact)
  GROUP BY customer_state
  ORDER BY customer_state ASC;
```

When you query the view, Vertica does not sort the data:

```
=> SELECT * FROM myview WHERE SUM > 2000000000;
    SUM     | customer_state
------------+----------------
 5225333668 | MI
 2832710696 | TN
14215397659 | TX
 4907216137 | CO
 2793284639 | MA
 3769455689 | CT
 3310667307 | IN
 2723441590 | AZ
 2642551509 | UT
 3330524215 | FL
 2128169759 | NV
29253817091 | CA
 4581840709 | IL
 2806150503 | PA
(14 rows)
```

To return sorted results, the outer query must include an ORDER BY clause:

```
=> SELECT * FROM myview WHERE SUM > 2000000000 ORDER BY customer_state ASC;
    SUM     | customer_state
------------+----------------
 2723441590 | AZ
29253817091 | CA
 4907216137 | CO
 3769455689 | CT
 3330524215 | FL
 4581840709 | IL
 3310667307 | IN
 2793284639 | MA
 5225333668 | MI
 2128169759 | NV
 2806150503 | PA
 2832710696 | TN
14215397659 | TX
 2642551509 | UT
(14 rows)
```

Run-time errors

If Vertica does not have to evaluate an expression that would generate a run-time error in order to answer a query, the run-time error might not occur.

For example, the following query returns an error, because TO_DATE cannot convert the string F to the specified date format:

```
=> SELECT TO_DATE('F','dd mm yyyy') FROM customer_dimension;
  ERROR: Invalid input for DD: "F"
```

Now create a view using the same query:

```
=> CREATE VIEW temp AS SELECT TO_DATE('F','dd mm yyyy')
  FROM customer_dimension;
  CREATE VIEW
```

In many cases, this view generates the same error message. For example:

```
=> SELECT * FROM temp;
  ERROR: Invalid input for DD: "F"
```

However, if you query that view with the COUNT function, Vertica returns with the desired results:

```
=> SELECT COUNT(*) FROM temp;
 COUNT
-------
  100
(1 row)
```

This behavior works as intended. You can create views that contain subqueries, where not every row is intended to pass the predicate.

Managing views

Obtaining view information

You can query system tables VIEWS and VIEW_COLUMNS to obtain information about existing views—for example, a view's definition and the attributes of columns that comprise that view. You can also query system table VIEW_TABLES to examine view-related dependencies—for example, to determine how many views reference a table before you drop it.

Renaming a view

Use ALTER VIEW to rename a view.

Dropping a view

Use DROP VIEW to drop a view. Only the specified view is dropped. Vertica does not support CASCADE functionality for views, and does not check for dependencies. Dropping a view causes any view that references it to fail.

Disabling and re-enabling views

If you drop a table that is referenced by a view, Vertica does not drop the view. However, attempts to use that view or access information about it from system table VIEW_COLUMNS return an error that the referenced table does not exist. If you restore that table, Vertica also re-enables usage of the view.

Flattened tables

Highly normalized database design often uses a star or snowflake schema model, comprising multiple large fact tables and many smaller dimension tables. Queries typically involve joins between a large fact table and multiple dimension tables. Depending on the number of tables and quantity of data that are joined, these queries can incur significant overhead.

To avoid this problem, some users create wide tables that combine all fact and dimension table columns that their queries require. These tables can dramatically speed up query execution. However, maintaining redundant sets of normalized and denormalized data has its own administrative costs.

Denormalized, or *flattened* , tables, can minimize these problems. Flattened tables can include columns that get their values by querying other tables. Operations on the source tables and flattened table are decoupled; changes in one are not automatically propagated to the other. This minimizes the overhead that is otherwise typical of denormalized tables.

A flattened table defines derived columns with one or both of the following column constraint clauses:

- DEFAULT *query-expression* sets the column value when the column is created with CREATE TABLE or ALTER TABLE...ADD COLUMN .
- SET USING *query-expression* sets the column value when the function REFRESH_COLUMNS is invoked.

In both cases, *query-expression* must return only one row and column value, or none. If the query returns no rows, the column value is set to NULL.

Like other tables defined in Vertica, you can add and remove DEFAULT and SET USING columns from a flattened table at any time. Vertica enforces dependencies between a flattened table and the tables that it queries. For details, see Modifying SET USING and DEFAULT columns .

In this section

- Flattened table example
- Creating flattened tables
- Required privileges
- DEFAULT versus SET USING
- Modifying SET USING and DEFAULT columns
- Rewriting SET USING queries
- Impact of SET USING columns on license limits

Flattened table example

In the following example, columns orderFact.cust_name and orderFact.cust_gender are defined as SET USING and DEFAULT columns, respectively. Both columns obtain their values by querying table custDim :

```
=> CREATE TABLE public.custDim(
      cid int PRIMARY KEY NOT NULL,
      name varchar(20),
      age int,
     gender varchar(1)
);

=> CREATE TABLE public.orderFact(
      order_id int PRIMARY KEY NOT NULL,
      order_date timestamp DEFAULT CURRENT_TIMESTAMP NOT NULL,
      cid int REFERENCES public.custDim(cid),
      cust_name varchar(20) SET USING (SELECT name FROM public.custDim WHERE (custDim.cid = orderFact.cid)),
      cust_gender varchar(1) DEFAULT (SELECT gender FROM public.custDim WHERE (custDim.cid = orderFact.cid)),
      amount numeric(12,2)
)
PARTITION BY order_date::DATE GROUP BY CALENDAR_HIERARCHY_DAY(order_date::DATE, 2, 2);
```

The following INSERT commands load data into both tables:

```
=> INSERT INTO custDim VALUES(1, 'Alice', 25, 'F');
=> INSERT INTO custDim VALUES(2, 'Boz', 30, 'M');
=> INSERT INTO custDim VALUES(3, 'Eva', 32, 'F');
=>
=> INSERT INTO orderFact (order_id, cid, amount) VALUES(100, 1, 15);
=> INSERT INTO orderFact (order_id, cid, amount) VALUES(200, 1, 1000);
=> INSERT INTO orderFact (order_id, cid, amount) VALUES(300, 2, -50);
=> INSERT INTO orderFact (order_id, cid, amount) VALUES(400, 3, 100);
=> INSERT INTO orderFact (order_id, cid, amount) VALUES(500, 2, 200);
=> COMMIT;
```

When you query the tables, Vertica returns the following result sets:

```
=> SELECT * FROM custDim;
 cid | name  | age | gender
-----+-------+-----+--------
   1 | Alice |  25 | F
   2 | Boz   |  30 | M
   3 | Eva   |  32 | F
(3 rows)

=> SELECT order_id, order_date::date, cid, cust_name, cust_gender, amount FROM orderFact ORDER BY cid;
 order_id | order_date | cid | cust_name | cust_gender | amount
----------+------------+-----+-----------+-------------+---------
      100 | 2018-12-31 |   1 |           | F           |   15.00
      200 | 2018-12-31 |   1 |           | F           | 1000.00
      300 | 2018-12-31 |   2 |           | M           |  -50.00
      500 | 2018-12-31 |   2 |           | M           |  200.00
      400 | 2018-12-31 |   3 |           | F           |  100.00
(5 rows)
```

Vertica automatically populates the DEFAULT column orderFact.cust_gender , but the SET USING column orderFact.cust_name remains NULL. You can automatically populate this column by calling the function REFRESH_COLUMNS on flattened table orderFact. This function invokes the SET USING query for column orderFact.cust_name and populates the column from the result set:

```
=> SELECT REFRESH_COLUMNS('orderFact', 'cust_name', 'REBUILD');

    REFRESH_COLUMNS
-------------------------------
refresh_columns completed
(1 row)

=> SELECT order_id, order_date::date, cid, cust_name, cust_gender, amount FROM orderFact ORDER BY cid;
 order_id | order_date | cid | cust_name | cust_gender | amount
----------+------------+-----+-----------+-------------+---------
      100 | 2018-12-31 |   1 | Alice     | F           |   15.00
      200 | 2018-12-31 |   1 | Alice     | F           | 1000.00
      300 | 2018-12-31 |   2 | Boz       | M           |  -50.00
      500 | 2018-12-31 |   2 | Boz       | M           |  200.00
      400 | 2018-12-31 |   3 | Eva       | F           |  100.00
(5 rows)
```

Creating flattened tables

A flattened table is typically a fact table where one or more columns query other tables for their values, through DEFAULT or SET USING constraints. DEFAULT and SET USING constraints can be used for columns of all data types. Like other columns, you can set these constraints when you create the flattened table, or any time thereafter by modifying the table DDL:

- CREATE TABLE ... (*column-name data-type* { DEFAULT | SET USING } *expression*)
- ALTER TABLE...ADD COLUMN *column-name* { DEFAULT | SET USING } *expression*
- ALTER TABLE...ALTER COLUMN *column-name* { SET DEFAULT | SET USING } *expression*

In all cases, the expressions that you set for these constraints are stored in the system table COLUMNS , in columns COLUMN_DEFAULT and COLUMN_SET_USING.

Supported expressions

DEFAULT and SET USING generally support the same expressions. These include:

- Queries
- Other columns in the same table
- Literals (constants)
- All operators supported by Vertica
- The following categories of functions:
  - Null-handling
  - User-defined scalar

- System information
- String
- Mathematical
- Formatting

For more information about DEFAULT and SET USING expressions, including restrictions, see Defining column values .

Required privileges

The following operations on flattened table require privileges as shown:

Operation	Object	Privileges
Retrieve data from a flattened table.	Schema	USAGE
	Flattened table	SELECT
Add SET USING or DEFAULT columns to a table.	Schemas (queried/flattened tables)	USAGE
	Queried tables	SELECT
	Target table	CREATE
INSERT data on a flattened table with SET USING and/or DEFAULT columns.	Schemas (queried/flattened tables)	USAGE
	Queried tables	SELECT
	Flattened table	INSERT
Run REFRESH_COLUMNS on a flattened table.	Schemas (queried/flattened tables)	USAGE
	Queried tables	SELECT
	Flattened table	SELECT, UPDATE

DEFAULT versus SET USING

Columns in a flattened table can query other tables with constraints DEFAULT and SET USING. In both cases, changes in the queried tables are not automatically propagated to the flattened table. The two constraints differ as follows:

DEFAULT columns

Vertica typically executes DEFAULT queries on new rows when they are added to the flattened table, through load operations such as INSERT and COPY, or when you create or alter a table with a new column that specifies a DEFAULT expression. In all cases, values in existing rows, including other columns with DEFAULT expressions, remain unchanged.

To refresh a column's default values, you must explicitly call UPDATE on that column as follows:

```
=> UPDATE table-name SET column-name=DEFAULT;
```

SET USING columns

Vertica executes SET USING queries only when you invoke the function REFRESH_COLUMNS . Load operations set SET USING columns in new rows to NULL. After the load, you must call REFRESH_COLUMNS to populate these columns from the queried tables. This can be useful in two ways: you can defer the overhead of updating the flattened table to any time that is convenient; and you can repeatedly query source tables for new data.

SET USING is especially useful for large flattened tables that reference data from multiple dimension tables. Often, only a small subset of SET USING columns are subject to change, and queries on the flattened table do not always require up-to-the-minute data. Given this scenario, you can refresh table content at regular intervals, or only during off-peak hours. One or both of these strategies can minimize overhead, and facilitate performance when querying large data sets.

You can control the scope of the refresh operation by calling REFRESH _COLUMNS in one of two modes:

- UPDATE : Marks original rows as deleted and replaces them with new rows. In order to save these updates, you must issue a COMMIT statement.
- REBUILD: Replaces all data in the specified columns. The rebuild operation is auto-committed.

If a flattened table is partitioned, you can reduce the overhead of calling REFRESH_COLUMNS in REBUILD mode by specifying one or more partition keys. Doing so limits the rebuild operation to the specified partitions. For details, see Partition-based REBUILD Operations .

Examples

> **Note**
> Examples use the custDim and orderFact tables described in Flattened table example .

The following UPDATE statement updates the custDim table:

```
=> UPDATE custDim SET name='Roz', gender='F' WHERE cid=2;
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
```

Changes are not propagated to flattened table orderFact , which includes SET USING and DEFAULT columns cust_name and cust_gender , respectively:

```
=> SELECT * FROM custDim ORDER BY cid;
 cid | name  | age | gender
-----+-------+-----+--------
   1 | Alice |  25 | F
   2 | Roz   |  30 | F
   3 | Eva   |  32 | F
(3 rows)

=> SELECT order_id, order_date::date, cid, cust_name, cust_gender, amount FROM orderFact ORDER BY cid;
 order_id | order_date | cid | cust_name | cust_gender | amount
----------+------------+-----+-----------+-------------+---------
      100 | 2018-12-31 |   1 | Alice     | F           |   15.00
      200 | 2018-12-31 |   1 | Alice     | F           | 1000.00
      300 | 2018-12-31 |   2 | Boz       | M           |  -50.00
      500 | 2018-12-31 |   2 | Boz       | M           |  200.00
      400 | 2018-12-31 |   3 | Eva       | F           |  100.00
(5 rows)
```

The following INSERT statement invokes the cust_gender column's DEFAULT query and sets that column to F . The load operation does not invoke the cust_name column's SET USING query, so cust_name is set to null:

```
=> INSERT INTO orderFact(order_id, cid, amount)  VALUES(500, 3, 750);
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
=>  SELECT order_id, order_date::date, cid, cust_name, cust_gender, amount FROM orderFact ORDER BY cid;
 order_id | order_date | cid | cust_name | cust_gender | amount
----------+------------+-----+-----------+-------------+---------
      100 | 2018-12-31 |   1 | Alice     | F           |   15.00
      200 | 2018-12-31 |   1 | Alice     | F           | 1000.00
      300 | 2018-12-31 |   2 | Boz       | M           |  -50.00
      500 | 2018-12-31 |   2 | Boz       | M           |  200.00
      400 | 2018-12-31 |   3 | Eva       | F           |  100.00
      500 | 2018-12-31 |   3 |           | F           |  750.00
(6 rows)
```

To update the values in cust_name , invoke its SET USING query by calling REFRESH_COLUMNS. REFRESH_COLUMNS executes cust_name 's SET USING query: it queries the name column in table custDim and updates cust_name with the following values:

- Sets cust_name in the new row to Eva .
- Returns updated values for cid=2 , and changes Boz to Roz .

```
=> SELECT REFRESH_COLUMNS ('orderFact','');
     REFRESH_COLUMNS
---------------------------
 refresh_columns completed
(1 row)

=> COMMIT;
COMMIT
=>  SELECT order_id, order_date::date, cid, cust_name, cust_gender, amount FROM orderFact ORDER BY cid;
 order_id | order_date | cid | cust_name | cust_gender | amount
----------+------------+-----+-----------+-------------+---------
      100 | 2018-12-31 |   1 | Alice     | F           |   15.00
      200 | 2018-12-31 |   1 | Alice     | F           | 1000.00
      300 | 2018-12-31 |   2 | Roz       | M           |  -50.00
      500 | 2018-12-31 |   2 | Roz       | M           |  200.00
      400 | 2018-12-31 |   3 | Eva       | F           |  100.00
      500 | 2018-12-31 |   3 | Eva       | F           |  750.00
(6 rows)
```

REFRESH_COLUMNS only affects the values in column cust_name . Values in column gender are unchanged, so settings for rows where cid=2 (Roz) remain set to M . To repopulate orderFact.cust_gender with default values from custDim.gender , call UPDATE on orderFact :

```
=> UPDATE orderFact SET cust_gender=DEFAULT WHERE cust_name='Roz';
 OUTPUT
--------
      2
(1 row)

=> COMMIT;
COMMIT
=> SELECT order_id, order_date::date, cid, cust_name, cust_gender, amount FROM orderFact ORDER BY cid;
 order_id | order_date | cid | cust_name | cust_gender | amount
----------+------------+-----+-----------+-------------+---------
      100 | 2018-12-31 |   1 | Alice     | F           |   15.00
      200 | 2018-12-31 |   1 | Alice     | F           | 1000.00
      300 | 2018-12-31 |   2 | Roz       | F           |  -50.00
      500 | 2018-12-31 |   2 | Roz       | F           |  200.00
      400 | 2018-12-31 |   3 | Eva       | F           |  100.00
      500 | 2018-12-31 |   3 | Eva       | F           |  750.00
(6 rows)
```

Modifying SET USING and DEFAULT columns

> **Note**
> Examples use the custDim and orderFact tables described in Flattened table example .

Modifying a SET USING and DEFAULT expression

ALTER TABLE...ALTER COLUMN can set an existing column to SET USING or DEFAULT, or change the query expression of an existing SET USING or DEFAULT column. In both cases, existing values remain unchanged. Vertica refreshes column values in the following cases:

- DEFAULT column: Refreshed only when you load new rows, or when you invoke UPDATE to set column values to DEFAULT .
- SET USING column: Refreshed only when you call REFRESH_COLUMNS on the table.

For example, you might set an entire column to NULL as follows:

```
=> ALTER TABLE orderFact ALTER COLUMN cust_name SET USING NULL;
ALTER TABLE
=> SELECT REFRESH_COLUMNS('orderFact', 'cust_name', 'REBUILD');
REFRESH_COLUMNS
---------------------------
refresh_columns completed
(1 row)

=> SELECT order_id, order_date::date, cid, cust_name, cust_gender, amount FROM orderFact ORDER BY cid;
 order_id | order_date | cid | cust_name | cust_gender | amount
----------+------------+-----+-----------+-------------+---------
   100 | 2018-12-31 |   1 |           | F           |   15.00
   200 | 2018-12-31 |   1 |           | F           | 1000.00
   300 | 2018-12-31 |   2 |           | M           |  -50.00
   500 | 2018-12-31 |   2 |           | M           |  200.00
   400 | 2018-12-31 |   3 |           | F           |  100.00
(5 rows)
```

For details, see Defining column values

Removing SET USING and DEFAULT constraints

You remove a column's SET USING or DEFAULT constraint with ALTER TABLE...ALTER COLUMN , as follows:

```
ALTER TABLE table-name ALTER COLUMN column-name DROP { SET USING | DEFAULT };
```

Vertica removes the constraint from the specified column, but the column and its data are otherwise unaffected. For example:

```
=> ALTER TABLE orderFact ALTER COLUMN cust_name DROP SET USING;
ALTER TABLE
```

Dropping columns queried by SET USING or DEFAULT

Vertica enforces dependencies between a flattened table and the tables that it queries. Attempts to drop a queried column or its table return an error unless the drop operation also includes the CASCADE option. Vertica implements CASCADE by removing the SET USING or DEFAULT constraint from the flattened table. The table column and its data are otherwise unaffected.

For example, attempts to drop column name in table custDim returns a rollback error, as this column is referenced by SET USING column orderFact.cust_gender :

```
=> ALTER TABLE custDim DROP COLUMN gender;
ROLLBACK 7301:  Cannot drop column "gender" since it is referenced in the default expression of table "public.orderFact", column "cust_gender"
```

To drop this column, use the CASCADE option:

```
=> ALTER TABLE custDim DROP COLUMN gender CASCADE;
ALTER TABLE
```

Vertica removes the DEFAULT constraint from orderFact.cust_gender as part of the drop operation. However, cust_gender retains the data that it previously queried from the dropped column custDim.gender :

```
=> SELECT EXPORT_TABLES('','orderFact');
                                        EXPORT_TABLES
-----------------------------------------------------------------------------------------------------------
CREATE TABLE public.orderFact
(
    order_id int NOT NULL,
    order_date timestamp NOT NULL DEFAULT (now())::timestamptz(6),
    cid int,
    cust_name varchar(20),
    cust_gender varchar(1) SET USING NULL,
    amount numeric(12,2),
    CONSTRAINT C_PRIMARY PRIMARY KEY (order_id) DISABLED
)
PARTITION BY ((orderFact.order_date)::date) GROUP BY (CASE WHEN ("datediff"('year', (orderFact.order_date)::date, ((now())::timestamptz(6))::date) >=
2) THEN (date_trunc('year', (orderFact.order_date)::date))::date WHEN ("datediff"('month', (orderFact.order_date)::date, ((now())::timestamptz(6))::date) >= 2)
THEN (date_trunc('month', (orderFact.order_date)::date))::date ELSE (orderFact.order_date)::date END);
ALTER TABLE public.orderFact ADD CONSTRAINT C_FOREIGN FOREIGN KEY (cid) references public.custDim (cid);

(1 row)

=> SELECT * FROM orderFact;
 order_id |         order_date         | cid | cust_name | cust_gender | amount
----------+----------------------------+-----+-----------+-------------+---------
      400 | 2021-01-05 13:27:56.026115 |   3 |           | F           |  100.00
      300 | 2021-01-05 13:27:56.026115 |   2 |           | F           |  -50.00
      200 | 2021-01-05 13:27:56.026115 |   1 |           | F           | 1000.00
      500 | 2021-01-05 13:30:18.138386 |   3 |           | F           |  750.00
      100 | 2021-01-05 13:27:56.026115 |   1 |           | F           |   15.00
      500 | 2021-01-05 13:27:56.026115 |   2 |           | F           |  200.00
(6 rows)
```

Rewriting SET USING queries

When you call REFRESH_COLUMNS on a flattened table 's SET USING (or DEFAULT USING) column, it executes the SET USING query by joining the target and source tables. By default, the source table is always the inner table of the join. In most cases, cardinality of the source table is less than the target table, so REFRESH_COLUMNS executes the join efficiently.

Occasionally—notably, when you call REFRESH_COLUMNS on a partitioned table—the source table can be larger than the target table. In this case, performance of the join operation can be suboptimal.

You can address this issue by enabling configuration parameter RewriteQueryForLargeDim . When enabled (1), Vertica rewrites the query, by reversing the inner and outer join between the target and source tables.

Important
Enable this parameter only if the SET USING source data is in a table that is larger than the target table. If the source data is in a table smaller than the target table, then enabling RewriteQueryForLargeDim can adversely affect refresh performance.

Impact of SET USING columns on license limits

Vertica does not count the data in denormalized columns towards your raw data license limit. SET USING columns obtain their data by querying columns in other tables. Only data from the source tables counts against your raw data license limit.

For a list of SET USING restrictions, see Defining column values .

You can remove a SET USING column so it counts toward your license limit with the following command:

```
=> ALTER TABLE table1 ALTER COLUMN column1 DROP SET USING;
```

SQL analytics

Vertica analytics are SQL functions based on the ANSI 99 standard. These functions handle complex analysis and reporting tasks—for example:

- Rank the longest-standing customers in a particular state.
- Calculate the moving average of retail volume over a specified time.
- Find the highest score among all students in the same grade.
- Compare the current sales bonus that salespersons received against their previous bonus.

Analytic functions return aggregate results but they do not group the result set. They return the group value multiple times, once per record. You can sort group values, or partitions, using a window ORDER BY clause, but the order affects only the function result set, not the entire query result set.

For details about supported functions, see Analytic functions .

In this section

- Invoking analytic functions
- Analytic functions versus aggregate functions
- Window partitioning
- Window ordering
- Window framing
- Named windows
- Analytic query examples
- Event-based windows
- Sessionization with event-based windows

Invoking analytic functions

You invoke analytic functions as follows:

```
analytic-function(arguments) OVER(
  [ window-partition-clause ]
  [ window-order-clause [ window-frame-clause ] ]
)
```

An analytic function's OVER clause contains up to three sub-clauses, which specify how to partition and sort function input, and how to frame input with respect to the current row. Function input is the result set that the query returns after it evaluates FROM , WHERE , GROUP BY , and HAVING clauses.

> **Note**
> Each function has its own OVER clause requirements. For example, some analytic functions do not support window order and window frame clauses.

For syntax details, see Analytic functions .

Function execution

An analytic function executes as follows:

1. Takes the input rows that the query returns after it performs all joins, and evaluates FROM , WHERE , GROUP BY , and HAVING clauses.
2. Groups input rows according to the window partition (PARTITION BY) clause. If this clause is omitted, all input rows are treated as a single partition.
3. Sorts rows within each partition according to window order (ORDER BY) clause.
4. If the OVER clause includes a window order clause, the function checks for a window frame clause and executes it as it processes each input row. If the OVER clause omits a window frame clause, the function treats the entire partition as a window frame.

Restrictions

- Analytic functions are allowed only in a query's SELECT and ORDER BY clauses.
- Analytic functions cannot be nested. For example, the following query throws an error:

```
=> SELECT MEDIAN(RANK() OVER(ORDER BY sal) OVER()).
```

Analytic functions versus aggregate functions

Like aggregate functions, analytic functions return aggregate results, but analytics do not group the result set. Instead, they return the group value multiple times with each record, allowing further analysis.

Analytic queries generally run faster and use fewer resources than aggregate queries.

Aggregate functions	**Analytic functions**
Return a single summary value.	Return the same number of rows as the input.
Define the groups of rows on which they operate through the SQL GROUP BY clause.	Define the groups of rows on which they operate through window partition and window frame clauses.

Examples

The examples below contrast the aggregate function COUNT with its analytic counterpart COUNT . The examples use the employees table as defined below:

```
CREATE TABLE employees(emp_no INT, dept_no INT);
INSERT INTO employees VALUES(1, 10);
INSERT INTO employees VALUES(2, 30);
INSERT INTO employees VALUES(3, 30);
INSERT INTO employees VALUES(4, 10);
INSERT INTO employees VALUES(5, 30);
INSERT INTO employees VALUES(6, 20);
INSERT INTO employees VALUES(7, 20);
INSERT INTO employees VALUES(8, 20);
INSERT INTO employees VALUES(9, 20);
INSERT INTO employees VALUES(10, 20);
INSERT INTO employees VALUES(11, 20);
COMMIT;
```

When you query this table, the following result set returns:

```
=> SELECT * FROM employees ORDER BY emp_no;
 emp_no | dept_no
--------+---------
      1 |      10
      2 |      30
      3 |      30
      4 |      10
      5 |      30
      6 |      20
      7 |      20
      8 |      20
      9 |      20
     10 |      20
     11 |      20
(11 rows)
```

Below, two queries use the COUNT function to count the number of employees in each department. The query on the left uses aggregate function COUNT ; the query on the right uses analytic function COUNT :

Aggregate COUNT	**Analytics COUNT**

`=> SELECT dept_no, COUNT(*) AS emp_count`
`FROM employees`
`GROUP BY dept_no ORDER BY dept_no;`
	`=> SELECT emp_no, dept_no, COUNT(*) OVER(`
	`PARTITION BY dept_no`
	`ORDER BY emp_no) AS emp_count`
	`FROM employees;`

`dept_no | emp_count`
`---------+-----------`
`10 | 2`
`20 | 6`
`30 | 3`
`(3 rows)`
	`emp_no | dept_no | emp_count`
	`--------+---------+-----------`
	`1 | 10 | 1`
	`4 | 10 | 2`
	`-------------------------------`
	`6 | 20 | 1`
	`7 | 20 | 2`
	`8 | 20 | 3`
	`9 | 20 | 4`
	`10 | 20 | 5`
	`11 | 20 | 6`
	`-------------------------------`
	`2 | 30 | 1`
	`3 | 30 | 2`
	`5 | 30 | 3`
	`(11 rows)`

Aggregate function COUNT returns one row per department for the number of employees in that department.	Within each dept_no partition analytic function COUNT returns a cumulative count of employees. The count is ordered by emp_no , as specified by the ORDER BY clause.

See also

- Analytic query examples

Window partitioning

Optionally specified in a function's OVER clause, a partition (PARTITION BY) clause groups input rows before the function processes them. Window partitioning using PARTITION NODES or PARTITION BEST is similar to an aggregate function's GROUP BY clause, except it returns exactly one result row per input row. Window partitioning using PARTITION ROW allows you to provide single-row partitions of input, allowing you to use window partitioning on 1:N transform functions. If you omit the window partition clause, the function treats all input rows as a single partition.

Specifying window partitioning

You specify window partitioning in the function's OVER clause, as follows:

```
{ PARTITION BY expression[,...] | PARTITION BEST | PARTITION NODES | PARTITION ROW }
```

For syntax details, see Window partition clause .

Examples

The examples in this topic use the allsales schema defined in Invoking analytic functions .

```
CREATE TABLE allsales(state VARCHAR(20), name VARCHAR(20), sales INT);
INSERT INTO allsales VALUES('MA', 'A', 60);
INSERT INTO allsales VALUES('NY', 'B', 20);
INSERT INTO allsales VALUES('NY', 'C', 15);
INSERT INTO allsales VALUES('MA', 'D', 20);
INSERT INTO allsales VALUES('MA', 'E', 50);
INSERT INTO allsales VALUES('NY', 'F', 40);
INSERT INTO allsales VALUES('MA', 'G', 10);
COMMIT;
```

The following query calculates the median of sales within each state. The analytic function is computed per partition and starts over again at the beginning of the next partition.

```
=> SELECT state, name, sales, MEDIAN(sales)
    OVER (PARTITION BY state) AS median from allsales;
```

The results are grouped into partitions for MA (35) and NY (20) under the median column.

```
 state | name | sales | median
-------+------+-------+--------
 NY    | C    |    15 |     20
 NY    | B    |    20 |     20
 NY    | F    |    40 |     20
-------------------------------
 MA    | G    |    10 |     35
 MA    | D    |    20 |     35
 MA    | E    |    50 |     35
 MA    | A    |    60 |     35
(7 rows)
```

The following query calculates the median of total sales among states. When you use OVER() with no parameters, there is one partition—the entire input:

```
=> SELECT state, sum(sales), median(SUM(sales))
    OVER () AS median FROM allsales GROUP BY state;
 state | sum | median
-------+-----+--------
 NY    |  75 |  107.5
 MA    | 140 |  107.5
(2 rows)
```

Sales larger than median (evaluation order)

Analytic functions are evaluated *after* all other clauses except the query's final SQL ORDER BY clause. So a query that asks for all rows with sales larger than the median returns an error because the WHERE clause is applied before the analytic function and column m does not yet exist:

```
=> SELECT name, sales,  MEDIAN(sales) OVER () AS m
  FROM allsales WHERE sales > m;
  ERROR 2624:  Column "m" does not exist
```

You can work around this by placing in a subquery the predicate WHERE sales > m :

```
=> SELECT * FROM
   (SELECT name, sales, MEDIAN(sales) OVER () AS m FROM allsales) sq
   WHERE sales > m;
 name | sales | m
------+-------+----
 F    |    40 | 20
 E    |    50 | 20
 A    |    60 | 20
(3 rows)
```

For more examples, see Analytic query examples .

Window ordering

Window ordering specifies how to sort rows that are supplied to the function. You specify window ordering through an ORDER BY clause in the function's OVER clause, as shown below. If the OVER clause includes a window partition clause , rows are sorted within each partition. An window order clause also creates a default window frame if none is explicitly specified.

The window order clause only specifies order within a window result set. The query can have its own ORDER BY clause outside the OVER clause. This has precedence over the window order clause and orders the final result set.

Specifying window order

You specify a window frame in the analytic function's OVER clause, as shown below:

```
ORDER BY { expression [ ASC | DESC [ NULLS { FIRST | LAST | AUTO } ] ]
 }[,...]
```

For syntax details, see Window order clause .

Analytic function usage

Analytic aggregation functions such as SUM support window order clauses.

Required Usage

The following functions require a window order clause:

- RANK
- DENSE_RANK
- LEAD
- LAG
- PERCENT_RANK
- CUME_DIST
- NTILE

Invalid Usage

You cannot use a window order clause with the following functions:

- PERCENTILE_CONT
- PERCENTILE_DISC
- MEDIAN

Examples

The examples below use the allsales table schema:

```
CREATE TABLE allsales(state VARCHAR(20), name VARCHAR(20), sales INT);
INSERT INTO allsales VALUES('MA', 'A', 60);
INSERT INTO allsales VALUES('NY', 'B', 20);
INSERT INTO allsales VALUES('NY', 'C', 15);
INSERT INTO allsales VALUES('MA', 'D', 20);
INSERT INTO allsales VALUES('MA', 'E', 50);
INSERT INTO allsales VALUES('NY', 'F', 40);
INSERT INTO allsales VALUES('MA', 'G', 10);
COMMIT;
```

Example 1

The following query orders sales inside each state partition:

```
=> SELECT state, sales, name, RANK() OVER(
 PARTITION BY state ORDER BY sales) AS RANK
 FROM allsales;
 state | sales | name | RANK
-------+-------+------+----------
 MA    |    10 | G    |       1
 MA    |    20 | D    |       2
 MA    |    50 | E    |       3
 MA    |    60 | A    |       4
---------------------------------
 NY    |    15 | C    |       1
 NY    |    20 | B    |       2
 NY    |    40 | F    |       3
(7 rows)
```

Example 2

The following query's final ORDER BY clause sorts results by name:

```
=> SELECT state, sales, name, RANK() OVER(
 PARTITION BY state ORDER BY sales) AS RANK
 FROM allsales ORDER BY name;
 state | sales | name | RANK
-------+-------+------+----------
 MA    |    60 | A    |       4
 NY    |    20 | B    |       2
 NY    |    15 | C    |       1
 MA    |    20 | D    |       2
 MA    |    50 | E    |       3
 NY    |    40 | F    |       3
 MA    |    10 | G    |       1
(7 rows)
```

Window framing

The window frame of an analytic function comprises a set of rows relative to the row that is currently being evaluated by the function. After the analytic function processes that row and its window frame, Vertica advances the current row and adjusts the frame boundaries accordingly. If the OVER clause also specifies a partition , Vertica also checks that frame boundaries do not cross partition boundaries. This process repeats until the function evaluates the last row of the last partition.

Specifying a window frame

You specify a window frame in the analytic function's OVER clause, as follows:

```
{ ROWS | RANGE } { BETWEEN start-point AND end-point } | start-point
```

***start-point* / *end-point* :**

```
{ UNBOUNDED {PRECEDING | FOLLOWING}
  | CURRENT ROW
  | constant-value  {PRECEDING | FOLLOWING}}
```

start-point and *end-point* specify the window frame's offset from the current row. Keywords ROWS and RANGE specify whether the offset is physical or logical. If you specify only a start point, Vertica creates a window from that point to the current row.

For syntax details, see Window frame clause .

Requirements

In order to specify a window frame, the OVER must also specify a window order (ORDER BY) clause. If the OVER clause includes a window order clause but omits specifying a window frame, the function creates a default frame that extends from the first row in the current partition to the current row. This is equivalent to the following clause:

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

Window aggregate functions

Analytic functions that support window frames are called window aggregates. They return information such as moving averages and cumulative results. To use the following functions as window (analytic) aggregates, instead of basic aggregates, the OVER clause must specify a window order clause and, optionally, a window frame clause. If the OVER clause omits specifying a window frame, the function creates a default window frame as described earlier.

The following analytic functions support window frames:

- AVG
- COUNT
- MAX / MIN
- SUM
- STDDEV / STDDEV_POP / STDDEV_SAMP
- VARIANCE / VAR_POP / VAR_SAMP

> **Note**
> Functions FIRST_VALUE and LAST_VALUE also support window frames, but they are only analytic functions with no aggregate counterpart. EXPONENTIAL_MOVING_AVERAGE , LAG , and LEAD analytic functions do not support window frames.

A window aggregate with an empty OVER clause creates no window frame. The function is used as a reporting function, where all input is treated as a single partition.

In this section

- Windows with a physical offset (ROWS)
- Windows with a logical offset (RANGE)
- Reporting aggregates

Windows with a physical offset (ROWS)

The keyword ROWS in a window frame clause specifies window dimensions as the number of rows relative to the current row. The value can be INTEGER data type only.

> **Note**
> The value returned by an analytic function with a physical offset is liable to produce nondeterministic results unless the ordering expression results in a unique ordering. To achieve unique ordering, the window order clause might need to specify multiple columns.

Examples

The examples on this page use the emp table schema:

```
CREATE TABLE emp(deptno INT, sal INT, empno INT);
INSERT INTO emp VALUES(10,101,1);
INSERT INTO emp VALUES(10,104,4);
INSERT INTO emp VALUES(20,100,11);
INSERT INTO emp VALUES(20,109,7);
INSERT INTO emp VALUES(20,109,6);
INSERT INTO emp VALUES(20,109,8);
INSERT INTO emp VALUES(20,110,10);
INSERT INTO emp VALUES(20,110,9);
INSERT INTO emp VALUES(30,102,2);
INSERT INTO emp VALUES(30,103,3);
INSERT INTO emp VALUES(30,105,5);
COMMIT;
```

The following query invokes COUNT to count the current row and the rows preceding it, up to two rows:

```
SELECT deptno, sal, empno,  COUNT(*) OVER
      (PARTITION BY deptno ORDER BY sal ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
     AS count FROM emp;
```

The OVER clause contains three components:

- Window partition clause PARTITION BY deptno
- Order by clause ORDER BY sal
- Window frame clause ROWS BETWEEN 2 PRECEDING AND CURRENT ROW . This clause defines window dimensions as extending from the current row through the two rows that precede it.

The query returns results that are divided into three partitions, indicated below as red lines. Within the second partition (deptno=20), COUNT processes the window frame clause as follows:

1. Creates the first window (green box). This window comprises a single row, as the current row (blue box) is also the the partition's first row. Thus, the value in the count column shows the number of rows in the current window, which is 1:

deptno	sal	empno	count
10	101	1	1
10	104	4	2
20	100	11	1
20	109	7	
20	109	6	
20	109	8	
20	110	10	
20	110	9	
30	102	2	1
30	103	3	2
30	105	5	3

2. After COUNT processes the partition's first row, it resets the current row to the partition's second row. The window now spans the current row and the row above it, so COUNT returns a value of 2:

deptno	sal	empno	count
10	101	1	1
10	104	4	2
20	100	11	1
20	109	7	2
20	109	6	
20	109	8	
20	110	10	
20	110	9	
30	102	2	1
30	103	3	2
30	105	5	3

3. After COUNT processes the partition's second row, it resets the current row to the partition's third row. The window now spans the current row and the two rows above it, so COUNT returns a value of 3:

deptno	sal	empno	count
10	101	1	1
10	104	4	2
20	100	11	1
20	109	7	2
20	109	6	3
20	109	8	
20	110	10	
20	110	9	
30	102	2	1
30	103	3	2
30	105	5	3

4. Thereafter, COUNT continues to process the remaining partition rows and moves the window accordingly, but the window dimensions (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) remain unchanged as three rows. Accordingly, the value in the count column also remains unchanged (3):

deptno	sal	empno	count
10	101	1	1
10	104	4	2
20	100	11	1
20	109	7	2
20	109	6	3
20	109	8	3
20	110	10	
20	110	9	
30	102	2	1
30	103	3	2
30	105	5	3

deptno	sal	empno	count
10	101	1	1
10	104	4	2
20	100	11	1
20	109	7	2
20	109	6	3
20	109	8	3
20	110	10	3
20	110	9	
30	102	2	1
30	103	3	2
30	105	5	3

deptno	sal	empno	count
10	101	1	1
10	104	4	2
20	100	11	1
20	109	7	2
20	109	6	3
20	109	8	3
20	110	10	3
20	110	9	3
30	102	2	1
30	103	3	2
30	105	5	3

Windows with a logical offset (RANGE)

The RANGE keyword defines an analytic window frame as a logical offset from the current row.

> **Note**
> The value returned by an analytic function with a logical offset is always deterministic.

For each row, an analytic function uses the window order clause (ORDER_BY) column or expression to calculate window frame dimensions as follows:

1. Within the current partition, evaluates the ORDER_BY value of the current row against the ORDER_BY values of contiguous rows.
2. Determines which of these rows satisfy the specified range requirements relative to the current row.
3. Creates a window frame that includes only those rows.
4. Executes on the current window.

Example

This example uses the table property_sales , which contains data about neighborhood home sales:

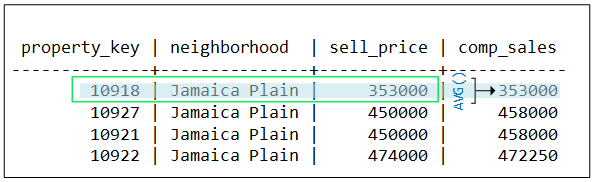
```
=> SELECT property_key, neighborhood, sell_price FROM property_sales ORDER BY neighborhood, sell_price;
 property_key | neighborhood  | sell_price
--------------+---------------+------------
        10918 | Jamaica Plain |     353000
        10921 | Jamaica Plain |     450000
        10927 | Jamaica Plain |     450000
        10922 | Jamaica Plain |     474000
        10919 | Jamaica Plain |     515000
        10917 | Jamaica Plain |     675000
        10924 | Jamaica Plain |     675000
        10920 | Jamaica Plain |     705000
        10923 | Jamaica Plain |     710000
        10926 | Jamaica Plain |     875000
        10925 | Jamaica Plain |     900000
        10930 | Roslindale    |     300000
        10928 | Roslindale    |     422000
        10932 | Roslindale    |     450000
        10929 | Roslindale    |     485000
        10931 | Roslindale    |     519000
        10938 | West Roxbury  |     479000
        10933 | West Roxbury  |     550000
        10937 | West Roxbury  |     550000
        10934 | West Roxbury  |     574000
        10935 | West Roxbury  |     598000
        10936 | West Roxbury  |     615000
        10939 | West Roxbury  |     720000
(23 rows)
```

The analytic function AVG can obtain the average of proximate selling prices within each neighborhood. The following query calculates for each home the average sale for all other neighborhood homes whose selling price was $50k higher or lower:

```
=> SELECT property_key, neighborhood, sell_price, AVG(sell_price) OVER(
    PARTITION BY neighborhood ORDER BY sell_price
    RANGE BETWEEN 50000 PRECEDING and 50000 FOLLOWING)::int AS comp_sales
    FROM property_sales ORDER BY neighborhood;
 property_key | neighborhood  | sell_price | comp_sales
--------------+---------------+------------+------------
        10918 | Jamaica Plain |     353000 |     353000
        10927 | Jamaica Plain |     450000 |     458000
        10921 | Jamaica Plain |     450000 |     458000
        10922 | Jamaica Plain |     474000 |     472250
        10919 | Jamaica Plain |     515000 |     494500
        10917 | Jamaica Plain |     675000 |     691250
        10924 | Jamaica Plain |     675000 |     691250
        10920 | Jamaica Plain |     705000 |     691250
        10923 | Jamaica Plain |     710000 |     691250
        10926 | Jamaica Plain |     875000 |     887500
        10925 | Jamaica Plain |     900000 |     887500
        10930 | Roslindale    |     300000 |     300000
        10928 | Roslindale    |     422000 |     436000
        10932 | Roslindale    |     450000 |     452333
        10929 | Roslindale    |     485000 |     484667
        10931 | Roslindale    |     519000 |     502000
        10938 | West Roxbury  |     479000 |     479000
        10933 | West Roxbury  |     550000 |     568000
        10937 | West Roxbury  |     550000 |     568000
        10934 | West Roxbury  |     574000 |     577400
        10935 | West Roxbury  |     598000 |     577400
        10936 | West Roxbury  |     615000 |     595667
        10939 | West Roxbury  |     720000 |     720000
(23 rows)
```

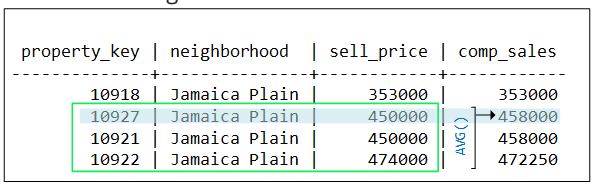
AVG processes this query as follows:

1. AVG evaluates row 1 of the first partition (Jamaica Plain), but finds no sales within $50k of this row's sell_price , ($353k). AVG creates a window that includes this row only, and returns an average of 353k for row 1:



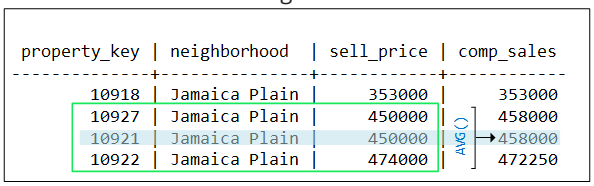
property_key	neighborhood	sell_price	comp_sales
10918	Jamaica Plain	353000	353000
10927	Jamaica Plain	450000	458000
10921	Jamaica Plain	450000	458000
10922	Jamaica Plain	474000	472250

2. AVG evaluates row 2 and finds three sell_price values within $50k of the current row. AVG creates a window that includes these three rows, and returns an average of 458k for row 2:



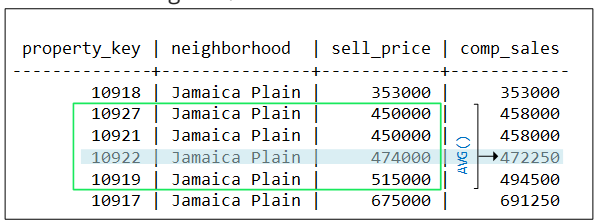
property_key	neighborhood	sell_price	comp_sales
10918	Jamaica Plain	353000	353000
10927	Jamaica Plain	450000	458000
10921	Jamaica Plain	450000	458000
10922	Jamaica Plain	474000	472250

3. AVG evaluates row 3 and finds the same three sell_price values within $50k of the current row. AVG creates a window identical to the one before, and returns the same average of 458k for row 3:



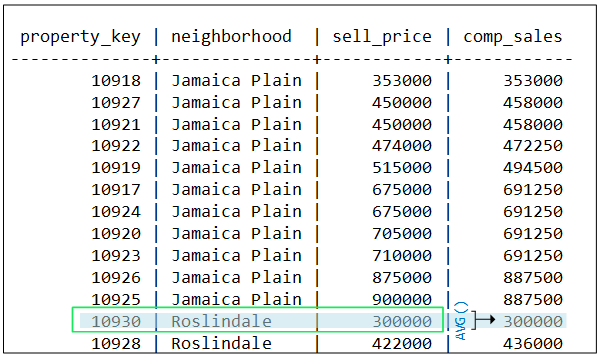
property_key	neighborhood	sell_price	comp_sales
10918	Jamaica Plain	353000	353000
10927	Jamaica Plain	450000	458000
10921	Jamaica Plain	450000	458000
10922	Jamaica Plain	474000	472250

4. AVG evaluates row 4 and finds four sell_price values within $50k of the current row. AVG expands its window to include rows 2 through 5, and returns an average of $472.25k for row 4:



property_key	neighborhood	sell_price	comp_sales
10918	Jamaica Plain	353000	353000
10927	Jamaica Plain	450000	458000
10921	Jamaica Plain	450000	458000
10922	Jamaica Plain	474000	472250
10919	Jamaica Plain	515000	494500
10917	Jamaica Plain	675000	691250

5. In similar fashion, AVG evaluates the remaining rows in this partition. When the function evaluates the first row of the second partition (Roslindale), it resets the window as follows:



property_key	neighborhood	sell_price	comp_sales
10918	Jamaica Plain	353000	353000
10927	Jamaica Plain	450000	458000
10921	Jamaica Plain	450000	458000
10922	Jamaica Plain	474000	472250
10919	Jamaica Plain	515000	494500
10917	Jamaica Plain	675000	691250
10924	Jamaica Plain	675000	691250
10920	Jamaica Plain	705000	691250
10923	Jamaica Plain	710000	691250
10926	Jamaica Plain	875000	887500
10925	Jamaica Plain	900000	887500
10930	Roslindale	300000	300000
10928	Roslindale	422000	436000

Restrictions

If RANGE specifies a constant value, that value's data type and the window's ORDER BY data type must be the same. The following exceptions apply:

- RANGE can specify INTERVAL Year to Month if the window order clause data type is one of following: TIMESTAMP , TIMESTAMP WITH TIMEZONE , or DATE . TIME and TIME WITH TIMEZONE are not supported.
- RANGE can specify INTERVAL Day to Second if the window order clause data is one of following: TIMESTAMP , TIMESTAMP WITH TIMEZONE , DATE , TIME , or TIME WITH TIMEZONE .

The window order clause must specify one of the following data types: NUMERIC , DATE/TIME , FLOAT or INTEGER . This requirement is ignored if the window specifies one of following frames:

- RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
- RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

Reporting aggregates

Some of the analytic functions that take the window-frame-clause are the reporting aggregates. These functions let you compare a partition's aggregate values with detail rows, taking the place of correlated subqueries or joins.

- AVG()
- COUNT()
- MAX() and MIN()
- SUM()
- STDDEV() , STDDEV_POP() , and STDDEV_SAMP()
- VARIANCE() , VAR_POP() , and VAR_SAMP()

If you use a window aggregate with an empty OVER() clause, the analytic function is used as a reporting function, where the entire input is treated as a single partition.

About standard deviation and variance functions

With standard deviation functions, a low standard deviation indicates that the data points tend to be very close to the mean, whereas high standard deviation indicates that the data points are spread out over a large range of values.

Standard deviation is often graphed and a distributed standard deviation creates the classic bell curve.

Variance functions measure how far a set of numbers is spread out.

Examples

Think of the window for reporting aggregates as a window defined as UNBOUNDED PRECEDING and UNBOUNDED FOLLOWING. The omission of a window-order-clause makes all rows in the partition also the window (reporting aggregates).

```
=> SELECT deptno, sal, empno, COUNT(sal)
  OVER (PARTITION BY deptno) AS COUNT FROM emp;
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     2
     10 | 104 |     4 |     2
-------------------------------
     20 | 110 |    10 |     6
     20 | 110 |     9 |     6
     20 | 109 |     7 |     6
     20 | 109 |     6 |     6
     20 | 109 |     8 |     6
     20 | 100 |    11 |     6
-------------------------------
     30 | 105 |     5 |     3
     30 | 103 |     3 |     3
     30 | 102 |     2 |     3
(11 rows)
```

If the OVER() clause in the above query contained a window-order-clause (for example, ORDER BY sal), it would become a moving window (window aggregate) query with a default window of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW:

```
=> SELECT deptno, sal, empno, COUNT(sal)  OVER (PARTITION BY deptno ORDER BY sal) AS COUNT FROM emp;
 deptno | sal | empno | count
--------+-----+-------+-------
    10 | 101 |     1 |     1
    10 | 104 |     4 |     2
-----------------------------
    20 | 100 |    11 |     1
    20 | 109 |     7 |     4
    20 | 109 |     6 |     4
    20 | 109 |     8 |     4
    20 | 110 |    10 |     6
    20 | 110 |     9 |     6
-----------------------------
    30 | 102 |     2 |     1
    30 | 103 |     3 |     2
    30 | 105 |     5 |     3
(11 rows)
```

What about LAST_VALUE?

You might wonder why you couldn't just use the LAST_VALUE() analytic function.

For example, for each employee, get the highest salary in the department:

```
=> SELECT deptno, sal, empno,LAST_VALUE(empno) OVER (PARTITION BY deptno ORDER BY sal) AS lv FROM emp;
```


Due to default window semantics, LAST_VALUE does not always return the last value of a partition. If you omit the window-frame-clause from the analytic clause, LAST_VALUE operates on this default window. Results, therefore, can seem non-intuitive because the function does not return the bottom of the current partition. It returns the bottom of the window, which continues to change along with the current input row being processed.

Remember the default window:

```
OVER (PARTITION BY deptno ORDER BY sal)
```

is the same as:

```
OVER(PARTITION BY deptno ORDER BY salROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```

```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  1
     10 | 104 |     4 |  4
     20 | 100 |    11 | 11
     20 | 109 |     7 |
     20 | 109 |     6 |
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |  2
     30 | 103 |     3 |  3
     30 | 105 |     5 |  5
```

```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  1
     10 | 104 |     4 |  4
     20 | 100 |    11 | 11
     20 | 109 |     7 |  7
     20 | 109 |     6 |
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |  2
     30 | 103 |     3 |  3
     30 | 105 |     5 |  5
```

```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  1
     10 | 104 |     4 |  4
     20 | 100 |    11 | 11
     20 | 109 |     7 |  7
     20 | 109 |     6 |  7
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |  2
     30 | 103 |     3 |  3
     30 | 105 |     5 |  5
```

```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  1
     10 | 104 |     4 |  4
     20 | 100 |    11 | 11
     20 | 109 |     7 |  7
     20 | 109 |     6 |  7
     20 | 109 |     8 |  7
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |  2
     30 | 103 |     3 |  3
     30 | 105 |     5 |  5
```

deptno	sal	empno	lv
10	101	1	1
10	104	4	4
20	100	11	11
20	109	7	7
20	109	6	7
20	109	8	7
20	110	10	10
20	110	9	
30	102	2	2
30	103	3	3
30	105	5	5

deptno	sal	empno	lv
10	101	1	1
10	104	4	4
20	100	11	11
20	109	7	7
20	109	6	7
20	109	8	7
20	110	10	10
20	110	9	10
30	102	2	2
30	103	3	3
30	105	5	5

If you want to return the last value of a partition, use UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

```
=> SELECT deptno, sal, empno, LAST_VALUE(empno)
OVER (PARTITION BY deptno ORDER BY sal
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS lv
FROM emp;
```


Vertica recommends that you use LAST_VALUE with the window-order-clause to produce deterministic results.

In the following example, empno 6, 7, and 8 have the same salary, so they are in adjacent rows. empno 8 appears first in this case but the order is not guaranteed.

deptno	sal	empno	lv
10	101	1	4
10	104	4	4
20	100	11	7
20	109	8	7
20	109	6	7
20	109	7	7
30	102	2	5
30	103	3	5
30	105	5	5

Notice in the output above, the last value is 7, which is the last row from the partition deptno = 20. If the rows have a different order, then the function returns a different value:

deptno	sal	empno	lv
10	101	1	4
10	104	4	4
20	100	11	6
20	109	8	6
20	109	7	6
20	109	6	6
30	102	2	5
30	103	3	5
30	105	5	5

Now the last value is 6, which is the last row from the partition deptno = 20. The solution is to add a unique key to the sort order. Even if the order of the query changes, the result will always be the same, and so deterministic.

```
=> SELECT deptno, sal, empno, LAST_VALUE(empno)
OVER (PARTITION BY deptno ORDER BY sal, empno
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as lv
FROM emp;
```

```
deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  4
     10 | 104 |     4 |  4
     20 | 100 |    11 |  8
     20 | 109 |     6 |  8
     20 | 109 |     7 |  8
     20 | 109 |     8 |  8
     30 | 102 |     2 |  5
     30 | 103 |     3 |  5
     30 | 105 |     5 |  5
```

Notice how the rows are now ordered by empno , the last value stays at 8, and it does not matter the order of the query.

Named windows

An analytic function's OVER clause can reference a named window, which encapsulates one or more window clauses: a window partition (PARTITION BY) clause and (optionally) a window order (ORDER BY) clause. Named windows can be useful when you write queries that invoke multiple analytic functions with similar OVER clause syntax—for example, they use the same partition clauses.

A query names a window as follows:

```
WINDOW window-name AS ( window-partition-clause [window-order-clause] );
```

The same query can name and reference multiple windows. All window names must be unique within the same query.

Examples

The following query invokes two analytic functions, RANK and DENSE_RANK . Because the two functions use the same partition and order clauses, the query names a window w that specifies both clauses. The two functions reference this window as follows:

```
=> SELECT employee_region region, employee_key, annual_salary,
    RANK() OVER w Rank,
    DENSE_RANK() OVER w "Dense Rank"
    FROM employee_dimension WINDOW w AS (PARTITION BY employee_region ORDER BY annual_salary);
          region           | employee_key | annual_salary | Rank | Dense Rank
-----------------------------------+--------------+---------------+------+------------
 West                              |         5248 |          1200 |    1 |          1
 West                              |         6880 |          1204 |    2 |          2
 West                              |         5700 |          1214 |    3 |          3
 West                              |         9857 |          1218 |    4 |          4
 West                              |         6014 |          1218 |    4 |          4
 West                              |         9221 |          1220 |    6 |          5
 West                              |         7646 |          1222 |    7 |          6
 West                              |         6621 |          1222 |    7 |          6
 West                              |         6488 |          1224 |    9 |          7
 West                              |         7659 |          1226 |   10 |          8
 West                              |         7432 |          1226 |   10 |          8
 West                              |         9905 |          1226 |   10 |          8
 West                              |         9021 |          1228 |   13 |          9
 West                              |         7855 |          1228 |   13 |          9
 West                              |         7119 |          1230 |   15 |         10
...
```

If the named window omits an order clause, the query's OVER clauses can specify their own order clauses. For example, you can modify the previous query so each function uses a different order clause. The named window is defined so it includes only a partition clause:

```
=> SELECT employee_region region, employee_key, annual_salary,
   RANK() OVER (w ORDER BY annual_salary DESC) Rank,
   DENSE_RANK() OVER (w ORDER BY annual_salary ASC) "Dense Rank"
   FROM employee_dimension WINDOW w AS (PARTITION BY employee_region);
         region          | employee_key | annual_salary | Rank | Dense Rank
---------------------------+--------------+---------------+------+------------
 West                      |         5248 |          1200 | 2795 |          1
 West                      |         6880 |          1204 | 2794 |          2
 West                      |         5700 |          1214 | 2793 |          3
 West                      |         6014 |          1218 | 2791 |          4
 West                      |         9857 |          1218 | 2791 |          4
 West                      |         9221 |          1220 | 2790 |          5
 West                      |         6621 |          1222 | 2788 |          6
 West                      |         7646 |          1222 | 2788 |          6
 West                      |         6488 |          1224 | 2787 |          7
 West                      |         7432 |          1226 | 2784 |          8
 West                      |         9905 |          1226 | 2784 |          8
 West                      |         7659 |          1226 | 2784 |          8
 West                      |         7855 |          1228 | 2782 |          9
 West                      |         9021 |          1228 | 2782 |          9
 West                      |         7119 |          1230 | 2781 |         10
...
```

Similarly, an OVER clause specifies a named window can also specify a window frame clause , provided the named window includes an order clause. This can be useful inasmuch as you cannot define a named windows to include a window frame clause.

For example, the following query defines a window that encapsulates partitioning and order clauses. The OVER clause invokes this window and also includes a window frame clause:

```
=> SELECT deptno, sal, empno,  COUNT(*) OVER (w ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS count
 FROM emp WINDOW w AS (PARTITION BY deptno ORDER BY sal);
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     8 |     2
     20 | 109 |     6 |     3
     20 | 109 |     7 |     3
     20 | 110 |    10 |     3
     20 | 110 |     9 |     3
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
(11 rows)
```

Recursive window references

A WINDOW clause can reference another window that is already named. For example, because named window w1 is defined before w2 , the WINDOW clause that defines w2 can reference w1 :

```
=> SELECT RANK() OVER(w1 ORDER BY sal DESC), RANK() OVER w2
  FROM EMP WINDOW w1 AS (PARTITION BY deptno), w2 AS (w1 ORDER BY sal);
```

Restrictions

- An OVER clause can reference only one named window.
- Each WINDOW clause within the same query must have a unique name.

Analytic query examples

The topics in this section show how to use analytic queries for calculations.

In this section

- Calculating a median value
- Getting price differential for two stocks
- Calculating the moving average
- Getting latest bid and ask results

Calculating a median value

A median is a numerical value that separates the higher half of a sample from the lower half. For example, you can retrieve the median of a finite list of numbers by arranging all observations from lowest value to highest value and then picking the middle one.

If the number of observations is even, then there is no single middle value; the median is the mean (average) of the two middle values.

The following example uses this table:

```
CREATE TABLE allsales(state VARCHAR(20), name VARCHAR(20), sales INT);
INSERT INTO allsales VALUES('MA', 'A', 60);
INSERT INTO allsales VALUES('NY', 'B', 20);
INSERT INTO allsales VALUES('NY', 'C', 15);
INSERT INTO allsales VALUES('MA', 'D', 20);
INSERT INTO allsales VALUES('MA', 'E', 50);
INSERT INTO allsales VALUES('NY', 'F', 40);
INSERT INTO allsales VALUES('MA', 'G', 10);
COMMIT;
```

You can use the analytic function MEDIAN to calculate the median of all sales in this table. In the following query, the function's OVER clause is empty, so the query returns the same aggregated value for each row of the result set:

```
=> SELECT name, sales, MEDIAN(sales) OVER() AS median FROM allsales;
 name | sales | median
------+-------+--------
 G    |    10 |     20
 C    |    15 |     20
 D    |    20 |     20
 B    |    20 |     20
 F    |    40 |     20
 E    |    50 |     20
 A    |    60 |     20
(7 rows)
```

You can modify this query to group sales by state and obtain the median for each one. To do so, include a window partition clause in the OVER clause:

```
=> SELECT state, name, sales, MEDIAN(sales) OVER(partition by state) AS median FROM allsales;
 state | name | sales | median
-------+------+-------+--------
 MA    | G    |    10 |     35
 MA    | D    |    20 |     35
 MA    | E    |    50 |     35
 MA    | A    |    60 |     35
 NY    | C    |    15 |     20
 NY    | B    |    20 |     20
 NY    | F    |    40 |     20
(7 rows)
```

Getting price differential for two stocks

The following subquery selects out two stocks of interest. The outer query uses the LAST_VALUE() and OVER() components of analytics, with IGNORE NULLS.

Schema

```
DROP TABLE Ticks CASCADE;


CREATE TABLE Ticks (ts TIMESTAMP, Stock varchar(10), Bid float);
INSERT INTO Ticks VALUES('2011-07-12 10:23:54', 'abc', 10.12);
INSERT INTO Ticks VALUES('2011-07-12 10:23:58', 'abc', 10.34);
INSERT INTO Ticks VALUES('2011-07-12 10:23:59', 'abc', 10.75);
INSERT INTO Ticks VALUES('2011-07-12 10:25:15', 'abc', 11.98);
INSERT INTO Ticks VALUES('2011-07-12 10:25:16', 'abc');
INSERT INTO Ticks VALUES('2011-07-12 10:25:22', 'xyz', 45.16);
INSERT INTO Ticks VALUES('2011-07-12 10:25:27', 'xyz', 49.33);
INSERT INTO Ticks VALUES('2011-07-12 10:31:12', 'xyz', 65.25);
INSERT INTO Ticks VALUES('2011-07-12 10:31:15', 'xyz');

COMMIT;
```

Ticks table

```
=> SELECT * FROM ticks;
        ts          | stock |  bid
--------------------+-------+-------
 2011-07-12 10:23:59 | abc   | 10.75
 2011-07-12 10:25:22 | xyz   | 45.16
 2011-07-12 10:23:58 | abc   | 10.34
 2011-07-12 10:25:27 | xyz   | 49.33
 2011-07-12 10:23:54 | abc   | 10.12
 2011-07-12 10:31:15 | xyz   |
 2011-07-12 10:25:15 | abc   | 11.98
 2011-07-12 10:25:16 | abc   |
 2011-07-12 10:31:12 | xyz   | 65.25
(9 rows)
```

Query

```
 => SELECT ts, stock, bid, last_value(price1 IGNORE NULLS)
   OVER(ORDER BY ts) - last_value(price2 IGNORE NULLS)
   OVER(ORDER BY ts)   as price_diff
 FROM
  (SELECT ts, stock, bid,
    CASE WHEN stock = 'abc' THEN bid ELSE NULL END AS price1,
    CASE WHEN stock = 'xyz' then bid ELSE NULL END AS price2
    FROM ticks
    WHERE stock IN ('abc','xyz')
  ) v1
 ORDER BY ts;
        ts          | stock |  bid  | price_diff
--------------------+-------+-------+------------
 2011-07-12 10:23:54 | abc   | 10.12 |
 2011-07-12 10:23:58 | abc   | 10.34 |
 2011-07-12 10:23:59 | abc   | 10.75 |
 2011-07-12 10:25:15 | abc   | 11.98 |
 2011-07-12 10:25:16 | abc   |       |
 2011-07-12 10:25:22 | xyz   | 45.16 |     -33.18
 2011-07-12 10:25:27 | xyz   | 49.33 |     -37.35
 2011-07-12 10:31:12 | xyz   | 65.25 |     -53.27
 2011-07-12 10:31:15 | xyz   |       |     -53.27
(9 rows)
```

Calculating the moving average

Calculating the moving average is useful to get an estimate about the trends in a data set. The moving average is the average of any subset of numbers over a period of time. For example, if you have retail data that spans over ten years, you could calculate a three year moving average, a four year moving average, and so on. This example calculates a 40-second moving average of bids for one stock. This examples uses the ticks table schema.

Query

```
=> SELECT ts, bid, AVG(bid)
  OVER(ORDER BY ts
     RANGE BETWEEN INTERVAL '40 seconds'
     PRECEDING AND CURRENT ROW)
FROM ticks
WHERE stock = 'abc'
GROUP BY bid, ts
ORDER BY ts;
        ts          |  bid  |     ?column?
--------------------+-------+------------------
 2011-07-12 10:23:54 | 10.12 |            10.12
 2011-07-12 10:23:58 | 10.34 |            10.23
 2011-07-12 10:23:59 | 10.75 | 10.4033333333333
 2011-07-12 10:25:15 | 11.98 |            11.98
 2011-07-12 10:25:16 |       |           11.98
(5 rows)
```

```
DROP TABLE Ticks CASCADE;



CREATE TABLE Ticks (ts TIMESTAMP, Stock varchar(10), Bid float);
INSERT INTO Ticks VALUES('2011-07-12 10:23:54', 'abc', 10.12);
INSERT INTO Ticks VALUES('2011-07-12 10:23:58', 'abc', 10.34);
INSERT INTO Ticks VALUES('2011-07-12 10:23:59', 'abc', 10.75);
INSERT INTO Ticks VALUES('2011-07-12 10:25:15', 'abc', 11.98);
INSERT INTO Ticks VALUES('2011-07-12 10:25:16', 'abc');
INSERT INTO Ticks VALUES('2011-07-12 10:25:22', 'xyz', 45.16);
INSERT INTO Ticks VALUES('2011-07-12 10:25:27', 'xyz', 49.33);
INSERT INTO Ticks VALUES('2011-07-12 10:31:12', 'xyz', 65.25);
INSERT INTO Ticks VALUES('2011-07-12 10:31:15', 'xyz');

COMMIT;
```

Getting latest bid and ask results

The following query fills in missing NULL values to create a full book order showing latest bid and ask price and size, by vendor id. Original rows have values for (typically) one price and one size, so use last_value with "ignore nulls" to find the most recent non-null value for the other pair each time there is an entry for the ID. Sequenceno provides a unique total ordering.

Schema:

```
=> CREATE TABLE bookorders(
   vendorid VARCHAR(100),
   date TIMESTAMP,
   sequenceno INT,
   askprice FLOAT,
   asksize INT,
   bidprice FLOAT,
   bidsize INT);
=> INSERT INTO bookorders VALUES('3325XPK','2011-07-12 10:23:54', 1, 10.12, 55, 10.23, 59);
=> INSERT INTO bookorders VALUES('3345XPZ','2011-07-12 10:23:55', 2, 10.55, 58, 10.75, 57);
=> INSERT INTO bookorders VALUES('445XPKF','2011-07-12 10:23:56', 3, 10.22, 43, 54);
=> INSERT INTO bookorders VALUES('445XPKF','2011-07-12 10:23:57', 3, 10.22, 59, 10.25, 61);
=> INSERT INTO bookorders VALUES('3425XPY','2011-07-12 10:23:58', 4, 11.87, 66, 11.90, 66);
=> INSERT INTO bookorders VALUES('3727XVK','2011-07-12 10:23:59', 5, 11.66, 51, 11.67, 62);
=> INSERT INTO bookorders VALUES('5325XYZ','2011-07-12 10:24:01', 6, 15.05, 44, 15.10, 59);
=> INSERT INTO bookorders VALUES('3675XVS','2011-07-12 10:24:05', 7, 15.43, 47, 58);
=> INSERT INTO bookorders VALUES('8972VUG','2011-07-12 10:25:15', 8, 14.95, 52, 15.11, 57);
COMMIT;
```

Query:

```
=> SELECT
   sequenceno Seq,
   date "Time",
   vendorid ID,
   LAST_VALUE (bidprice IGNORE NULLS)
    OVER (PARTITION BY vendorid ORDER BY sequenceno
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
   AS "Bid Price",
   LAST_VALUE (bidsize IGNORE NULLS)
    OVER (PARTITION BY vendorid ORDER BY sequenceno
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
   AS "Bid Size",
   LAST_VALUE (askprice IGNORE NULLS)
    OVER (PARTITION BY vendorid ORDER BY sequenceno
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
   AS "Ask Price",
   LAST_VALUE (asksize IGNORE NULLS)
    OVER (PARTITION BY vendorid order by sequenceno
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
   AS  "Ask Size"
 FROM bookorders
ORDER BY sequenceno;
 Seq |        Time         |   ID    | Bid Price | Bid Size | Ask Price | Ask Size
-----+---------------------+---------+-----------+----------+-----------+----------
   1 | 2011-07-12 10:23:54 | 3325XPK |     10.23 |       59 |     10.12 |       55
   2 | 2011-07-12 10:23:55 | 3345XPZ |     10.75 |       57 |     10.55 |       58
   3 | 2011-07-12 10:23:57 | 445XPKF |     10.25 |       61 |     10.22 |       59
   3 | 2011-07-12 10:23:56 | 445XPKF |      54 |          |     10.22 |       43
   4 | 2011-07-12 10:23:58 | 3425XPY |      11.9 |       66 |     11.87 |       66
   5 | 2011-07-12 10:23:59 | 3727XVK |     11.67 |       62 |     11.66 |       51
   6 | 2011-07-12 10:24:01 | 5325XYZ |      15.1 |       59 |     15.05 |       44
   7 | 2011-07-12 10:24:05 | 3675XVS |      58 |          |     15.43 |       47
   8 | 2011-07-12 10:25:15 | 8972VUG |     15.11 |       57 |     14.95 |       52
(9 rows)
```

Event-based windows

Event-based windows let you break time series data into windows that border on significant events within the data. This is especially relevant in financial data where analysis often focuses on specific events as triggers to other activity.

Vertica provides two event-based window functions that are not part of the SQL-99 standard:

- CONDITIONAL_CHANGE_EVENT assigns an event window number to each row, starting from 0, and increments by 1 when the result of evaluating the argument expression on the current row differs from that on the previous row. This function is similar to the analytic function ROW_NUMBER , which assigns a unique number, sequentially, starting from 1, to each row within a partition.
- CONDITIONAL_TRUE_EVENT assigns an event window number to each row, starting from 0, and increments the number by 1 when the result of the boolean argument expression evaluates true.

Both functions are described in greater detail below.

> **Note**
> CONDITIONAL_CHANGE_EVENT and CONDITIONAL_TRUE_EVENT do not support window framing .

Example schema

The examples on this page use the following schema:

```
CREATE TABLE TickStore3 (ts TIMESTAMP, symbol VARCHAR(8), bid FLOAT);
CREATE PROJECTION TickStore3_p (ts, symbol, bid) AS SELECT * FROM TickStore3 ORDER BY ts, symbol, bid UNSEGMENTED ALL NODES;
INSERT INTO TickStore3 VALUES ('2009-01-01 03:00:00', 'XYZ', 10.0);
INSERT INTO TickStore3 VALUES ('2009-01-01 03:00:03', 'XYZ', 11.0);
INSERT INTO TickStore3 VALUES ('2009-01-01 03:00:06', 'XYZ', 10.5);
INSERT INTO TickStore3 VALUES ('2009-01-01 03:00:09', 'XYZ', 11.0);
COMMIT;
```

Using CONDITIONAL_CHANGE_EVENT

The analytical function CONDITIONAL_CHANGE_EVENT returns a sequence of integers indicating event window numbers, starting from 0. The function increments the event window number when the result of evaluating the function expression on the current row differs from the previous value.

In the following example, the first query returns all records from the TickStore3 table. The second query uses the CONDITIONAL_CHANGE_EVENT function on the bid column. Since each bid row value is different from the previous value, the function increments the window ID from 0 to 3:

```
SELECT ts, symbol, bidFROM Tickstore3 ORDER BY ts;
```

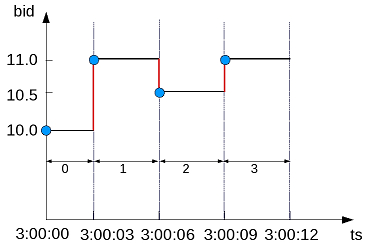
```
SELECT CONDITIONAL_CHANGE_EVENT(bid)
  OVER(ORDER BY ts) FROM Tickstore3;
```

```
        ts         | symbol | bid
--------------------+--------+------
2009-01-01 03:00:00 | XYZ    |   10
2009-01-01 03:00:03 | XYZ    |   11
2009-01-01 03:00:06 | XYZ    | 10.5
2009-01-01 03:00:09 | XYZ    |   11
(4 rows)
```

==>

```
        ts         | symbol | bid  | cce
--------------------+--------+------+-----
2009-01-01 03:00:00 | XYZ    |   10 | 0
2009-01-01 03:00:03 | XYZ    |   11 | 1
2009-01-01 03:00:06 | XYZ    | 10.5 | 2
2009-01-01 03:00:09 | XYZ    |   11 | 3
(4 rows)
```

The following figure is a graphical illustration of the change in the bid price. Each value is different from its previous one, so the window ID increments for each time slice:



So the window ID starts at 0 and increments at every change in from the previous value.

In this example, the bid price changes from $10 to $11 in the second row, but then stays the same. CONDITIONAL_CHANGE_EVENT increments the event window ID in row 2, but not subsequently:

```
SELECT ts, symbol, bidFROM Tickstore3 ORDER BY ts;
```

```
SELECT CONDITIONAL_CHANGE_EVENT(bid)
  OVER(ORDER BY ts) FROM Tickstore3;
```

```
        ts         | symbol | bid
--------------------+--------+------
2009-01-01 03:00:00 | XYZ    |   10
2009-01-01 03:00:03 | XYZ    |   11
2009-01-01 03:00:06 | XYZ    |   11
2009-01-01 03:00:09 | XYZ    |   11
```

==>

```
        ts         | symbol | bid  | cce
--------------------+--------+------+-----
2009-01-01 03:00:00 | XYZ    |   10 | 0
2009-01-01 03:00:03 | XYZ    |   11 | 1
2009-01-01 03:00:06 | XYZ    |   11 | 1
2009-01-01 03:00:09 | XYZ    |   11 | 1
```

The following figure is a graphical illustration of the change in the bid price at 3:00:03 only. The price stays the same at 3:00:06 and 3:00:09, so the window ID remains at 1 for each time slice after the change:

```
![CCE 2](/images/cce2.png)
```

Using CONDITIONAL_TRUE_EVENT

Like CONDITIONAL_CHANGE_EVENT , the analytic function CONDITIONAL_TRUE_EVENT also returns a sequence of integers indicating event window numbers, starting from 0. The two functions differ as follows:

- CONDITIONAL_TRUE_EVENT increments the window ID each time its expression evaluates to true.
- CONDITIONAL_CHANGE_EVENT increments on a comparison expression with the previous value.

In the following example, the first query returns all records from the TickStore3 table. The second query uses CONDITIONAL_TRUE_EVENT to test whether the current bid is greater than a given value (10.6). Each time the expression tests true, the function increments the window ID. The first time the function increments the window ID is on row 2, when the value is 11. The expression tests false for the next row (value is not greater than 10.6), so the function does not increment the event window ID. In the final row, the expression is true for the given condition, and the function increments the window:

```
SELECT ts, symbol, bidFROM Tickstore3 ORDER BY ts;
```

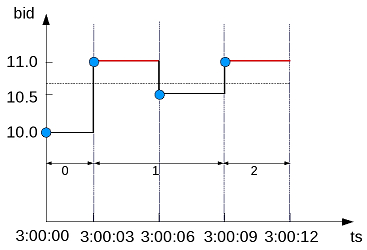
```
SELECT CONDITIONAL_TRUE_EVENT(bid > 10.6)
 OVER(ORDER BY ts) FROM Tickstore3;
```

```
        ts          | symbol | bid
--------------------+--------+------
2009-01-01 03:00:00 | XYZ    |   10
2009-01-01 03:00:03 | XYZ    |   11
2009-01-01 03:00:06 | XYZ    | 10.5
2009-01-01 03:00:09 | XYZ    |   11
```

==>

```
        ts          | symbol | bid  | cte---------------------+--------+------+-----
2009-01-01 03:00:00 | XYZ    |   10 | 0
2009-01-01 03:00:03 | XYZ    |   11 | 1
2009-01-01 03:00:06 | XYZ    | 10.5 | 1
2009-01-01 03:00:09 | XYZ    |   11 | 2
```

The following figure is a graphical illustration that shows the bid values and window ID changes. Because the bid value is greater than $10.6 on only the second and fourth time slices (3:00:03 and 3:00:09), the window ID returns <0, **1** ,1, **2** >:



In the following example, the first query returns all records from the TickStore3 table, ordered by the tickstore values (ts). The second query uses CONDITIONAL_TRUE_EVENT to increment the window ID each time the bid value is greater than 10.6. The first time the function increments the event window ID is on row 2, where the value is 11. The window ID then increments each time after that, because the expression (bid > 10.6) tests true for each time slice:

```
SELECT ts, symbol, bidFROM Tickstore3 ORDER BY ts;
```

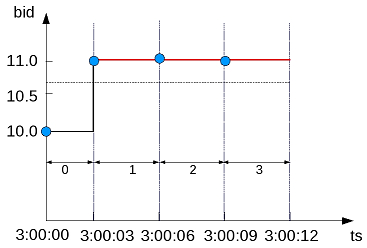
```
SELECT CONDITIONAL_TRUE_EVENT(bid > 10.6)
 OVER(ORDER BY ts) FROM Tickstore3;
```

```
        ts          | symbol | bid
--------------------+--------+------
2009-01-01 03:00:00 | XYZ    |   10
2009-01-01 03:00:03 | XYZ    |   11
2009-01-01 03:00:06 | XYZ    |   11
2009-01-01 03:00:09 | XYZ    |   11
```

==>

```
        ts          | symbol | bid  | cte
--------------------+--------+------+-----
2009-01-01 03:00:00 | XYZ    |   10 | 0
2009-01-01 03:00:03 | XYZ    |   11 | 1
2009-01-01 03:00:06 | XYZ    |   11 | 2
2009-01-01 03:00:09 | XYZ    |   11 | 3
```

The following figure is a graphical illustration that shows the bid values and window ID changes. The bid value is greater than 10.6 on the second time slice (3:00:03) and remains for the remaining two time slices. The function increments the event window ID each time because the expression tests true:



Advanced use of event-based windows

In event-based window functions, the condition expression accesses values from the current row only. To access a previous value, you can use a more powerful event-based window that allows the window event condition to include previous data points. For example, analytic function LAG (x, n) retrieves the value of column x in the *n* th to last input record. In this case, LAG shares the OVER specifications of the CONDITIONAL_CHANGE_EVENT or CONDITIONAL_TRUE_EVENT function expression.

In the following example, the first query returns all records from the TickStore3 table. The second query uses CONDITIONAL_TRUE_EVENT with the LAG function in its boolean expression. In this case, CONDITIONAL_TRUE_EVENT increments the event window ID each time the bid value on the current row is less than the previous value. The first time CONDITIONAL_TRUE_EVENT increments the window ID starts on the third time slice, when the expression tests true. The current value (10.5) is less than the previous value. The window ID is not incremented in the last row because the final value is greater than the previous row:

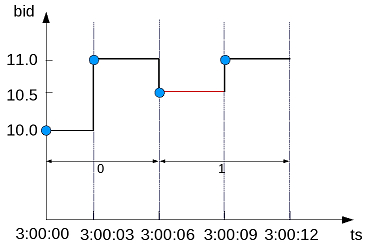
```
SELECT ts, symbol, bidFROM Tickstore3 ORDER BY ts;
```

```
SELECT CONDITIONAL_TRUE_EVENT(bid < LAG(bid))
 OVER(ORDER BY ts) FROM Tickstore;
```

```
        ts          | symbol | bid
--------------------+--------+------
2009-01-01 03:00:00 | XYZ    |   10
2009-01-01 03:00:03 | XYZ    |   11
2009-01-01 03:00:06 | XYZ    | 10.5
2009-01-01 03:00:09 | XYZ    |   11
```

```
        ts          | symbol | bid  | cte
--------------------+--------+------+-----
2009-01-01 03:00:00 | XYZ    |   10 | 0
2009-01-01 03:00:03 | XYZ    |   11 | 0
2009-01-01 03:00:06 | XYZ    | 10.5 | 1
2009-01-01 03:00:09 | XYZ    |   11 | 1
```

The following figure illustrates the second query above. When the bid price is less than the previous value, the window ID gets incremented, which occurs only in the third time slice (3:00:06):



See also

- Sessionization with event-based windows
- Time series analytics

Sessionization with event-based windows

Sessionization, a special case of event-based windows, is a feature often used to analyze click streams, such as identifying web browsing sessions from recorded web clicks.

In Vertica, given an input clickstream table, where each row records a Web page click made by a particular user (or IP address), the sessionization computation attempts to identify Web browsing sessions from the recorded clicks by grouping the clicks from each user based on the time-intervals between the clicks. If two clicks from the same user are made too far apart in time, as defined by a time-out threshold, the clicks are treated as though they are from two different browsing sessions.

Example Schema The examples in this topic use the following WebClicks schema to represent a simple clickstream table:

```
CREATE TABLE WebClicks(userId INT, timestamp TIMESTAMP);
INSERT INTO WebClicks VALUES (1, '2009-12-08 3:00:00 pm');
INSERT INTO WebClicks VALUES (1, '2009-12-08 3:00:25 pm');
INSERT INTO WebClicks VALUES (1, '2009-12-08 3:00:45 pm');
INSERT INTO WebClicks VALUES (1, '2009-12-08 3:01:45 pm');
INSERT INTO WebClicks VALUES (2, '2009-12-08 3:02:45 pm');
INSERT INTO WebClicks VALUES (2, '2009-12-08 3:02:55 pm');
INSERT INTO WebClicks VALUES (2, '2009-12-08 3:03:55 pm');
COMMIT;
```

The input table WebClicks contains the following rows:

```
=> SELECT * FROM WebClicks;
 userId |      timestamp
--------+---------------------
      1 | 2009-12-08 15:00:00
      1 | 2009-12-08 15:00:25
      1 | 2009-12-08 15:00:45
      1 | 2009-12-08 15:01:45
      2 | 2009-12-08 15:02:45
      2 | 2009-12-08 15:02:55
      2 | 2009-12-08 15:03:55
(7 rows)
```

In the following query, sessionization performs computation on the SELECT list columns, showing the difference between the current and previous timestamp value using LAG() . It evaluates to true and increments the window ID when the difference is greater than 30 seconds.

```
=> SELECT userId, timestamp,
    CONDITIONAL_TRUE_EVENT(timestamp - LAG(timestamp) > '30 seconds')
    OVER(PARTITION BY userId ORDER BY timestamp) AS session FROM WebClicks;
 userId |      timestamp      | session
--------+---------------------+---------
      1 | 2009-12-08 15:00:00 |       0
      1 | 2009-12-08 15:00:25 |       0
      1 | 2009-12-08 15:00:45 |       0
      1 | 2009-12-08 15:01:45 |       1
      2 | 2009-12-08 15:02:45 |       0
      2 | 2009-12-08 15:02:55 |       0
      2 | 2009-12-08 15:03:55 |       1
(7 rows)
```

In the output, the session column contains the window ID from the CONDITIONAL_TRUE_EVENT function. The window ID evaluates to true on row 4 (timestamp 15:01:45), and the ID that follows row 4 is zero because it is the start of a new partition (for user ID 2), and that row does not evaluate to true until the last line in the output.

You might want to give users different time-out thresholds. For example, one user might have a slower network connection or be multi-tasking, while another user might have a faster connection and be focused on a single Web site, doing a single task.

To compute an adaptive time-out threshold based on the last 2 clicks, use CONDITIONAL_TRUE_EVENT with LAG to return the average time between the last 2 clicks with a grace period of 3 seconds:

```
=> SELECT userId, timestamp, CONDITIONAL_TRUE_EVENT(timestamp - LAG(timestamp) >
(LAG(timestamp, 1) - LAG(timestamp, 3)) / 2 + '3 seconds')
OVER(PARTITION BY userId ORDER BY timestamp) AS session
FROM WebClicks;
 userId |      timestamp      | session
--------+---------------------+---------
      2 | 2009-12-08 15:02:45 |       0
      2 | 2009-12-08 15:02:55 |       0
      2 | 2009-12-08 15:03:55 |       0
      1 | 2009-12-08 15:00:00 |       0
      1 | 2009-12-08 15:00:25 |       0
      1 | 2009-12-08 15:00:45 |       0
      1 | 2009-12-08 15:01:45 |       1
(7 rows)
```

Note
You cannot define a moving window in time series data. For example, if the query is evaluating the first row and there's no data, it will be the current row. If you have a lag of 2, no results are returned until the third row.

See also

- Event-based windows
- CONDITIONAL_TRUE_EVENT [analytic]

Machine learning for predictive analytics

Vertica provides a number of machine learning functions for performing in-database analysis. These functions perform data preparation, model training, model management, and predictive tasks. Vertica supports the following in-database machine learning algorithms:

- Regression algorithms : Linear regression, random forest, SVM, XGBoost
- Classification algorithms : Logistic regression, naive Bayes, random forest, SVM, XGBoost
- Clustering algorithms : K-means, bisecting k-means
- Time series forecasting : Autoregression, moving-average, ARIMA

For a scikit-like machine learning library that integrates directly with the data in your Vertica database, see VerticaPy .

For more information about specific machine learning functions, see Machine learning functions .

In this section

- Download the machine learning example data
- Data preparation
- Regression algorithms
- Classification algorithms
- Clustering algorithms
- Time series forecasting
- Model management
- Using external models with Vertica

Download the machine learning example data

You need several data sets to run the machine learning examples. You can download these data sets from the Vertica GitHub repository.

Important
The GitHub examples are based on the latest Vertica version. If you note differences, please upgrade to the latest version.

You can download the example data in either of two ways:

- Download the ZIP file. Extract the contents of the file into a directory.
- Clone the Vertica Machine Learning GitHub repository. Using a terminal window, run the following command:

```
$ git clone https://github.com/vertica/Machine-Learning-Examples
```

Loading the example data

You can load the example data by doing one of the following. Note that models are not automatically dropped. You must either rerun the load_ml_data.sql script to drop models or manually drop them.

- Copying and pasting the DDL and DML operations in load_ml_data.sql in a vsql prompt or another Vertica client.
- Running the following command from a terminal window within the data folder in the Machine-Learning-Examples directory:

```
$ /opt/vertica/bin/vsql -d <name of your database> -f load_ml_data.sql
```

You must also load the naive_bayes_data_prepration.sql script in the Machine-Learning-Examples directory:

```
$ /opt/vertica/bin/vsql -d <name of your database> -f ./naive_bayes/naive_bayes_data_preparation.sql
```

Example data descriptions

The repository contains the following data sets.

Name	Description
agar_dish	Synthetic data set meant to represent clustering of bacteria on an agar dish. Contains the following columns: id, x-coordinate, and y-coordinate.
agar_dish_2	125 rows sampled randomly from the original 500 rows of the agar_dish data set.
agar_dish_1	375 rows sampled randomly from the original 500 rows of the agar_dish data set.
baseball	Contains statistics from a fictional baseball league. The statistics included are: first name, last name, date of birth, team name, homeruns, hits, batting average, and salary.
daily-min-temperatures	Contains data on the daily minimum temperature in Melbourne, Australia from 1981 through 1990.
dem_votes	Contains data on the number of yes and no votes by Democrat members of U.S. Congress for each of the 16 votes in the house84 data set. The table must be populated by running the naive_bayes_data_prepration.sql script. Contains the following columns: vote, yes, no.
faithful	Wait times between eruptions and the duration of the eruption for the Old Faithful geyser in Yellowstone National Park, Wyoming, USA. **Reference** Härdle, W. (1991) *Smoothing Techniques with Implementation in S* . New York: Springer. Azzalini, A. and Bowman, A. W. (1990). A look at some data on the Old Faithful geyser. *Applied Statistics* 39, 357–365.
faithful_testing	Roughly 60% of the original 272 rows of the faithful data set.
faithful_training	Roughly 40% of the original 272 rows of the faithful data set.
house84	The house84 data set includes votes for each of the U.S. House of Representatives Congress members on 16 votes. Contains the following columns: id, party, vote1, vote2, vote3, vote4, vote5, vote6, vote7, vote8, vote9, vote10, vote11, vote12, vote13, vote14, vote15, vote16. **Reference** Congressional Quarterly Almanac, 98th Congress, 2nd session 1984, Volume XL: Congressional Quarterly Inc. Washington, D.C., 1985.

iris	The iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are Iris setosa, versicolor, and virginica. **Reference** Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language* . Wadsworth & Brooks/Cole.
iris1	90 rows sampled randomly from the original 150 rows in the iris data set.
iris2	60 rows sampled randomly from the original 150 rows in the iris data set.
mtcars	The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models). **Reference** Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics* , 37, 391–411.
rep_votes	Contains data on the number of yes and no votes by Republican members of U.S. Congress for each of the 16 votes in the house84 data set. The table must be populated by running the naive_bayes_data_prepration.sql script. Contains the following columns: vote, yes, no.
salary_data	Contains fictional employee data. The data included are: employee id, first name, last name, years worked, and current salary.
transaction_data	Contains fictional credit card transactions with a BOOLEAN column indicating whether there was fraud associated with the transaction. The data included are: first name, last name, store, cost, and fraud.
titanic_testing	Contains passenger information from the Titanic ship including sex, age, passenger class, and whether or not they survived.
titanic_training	Contains passenger information from the Titanic ship including sex, age, passenger class, and whether or not they survived.
world	Contains country-specific information about human development using HDI, GDP, and CO2 emissions.

Data preparation

Before you can analyze your data, you must prepare it. You can do the following data preparation tasks in Vertica:

In this section

- Balancing imbalanced data
- Detect outliers
- Encoding categorical columns
- Imputing missing values
- Normalizing data
- PCA (principal component analysis)
- Sampling data
- SVD (singular value decomposition)

Balancing imbalanced data

Imbalanced data occurs when an uneven distribution of classes occurs in the data. Building a predictive model on the imbalanced data set would cause a model that appears to yield high accuracy but does not generalize well to the new data in the minority class. To prevent creating models with false levels of accuracy, you should rebalance your imbalanced data before creating a predictive model.

Before you begin the example, load the Machine Learning sample data .
You see imbalanced data a lot in financial transaction data where the majority of the transactions are not fraudulent and a small number of the transactions are fraudulent, as shown in the following example.

1. View the distribution of the classes.

```
=> SELECT fraud, COUNT(fraud) FROM transaction_data GROUP BY fraud;
 fraud | COUNT
-------+-------
 TRUE  |    19
 FALSE |   981
(2 rows)
```

2. Use the BALANCE function to create a more balanced data set.

```
=> SELECT BALANCE('balance_fin_data', 'transaction_data', 'fraud', 'under_sampling'
                  USING PARAMETERS sampling_ratio = 0.2);
        BALANCE
--------------------------
 Finished in 1 iteration

(1 row)
```

3. View the new distribution of the classifiers.

```
=> SELECT fraud, COUNT(fraud) FROM balance_fin_data GROUP BY fraud;
 fraud | COUNT
-------+-------
 t     |    19
 f     |   236
(2 rows)
```

See also

- BALANCE

Detect outliers

Outliers are data points that greatly differ from other data points in a dataset. You can use outlier detection for applications such as fraud detection and system health monitoring, or you can detect outliers to then remove them from your data. If you leave outliers in your data when training a machine learning model, your resultant model is at risk for bias and skewed predictions. Vertica supports two methods for detecting outliers: the DETECT_OUTLIERS function and the IFOREST algorithm.

> **Note**
> The examples below use the baseball dataset from the machine learning example data. If you haven't already, Download the machine learning example data .

Isolation forest

Isolation forest (iForest) is an unsupervised algorithm that operates on the assumption that outliers are few and different. This assumption makes outliers susceptible to a separation mechanism called isolation. Instead of comparing data instances to a constructed normal distribution of each data feature, isolation focuses on outliers themselves.

To isolate outliers directly, iForest builds binary tree structures named isolation trees (iTrees) to model the feature space. These iTrees randomly and recursively split the feature space so that each node of the tree represents a feature subspace. For instance, the first split divides the whole feature space into two subspaces, which are represented by the two child nodes of the root node. A data instance is considered isolated when it is the only member of a feature subspace. Because outliers are assumed to be few and different, outliers are likely to be isolated sooner than normal data instances.

In order to improve the robustness of the algorithm, iForest builds an ensemble of iTrees, which each separate the feature space differently. The algorithm calculates the average path length needed to isolate a data instance across all iTrees. This average path length helps determine the anomaly_score for each data instance in a dataset. The data instances with an anomaly_score above a given threshold are considered outliers.

You do not need a large dataset to train an iForest, and even a small sample should suffice to train an accurate model. The data can have columns of types CHAR, VARCHAR, BOOL, INT, or FLOAT.

After you have a trained an iForest model, you can use the APPLY_IFOREST function to detect outliers in any new data added to the dataset.

The following example demonstrates how to train an iForest model and detect outliers on the baseball dataset.

To build and train an iForest model, call IFOREST :

```
=> SELECT IFOREST('baseball_outliers','baseball','hr, hits, salary' USING PARAMETERS max_depth=30, nbins=100);
IFOREST
----------
Finished
(1 row)
```

You can view a summary of the trained model using GET_MODEL_SUMMARY :

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='baseball_outliers');
GET_MODEL_SUMMARY
---------------------------------------------------------------------------------------------------------------------------------------

===========
call_string
===========
SELECT iforest('public.baseball_outliers', 'baseball', 'hr, hits, salary' USING PARAMETERS exclude_columns='', ntree=100, sampling_size=0.632,
col_sample_by_tree=1, max_depth=30, nbins=100);

=======
details
=======
predictor|      type
---------+----------------
   hr    |      int
  hits   |      int
 salary  |float or numeric

===============
Additional Info
===============
       Name       |Value
------------------+-----
tree_count        | 100
rejected_row_count|  0
accepted_row_count|1000

(1 row)
```

You can apply the trained iForest model to the baseball dataset with APPLY_IFOREST . To view only the data instances that are identified as outliers, you can run the following query:

```
=> SELECT * FROM (SELECT first_name, last_name, APPLY_IFOREST(hr, hits, salary USING PARAMETERS model_name='baseball_outliers',
threshold=0.6)
   AS predictions FROM baseball) AS outliers WHERE predictions.is_anomaly IS true;
 first_name | last_name |                      predictions
------------+-----------+-------------------------------------------------------
 Jacqueline | Richards  | {"anomaly_score":0.8572338674053986,"is_anomaly":true}
 Debra      | Hall      | {"anomaly_score":0.6007846156043213,"is_anomaly":true}
 Gerald     | Fuller    | {"anomaly_score":0.6813650107767862,"is_anomaly":true}
(3 rows)
```

Instead of specifying a threshold value for APPLY_IFOREST, you can set the contamination parameter. This parameter sets a threshold so that the ratio of training data points labeled as outliers is approximately equal to the value of contamination :

```
=> SELECT * FROM (SELECT first_name, last_name, APPLY_IFOREST(team, hr, hits, avg, salary USING PARAMETERS
model_name='baseball_anomalies',
   contamination = 0.1) AS predictions FROM baseball) AS outliers WHERE predictions.is_anomaly IS true;
 first_name | last_name |                      predictions
------------+-----------+-----------------------------------------------------------
 Marie      | Fields    | {"anomaly_score":0.5307715717521868,"is_anomaly":true}
 Jacqueline | Richards  | {"anomaly_score":0.777757463074347,"is_anomaly":true}
 Debra      | Hall      | {"anomaly_score":0.5714649698133808,"is_anomaly":true}
 Gerald     | Fuller    | {"anomaly_score":0.5980549926114661,"is_anomaly":true}
(4 rows)
```

DETECT_OUTLIERS

The DETECT_OUTLIERS function assumes a normal distribution for each data dimension, and then identifies data instances that differ strongly from the normal profile of any dimension. The function uses the robust z-score detection method to normalize each input column. If a data instance contains a normalized value greater than a specified threshold, it is identified as an outlier. The function outputs a table that contains all the outliers.

The function accepts data with only numeric input columns, treats each column independently, and assumes a Gaussian distribution on each column. If you want to detect outliers in new data added to the dataset, you must rerun DETECT_OUTLIERS .

The following example demonstrates how you can detect the outliers in the baseball dataset based on the hr, hits, and salary columns. The DETECT_OUTLIERS function creates a table containing the outliers with the input and key columns:

```
=> SELECT DETECT_OUTLIERS('baseball_hr_hits_salary_outliers', 'baseball', 'hr, hits, salary', 'robust_zscore'
                    USING PARAMETERS outlier_threshold=3.0);
    DETECT_OUTLIERS
-------------------------
 Detected 5 outliers

(1 row)
```

To view the outliers, query the output table containing the outliers:

```
=> SELECT * FROM baseball_hr_hits_salary_outliers;
id  | first_name | last_name |    dob     |   team    |   hr   |  hits  |  avg  |       salary
----+------------+-----------+------------+-----------+--------+--------+-------+---------------------
73  | Marie      | Fields    | 1985-11-23 | Mauv      |   8888 |     34 | 0.283 | 9.99999999341471e+16
89  | Jacqueline | Richards  | 1975-10-06 | Pink      |  273333 | 4490260 | 0.324 |  4.4444444444828e+17
87  | Jose       | Stephens  | 1991-07-20 | Green     |     80 |   64253 |  0.69 |          16032567.12
222 | Gerald     | Fuller    | 1991-02-13 | Goldenrod | 3200000 |    216 | 0.299 |          37008899.76
147 | Debra      | Hall      | 1980-12-31 | Maroon    | 1100037 |    230 | 0.431 |          9000101403
(5 rows)
```

You can create a view omitting the outliers from the table:

```
=> CREATE VIEW clean_baseball AS
   SELECT * FROM baseball WHERE id NOT IN (SELECT id FROM baseball_hr_hits_salary_outliers);
CREATE VIEW
```

See also

- DETECT_OUTLIERS
- IFOREST
- APPLY_IFOREST
- READ_TREE

Encoding categorical columns

Many machine learning algorithms cannot work with categorical data. To accommodate such algorithms, categorical data must be converted to numerical data before training. Directly mapping the categorical values into indices is not enough. For example, if your categorical feature has three distinct values "red", "green" and "blue", replacing them with 1, 2 and 3 may have a negative impact on the training process because algorithms usually rely on some kind of numerical distances between values to discriminate between them. In this case, the Euclidean distance from 1 to 3 is twice the distance from 1 to 2, which means the training process will think that "red" is much more different than "blue", while it is more similar to

"green". Alternatively, one hot encoding maps each categorical value to a binary vector to avoid this problem. For example, "red" can be mapped to [1,0,0], "green" to [0,1,0] and "blue" to [0,0,1]. Now, the pair-wise distances between the three categories are all the same. One hot encoding allows you to convert categorical variables to binary values so that you can use different machine learning algorithms to evaluate your data.

The following example shows how you can apply one hot encoding to the Titanic data set. If you would like to read more about this data set, see the Kaggle site .

Suppose you want to use a logistic regression classifier to predict which passengers survived the sinking of the Titanic. You cannot use categorical features for logistic regression without one hot encoding. This data set has two categorical features that you can use. The "sex" feature can be either male or female. The "embarkation_point" feature can be one of the following:

- S for Southampton
- Q for Queenstown
- C for Cherbourg

Before you begin the example, load the Machine Learning sample data .

1. Run the ONE_HOT_ENCODER_FIT function on the training data:

```
=> SELECT ONE_HOT_ENCODER_FIT('titanic_encoder', 'titanic_training', 'sex, embarkation_point');
ONE_HOT_ENCODER_FIT
---------------------
Success
```

(1 row)

2. View a summary of the titanic_encoder model:

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='titanic_encoder');
GET_MODEL_SUMMARY
-------------------------------------------------------------------------------------------
===========
call_string
===========
SELECT one_hot_encoder_fit('public.titanic_encoder','titanic_training','sex, embarkation_point'
USING PARAMETERS exclude_columns='', output_view='', extra_levels='{}');
==================
varchar_categories
==================
category_name    |category_level|category_level_index
-----------------+--------------+--------------------
embarkation_point|      C       |         0
embarkation_point|      Q       |         1
embarkation_point|      S       |         2
embarkation_point|              |         3
sex              |    female    |         0
sex              |     male     |         1
(1 row)
```

3. Run the GET_MODEL_ATTRIBUTE function. This function returns the categorical levels in their native data types, so they can be compared easily with the original table:

```
=> SELECT * FROM (SELECT GET_MODEL_ATTRIBUTE(USING PARAMETERS model_name='titanic_encoder',
attr_name='varchar_categories')) AS attrs INNER JOIN (SELECT passenger_id, name, sex, age,
embarkation_point FROM titanic_training) AS original_data ON attrs.category_level
ILIKE original_data.embarkation_point ORDER BY original_data.passenger_id LIMIT 10;
category_name     | category_level | category_level_index | passenger_id |name
|  sex   | age | embarkation_point
------------------+----------------+----------------------+--------------+----------------------------
----------------------+--------+-----+-------------------
embarkation_point | S              |                    2 |            1 | Braund, Mr. Owen Harris
| male   |  22 | S
embarkation_point | C              |                    0 |            2 | Cumings, Mrs. John Bradley
(Florence Briggs Thayer | female |  38 | C
embarkation_point | S              |                    2 |            3 | Heikkinen, Miss. Laina
| female |  26 | S
embarkation_point | S              |                    2 |            4 | Futrelle, Mrs. Jacques Heath
(Lily May Peel)       | female |  35 | S
embarkation_point | S              |                    2 |            5 | Allen, Mr. William Henry
| male   |  35 | S
embarkation_point | Q              |                    1 |            6 | Moran, Mr. James
| male   |     | Q
embarkation_point | S              |                    2 |            7 | McCarthy, Mr. Timothy J
| male   |  54 | S
embarkation_point | S              |                    2 |            8 | Palsson, Master. Gosta Leonard
| male   |   2 | S
embarkation_point | S              |                    2 |            9 | Johnson, Mrs. Oscar W
(Elisabeth Vilhelmina Berg)  | female |  27 | S
embarkation_point | C              |                    0 |           10 | Nasser, Mrs. Nicholas
(Adele Achem)                | female |  14 | C
(10 rows)
```

4. Run the APPLY_ONE_HOT_ENCODER function on both the training and testing data:

```
=> CREATE VIEW titanic_training_encoded AS SELECT passenger_id, survived, pclass, sex_1, age,
sibling_and_spouse_count, parent_and_child_count, fare, embarkation_point_1, embarkation_point_2
FROM (SELECT APPLY_ONE_HOT_ENCODER(* USING PARAMETERS model_name='titanic_encoder')
FROM titanic_training) AS sq;

CREATE VIEW

=> CREATE VIEW titanic_testing_encoded AS SELECT passenger_id, name, pclass, sex_1, age,
sibling_and_spouse_count, parent_and_child_count, fare, embarkation_point_1, embarkation_point_2
FROM (SELECT APPLY_ONE_HOT_ENCODER(* USING PARAMETERS model_name='titanic_encoder')
FROM titanic_testing) AS sq;
CREATE VIEW
```

5. Then, train a logistic regression classifier on the training data, and execute the model on the testing data:

```
=> SELECT LOGISTIC_REG('titanic_log_reg', 'titanic_training_encoded', 'survived', '*'
USING PARAMETERS exclude_columns='passenger_id, survived');
LOGISTIC_REG
---------------------------
Finished in 5 iterations
(1 row)

=> SELECT passenger_id, name, PREDICT_LOGISTIC_REG(pclass, sex_1, age, sibling_and_spouse_count,
parent_and_child_count, fare, embarkation_point_1, embarkation_point_2 USING PARAMETERS
model_name='titanic_log_reg') FROM titanic_testing_encoded ORDER BY passenger_id LIMIT 10;
passenger_id |                 name                 | PREDICT_LOGISTIC_REG
-------------+----------------------------------------+----------------------
893          | Wilkes, Mrs. James (Ellen Needs)       |                    0
894          | Myles, Mr. Thomas Francis              |                    0
895          | Wirz, Mr. Albert                       |                    0
896          | Hirvonen, Mrs. Alexander (Helga E Lindqvist) |                    1
897          | Svensson, Mr. Johan Cervin             |                    0
898          | Connolly, Miss. Kate                   |                    1
899          | Caldwell, Mr. Albert Francis           |                    0
900          | Abrahim, Mrs. Joseph (Sophie Halaut Easu)    |                    1
901          | Davies, Mr. John Samuel                |                    0
902          | Ilieff, Mr. Ylio                       |
(10 rows)
```

Imputing missing values

You can use the IMPUTE function to replace missing data with the most frequent value or with the average value in the same column. This impute example uses the small_input_impute table. Using the function, you can specify either the mean or mode method.

These examples show how you can use the IMPUTE function on the small_input_impute table.

Before you begin the example, load the Machine Learning sample data .
First, query the table so you can see the missing values:

```
=> SELECT * FROM small_input_impute;
pid | pclass | gender |    x1     |    x2     |    x3     | x4 | x5 | x6
----+--------+--------+-----------+-----------+-----------+----+----+----
5   |      0 |      1 | -2.590837 | -2.892819 |  -2.70296 |  2 | t  | C
7   |      1 |      1 |  3.829239 |   3.08765 |  Infinity |    | f  | C
13  |      0 |      0 | -9.060605 | -9.390844 | -9.559848 |  6 | t  | C
15  |      0 |      1 | -2.590837 | -2.892819 |  -2.70296 |  2 | f  | A
16  |      0 |      1 | -2.264599 | -2.615146 |  -2.10729 | 11 | f  | A
19  |      1 |      1 |           |  3.841606 |  3.754375 | 20 | t  |
1   |      0 |      0 | -9.445818 | -9.740541 | -9.786974 |  3 | t  | A
1   |      0 |      0 | -9.445818 | -9.740541 | -9.786974 |  3 | t  | A
2   |      0 |      0 | -9.618292 | -9.308881 | -9.562255 |  4 | t  | A
3   |      0 |      0 | -9.060605 | -9.390844 | -9.559848 |  6 | t  | B
4   |      0 |      0 | -2.264599 | -2.615146 |  -2.10729 | 15 | t  | B
6   |      0 |      1 | -2.264599 | -2.615146 |  -2.10729 | 11 | t  | C
8   |      1 |      1 |  3.273592 |           |  3.477332 | 18 | f  | B
10  |      1 |      1 |           |  3.841606 |  3.754375 | 20 | t  | A
18  |      1 |      1 |  3.273592 |           |  3.477332 | 18 | t  | B
20  |      1 |      1 |           |  3.841606 |  3.754375 | 20 |    | C
9   |      1 |      1 |           |  3.841606 |  3.754375 | 20 | f  | B
11  |      0 |      0 | -9.445818 | -9.740541 | -9.786974 |  3 | t  | B
12  |      0 |      0 | -9.618292 | -9.308881 | -9.562255 |  4 | t  | C
14  |      0 |      0 | -2.264599 | -2.615146 |  -2.10729 | 15 | f  | A
17  |      1 |      1 |  3.829239 |   3.08765 |  Infinity |    | f  | B
(21 rows)
```

Specify the mean method

Execute the IMPUTE function, specifying the mean method:

```
=> SELECT IMPUTE('output_view','small_input_impute', 'pid, x1,x2,x3,x4','mean'
                USING PARAMETERS exclude_columns='pid');
IMPUTE
--------------------------
Finished in 1 iteration
(1 row)
```

View output_view to see the imputed values:

```
=> SELECT * FROM output_view;
pid | pclass | gender |        x1         |        x2         |        x3         | x4 | x5 | x6
----+--------+--------+-------------------+-------------------+-------------------+----+----+----
5   |      0 |      1 |         -2.590837 |         -2.892819 |          -2.70296 |  2 | t  | C
7   |      1 |      1 |          3.829239 |           3.08765 | -3.12989705263158 | 11 | f  | C
13  |      0 |      0 |         -9.060605 |         -9.390844 |         -9.559848 |  6 | t  | C
15  |      0 |      1 |         -2.590837 |         -2.892819 |          -2.70296 |  2 | f  | A
16  |      0 |      1 |         -2.264599 |         -2.615146 |          -2.10729 | 11 | f  | A
19  |      1 |      1 | -3.86645035294118 |          3.841606 |          3.754375 | 20 | t  |
9   |      1 |      1 | -3.86645035294118 |          3.841606 |          3.754375 | 20 | f  | B
11  |      0 |      0 |         -9.445818 |         -9.740541 |         -9.786974 |  3 | t  | B
12  |      0 |      0 |         -9.618292 |         -9.308881 |         -9.562255 |  4 | t  | C
14  |      0 |      0 |         -2.264599 |         -2.615146 |          -2.10729 | 15 | f  | A
17  |      1 |      1 |          3.829239 |           3.08765 | -3.12989705263158 | 11 | f  | B
1   |      0 |      0 |         -9.445818 |         -9.740541 |         -9.786974 |  3 | t  | A
1   |      0 |      0 |         -9.445818 |         -9.740541 |         -9.786974 |  3 | t  | A
2   |      0 |      0 |         -9.618292 |         -9.308881 |         -9.562255 |  4 | t  | A
3   |      0 |      0 |         -9.060605 |         -9.390844 |         -9.559848 |  6 | t  | B
4   |      0 |      0 |         -2.264599 |         -2.615146 |          -2.10729 | 15 | t  | B
6   |      0 |      1 |         -2.264599 |         -2.615146 |          -2.10729 | 11 | t  | C
8   |      1 |      1 |          3.273592 | -3.22766163157895 |          3.477332 | 18 | f  | B
10  |      1 |      1 | -3.86645035294118 |          3.841606 |          3.754375 | 20 | t  | A
18  |      1 |      1 |          3.273592 | -3.22766163157895 |          3.477332 | 18 | t  | B
20  |      1 |      1 | -3.86645035294118 |          3.841606 |          3.754375 | 20 |    | C
(21 rows)
```

You can also execute the IMPUTE function, specifying the mean method and using the partition_columns parameter. This parameter works similarly to the GROUP_BY clause:

```
=> SELECT IMPUTE('output_view_group','small_input_impute', 'pid, x1,x2,x3,x4','mean'
USING PARAMETERS exclude_columns='pid', partition_columns='pclass,gender');
impute
--------------------------
Finished in 1 iteration
(1 row)
```

View output_view_group to see the imputed values:

```
=> SELECT * FROM output_view_group;
pid | pclass | gender |    x1     |        x2        |        x3        | x4 | x5 | x6
----+--------+--------+-----------+------------------+------------------+----+----+----
5   |      0 |      1 | -2.590837 |       -2.892819 |        -2.70296 |  2 | t  | C
7   |      1 |      1 |  3.829239 |         3.08765 | 3.66202733333333 | 19 | f  | C
13  |      0 |      0 | -9.060605 |       -9.390844 |       -9.559848 |  6 | t  | C
15  |      0 |      1 | -2.590837 |       -2.892819 |        -2.70296 |  2 | f  | A
16  |      0 |      1 | -2.264599 |       -2.615146 |        -2.10729 | 11 | f  | A
19  |      1 |      1 | 3.5514155 |        3.841606 |        3.754375 | 20 | t  |
1   |      0 |      0 | -9.445818 |       -9.740541 |       -9.786974 |  3 | t  | A
1   |      0 |      0 | -9.445818 |       -9.740541 |       -9.786974 |  3 | t  | A
2   |      0 |      0 | -9.618292 |       -9.308881 |       -9.562255 |  4 | t  | A
3   |      0 |      0 | -9.060605 |       -9.390844 |       -9.559848 |  6 | t  | B
4   |      0 |      0 | -2.264599 |       -2.615146 |        -2.10729 | 15 | t  | B
6   |      0 |      1 | -2.264599 |       -2.615146 |        -2.10729 | 11 | t  | C
8   |      1 |      1 |  3.273592 | 3.59028733333333 |        3.477332 | 18 | f  | B
10  |      1 |      1 | 3.5514155 |        3.841606 |        3.754375 | 20 | t  | A
18  |      1 |      1 |  3.273592 | 3.59028733333333 |        3.477332 | 18 | t  | B
20  |      1 |      1 | 3.5514155 |        3.841606 |        3.754375 | 20 |    | C
9   |      1 |      1 | 3.5514155 |        3.841606 |        3.754375 | 20 | f  | B
11  |      0 |      0 | -9.445818 |       -9.740541 |       -9.786974 |  3 | t  | B
12  |      0 |      0 | -9.618292 |       -9.308881 |       -9.562255 |  4 | t  | C
14  |      0 |      0 | -2.264599 |       -2.615146 |        -2.10729 | 15 | f  | A
17  |      1 |      1 |  3.829239 |         3.08765 | 3.66202733333333 | 19 | f  | B
(21 rows)
```

Specify the mode method

Execute the IMPUTE function, specifying the mode method:

```
=> SELECT impute('output_view_mode','small_input_impute', 'pid, x5,x6','mode'
                 USING PARAMETERS exclude_columns='pid');
impute
--------------------------
Finished in 1 iteration
(1 row)
```

View output_view_mode to see the imputed values:

```
=> SELECT * FROM output_view_mode;
pid | pclass | gender |    x1     |    x2     |    x3     | x4 | x5 | x6
----+--------+--------+-----------+-----------+-----------+----+----+----
5   |      0 |      1 | -2.590837 | -2.892819 |  -2.70296 |  2 | t  | C
7   |      1 |      1 |  3.829239 |   3.08765 |  Infinity |    | f  | C
13  |      0 |      0 | -9.060605 | -9.390844 | -9.559848 |  6 | t  | C
15  |      0 |      1 | -2.590837 | -2.892819 |  -2.70296 |  2 | f  | A
16  |      0 |      1 | -2.264599 | -2.615146 |  -2.10729 | 11 | f  | A
19  |      1 |      1 |           |  3.841606 |  3.754375 | 20 | t  | B
1   |      0 |      0 | -9.445818 | -9.740541 | -9.786974 |  3 | t  | A
1   |      0 |      0 | -9.445818 | -9.740541 | -9.786974 |  3 | t  | A
2   |      0 |      0 | -9.618292 | -9.308881 | -9.562255 |  4 | t  | A
3   |      0 |      0 | -9.060605 | -9.390844 | -9.559848 |  6 | t  | B
4   |      0 |      0 | -2.264599 | -2.615146 |  -2.10729 | 15 | t  | B
6   |      0 |      1 | -2.264599 | -2.615146 |  -2.10729 | 11 | t  | C
8   |      1 |      1 |  3.273592 |           |  3.477332 | 18 | f  | B
10  |      1 |      1 |           |  3.841606 |  3.754375 | 20 | t  | A
18  |      1 |      1 |  3.273592 |           |  3.477332 | 18 | t  | B
20  |      1 |      1 |           |  3.841606 |  3.754375 | 20 | t  | C
9   |      1 |      1 |           |  3.841606 |  3.754375 | 20 | f  | B
11  |      0 |      0 | -9.445818 | -9.740541 | -9.786974 |  3 | t  | B
12  |      0 |      0 | -9.618292 | -9.308881 | -9.562255 |  4 | t  | C
14  |      0 |      0 | -2.264599 | -2.615146 |  -2.10729 | 15 | f  | A
17  |      1 |      1 |  3.829239 |   3.08765 |  Infinity |    | f  | B
(21 rows)
```

You can also execute the IMPUTE function, specifying the mode method and using the partition_columns parameter. This parameter works similarly to the GROUP_BY clause:

```
=> SELECT impute('output_view_mode_group','small_input_impute', 'pid, x5,x6','mode'
                 USING PARAMETERS exclude_columns='pid',partition_columns='pclass,gender');
impute
--------------------------
Finished in 1 iteration
(1 row)
```

View output_view_mode_group to see the imputed values:

```
=> SELECT * FROM output_view_mode_group;
pid | pclass | gender |    x1    |    x2    |    x3    | x4 | x5 | x6
----+--------+--------+----------+----------+----------+----+----+----
1   |      0 |      0 | -9.445818 | -9.740541 | -9.786974 |  3 | t  | A
1   |      0 |      0 | -9.445818 | -9.740541 | -9.786974 |  3 | t  | A
2   |      0 |      0 | -9.618292 | -9.308881 | -9.562255 |  4 | t  | A
3   |      0 |      0 | -9.060605 | -9.390844 | -9.559848 |  6 | t  | B
4   |      0 |      0 | -2.264599 | -2.615146 |  -2.10729 | 15 | t  | B
13  |      0 |      0 | -9.060605 | -9.390844 | -9.559848 |  6 | t  | C
11  |      0 |      0 | -9.445818 | -9.740541 | -9.786974 |  3 | t  | B
12  |      0 |      0 | -9.618292 | -9.308881 | -9.562255 |  4 | t  | C
14  |      0 |      0 | -2.264599 | -2.615146 |  -2.10729 | 15 | f  | A
5   |      0 |      1 | -2.590837 | -2.892819 |  -2.70296 |  2 | t  | C
15  |      0 |      1 | -2.590837 | -2.892819 |  -2.70296 |  2 | f  | A
16  |      0 |      1 | -2.264599 | -2.615146 |  -2.10729 | 11 | f  | A
6   |      0 |      1 | -2.264599 | -2.615146 |  -2.10729 | 11 | t  | C
7   |      1 |      1 |  3.829239 |   3.08765 |  Infinity |    | f  | C
19  |      1 |      1 |           |  3.841606 |  3.754375 | 20 | t  | B
9   |      1 |      1 |           |  3.841606 |  3.754375 | 20 | f  | B
17  |      1 |      1 |  3.829239 |   3.08765 |  Infinity |    | f  | B
8   |      1 |      1 |  3.273592 |           |  3.477332 | 18 | f  | B
10  |      1 |      1 |           |  3.841606 |  3.754375 | 20 | t  | A
18  |      1 |      1 |  3.273592 |           |  3.477332 | 18 | t  | B
20  |      1 |      1 |           |  3.841606 |  3.754375 | 20 | f  | C
(21 rows)
```

See also

IMPUTE

Normalizing data

The purpose of normalization is, primarily, to scale numeric data from different columns down to an equivalent scale. For example, suppose you execute the LINEAR_REG function on a data set with two feature columns, current_salary and years_worked . The output value you are trying to predict is a worker's future salary. The values in the current_salary column are likely to have a far wider range, and much larger values, than the values in the years_worked column. Therefore, the values in the current_salary column can overshadow the values in the years_worked column, thus skewing your model.

Vertica offers the following data preparation methods which use normalization. These methods are:

- MinMax
  Using the MinMax normalization method, you can normalize the values in both of these columns to be within a distribution of values between 0 and 1. Doing so allows you to compare values on very different scales to one another by reducing the dominance of one column over the other.
- Z-score
  Using the Z-score normalization method, you can normalize the values in both of these columns to be the number of standard deviations an observation is from the mean of each column. This allows you to compare your data to a normally distributed random variable.
- Robust Z-score
  Using the Robust Z-score normalization method, you can lessen the influence of outliers on Z-score calculations. Robust Z-score normalization uses the median value as opposed to the mean value used in Z-score. By using the median instead of the mean, it helps remove some of the influence of outliers in the data.

Normalizing data results in the creation of a view where the normalized data is saved. The output_view option in the NORMALIZE function determines name of the view .

Normalizing salary data using MinMax

The following example shows how you can normalize the salary_data table using the MinMax normalization method.

Before you begin the example, load the Machine Learning sample data .

```
=> SELECT NORMALIZE('normalized_salary_data', 'salary_data', 'current_salary, years_worked', 'minmax');
       NORMALIZE
-------------------------
 Finished in 1 iteration

(1 row)

=> SELECT * FROM normalized_salary_data;
employee_id | first_name  | last_name  |     years_worked     |    current_salary
------------+-------------+------------+----------------------+----------------------
189         | Shawn       | Moore      | 0.350000000000000000 | 0.437246565765357217
518         | Earl        | Shaw       | 0.100000000000000000 | 0.978867411144492943
1126        | Susan       | Alexander  | 0.250000000000000000 | 0.909048995710749580
1157        | Jack        | Stone      | 0.100000000000000000 | 0.601863084103319918
1277        | Scott       | Wagner     | 0.050000000000000000 | 0.455949209228501786
3188        | Shirley     | Flores     | 0.400000000000000000 | 0.538816771536005140
3196        | Andrew      | Holmes     | 0.900000000000000000 | 0.183954046444834949
3430        | Philip      | Little     | 0.100000000000000000 | 0.735279557092379495
3522        | Jerry       | Ross       | 0.800000000000000000 | 0.671828883472214349
3892        | Barbara     | Flores     | 0.350000000000000000 | 0.092901007123556866


.
.
.
(1000 rows)
```

Normalizing salary data using z-score

The following example shows how you can normalize the salary_data table using the Z-score normalization method.

Before you begin the example, load the Machine Learning sample data .

```
=> SELECT NORMALIZE('normalized_z_salary_data', 'salary_data', 'current_salary, years_worked',
                    'zscore');
       NORMALIZE
-------------------------
 Finished in 1 iteration

(1 row)

=> SELECT * FROM normalized_z_salary_data;
employee_id | first_name  | last_name  |     years_worked     |    current_salary
------------+-------------+------------+----------------------+----------------------
189         | Shawn       | Moore      |  -0.524447274157005  |   -0.221041249770669
518         | Earl        | Shaw       |   -1.35743214416495  |     1.66054215981221
1126        | Susan       | Alexander  |  -0.857641222160185  |     1.41799393943946
1157        | Jack        | Stone      |   -1.35743214416495  |    0.350834283622416
1277        | Scott       | Wagner     |   -1.52402911816654  |   -0.156068522159045
3188        | Shirley     | Flores     |  -0.357850300155415  |    0.131812255991634
3196        | Andrew      | Holmes     |    1.30811943986048  |    -1.10097599783475
3430        | Philip      | Little     |   -1.35743214416495  |    0.814321286168547
3522        | Jerry       | Ross       |   0.974925491857304  |    0.593894513770248
3892        | Barbara     | Flores     |  -0.524447274157005  |    -1.41729301118583


.
.
.
(1000 rows)
```

Normalizing salary data using robust z-score

The following example shows how you can normalize the salary_data table using the robust Z-score normalization method.

Before you begin the example, load the Machine Learning sample data .

```
=> SELECT NORMALIZE('normalized_robustz_salary_data', 'salary_data', 'current_salary, years_worked', 'robust_zscore');
       NORMALIZE
-------------------------
 Finished in 1 iteration

(1 row)

=> SELECT * FROM normalized_robustz_salary_data;
employee_id | first_name  | last_name |    years_worked    |    current_salary
------------+-------------+-----------+--------------------+----------------------
189         | Shawn       | Moore     | -0.404694455685957 | -0.158933849655499140
518         | Earl        | Shaw      | -1.079185215162552 |  1.317126172796275889
1126        | Susan       | Alexander | -0.674490759476595 |  1.126852528914384584
1157        | Jack        | Stone     | -1.079185215162552 |  0.289689691751547422
1277        | Scott       | Wagner    | -1.214083367057871 | -0.107964200747705902
3188        | Shirley     | Flores    | -0.269796303790638 |  0.117871818902746738
3196        | Andrew      | Holmes    |  1.079185215162552 | -0.849222942006447161
3430        | Philip      | Little    | -1.079185215162552 |  0.653284859470426481
3522        | Jerry       | Ross      |  0.809388911371914 |  0.480364995828913355
3892        | Barbara     | Flores    | -0.404694455685957 | -1.097366550974798397
3939        | Anna        | Walker    | -0.944287063267233 |  0.414956177842775781
4165        | Martha      | Reyes     |  0.269796303790638 |  0.773947701782753329
4335        | Phillip     | Wright    | -1.214083367057871 |  1.218843012657445647
4534        | Roger       | Harris    |  1.079185215162552 |  1.155185021164402608
4806        | John        | Robinson  |  0.809388911371914 | -0.494320112876813908
4881        | Kelly       | Welch     |  0.134898151895319 | -0.540778808820045933
4889        | Jennifer    | Arnold    |  1.214083367057871 | -0.299762093576526566
5067        | Martha      | Parker    |  0.000000000000000 |  0.719991348857328239
5523        | John        | Martin    | -0.269796303790638 | -0.411248545269163826
6004        | Nicole      | Sullivan  |  0.269796303790638 |  1.065141044522487821
6013        | Harry       | Woods     | -0.944287063267233 |  1.005664438654129376
6240        | Norma       | Martinez  |  1.214083367057871 |  0.762412844887071691

.
.
.
(1000 rows)
```

See also

- NORMALIZE

PCA (principal component analysis)

Principal Component Analysis (PCA) is a technique that reduces the dimensionality of data while retaining the variation present in the data. In essence, a new coordinate system is constructed so that data variation is strongest along the first axis, less strong along the second axis, and so on. Then, the data points are transformed into this new coordinate system. The directions of the axes are called principal components.

If the input data is a table with p columns, there could be maximum p principal components. However, it's usually the case that the data variation along the direction of some k-th principal component becomes almost negligible, which allows us to keep only the first k components. As a result, the new coordinate system has fewer axes. Hence, the transformed data table has only k columns instead of p. It is important to remember that the k output columns are not simply a subset of p input columns. Instead, each of the k output columns is a combination of all p input columns.

You can use the following functions to train and apply the PCA model:

- APPLY_INVERSE_PCA
- APPLY_PCA
- PCA

For a complete example, see Dimension reduction using PCA .

In this section

- Dimension reduction using PCA

Dimension reduction using PCA

This PCA example uses a data set with a large number of columns named world. The example shows how you can apply PCA to all columns in the data set (except HDI) and reduce them into two dimensions.

Before you begin the example, load the Machine Learning sample data .

1. Create the PCA model, named pcamodel .

```
=> SELECT PCA ('pcamodel', 'world','country,HDI,em1970,em1971,em1972,em1973,em1974,em1975,em1976,em1977,
em1978,em1979,em1980,em1981,em1982,em1983,em1984 ,em1985,em1986,em1987,em1988,em1989,em1990,em1991,em1992,
em1993,em1994,em1995,em1996,em1997,em1998,em1999,em2000,em2001,em2002,em2003,em2004,em2005,em2006,em2007,
em2008,em2009,em2010,gdp1970,gdp1971,gdp1972,gdp1973,gdp1974,gdp1975,gdp1976,gdp1977,gdp1978,gdp1979,gdp1980,
gdp1981,gdp1982,gdp1983,gdp1984,gdp1985,gdp1986,gdp1987,gdp1988,gdp1989,gdp1990,gdp1991,gdp1992,gdp1993,
gdp1994,gdp1995,gdp1996,gdp1997,gdp1998,gdp1999,gdp2000,gdp2001,gdp2002,gdp2003,gdp2004,gdp2005,gdp2006,
gdp2007,gdp2008,gdp2009,gdp2010' USING PARAMETERS exclude_columns='HDI, country');

PCA
------------------------------------------------------------------
Finished in 1 iterations.
Accepted Rows: 96  Rejected Rows: 0
(1 row)
```

2. View the summary output of pcamodel .

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='pcamodel');
GET_MODEL_SUMMARY
-------------------------------------------------------------------------------
```

3. Next, apply PCA to a select few columns, with the exception of HDI and country.

```
=> SELECT APPLY_PCA (HDI,country,em1970,em2010,gdp1970,gdp2010 USING PARAMETERS model_name='pcamodel',
exclude_columns='HDI,country', key_columns='HDI,country',cutoff=.3) OVER () FROM world;
HDI   |       country       |       col1
------+---------------------+-------------------
0.886 | Belgium             | -36288.1191849017
0.699 | Belize              |   -81740.32711562
0.427 | Benin               | -122666.882708325
0.805 | Chile               | -161356.484748602
0.687 | China               | -202634.254216416
0.744 | Costa Rica          | -242043.080125449
0.4   | Cote d'Ivoire       | -283330.394428932
0.776 | Cuba                | -322625.857541772
0.895 | Denmark             | -356086.311721071
0.644 | Egypt               | -403634.743992772
.
.
.
(96 rows)
```

4. Then, optionally apply the inverse function to transform the data back to its original state. This example shows an abbreviated output, only for the first record. There are 96 records in total.

```
=> SELECT APPLY_INVERSE_PCA (HDI,country,em1970,em2010,gdp1970,gdp2010 USING PARAMETERS model_name='pcamodel',
exclude_columns='HDI,country', key_columns='HDI,country') OVER () FROM world limit 1;
-[ RECORD 1 ]--------------
HDI     | 0.886
country | Belgium
em1970  | 3.74891915022521
em1971  | 26.091852917619
em1972  | 22.0262860721982
em1973  | 24.8214492074202
em1974  | 20.9486650320945
em1975  | 29.5717692117088
em1976  | 17.4373459783249
em1977  | 33.1895610966146
```

em1978 | 15.6251407781098
em1979 | 14.9560299812815
em1980 | 18.0870223053504
em1981 | -6.23151505146251
em1982 | -7.12300504708672
em1983 | -7.52627957856581
em1984 | -7.17428622245234
em1985 | -9.04899186621455
em1986 | -10.5098581697156
em1987 | -7.97146984849547
em1988 | -8.85458031319287
em1989 | -8.78422101747477
em1990 | -9.61931854722004
em1991 | -11.6411235452067
em1992 | -12.8882752879355
em1993 | -15.0647523842803
em1994 | -14.3266175918398
em1995 | -9.07603254825782
em1996 | -9.32002671928241
em1997 | -10.0209028262361
em1998 | -6.70882735196004
em1999 | -7.32575918131333
em2000 | -10.3113551933996
em2001 | -11.0162573094354
em2002 | -10.886264397431
em2003 | -8.96078372850612
em2004 | -11.5157129257881
em2005 | -12.5048269019293
em2006 | -12.2345161132594
em2007 | -8.92504587601715
em2008 | -12.1136551375247
em2009 | -10.1144380511421
em2010 | -7.72468307053519
gdp1970 | 10502.1047183969
gdp1971 | 9259.97560190599
gdp1972 | 6593.98178532712
gdp1973 | 5325.33813328068
gdp1974 | -899.029529832931
gdp1975 | -3184.93671107899
gdp1976 | -4517.68204331439
gdp1977 | -3322.9509067019
gdp1978 | -33.8221923368737
gdp1979 | 2882.50573071066
gdp1980 | 3638.74436577365
gdp1981 | 2211.77365027338
gdp1982 | 5811.44631880621
gdp1983 | 7365.75180165581
gdp1984 | 10465.1797058904
gdp1985 | 12312.7219748196
gdp1986 | 12309.0418293413
gdp1987 | 13695.5173269466
gdp1988 | 12531.9995299889
gdp1989 | 13009.2244205049
gdp1990 | 10697.6839797576
gdp1991 | 6835.94651304181
gdp1992 | 4275.67753277099
gdp1993 | 3382.29408813394
gdp1994 | 3703.65406726311
gdp1995 | 4238.17659535371
gdp1996 | 4692.48744219914
gdp1997 | 4539.23538342266

```
gdp1998 | 5886.78983381162
gdp1999 | 7527.72448728762
gdp2000 | 7646.05563584361
gdp2001 | 9053.22077886667
gdp2002 | 9914.82548013531
gdp2003 | 9201.64413455221
gdp2004 | 9234.70123279344
gdp2005 | 9565.5457350936
gdp2006 | 9569.86316415438
gdp2007 | 9104.60260145907
gdp2008 | 8182.8163827425
gdp2009 | 6279.93197775805
gdp2010 | 4274.40397281553
```

See also

- APPLY_INVERSE_PCA
- APPLY_PCA
- PCA

Sampling data

The goal of data sampling is to take a smaller, more manageable sample of a much larger data set. With a sample data set, you can produce predictive models or use it to help you tune your database. The following example shows how you can use the TABLESAMPLE clause to create a sample of your data.

Sampling data from a table

Before you begin the example, load the Machine Learning sample data .
Using the baseball table, create a new table named baseball_sample containing a 25% sample of baseball . Remember, TABLESAMPLE does not guarantee that the exact percentage of records defined in the clause are returned.

```
=> CREATE TABLE baseball_sample AS SELECT * FROM baseball TABLESAMPLE(25);
CREATE TABLE
=> SELECT * FROM baseball_sample;
 id  | first_name | last_name |    dob     |    team    | hr  | hits |  avg  |   salary
-----+------------+-----------+------------+------------+-----+------+-------+-------------
   4 | Amanda     | Turner    | 1997-12-22 | Maroon     |  58 |  177 | 0.187 |     8047721
  20 | Jesse      | Cooper    | 1983-04-13 | Yellow     |  97 |   39 | 0.523 |     4252837
  22 | Randy      | Peterson  | 1980-05-28 | Orange     |  14 |   16 | 0.141 |  11827728.1
  24 | Carol      | Harris    | 1991-04-02 | Fuscia     |  96 |   12 | 0.456 |  40572253.6
  32 | Rose       | Morrison  | 1977-07-26 | Goldenrod  |  27 |  153 | 0.442 | 14510752.49
  50 | Helen      | Medina    | 1987-12-26 | Maroon     |  12 |  150 |  0.54 | 32169267.91
  70 | Richard    | Gilbert   | 1983-07-13 | Khaki      |   1 |  250 | 0.213 | 40518422.76
  81 | Angela     | Cole      | 1991-08-16 | Violet     |  87 |  136 | 0.706 | 42875181.51
  82 | Elizabeth  | Foster    | 1994-04-30 | Indigo     |  46 |  163 | 0.481 | 33896975.53
  98 | Philip     | Gardner   | 1992-05-06 | Puce       |  39 |  239 | 0.697 | 20967480.67
 102 | Ernest     | Freeman   | 1983-10-05 | Turquoise  |  46 |   77 | 0.564 | 21444463.92
.
.
.
(227 rows)
```

With your sample you can create a predictive model, or tune your database.

See also

- FROM clause (for more information about the TABLESAMPLE clause)

SVD (singular value decomposition)

Singular Value Decomposition (SVD) is a matrix decomposition method that allows you to approximate matrix X with dimensions n-by-p as a product of 3 matrices: X(n-by-p) = U(n-by-k).S(k-byk).VT(k-by-p) where k is an integer from 1 to p, and S is a diagonal matrix. Its diagonal has non-negative values, called singular values, sorted from the largest, at the top left, to the smallest, at the bottom right. All other elements of S are zero.

In practice, the matrix V(p-by-k), which is the transposed version of VT, is more preferred.

If k (an input parameter to the decomposition algorithm, also known as the number of components to keep in the output) is equal to p, the decomposition is exact. If k is less than p, the decomposition becomes an approximation.

An application of SVD is lossy data compression. For example, storing X required n.p elements, while storing the three matrices U, S, and VT requires storing n.k + k + k.p elements. If n=1000, p=10, and k=2, storing X would require 10,000 elements while storing the approximation would require 2,000+4+20 = 2,024 elements. A smaller value of k increases the savings in storage space, while a larger value of k gives a more accurate approximation.

Depending on your data, the singular values may decrease rapidly, which allows you to choose a value of k that is much smaller than the value of p.

Another common application of SVD is to perform the principal component analysis.

You can use the following functions to train and apply the SVD model:

- APPLY_INVERSE_SVD
- APPLY_SVD
- SVD

For a complete example, see Computing SVD .

In this section

- Computing SVD

Computing SVD

This SVD example uses a small data set named small_svd. The example shows you how to compute SVD using the given data set. The table is a matrix of numbers. The singular value decomposition is computed using the SVD function. This example computes the SVD of the table matrix and assigns it to a new object, which contains one vector, and two matrices, U and V. The vector contains the singular values. The first matrix, U contains the left singular vectors and V contains the right singular vectors.

Before you begin the example, load the Machine Learning sample data .

1. Create the SVD model, named svdmodel .

```
=> SELECT SVD ('svdmodel', 'small_svd', 'x1,x2,x3,x4');
SVD
-------------------------------------------------------------
Finished in 1 iterations.
Accepted Rows: 8  Rejected Rows: 0
(1 row)
```

2. View the summary output of svdmodel .

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='svdmodel');
GET_MODEL_SUMMARY
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------
=======
columns
=======
index|name
-----+----
1    | x1
2    | x2
3    | x3
4    | x4
===============
singular_values
===============
index| value  |explained_variance|accumulated_explained_variance
-----+--------+------------------+------------------------------
1    |22.58748|      0.95542     |             0.95542
2    | 3.79176|      0.02692     |             0.98234
3    | 2.55864|      0.01226     |             0.99460
4    | 1.69756|      0.00540     |             1.00000
======================
right_singular_vectors
======================
index|vector1 |vector2 |vector3 |vector4
-----+--------+--------+--------+--------
1    | 0.58736| 0.08033| 0.74288|-0.31094
2    | 0.26661| 0.78275|-0.06148| 0.55896
3    | 0.71779|-0.13672|-0.64563|-0.22193
4    | 0.26211|-0.60179| 0.16587| 0.73596
========
counters
========
counter_name      |counter_value
------------------+-------------
accepted_row_count|      8
rejected_row_count|      0
iteration_count   |      1
===========
call_string
===========
SELECT SVD('public.svdmodel', 'small_svd', 'x1,x2,x3,x4');
(1 row)
```

3. Create a new table, named Umat to obtain the values for U.

```
=> CREATE TABLE Umat AS SELECT APPLY_SVD(id, x1, x2, x3, x4 USING PARAMETERS model_name='svdmodel',
exclude_columns='id', key_columns='id') OVER() FROM small_svd;
CREATE TABLE
```

4. View the results in the Umat table. This table transforms the matrix into a new coordinates system.

```
=> SELECT * FROM Umat ORDER BY id;
 id |       col1        |       col2        |       col3        |       col4
----+-------------------+-------------------+-------------------+-------------------
1   | -0.494871802886819 | -0.161721379259287 |  0.0712816417153664 | -0.473145877877408
2   |  -0.17652411036246 | 0.0753183783382909 |  -0.678196192333598 | 0.0567124770173372
3   | -0.150974762654569 | -0.589561842046029 | 0.00392654610109522 |  0.360011163271921
4   |  -0.44849499240202 |  0.347260956311326 |   0.186958376368345 |  0.378561270493651
5   | -0.494871802886819 | -0.161721379259287 |  0.0712816417153664 | -0.473145877877408
6   |  -0.17652411036246 | 0.0753183783382909 |  -0.678196192333598 | 0.0567124770173372
7   | -0.150974762654569 | -0.589561842046029 | 0.00392654610109522 |  0.360011163271921
8   |  -0.44849499240202 |  0.347260956311326 |   0.186958376368345 |  0.378561270493651
(8 rows)
```

5. Then, we can optionally transform the data back by converting it from Umat to Xmat. First, we must create the Xmat table and then apply the APPLY_INVERSE_SVD function to the table:

```
=> CREATE TABLE Xmat AS SELECT APPLY_INVERSE_SVD(* USING PARAMETERS model_name='svdmodel',
exclude_columns='id', key_columns='id') OVER() FROM Umat;
CREATE TABLE
```

6. Then view the data from the Xmat table that was created:

```
=> SELECT id, x1::NUMERIC(5,1), x2::NUMERIC(5,1), x3::NUMERIC(5,1), x4::NUMERIC(5,1) FROM Xmat
ORDER BY id;
id | x1  | x2  | x3  | x4
---+-----+-----+-----+-----
1  | 7.0 | 3.0 | 8.0 | 2.0
2  | 1.0 | 1.0 | 4.0 | 1.0
3  | 2.0 | 3.0 | 2.0 | 0.0
4  | 6.0 | 2.0 | 7.0 | 4.0
5  | 7.0 | 3.0 | 8.0 | 2.0
6  | 1.0 | 1.0 | 4.0 | 1.0
7  | 2.0 | 3.0 | 2.0 | 0.0
8  | 6.0 | 2.0 | 7.0 | 4.0
(8 rows)
```

See also

- APPLY_INVERSE_SVD
- APPLY_SVD
- SVD

Regression algorithms

Regression is an important and popular machine learning tool that makes predictions from data by learning the relationship between some features of the data and an observed value response. Regression is used to make predictions about profits, sales, temperature, stocks, and more. For example, you could use regression to predict the price of a house based on the location, the square footage, the size of the lot, and so on. In this example, the house's value is the response, and the other factors, such as location, are the features.

The optimal set of coefficients found for the regression's equation is known as the model. The relationship between the outcome and the features is summarized in the model, which can then be applied to different data sets, where the outcome value is unknown.

In this section

- Autoregression
- Linear regression
- Poisson regression
- Random forest for regression
- SVM (support vector machine) for regression
- XGBoost for regression

Autoregression

See the autoregressive model example under time series models.

Linear regression

Using linear regression, you can model the linear relationship between independent variables, or features, and a dependent variable, or outcome. You can build linear regression models to:

- Fit a predictive model to a training data set of independent variables and some dependent variable. Doing so allows you to use feature variable values to make predictions on outcomes. For example, you can predict the amount of rain that will fall on a particular day of the year.
- Determine the strength of the relationship between an independent variable and some outcome variable. For example, suppose you want to determine the importance of various weather variables on the outcome of how much rain will fall. You can build a linear regression model based on observations of weather patterns and rainfall to find the answer.

Unlike Logistic regression , which you use to determine a binary classification outcome, linear regression is primarily used to predict continuous numerical outcomes in linear relationships.

You can use the following functions to build a linear regression model, view the model, and use the model to make predictions on a set of test data:

- LINEAR_REG
- PREDICT_LINEAR_REG
- GET_MODEL_SUMMARY

For a complete example of how to use linear regression on a table in Vertica, see Building a linear regression model .

In this section

- Building a linear regression model

Building a linear regression model

This linear regression example uses a small data set named faithful. The data set contains the intervals between eruptions and the duration of eruptions for the Old Faithful geyser in Yellowstone National Park. The duration of each eruption can be between 1.5 and 5 minutes. The length of intervals between eruptions and of each eruption varies. However, you can estimate the time of the next eruption based on the duration of the previous eruption. The example shows how you can build a model to predict the value of eruptions , given the value of the waiting feature.

Before you begin the example, load the Machine Learning sample data .

1. Create the linear regression model, named linear_reg_faithful , using the faithful_training training data:

```
=> SELECT LINEAR_REG('linear_reg_faithful', 'faithful_training', 'eruptions', 'waiting'
  USING PARAMETERS optimizer='BFGS');
      LINEAR_REG
---------------------------
 Finished in 6 iterations

(1 row)
```

2. View the summary output of linear_reg_faithful :

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='linear_reg_faithful');
-------------------------------------------------------------------------------
=======
details
=======
predictor|coefficient|std_err |t_value |p_value
---------+-----------+--------+--------+--------
Intercept| -2.06795  | 0.21063|-9.81782| 0.00000
waiting  |  0.07876  | 0.00292|26.96925| 0.00000

==============
regularization
==============
type| lambda
----+--------
none| 1.00000

===========
call_string
===========
linear_reg('public.linear_reg_faithful', 'faithful_training', '"eruptions"', 'waiting'
USING PARAMETERS optimizer='bfgs', epsilon=1e-06, max_iterations=100,
regularization='none', lambda=1)

===============
Additional Info
===============
Name             |Value
-----------------+-----
iteration_count   |  3
rejected_row_count|  0
accepted_row_count| 162
(1 row)
```

3. Create a table that contains the response values from running the PREDICT_LINEAR_REG function on your test data. Name this table pred_faithful_results . View the results in the pred_faithful_results table:

```
=> CREATE TABLE pred_faithful_results AS
   (SELECT id, eruptions, PREDICT_LINEAR_REG(waiting USING PARAMETERS model_name='linear_reg_faithful')
   AS pred FROM faithful_testing);
CREATE TABLE

=> SELECT * FROM pred_faithful_results ORDER BY id;
 id  | eruptions |       pred
-----+-----------+------------------
   4 |     2.283 |  2.8151271587036
   5 |     4.533 | 4.62659045686076
   8 |       3.6 | 4.62659045686076
   9 |      1.95 | 1.94877514654148
  11 |     1.833 | 2.18505296804024
  12 |     3.917 | 4.54783118302784
  14 |      1.75 |  1.6337380512098
  20 |      4.25 | 4.15403481386324
  22 |      1.75 |  1.6337380512098
.
.
.
(110 rows)
```

Calculating the mean squared error (MSE)

You can calculate how well your model fits the data using the MSE function. MSE returns the average of the squared differences between actual value and predicted values.

```
=> SELECT MSE (eruptions::float, pred::float) OVER() FROM
  (SELECT eruptions, pred FROM pred_faithful_results) AS prediction_output;
       mse       |                  Comments
-----------------+----------------------------------------------
 0.252925741352641 | Of 110 rows, 110 were used and 0 were ignored
(1 row)
```

See also

- LINEAR_REG
- PREDICT_LINEAR_REG
- GET_MODEL_SUMMARY
- RSQUARED
- MSE

Poisson regression

Using Poisson regression, you can model count data. Poisson regression offers an alternative to linear regression or logistic regression and is useful when the target variable describes event frequency (event count in a fixed interval of time). Linear regression is preferable if you aim to predict continuous numerical outcomes in linear relationships, while logistic regression is used for predicting a binary classification.

You can use the following functions to build a Poisson regression model, view the model, and use the model to make predictions on a set of test data:

- POISSON_REG
- PREDICT_POISSON_REG
- GET_MODEL_SUMMARY

Random forest for regression

The Random Forest for regression algorithm creates an ensemble model of regression trees. Each tree is trained on a randomly selected subset of the training data. The algorithm predicts the value that is the mean prediction of the individual trees.

You can use the following functions to train the Random Forest model, and use the model to make predictions on a set of test data:

- GET_MODEL_SUMMARY
- PREDICT_RF_REGRESSOR
- RF_REGRESSOR

For a complete example of how to use the Random Forest for regression algorithm in Vertica, see Building a random forest regression model .

In this section

- Building a random forest regression model

Building a random forest regression model

This example uses the "mtcars" dataset to create a random forest model to predict the value of carb (the number of carburetors).

Before you begin the example, load the Machine Learning sample data .

1. Use RF_REGRESSOR to create the random forest model myRFRegressorModel using the mtcars training data. View the summary output of the model with GET_MODEL_SUMMARY :

```
=> SELECT RF_REGRESSOR ('myRFRegressorModel', 'mtcars', 'carb', 'mpg, cyl, hp, drat, wt' USING PARAMETERS
ntree=100, sampling_size=0.3);
RF_REGRESSOR
--------------
Finished
(1 row)


=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='myRFRegressorModel');
-------------------------------------------------------------------------------
===========
call_string
===========
SELECT rf_regressor('public.myRFRegressorModel', 'mtcars', '"carb"', 'mpg, cyl, hp, drat, wt'
USING PARAMETERS exclude_columns='', ntree=100, mtry=1, sampling_size=0.3, max_depth=5, max_breadth=32,
min_leaf_size=5, min_info_gain=0, nbins=32);

=======
details
=======
predictor|type
---------+-----
mpg      |float
cyl      | int
hp       | int
drat     |float
wt       |float
================
Additional Info
================
Name              |Value
------------------+-----
tree_count        | 100
rejected_row_count|  0
accepted_row_count| 32
(1 row)
```

2. Use PREDICT_RF_REGRESSOR to predict the number of carburetors:

```
=> SELECT PREDICT_RF_REGRESSOR (mpg,cyl,hp,drat,wt
USING PARAMETERS model_name='myRFRegressorModel') FROM mtcars;
PREDICT_RF_REGRESSOR
---------------------
2.94774203574204
2.6954087024087
2.6954087024087
2.89906346431346
2.97688489288489
2.97688489288489
2.7086587024087
2.92078965478965
2.97688489288489
2.7086587024087
2.95621822621823
2.82255155955156
2.7086587024087
2.7086587024087
2.85650394050394
2.85650394050394
2.97688489288489
2.95621822621823
2.6954087024087
2.6954087024087
2.84493251193251
2.97688489288489
2.97688489288489
2.8856467976468
2.6954087024087
2.92078965478965
2.97688489288489
2.97688489288489
2.7934087024087
2.7934087024087
2.7086587024087
2.72469441669442
(32 rows)
```

SVM (support vector machine) for regression

Support Vector Machine (SVM) for regression predicts continuous ordered variables based on the training data.

Unlike Logistic regression , which you use to determine a binary classification outcome, SVM for regression is primarily used to predict continuous numerical outcomes.

You can use the following functions to build an SVM for regression model, view the model, and use the model to make predictions on a set of test data:

- SVM_REGRESSOR
- PREDICT_SVM_REGRESSOR
- GET_MODEL_SUMMARY

For a complete example of how to use the SVM algorithm in Vertica, see Building an SVM for regression model .

In this section

- Building an SVM for regression model

Building an SVM for regression model

This SVM for regression example uses a small data set named faithful, based on the Old Faithful geyser in Yellowstone National Park. The data set contains values about the waiting time between eruptions and the duration of eruptions of the geyser. The example shows how you can build a model to predict the value of eruptions , given the value of the waiting feature.

Before you begin the example, load the Machine Learning sample data .

1. Create the SVM model, named svm_faithful , using the faithful_training training data:

```
=> SELECT SVM_REGRESSOR('svm_faithful', 'faithful_training', 'eruptions', 'waiting'
                    USING PARAMETERS error_tolerance=0.1, max_iterations=100);
       SVM_REGRESSOR
---------------------------
 Finished in 5 iterations
Accepted Rows: 162   Rejected Rows: 0
(1 row)
```

2. View the summary output of svm_faithful :

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='svm_faithful');


---------------------------------------------------------------------------
=======
details
=======

============================
Predictors and Coefficients
============================
         |Coefficients
---------+------------
Intercept|  -1.59007
waiting  |   0.07217
===========
call_string
===========
Call string:
SELECT svm_regressor('public.svm_faithful', 'faithful_training', '"eruptions"',
'waiting'USING PARAMETERS error_tolerance = 0.1, C=1, max_iterations=100,
epsilon=0.001);

===============
Additional Info
===============
Name             |Value
-----------------+-----
accepted_row_count| 162
rejected_row_count|  0
iteration_count  |  5
(1 row)
```

3. Create a new table that contains the response values from running the PREDICT_SVM_REGRESSOR function on your test data. Name this table pred_faithful_results. View the results in the pred_faithful_results table:

```
=> CREATE TABLE pred_faithful AS
      (SELECT id, eruptions, PREDICT_SVM_REGRESSOR(waiting USING PARAMETERS model_name='svm_faithful')
       AS pred FROM faithful_testing);
CREATE TABLE
=> SELECT * FROM pred_faithful ORDER BY id;
 id  | eruptions |       pred
-----+-----------+------------------
   4 |     2.283 | 2.88444568755189
   5 |     4.533 | 4.54434581879796
   8 |       3.6 | 4.54434581879796
   9 |      1.95 | 2.09058040739072
  11 |     1.833 | 2.30708912016195
  12 |     3.917 | 4.47217624787422
  14 |      1.75 | 1.80190212369576
  20 |      4.25 | 4.11132839325551
  22 |      1.75 | 1.80190212369576
.
.
.
(110 rows)
```

Calculating the mean squared error (MSE)

You can calculate how well your model fits the data is by using the MSE function. MSE returns the average of the squared differences between actual value and predicted values.

```
=> SELECT MSE(obs::float, prediction::float) OVER()
   FROM (SELECT eruptions AS obs, pred AS prediction
         FROM pred_faithful) AS prediction_output;
       mse        |                  Comments
------------------+----------------------------------------------
 0.254499811834235 | Of 110 rows, 110 were used and 0 were ignored
(1 row)
```

See also

- SVM (support vector machine) for regression
- SVM_REGRESSOR
- PREDICT_SVM_REGRESSOR
- GET_MODEL_SUMMARY

XGBoost for regression

XGBoost (eXtreme Gradient Boosting) is a popular supervised-learning algorithm used for regression and classification on large datasets. It uses sequentially-built shallow decision trees to provide accurate results and a highly-scalable training method that avoids overfitting.

The following XGBoost functions create and perform predictions with a regression model:

- XGB_REGRESSOR
- PREDICT_XGB_REGRESSOR

Example

This example uses a small data set named "mtcars", which contains design and performance data for 32 automobiles from 1973-1974, and creates an XGBoost regression model to predict the value of the variable carb (the number of carburetors).

Before you begin the example, load the Machine Learning sample data .

1. Use XGB_REGRESSOR to create the XGBoost regression model xgb_cars from the mtcars dataset:

```
=> SELECT XGB_REGRESSOR ('xgb_cars', 'mtcars', 'carb', 'mpg, cyl, hp, drat, wt'
    USING PARAMETERS learning_rate=0.5);
 XGB_REGRESSOR
---------------
 Finished
(1 row)
```

You can then view a summary of the model with GET_MODEL_SUMMARY :

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='xgb_cars');
                GET_MODEL_SUMMARY
-----------------------------------------------------------
===========
call_string
===========
xgb_regressor('public.xgb_cars', 'mtcars', '"carb"', 'mpg, cyl, hp, drat, wt'
USING PARAMETERS exclude_columns='', max_ntree=10, max_depth=5, nbins=32, objective=squarederror,
split_proposal_method=global, epsilon=0.001, learning_rate=0.5, min_split_loss=0, weight_reg=0, sampling_size=1)

=======
details
=======
predictor|      type
---------+----------------
   mpg   |float or numeric
   cyl   |      int
   hp    |      int
  drat   |float or numeric
   wt    |float or numeric

================
Additional Info
================
       Name       |Value
------------------+-----
    tree_count    | 10
rejected_row_count|  0
accepted_row_count| 32

(1 row)
```

2. Use PREDICT_XGB_REGRESSOR to predict the number of carburetors:

```
=> SELECT carb, PREDICT_XGB_REGRESSOR (mpg,cyl,hp,drat,wt USING PARAMETERS model_name='xgb_cars') FROM mtcars;
 carb | PREDICT_XGB_REGRESSOR
------+-----------------------
    4 |     4.00335213618023
    2 |      2.0038188946536
    6 |     5.98866003194438
    1 |     1.01774386191546
    2 |      1.9959801016274
    2 |      2.0038188946536
    4 |     3.99545403625739
    8 |     7.99211056556231
    2 |     1.99291901733151
    3 |      2.9975688946536
    3 |      2.9975688946536
    1 |     1.00320357711227
    2 |      2.0038188946536
    4 |     3.99545403625739
    4 |     4.00124134679445
    1 |     1.00759516721382
    4 |     3.99700517763435
    4 |     3.99580193056138
    4 |     4.00009088187525
    3 |      2.9975688946536
    2 |     1.98625064560888
    1 |     1.00355294416998
    2 |     2.00666247039502
    1 |     1.01682931210169
    4 |     4.00124134679445
    1 |     1.01007809485918
    2 |     1.98438405824605
    4 |     3.99580193056138
    2 |     1.99291901733151
    4 |     4.00009088187525
    2 |      2.0038188946536
    1 |     1.00759516721382
(32 rows)
```

Classification algorithms

Classification is an important and popular machine learning tool that assigns items in a data set to different categories. Classification is used to predict risk over time, in fraud detection, text categorization, and more. Classification functions begin with a data set where the different categories are known. For example, suppose you want to classify students based on how likely they are to get into graduate school. In addition to factors like admission score exams and grades, you could also track work experience.

Binary classification means the outcome, in this case, admission, only has two possible values: admit or do not admit. Multiclass outcomes have more than two values. For example, low, medium, or high chance of admission. During the training process, classification algorithms find the relationship between the outcome and the features. This relationship is summarized in the model, which can then be applied to different data sets, where the categories are unknown.

In this section

- Logistic regression
- Naive bayes
- Random forest for classification
- SVM (support vector machine) for classification
- XGBoost for classification

Logistic regression

Using logistic regression, you can model the relationship between independent variables, or features, and some dependent variable, or outcome. The outcome of logistic regression is always a binary value.

You can build logistic regression models to:

- Fit a predictive model to a training data set of independent variables and some binary dependent variable. Doing so allows you to make predictions on outcomes, such as whether a piece of email is spam mail or not.
- Determine the strength of the relationship between an independent variable and some binary outcome variable. For example, suppose you want to determine whether an email is spam or not. You can build a logistic regression model, based on observations of the properties of email messages. Then, you can determine the importance of various properties of an email message on that outcome.

You can use the following functions to build a logistic regression model, view the model, and use the model to make predictions on a set of test data:

- LOGISTIC_REG
- PREDICT_LOGISTIC_REG
- GET_MODEL_SUMMARY

For a complete programming example of how to use logistic regression on a table in Vertica, see Building a logistic regression model .

In this section

- Building a logistic regression model

Building a logistic regression model

This logistic regression example uses a small data set named mtcars. The example shows how to build a model that predicts the value of am , which indicates whether the car has an automatic or a manual transmission. It uses the given values of all the other features in the data set.

In this example, roughly 60% of the data is used as training data to create a model. The remaining 40% is used as testing data against which you can test your logistic regression model.

Before you begin the example, load the Machine Learning sample data .

1. Create the logistic regression model, named logistic_reg_mtcars , using the mtcars_train training data.

```
=> SELECT LOGISTIC_REG('logistic_reg_mtcars', 'mtcars_train', 'am', 'cyl, wt'
  USING PARAMETERS exclude_columns='hp');
      LOGISTIC_REG
---------------------------
 Finished in 15 iterations

(1 row)
```

2. View the summary output of logistic_reg_mtcars .

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='logistic_reg_mtcars');
-------------------------------------------------------------------------------
=======
details
=======
predictor|coefficient|  std_err  |z_value |p_value
---------+-----------+-----------+--------+--------
Intercept| 262.39898 |44745.77338| 0.00586| 0.99532
cyl      | 16.75892  |5987.23236 | 0.00280| 0.99777
wt       |-119.92116 |17237.03154|-0.00696| 0.99445

==============
regularization
==============
type| lambda
----+--------
none| 1.00000

===========
call_string
===========
logistic_reg('public.logistic_reg_mtcars', 'mtcars_train', '"am"', 'cyl, wt'
USING PARAMETERS exclude_columns='hp', optimizer='newton', epsilon=1e-06,
max_iterations=100, regularization='none', lambda=1)


================
Additional Info
================
Name             |Value
-----------------+-----
iteration_count   | 20
rejected_row_count|  0
accepted_row_count| 20
(1 row)
```

3. Create a table named mtcars_predict_results . Populate this table with the prediction outputs you obtain from running the PREDICT_LOGISTIC_REG function on your test data. View the results in the mtcars_predict_results table.

```
=> CREATE TABLE mtcars_predict_results AS
  (SELECT car_model, am, PREDICT_LOGISTIC_REG(cyl, wt
  USING PARAMETERS model_name='logistic_reg_mtcars')
  AS Prediction FROM mtcars_test);
CREATE TABLE


=> SELECT * FROM mtcars_predict_results;
  car_model    | am | Prediction
---------------+----+------------
 AMC Javelin    | 0 |          0
 Hornet 4 Drive | 0 |          0
 Maserati Bora  | 1 |          0
 Merc 280       | 0 |          0
 Merc 450SL     | 0 |          0
 Toyota Corona  | 0 |          1
 Volvo 142E     | 1 |          1
 Camaro Z28     | 0 |          0
 Datsun 710     | 1 |          1
 Honda Civic    | 1 |          1
 Porsche 914-2  | 1 |          1
 Valiant        | 0 |          0
(12 rows)
```

4. Evaluate the accuracy of the PREDICT_LOGISTIC_REG function, using the CONFUSION_MATRIX evaluation function.

```
=> SELECT CONFUSION_MATRIX(obs::int, pred::int USING PARAMETERS num_classes=2) OVER()
  FROM (SELECT am AS obs, Prediction AS pred FROM mtcars_predict_results) AS prediction_output;
 class | 0 | 1 |                   comment
-------+---+---+-------------------------------------------------
     0 | 6 | 1 |
     1 | 1 | 4 | Of 12 rows, 12 were used and 0 were ignored
(2 rows)
```

   In this case, PREDICT_LOGISTIC_REG correctly predicted that four out of five cars with a value of 1 in the am column have a value of 1 . Out of the seven cars which had a value of 0 in the am column, six were correctly predicted to have the value 0 . One car was incorrectly classified as having the value 1 .

See also

- CONFUSION_MATRIX
- LIFT_TABLE
- LOGISTIC_REG
- PREDICT_LOGISTIC_REG
- GET_MODEL_SUMMARY

Naive bayes

You can use the Naive Bayes algorithm to classify your data when features can be assumed independent. The algorithm uses independent features to calculate the probability of a specific class. For example, you might want to predict the probability that an email is spam. In that case, you would use a corpus of words associated with spam to calculate the probability the email's content is spam.

You can use the following functions to build a Naive Bayes model, view the model, and use the model to make predictions on a set of test data:

- NAIVE_BAYES
- PREDICT_NAIVE_BAYES
- PREDICT_NAIVE_BAYES_CLASSES
- GET_MODEL_SUMMARY

For a complete example of how to use the Naive Bayes algorithm in Vertica, see Classifying data using naive bayes .

In this section

- Classifying data using naive bayes

Classifying data using naive bayes

This Naive Bayes example uses the HouseVotes84 data set to show you how to build a model. With this model, you can predict which party the member of the United States Congress is affiliated based on their voting record. To aid in classifying the data it has been cleaned, and any missed votes have been replaced. The cleaned data replaces missed votes with the voter's party majority vote. For example, suppose a member of the Democrats had a missing value for vote1 and majority of the Democrats voted in favor. This example replaces all missing Democrats' votes for vote1 with a vote in favor.

In this example, approximately 75% of the cleaned HouseVotes84 data is randomly selected and copied to a training table. The remaining cleaned HouseVotes84 data is used as a testing table.

Before you begin the example, load the Machine Learning sample data .
You must also load the naive_bayes_data_prepration.sql script:

```
$ /opt/vertica/bin/vsql -d <name of your database> -f naive_bayes_data_preparation.sql
```

1. Create the Naive Bayes model, named naive_house84_model , using the house84_train training data.

```
=> SELECT NAIVE_BAYES('naive_house84_model', 'house84_train', 'party',
                      '*' USING PARAMETERS exclude_columns='party, id');
                NAIVE_BAYES
-----------------------------------------------------
 Finished. Accepted Rows: 315  Rejected Rows: 0
(1 row)
```

2. Create a new table, named predicted_party_naive . Populate this table with the prediction outputs you obtain from the PREDICT_NAIVE_BAYES function on your test data.

```
=> CREATE TABLE predicted_party_naive
    AS SELECT party,
        PREDICT_NAIVE_BAYES (vote1, vote2, vote3, vote4, vote5,
                             vote6, vote7, vote8, vote9, vote10,
                             vote11, vote12, vote13, vote14,
                             vote15, vote16
                              USING PARAMETERS model_name = 'naive_house84_model',
                                               type = 'response') AS Predicted_Party
      FROM house84_test;
CREATE TABLE
```

3. Calculate the accuracy of the model's predictions.

```
=> SELECT  (Predictions.Num_Correct_Predictions / Count.Total_Count) AS Percent_Accuracy
   FROM (  SELECT COUNT(Predicted_Party) AS Num_Correct_Predictions
       FROM predicted_party_naive
       WHERE party = Predicted_Party
        ) AS Predictions,
        (  SELECT COUNT(party) AS Total_Count
             FROM predicted_party_naive
          ) AS Count;
  Percent_Accuracy
----------------------
 0.933333333333333333
(1 row)
```

The model correctly predicted the party of the members of Congress based on their voting patterns with 93% accuracy.

Viewing the probability of each class

You can also view the probability of each class. Use PREDICT_NAIVE_BAYES_CLASSES to see the probability of each class.

```
=> SELECT PREDICT_NAIVE_BAYES_CLASSES (id, vote1, vote2, vote3, vote4, vote5,
                                vote6, vote7, vote8, vote9, vote10,
                                vote11, vote12, vote13, vote14,
                                vote15, vote16
                                USING PARAMETERS model_name = 'naive_house84_model',
                                            key_columns = 'id', exclude_columns = 'id',
                                            classes = 'democrat, republican')
      OVER() FROM house84_test;
 id  | Predicted |    Probability    |      democrat       |      republican
-----+-----------+-------------------+---------------------+----------------------
 368 | democrat  |                 1 |                   1 |                    0
 372 | democrat  |                 1 |                   1 |                    0
 374 | democrat  |                 1 |                   1 |                    0
 378 | republican | 0.999999962214987 | 3.77850125111219e-08 |    0.999999962214987
 384 | democrat  |                 1 |                   1 |                    0
 387 | democrat  |                 1 |                   1 |                    0
 406 | republican | 0.999999945980143 | 5.40198564592332e-08 |    0.999999945980143
 419 | democrat  |                 1 |                   1 |                    0
 421 | republican | 0.922808855631005 |   0.0771911443689949 |    0.922808855631005
.
.
.
(109 rows)
```

See also

- NAIVE_BAYES
- PREDICT_NAIVE_BAYES
- PREDICT_NAIVE_BAYES_CLASSES

Random forest for classification

The Random Forest algorithm creates an ensemble model of decision trees. Each tree is trained on a randomly selected subset of the training data.

You can use the following functions to train the Random Forest model, and use the model to make predictions on a set of test data:

- RF_CLASSIFIER
- PREDICT_RF_CLASSIFIER
- PREDICT_RF_CLASSIFIER_CLASSES
- GET_MODEL_SUMMARY

For a complete example of how to use the Random Forest algorithm in Vertica, see Classifying data using random forest .

In this section

- Classifying data using random forest

Classifying data using random forest

This random forest example uses a data set named iris. The example contains four variables that measure various parts of the iris flower to predict its species.

Before you begin the example, make sure that you have followed the steps in Download the machine learning example data .

1. Use RF_CLASSIFIER to create the random forest model, named rf_iris , using the iris data. View the summary output of the model with GET_MODEL_SUMMARY :

```
=> SELECT RF_CLASSIFIER ('rf_iris', 'iris', 'Species', 'Sepal_Length, Sepal_Width, Petal_Length, Petal_Width'
USING PARAMETERS ntree=100, sampling_size=0.5);

       RF_CLASSIFIER
----------------------------
Finished training

(1 row)


=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='rf_iris');
-------------------------------------------------------------------------------
===========
call_string
===========
SELECT rf_classifier('public.rf_iris', 'iris', '"species"', 'Sepal_Length, Sepal_Width, Petal_Length,
Petal_Width' USING PARAMETERS exclude_columns='', ntree=100, mtry=2, sampling_size=0.5, max_depth=5,
max_breadth=32, min_leaf_size=1, min_info_gain=0, nbins=32);

=======
details
=======
predictor   |type
------------+-----
sepal_length|float
sepal_width |float
petal_length|float
petal_width |float

================
Additional Info
================
Name              |Value
------------------+-----
tree_count        | 100
rejected_row_count|  0
accepted_row_count| 150
(1 row)
```

2. Apply the classifier to the test data with PREDICT_RF_CLASSIFIER :

```
=> SELECT PREDICT_RF_CLASSIFIER (Sepal_Length, Sepal_Width, Petal_Length, Petal_Width
                        USING PARAMETERS model_name='rf_iris') FROM iris1;

PREDICT_RF_CLASSIFIER
----------------------
setosa
setosa
setosa
.
.
.
versicolor
versicolor
versicolor
.
.
.
virginica
virginica
virginica
.
.
.
(90 rows)
```

3. Use PREDICT_RF_CLASSIFIER_CLASSES to view the probability of each class:

```
=> SELECT PREDICT_RF_CLASSIFIER_CLASSES(Sepal_Length, Sepal_Width, Petal_Length, Petal_Width
                        USING PARAMETERS model_name='rf_iris') OVER () FROM iris1;
predicted |    probability
-----------+-------------------
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |             0.99
.
.
.
(90 rows)
```

SVM (support vector machine) for classification

Support Vector Machine (SVM) is a classification algorithm that assigns data to one category or the other based on the training data. This algorithm implements linear SVM, which is highly scalable.

You can use the following functions to train the SVM model, and use the model to make predictions on a set of test data:

- SVM_CLASSIFIER
- PREDICT_SVM_CLASSIFIER
- GET_MODEL_SUMMARY

You can also use the following evaluation functions to gain further insights:

- CONFUSION_MATRIX
- DETECT_OUTLIERS
- ERROR_RATE

- ROC

For a complete example of how to use the SVM algorithm in Vertica, see Classifying data using SVM (support vector machine) .

The implementation of the SVM algorithm in Vertica is based on the paper Distributed Newton Methods for Regularized Logistic Regression .

In this section

- Classifying data using SVM (support vector machine)

Classifying data using SVM (support vector machine)

This SVM example uses a small data set named mtcars. The example shows how you can use the SVM_CLASSIFIER function to train the model to predict the value of am (the transmission type, where 0 = automatic and 1 = manual) using the PREDICT_SVM_CLASSIFIER function.

Before you begin the example, load the Machine Learning sample data .

1. Create the SVM model, named svm_class , using the mtcars_train training data.

```
=> SELECT SVM_CLASSIFIER('svm_class', 'mtcars_train', 'am', 'cyl, mpg, wt, hp, gear'
                         USING PARAMETERS exclude_columns='gear');
SVM_CLASSIFIER
--------------------------------------------------------------------
Finished in 12 iterations.
Accepted Rows: 20  Rejected Rows: 0
(1 row)
```

2. View the summary output of `svm_class` .

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='svm_class');
-------------------------------------------------------------------------------


=======
details
=======
predictor|coefficient
---------+-----------
Intercept| -0.02006
cyl      |  0.15367
mpg      |  0.15698
wt       | -1.78157
hp       |  0.00957

===========
call_string
===========
SELECT svm_classifier('public.svm_class', 'mtcars_train', '"am"', 'cyl, mpg, wt, hp, gear'
USING PARAMETERS exclude_columns='gear', C=1, max_iterations=100, epsilon=0.001);

===============
Additional Info
===============
Name             |Value
-----------------+-----
accepted_row_count| 20
rejected_row_count|  0
iteration_count   | 12
(1 row)
```

3. Create a new table, named svm_mtcars_predict . Populate this table with the prediction outputs you obtain from running the PREDICT_SVM_CLASSIFIER function on your test data.

```
=> CREATE TABLE svm_mtcars_predict AS
  (SELECT car_model, am, PREDICT_SVM_CLASSIFIER(cyl, mpg, wt, hp
                                 USING PARAMETERS model_name='svm_class')
                                 AS Prediction FROM mtcars_test);
CREATE TABLE
```

4. View the results in the svm_mtcars_predict table.

```
=> SELECT * FROM svm_mtcars_predict;
car_model     | am | Prediction
------------- +----+------------
Toyota Corona |  0 |          1
Camaro Z28    |  0 |          0
Datsun 710    |  1 |          1
Valiant       |  0 |          0
Volvo 142E    |  1 |          1
AMC Javelin   |  0 |          0
Honda Civic   |  1 |          1
Hornet 4 Drive|  0 |          0
Maserati Bora |  1 |          1
Merc 280      |  0 |          0
Merc 450SL    |  0 |          0
Porsche 914-2 |  1 |          1
(12 rows)
```

5. Evaluate the accuracy of the PREDICT_SVM_CLASSIFIER function, using the CONFUSION_MATRIX evaluation function.

```
=> SELECT CONFUSION_MATRIX(obs::int, pred::int USING PARAMETERS num_classes=2) OVER()
       FROM (SELECT am AS obs, Prediction AS pred FROM svm_mtcars_predict) AS prediction_output;
 class | 0 | 1 |                  comment
-------+---+---+-----------------------------------------------
     0 | 6 | 1 |
     1 | 0 | 5 | Of 12 rows, 12 were used and 0 were ignored
(2 rows)
```

   In this case, PREDICT_SVM_CLASSIFIER correctly predicted that the cars with a value of 1 in the am column have a value of 1 . No cars were incorrectly classified. Out of the seven cars which had a value of 0 in the am column, six were correctly predicted to have the value 0 . One car was incorrectly classified as having the value 1 .

See also

- SVM (support vector machine) for classification
- SVM_CLASSIFIER
- PREDICT_SVM_CLASSIFIER

XGBoost for classification

XGBoost (eXtreme Gradient Boosting) is a popular supervised-learning algorithm used for regression and classification on large datasets. It uses sequentially-built shallow decision trees to provide accurate results and a highly-scalable training method that avoids overfitting.

The following XGBoost functions create and perform predictions with a classification model:

- XGB_CLASSIFIER
- PREDICT_XGB_CLASSIFIER
- PREDICT_XGB_CLASSIFIER_CLASSES

Example

This example uses the "iris" dataset, which contains measurements for various parts of a flower, and can be used to predict its species and creates an XGBoost classifier model to classify the species of each flower.

Before you begin the example, load the Machine Learning sample data .

1. Use XGB_CLASSIFIER to create the XGBoost classifier model xgb_iris using the iris dataset:

```
=> SELECT XGB_CLASSIFIER ('xgb_iris', 'iris', 'Species', 'Sepal_Length, Sepal_Width, Petal_Length, Petal_Width'
   USING PARAMETERS max_ntree=10, max_depth=5, weight_reg=0.1, learning_rate=1);
 XGB_CLASSIFIER
----------------
 Finished
(1 row)
```

You can then view a summary of the model with GET_MODEL_SUMMARY :

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='xgb_iris');
                                                                                                                GET_MODEL_SUMMARY
-----------------------------------------------------
===========
call_string
===========
xgb_classifier('public.xgb_iris', 'iris', '"species"', 'Sepal_Length, Sepal_Width, Petal_Length, Petal_Width'
USING PARAMETERS exclude_columns='', max_ntree=10, max_depth=5, nbins=32, objective=crossentropy,
split_proposal_method=global, epsilon=0.001, learning_rate=1, min_split_loss=0, weight_reg=0.1, sampling_size=1)

=======
details
=======
 predictor  |      type
------------+----------------
sepal_length|float or numeric
sepal_width |float or numeric
petal_length|float or numeric
petal_width |float or numeric


===============
Additional Info
===============
       Name       |Value
------------------+-----
    tree_count    |  10
rejected_row_count|  0
accepted_row_count| 150

(1 row)
```

2. Use PREDICT_XGB_CLASSIFIER to apply the classifier to the test data:

```
=> SELECT PREDICT_XGB_CLASSIFIER (Sepal_Length, Sepal_Width, Petal_Length, Petal_Width
   USING PARAMETERS model_name='xgb_iris') FROM iris1;
 PREDICT_XGB_CLASSIFIER
------------------------
 setosa
 setosa
 setosa
 .
 .
 .
 versicolor
 versicolor
 versicolor
 .
 .
 .
 virginica
 virginica
 virginica
 .
 .
 .

(90 rows)
```

3. Use PREDICT_XGB_CLASSIFIER_CLASSES to view the probability of each class:

```
=> SELECT PREDICT_XGB_CLASSIFIER_CLASSES(Sepal_Length, Sepal_Width, Petal_Length, Petal_Width
   USING PARAMETERS model_name='xgb_iris') OVER (PARTITION BEST) FROM iris1;
 predicted |    probability
-----------+-------------------
 setosa    |   0.9999650465368
 setosa    |   0.9999650465368
 setosa    |   0.9999650465368
 setosa    |   0.9999650465368
 setosa    | 0.999911552783011
 setosa    |   0.9999650465368
 setosa    |   0.9999650465368
 setosa    |   0.9999650465368
 setosa    |   0.9999650465368
 setosa    |   0.9999650465368
 setosa    |   0.9999650465368
 setosa    |   0.9999650465368
 versicolor |  0.99991871763563
 .
 .
 .
(90 rows)
```

Clustering algorithms

Clustering is an important and popular machine learning tool used to find clusters of items in a data set that are similar to one another. The goal of clustering is to create clusters with a high number of objects that are similar. Similar to classification, clustering segments the data. However, in clustering, the categorical groups are not defined.

Clustering data into related groupings has many useful applications. If you already know how many clusters your data contains, the K-means algorithm may be sufficient to train your model and use that model to predict cluster membership for new data points.

However, in the more common case, you do not know before analyzing the data how many clusters it contains. In these cases, the Bisecting k-means algorithm is much more effective at finding the correct clusters in your data.

Both k-means and bisecting k-means predict the clusters for a given data set. A model trained using either algorithm can then be used to predict the cluster to which new data points are assigned.

Clustering can be used to find anomalies in data and find natural groups of data. For example, you can use clustering to analyze a geographical region and determine which areas of that region are most likely to be hit by an earthquake. For a complete example, see Earthquake Cluster Analysis Using the KMeans Approach .

In Vertica, clustering is computed based on Euclidean distance. Through this computation, data points are assigned to the cluster with the nearest center.

In this section

- K-means
- K-prototypes
- Bisecting k-means

K-means

You can use the *k-means* clustering algorithm to cluster data points into *k* different groups based on similarities between the data points.

k-means partitions *n* observations into *k* clusters. Through this partitioning, k-means assigns each observation to the cluster with the nearest mean, or *cluster center* .

For a complete example of how to use k-means on a table in Vertica, see Clustering data using k-means .

In this section

- Clustering data using k-means

Clustering data using k-means

This k-means example uses two small data sets: agar_dish_1 and agar_dish_2 . Using the numeric data in the agar_dish_1 data set, you can cluster the data into *k* clusters. Then, using the created k-means model, you can run APPLY_KMEANS on agar_dish_2 and assign them to the clusters created in your original model.

Before you begin the example, load the Machine Learning sample data .

Clustering training data into *k* clusters

1. Create the k-means model, named agar_dish_kmeans using the agar_dish_1 table data.

```
=> SELECT KMEANS('agar_dish_kmeans', 'agar_dish_1', '*', 5
                USING PARAMETERS exclude_columns ='id', max_iterations=20, output_view='agar_1_view',
                key_columns='id');
          KMEANS
---------------------------
 Finished in 7 iterations


(1 row)
```

   The example creates a model named agar_dish_kmeans and a view containing the results of the model named agar_1_view . You might get different results when you run the clustering algorithm. This is because KMEANS randomly picks initial centers by default.

2. View the output of agar_1_view .

```
=> SELECT * FROM agar_1_view;
 id  | cluster_id
-----+------------
  2 |          4
  5 |          4
  7 |          4
  9 |          4
 13 |          4
.
.
.
(375 rows)
```

3. Because you specified the number of clusters as 5, verify that the function created five clusters. Count the number of data points within each cluster.

```
=> SELECT cluster_id, COUNT(cluster_id) as Total_count
   FROM agar_1_view
   GROUP BY cluster_id;
 cluster_id | Total_count
------------+-------------
          0 |          76
          2 |          80
          1 |          74
          3 |          73
          4 |          72
(5 rows)
```

From the output, you can see that five clusters were created: 0 , 1 , 2 , 3 , and 4 .

You have now successfully clustered the data from agar_dish_1.csv into five distinct clusters.

Summarizing your model

View the summary output of agar_dish_means using the GET_MODEL_SUMMARY function.

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='agar_dish_kmeans');
-------------------------------------------------------------------------------
=======
centers
=======
x       |   y
--------+--------
0.49708 | 0.51116
-7.48119|-7.52577
-1.56238|-1.50561
-3.50616|-3.55703
-5.52057|-5.49197

=======
metrics
=======
Evaluation metrics:
  Total Sum of Squares: 6008.4619
  Within-Cluster Sum of Squares:
    Cluster 0: 12.083548
    Cluster 1: 12.389038
    Cluster 2: 12.639238
    Cluster 3: 11.210146
    Cluster 4: 12.994356
  Total Within-Cluster Sum of Squares: 61.316326
  Between-Cluster Sum of Squares: 5947.1456
  Between-Cluster SS / Total SS: 98.98%
Number of iterations performed: 2
Converged: True
Call:
kmeans('public.agar_dish_kmeans', 'agar_dish_1', '*', 5
USING PARAMETERS exclude_columns='id', max_iterations=20, epsilon=0.0001, init_method='kmeanspp',
distance_method='euclidean', output_view='agar_view_1', key_columns='id')
(1 row)
```

Clustering data using a k-means model

Using agar_dish_kmeans , the k-means model you just created, you can assign the points in agar_dish_2 to cluster centers.

Create a table named kmeans_results , using the agar_dish_2 table as your input table and the agar_dish_kmeans model for your initial cluster centers.

Add only the relevant feature columns to the arguments in the APPLY_KMEANS function.

```
=> CREATE TABLE kmeans_results AS
    (SELECT id,
          APPLY_KMEANS(x, y
                USING PARAMETERS
                      model_name='agar_dish_kmeans') AS cluster_id
     FROM agar_dish_2);
```

The kmeans_results table shows that the agar_dish_kmeans model correctly clustered the agar_dish_2 data.

See also

- APPLY_KMEANS
- KMEANS
- GET_MODEL_SUMMARY

K-prototypes

You can use the *k-prototypes* clustering algorithm to cluster mixed data into different groups based on similarities between the data points. The k-prototypes algorithm extends the functionality of k-means clustering, which is limited to numerical data, by combining it with k-modes clustering, a clustering algorithm for categorical data.

See the syntax for k-prototypes here .

Bisecting k-means

The *bisecting k-means* clustering algorithm combines k-means clustering with divisive hierarchy clustering. With bisecting k-means, you get not only the clusters but also the hierarchical structure of the clusters of data points.

This hierarchy is more informative than the unstructured set of flat clusters returned by K-means . The hierarchy shows how the clustering results would look at every step of the process of bisecting clusters to find new clusters. The hierarchy of clusters makes it easier to decide the number of clusters in the data set.

Given a hierarchy of *k* clusters produced by bisecting k-means, you can easily calculate any prediction of the form: Assume the data contain only *k'* clusters, where *k'* is a number that is smaller than or equal to the *k* used to train the model.

For a complete example of how to use bisecting k-means to analyze a table in Vertica, see Clustering data hierarchically using bisecting k-means .

In this section

- Clustering data hierarchically using bisecting k-means

Clustering data hierarchically using bisecting k-means

This bisecting k-means example uses two small data sets named agar_dish_training and agar_dish_testing. Using the numeric data in the agar_dish_training data set, you can cluster the data into *k* clusters. Then, using the resulting bisecting k-means model, you can run APPLY_BISECTING_KMEANS on agar_dish_testing and assign the data to the clusters created in your trained model. Unlike regular k-means (also provided in Vertica), bisecting k-means allows you to predict with any number of clusters less than or equal to *k* . So if you train the model with *k* =5 but later decide to predict with *k* =2, you do not have to retrain the model; just run APPLY_BISECTING_KMEANS with *k* =2.

Before you begin the example, load the Machine Learning sample data . For this example, we load agar_dish_training.csv and agar_dish_testing.csv .

Clustering training data into *k* clusters to train the model

1. Create the bisecting k-means model, named agar_dish_bkmeans, using the agar_dish_training table data.

   ```
   => SELECT BISECTING_KMEANS('agar_dish_bkmeans', 'agar_dish_training', '*', 5 USING PARAMETERS exclude_columns='id', key_columns='id',
   output_view='agar_1_view');
    BISECTING_KMEANS
   ------------------
    Finished.
   (1 row)
   ```

   This example creates a model named agar_dish_bkmeans and a view containing the results of the model named agar_1_view. You might get slightly different results when you run the clustering algorithm. This is because BISECTING_KMEANS uses random numbers to generate the best clusters.
2. View the output of agar_1_view.

```
=> SELECT * FROM agar_1_view;
 id  | cluster_id
-----+------------
  2 |          4
  5 |          4
  7 |          4
  9 |          4
...
```

Here we can see the id of each point in the agar_dish_training table and which cluster it has been assigned to.

3. Because we specified the number of clusters as 5, verify that the function created five clusters by counting the number of data points within each cluster.

```
=> SELECT cluster_id, COUNT(cluster_id) as Total_count FROM agar_1_view GROUP BY cluster_id;
 cluster_id | Total_count
------------+-------------
          5 |          76
          7 |          73
          8 |          74
          4 |          72
          6 |          80
(5 rows)
```

You may wonder why the cluster_ids do not start at 0 or 1. The reason is that the bisecting k-means algorithm generates many more clusters than k-means, and then outputs the ones that are needed for the designated value of *k* . We will see later why this is useful.

You have now successfully clustered the data from agar_dish_training.csv into five distinct clusters.

Summarizing your model

View the summary output of agar_dish_bkmeans using the GET_MODEL_SUMMARY function.

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='agar_dish_bkmeans');
======
BKTree
======
center_id|   x    |   y    | withinss |totWithinss|bisection_level|cluster_size|parent|left_child|right_child
---------+--------+--------+----------+-----------+---------------+------------+------+----------+-----------
   0    |-3.59450|-3.59371|6008.46192|6008.46192 |       0       |    375     |      |    1     |     2
   1    |-6.47574|-6.48280|336.41161 |1561.29110 |       1       |    156     |  0   |    5     |     6
   2    |-1.54210|-1.53574|1224.87949|1561.29110 |       1       |    219     |  0   |    3     |     4
   3    |-2.54088|-2.53830|317.34228 | 665.83744 |       2       |    147     |  2   |    7     |     8
   4    | 0.49708| 0.51116| 12.08355 | 665.83744 |       2       |     72     |  2   |          |
   5    |-7.48119|-7.52577| 12.38904 | 354.80922 |       3       |     76     |  1   |          |
   6    |-5.52057|-5.49197| 12.99436 | 354.80922 |       3       |     80     |  1   |          |
   7    |-1.56238|-1.50561| 12.63924 | 61.31633  |       4       |     73     |  3   |          |
   8    |-3.50616|-3.55703| 11.21015 | 61.31633  |       4       |     74     |  3   |          |


=======
Metrics
=======
                    Measure                     |  Value
------------------------------------------------------------+----------
               Total sum of squares             |6008.46192
         Total within-cluster sum of squares    | 61.31633
          Between-cluster sum of squares        |5947.14559
Between-cluster sum of squares / Total sum of squares| 98.97950
      Sum of squares for cluster 1, center_id 5      | 12.38904
      Sum of squares for cluster 2, center_id 6      | 12.99436
      Sum of squares for cluster 3, center_id 7      | 12.63924
      Sum of squares for cluster 4, center_id 8      | 11.21015
      Sum of squares for cluster 5, center_id 4      | 12.08355


===========
call_string
===========
bisecting_kmeans('agar_dish_bkmeans', 'agar_dish_training', '*', 5
USING PARAMETERS exclude_columns='id', bisection_iterations=1, split_method='SUM_SQUARES', min_divisible_cluster_size=2,
distance_method='euclidean', kmeans_center_init_method='kmeanspp', kmeans_epsilon=0.0001, kmeans_max_iterations=10, output_view=''agar_1_view'',
key_columns=''id'')

===============
Additional Info
===============
       Name         |Value
--------------------+-----
  num_of_clusters   |  5
dimensions_of_dataset|  2
num_of_clusters_found|  5
 height_of_BKTree   |  4

(1 row)
```

Here we can see the details of all the intermediate clusters created by bisecting k-means during training, some metrics for evaluating the quality of the clustering (the lower the sum of squares, the better), the specific parameters with which the algorithm was trained, and some general information about the data algorithm.

Clustering testing data using a bisecting k-means model

Using agar_dish_bkmeans, the bisecting k-means model you just created, you can assign the points in agar_dish_testing to cluster centers.

1. Create a table named bkmeans_results, using the agar_dish_testing table as your input table and the agar_dish_bkmeans model for your cluster centers. Add only the relevant feature columns to the arguments in the APPLY_BISECTING_KMEANS function.

```
=> CREATE TABLE bkmeans_results_k5 AS
      (SELECT id,
            APPLY_BISECTING_KMEANS(x, y
                  USING PARAMETERS
                        model_name='agar_dish_bkmeans', number_clusters=5) AS cluster_id
       FROM agar_dish_testing);
=> SELECT cluster_id, COUNT(cluster_id) as Total_count FROM bkmeans_results_k5 GROUP BY cluster_id;
 cluster_id | Total_count
------------+-------------
          5 |          24
          4 |          28
          6 |          20
          8 |          26
          7 |          27
(5 rows)
```

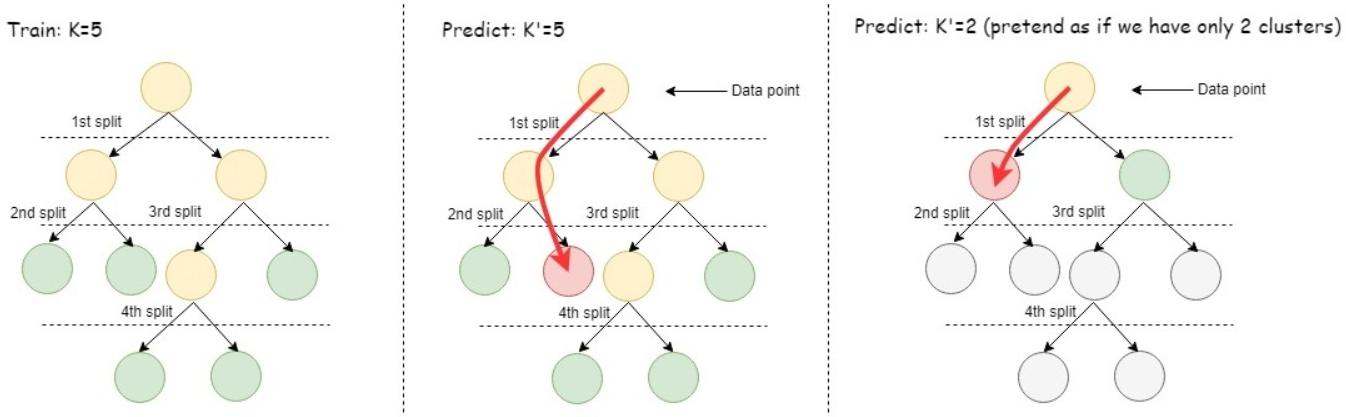
   The bkmeans_results_k5 table shows that the agar_dish_bkmeans model correctly clustered the agar_dish_testing data.
2. The real advantage of using bisecting k-means is that the model it creates can cluster data into any number of clusters less than or equal to the *k* with which it was trained. Now you could cluster the above testing data into 3 clusters instead of 5, without retraining the model:

```
=> CREATE TABLE bkmeans_results_k3 AS
      (SELECT id,
            APPLY_BISECTING_KMEANS(x, y
                  USING PARAMETERS
                        model_name='agar_dish_bkmeans', number_clusters=3) AS cluster_id
       FROM agar_dish_testing);
=> SELECT cluster_id, COUNT(cluster_id) as Total_count FROM bkmeans_results_k3 GROUP BY cluster_id;
 cluster_id | Total_count
------------+-------------
          4 |          28
          3 |          53
          1 |          44
(3 rows)
```

Prediction using the trained bisecting k-means model

To cluster data using a trained model, the bisecting k-means algorithm starts by comparing the incoming data point with the child cluster centers of the root cluster node. The algorithm finds which of those centers the data point is closest to. Then the data point is compared with the child cluster centers of the closest child of the root. The prediction process continues to iterate until it reaches a leaf cluster node. Finally, the point is assigned to the closest leaf cluster. The following picture gives a simple illustration of the training process and prediction process of the bisecting k-means algorithm. An advantage of using bisecting k-means is that you can predict using any value of *k* from 2 to the largest *k* value the model was trained on.

The model in the picture below was trained on *k* =5. The middle picture shows using the model to predict with *k* =5, in other words, match the incoming data point to the center with the closest value, in the level of the hierarchy where there are 5 leaf clusters. The picture on the right shows using the model to predict *as if k* =2, in other words, first compare the incoming data point to the leaf clusters at the level where there were only two clusters, then match the data point to the closer of those two cluster centers. This approach is faster than predicting with k-means.



See also

- APPLY_BISECTING_KMEANS
- BISECTING_KMEANS
- GET_MODEL_SUMMARY

Time series forecasting

Time series models are trained on stationary time series (that is, time series where the mean doesn't change over time) of stochastic processes with consistent time steps. These algorithms forecast future values by taking into account the influence of values at some number of preceding timesteps (lags).

Examples of applicable datasets include those for temperature, stock prices, earthquakes, product sales, etc.

To normalize datasets with inconsistent timesteps, see Gap filling and interpolation (GFI) .

In this section

- ARIMA model example
- Autoregressive model example
- Moving-average model example

ARIMA model example

Autoregressive integrated moving average (ARIMA) models combine the abilities of AUTOREGRESSOR and MOVING_AVERAGE models by making future predictions based on both preceding time series values and errors of previous predictions. At model training time, you specify the number of preceding values and previous prediction errors that the model will use to calculate predictions.

You can use the following functions to train and make predictions with ARIMA models:

- ARIMA : Creates and trains an ARIMA model
- PREDICT_ARIMA : Applies a trained ARIMA model to an input relation or makes predictions using the in-sample data

These functions require time series data with consistent timesteps. To normalize a time series with inconsistent timesteps, see Gap filling and interpolation (GFI) .

The following example trains an ARIMA model on a daily temperature dataset and then makes temperature predictions with both possible prediction methods—using the in-sample data and applying the model to an input relation.

Load the training data

Before you begin the example, load the Machine Learning sample data .
This examples uses the daily-min-temperatures dataset, which contains data on the daily minimum temperature in Melbourne, Australia from 1981 through 1990. The data is available in the temp_data table:

```
=> SELECT * FROM temp_data;
        time         | Temperature
---------------------+-------------
 1981-01-01 00:00:00 |        20.7
 1981-01-02 00:00:00 |        17.9
 1981-01-03 00:00:00 |        18.8
 1981-01-04 00:00:00 |        14.6
 1981-01-05 00:00:00 |        15.8
 ...
 1990-12-27 00:00:00 |          14
 1990-12-28 00:00:00 |        13.6
 1990-12-29 00:00:00 |        13.5
 1990-12-30 00:00:00 |        15.7
 1990-12-31 00:00:00 |          13
(3650 rows)

=> SELECT COUNT(*) FROM temp_data;
 COUNT
-------
3650
(1 row)
```

Train the ARIMA model

After you load the data, you can use the ARIMA function to create and train an ARIMA model. For this example, the model is trained with lags of p =3 and q =3, taking the value and prediction error of three previous time steps into account for each prediction:

```
=> SELECT ARIMA('arima_temp', 'temp_data', 'temperature', 'time' USING PARAMETERS p=3, q=3);
                          ARIMA
-----------------------------------------------------------------
Finished in 20 iterations.
3650 elements accepted, 0 elements rejected.

(1 row)
```

You can view a summary of the model with the GET_MODEL_SUMMARY function:

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='arima_temp');
               GET_MODEL_SUMMARY
-------------------------------------------------


============
coefficients
============
parameter| value
---------+--------
  phi_1  | 0.64189
  phi_2  | 0.46667
  phi_3  |-0.11777
 theta_1 |-0.05109
 theta_2 |-0.58699
 theta_3 |-0.15882



==============
regularization
==============
none

===============
timeseries_name
===============
temperature

==============
timestamp_name
==============
time

==============
missing_method
==============
linear_interpolation

===========
call_string
===========
ARIMA('public.arima_temp', 'temp_data', 'temperature', 'time' USING PARAMETERS p=3, d=0, q=3, missing='linear_interpolation', init_method='Zero',
epsilon=1e-06, max_iterations=100);

===============
Additional Info
===============
       Name       | Value
------------------+--------
        p         |   3
        q         |   3
        d         |   0
       mean       |11.17775
      lambda      | 1.00000
mean_squared_error| 5.80490
rejected_row_count|   0
accepted_row_count|  3650

(1 row)
```

Make predictions

After you train the ARIMA model, you can call the PREDICT_ARIMA function to predict future temperatures. This function supports making predictions using the in-sample data that the model was trained on or applying the model to an input relation.

Using in-sample data

The following example makes predictions using the in-sample data that the model was trained on. The model begins prediction at the end of the temp_data table and returns predicted values for ten timesteps:

```
=> SELECT PREDICT_ARIMA(USING PARAMETERS model_name='arima_temp', start=0, npredictions=10) OVER();
   prediction
------------------
12.9745063293842
13.4389080858551
13.3955791360528
13.3551146487462
13.3149336514747
13.2750516811057
13.2354710353376
13.1961939790513
13.1572226788109
13.1185592045127
(10 rows)
```

For both prediction methods, if you want the function to return the standard error of each prediction, you can set output_standard_errors to true:

```
=> SELECT PREDICT_ARIMA(USING PARAMETERS model_name='arima_temp', start=0, npredictions=10, output_standard_errors=true) OVER();
    prediction    |     std_err
------------------+------------------
 12.9745063293842 | 1.00621890780865
 13.4389080858551 | 1.45340836833232
 13.3955791360528 | 1.61041524562932
 13.3551146487462 | 1.76368421116143
 13.3149336514747 | 1.91223938476627
 13.2750516811057 | 2.05618464609977
 13.2354710353376 | 2.19561771498385
 13.1961939790513 | 2.33063553781651
 13.1572226788109 | 2.46133422924445
 13.1185592045127 | 2.58780904243988
(10 rows)
```

Applying to an input relation

You can also apply the model to an input relation, in this case the temp_data training set:

```
=> SELECT PREDICT_ARIMA(temperature USING PARAMETERS model_name='arima_temp', start=3651, npredictions=10, output_standard_errors=true)
OVER(ORDER BY time) FROM temp_data;
    prediction    |     std_err
------------------+------------------
 12.9745063293842 | 1.00621890780865
 13.4389080858551 | 1.45340836833232
 13.3955791360528 | 1.61041524562932
 13.3551146487462 | 1.76368421116143
 13.3149336514747 | 1.91223938476627
 13.2750516811057 | 2.05618464609977
 13.2354710353376 | 2.19561771498385
 13.1961939790513 | 2.33063553781651
 13.1572226788109 | 2.46133422924445
 13.1185592045127 | 2.58780904243988
(10 rows)
```

Because the same data and relative start index were provided to both prediction methods, the model predictions for each method are identical.

When applying a model to an input relation, you can set add_mean to false so that the function returns the predicted difference from the mean instead of the sum of the model mean and the predicted difference:

```
=> SELECT PREDICT_ARIMA(temperature USING PARAMETERS model_name='arima_temp', start=3680, npredictions=10, add_mean=false)
OVER(ORDER BY time) FROM temp_data;
  prediction
------------------
1.2026877112171
1.17114068517961
1.13992534953432
1.10904183333367
1.0784901998692
1.04827044781798
1.01838251238116
0.98882626641461
0.959601521551628
0.93070802931751
(10 rows)
```

See also

- ARIMA
- PREDICT_ARIMA
- GET_MODEL_SUMMARY
- MSE

Autoregressive model example

Autoregressive models predict future values of a time series based on the preceding values. More specifically, the user-specified *lag* determines how many previous timesteps it takes into account during computation, and predicted values are linear combinations of the values at each lag.

Use the following functions when training and predicting with autoregressive models. Note that these functions require datasets with consistent timesteps.

- AUTOREGRESSOR : trains an autoregressive model
- PREDICT_AUTOREGRESSOR : applies the model to a dataset to make predictions

To normalize datasets with inconsistent timesteps, see Gap filling and interpolation (GFI) .

Example

1. Load the datasets from the Machine-Learning-Examples repository.
   This example uses the daily-min-temperatures dataset, which contains data on the daily minimum temperature in Melbourne, Australia from 1981 through 1990:
   ```
   => SELECT * FROM temp_data;
          time         | Temperature
   --------------------+-------------
    1981-01-01 00:00:00 |        20.7
    1981-01-02 00:00:00 |        17.9
    1981-01-03 00:00:00 |        18.8
    1981-01-04 00:00:00 |        14.6
    1981-01-05 00:00:00 |        15.8
   ...
    1990-12-27 00:00:00 |          14
    1990-12-28 00:00:00 |        13.6
    1990-12-29 00:00:00 |        13.5
    1990-12-30 00:00:00 |        15.7
    1990-12-31 00:00:00 |          13
   (3650 rows)
   ```
2. Use AUTOREGRESSOR to create the autoregressive model AR_temperature from the temp_data dataset. In this case, the model is trained with a lag of p =3, taking the previous 3 entries into account for each estimation:
   ```
   => SELECT AUTOREGRESSOR('AR_temperature', 'temp_data', 'Temperature', 'time' USING PARAMETERS p=3);
                       AUTOREGRESSOR
   ------------------------------------------------------------
    Finished. 3650 elements accepted, 0 elements rejected.
   (1 row)
   ```

You can view a summary of the model with GET_MODEL_SUMMARY :

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='AR_temperature');

 GET_MODEL_SUMMARY
-------------------

============
coefficients
============
parameter| value
---------+--------
  alpha  | 1.88817
phi_(t-1)| 0.70004
phi_(t-2)|-0.05940
phi_(t-3)| 0.19018


===================
mean_squared_error
===================
not evaluated

===========
call_string
===========
autoregressor('public.AR_temperature', 'temp_data', 'temperature', 'time'
USING PARAMETERS p=3, missing=linear_interpolation, regularization='none', lambda=1, compute_mse=false);

===============
Additional Info
===============
       Name       |Value
------------------+-----
    lag_order     |  3
rejected_row_count|  0
accepted_row_count|3650
(1 row)
```

3. Use PREDICT_AUTOREGRESSOR to predict future temperatures. The following query start s the prediction at the end of the dataset and returns 10 predictions.

```
=> SELECT PREDICT_AUTOREGRESSOR(Temperature USING PARAMETERS model_name='AR_temperature', npredictions=10) OVER(ORDER BY time) FROM temp_data;

    prediction
------------------
 12.6235419917807
 12.9387860506032
 12.6683380680058
 12.3886937385419
 12.2689506237424
 12.1503023330142
 12.0211734746741
 11.9150531529328
 11.825870404008
 11.7451846722395
(10 rows)
```

Moving-average model example

Moving average models use the errors of previous predictions to make future predictions. More specifically, the user-specified *lag* determines how many previous predictions and errors it takes into account during computation.

Use the following functions when training and predicting with moving-average models. Note that these functions require datasets with consistent timesteps.

- MOVING_AVERAGE : trains a moving-average model
- PREDICT_MOVING_AVERAGE : applies the model to a dataset to make predictions

To normalize datasets with inconsistent timesteps, see Gap filling and interpolation (GFI) .

Example

1. Load the datasets from the Machine-Learning-Examples repository.
   This example uses the daily-min-temperatures dataset, which contains data on the daily minimum temperature in Melbourne, Australia from 1981 through 1990:

```
=> SELECT * FROM temp_data;
        time         | Temperature
---------------------+-------------
 1981-01-01 00:00:00 |        20.7
 1981-01-02 00:00:00 |        17.9
 1981-01-03 00:00:00 |        18.8
 1981-01-04 00:00:00 |        14.6
 1981-01-05 00:00:00 |        15.8
...
 1990-12-27 00:00:00 |          14
 1990-12-28 00:00:00 |        13.6
 1990-12-29 00:00:00 |        13.5
 1990-12-30 00:00:00 |        15.7
 1990-12-31 00:00:00 |          13
(3650 rows)
```

2. Use MOVING_AVERAGE to create the moving-average model MA_temperature from the temp_data dataset. In this case, the model is trained with a lag of p =3, taking the error of 3 previous predictions into account for each estimation:

```
=> SELECT MOVING_AVERAGE('MA_temperature', 'temp_data', 'temperature', 'time' USING PARAMETERS q=3, missing='linear_interpolation',
regularization='none', lambda=1);
                    MOVING_AVERAGE
-----------------------------------------------------------
 Finished. 3650 elements accepted, 0 elements rejected.
(1 row)
```

   You can view a summary of the model with GET_MODEL_SUMMARY :

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='MA_temperature');

 GET_MODEL_SUMMARY
--------------------

=============
coefficients
=============
parameter| value
---------+--------
phi_(t-0)|-0.90051
phi_(t-1)|-0.10621
phi_(t-2)| 0.07173


================
timeseries_name
================
temperature

===============
timestamp_name
===============
time

===========
call_string
===========
moving_average('public.MA_temperature', 'temp_data', 'temperature', 'time'
USING PARAMETERS q=3, missing=linear_interpolation, regularization='none', lambda=1);

================
Additional Info
================
       Name       | Value
------------------+--------
       mean       |11.17780
    lag_order     |   3
      lambda      | 1.00000
rejected_row_count|   0
accepted_row_count|  3650

(1 row)
```

3. Use PREDICT_MOVING_AVERAGE to predict future temperatures. The following query start s the prediction at the end of the dataset and returns 10 predictions.

```
=> SELECT PREDICT_MOVING_AVERAGE(Temperature USING PARAMETERS model_name='MA_temperature', npredictions=10) OVER(ORDER BY time) FROM temp_data;

   prediction
------------------
13.1324365636272
12.8071086272833
12.7218966671721
12.6011086656032
12.506624729879
12.4148247026733
12.3307873804812
12.2521385975133
12.1789741993396
12.1107640076638
(10 rows)
```

Model management

Vertica provides a number of tools to manage existing models. You can view model summaries and attributes, alter model characteristics like name and privileges, drop models, and version models.

In this section

- Model versioning
- Altering models
- Dropping models
- Managing model security
- Viewing model attributes
- Summarizing models
- Viewing models

Model versioning

Model versioning provides an infrastructure to track and manage the status of registered models in a database. The versioning infrastructure supports a collaborative environment where multiple users can submit candidate models for individual applications, which are identified by their *registered_name* . Models in Vertica are by default unregistered, but any user with sufficient privileges can register a model to an application and add it to the versioning environment.

Register models

When a user has a candidate model ready to submit to an application, they can register the model using the REGISTER_MODEL function, which accepts a *model_name* and *registered_name* . The registered model is assigned an initial status of 'under_review' and is visible in the REGISTERED_MODELS system table to users with USAGE privileges.

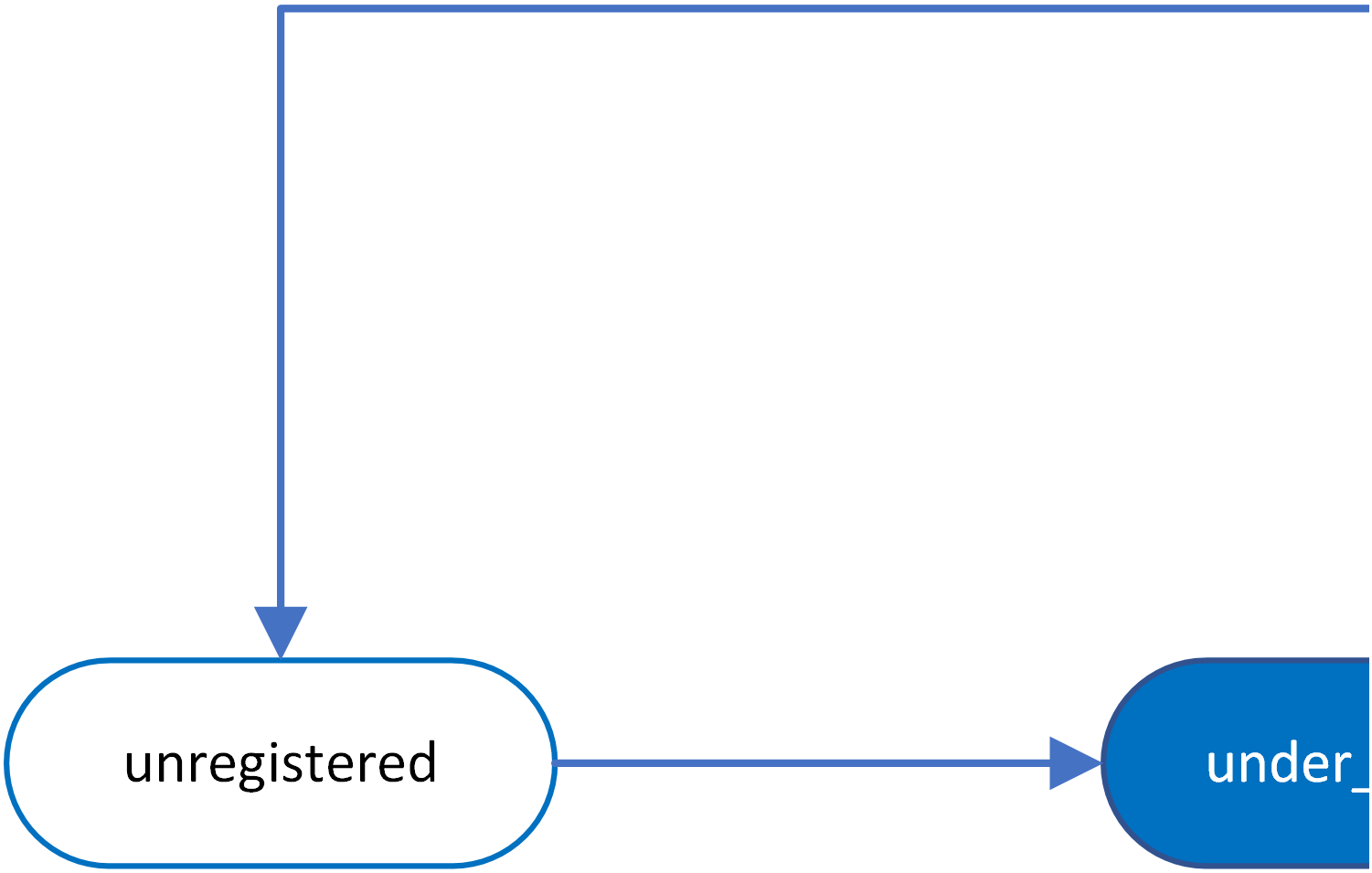
All registered models under a given *registered_name* are considered different versions of the same model, regardless of trainer, algorithm type, or production status. If a model is the first to be registered to a given *registered_name* , the model is assigned a *registered_version* of one. Otherwise, newly registered models are assigned an incremented *registered_version* of $n + 1$, where n is the number of models already registered to the given *registered_name* . Each registered model can be uniquely identified by the combination of *registered_name* and *registered_version* .

Change model status

After a model is registered, the model owner is automatically changed to superuser and the previous owner is given USAGE privileges. The MLSUPERVISOR role has full privileges over all registered models, as does dbadmin. Only users with these two roles can call the CHANGE_MODEL_STATUS function to change the status of registered models. There are six possible statuses for registered models:

- under_review : Status assigned to newly registered models.
- staging : Model is targeted for A/B testing against the model currently in production.
- production : Model is in production for its specified application. Only one model can be in production for a given *registered_name* at one time.
- archived : Status of models that were previously in production. Archived models can be returned to production at any time.
- declined : Model is no longer in consideration for production.
- unregistered : Model is removed from the versioning environment. The model does not appear in the REGISTERED_MODELS system table.

The following diagram depicts the valid status transitions:



Note
If you change the status of a model to 'production' and there is already a model in production under the given *registered_name* , the status of the new model is set to 'production' and that of the old model to 'archived'.

You can view the status history of registered models with the MODEL_STATUS_HISTORY system table, which includes models that have been unregistered or dropped. Only superusers or users to whom they have granted sufficient privileges can query the table. Vertica recommends granting access on the table to the MLSUPERVISOR role.

Only users with the MLSUPERVISOR role, or those granted equivalent privileges, can drop or alter registered models. As with unregistered models, you can drop and alter registered models with the DROP MODEL and ALTER MODEL commands. Dropped models no longer appear in the REGISTERED_MODELS system table.

To make predictions with a registered model, you must use the [*schema_name.*] *model_name* and predict function for the appropriate model type. You can find all necessary information in the REGISTERED_MODELS system table.

Model registration does not effect the process of exporting models. However, model registration information, such as *registered_version* , is not captured in exported models. All imported models are initially unregistered, but each category of imported model—native Vertica, PMML, and TensorFlow—is compatible with model versioning.

Examples

The following example demonstrates a possible model versioning workflow. This workflow begins with registering multiple models to an application, continues with an MLSUPERVISOR managing and changing the status of those models, and concludes with one of the models in production.

First, the models must be registered to an application. In this case, the models, native_linear_reg and linear_reg_spark1 , are registered as the linear_reg_app application:

```
=> SELECT REGISTER_MODEL('native_linear_reg', 'linear_reg_app');
                    REGISTER_MODEL
-----------------------------------------------------------------------
Model [native_linear_reg] is registered as [linear_reg_app], version [2]
(1 row)

=> SELECT REGISTER_MODEL('linear_reg_spark1', 'linear_reg_app');
                    REGISTER_MODEL
-----------------------------------------------------------------------
Model [linear_reg_spark1] is registered as [linear_reg_app], version [3]
(1 row)
```

You can query the REGISTERED_MODELS system table to view details about the newly registered models, such as the version number and model status:

```
=> SELECT * FROM REGISTERED_MODELS;
 registered_name | registered_version |    status    |       registered_time        |     model_id     | schema_name |    model_name    |      model_type      |
category
-----------------+--------------------+--------------+------------------------------+------------------+-------------+------------------+----------------------+----------------
 linear_reg_app  |                  3 | UNDER_REVIEW | 2023-01-26 09:52:00.082166-04 | 45035996273714020 | public      | linear_reg_spark1 |
PMML_REGRESSION_MODEL | PMML
 linear_reg_app  |                  2 | UNDER_REVIEW | 2023-01-26 09:51:04.553102-05 | 45035996273850350 | public      | native_linear_reg |
LINEAR_REGRESSION     | VERTICA_MODELS
 linear_reg_app  |                  1 | PRODUCTION   | 2023-01-24 05:29:25.990626-02 | 45035996273853740 | public      | linear_reg_newton |
LINEAR_REGRESSION     | VERTICA_MODELS
 logistic_reg_app |                  1 | PRODUCTION   | 2023-01-23 08:49:25.990626-02 | 45035996273853740 | public      | log_reg_cgd       |
LOGISTIC_REGRESSION   | VERTICA_MODELS
(4 rows)
```

If you query the MODELS system table, you can see that the model owner has automatically been changed to superuser, dbadmin in this case:

```
=> SELECT model_name, owner_name FROM MODELS WHERE model_name IN ('native_linear_reg', 'linear_reg_spark1');
    model_name    | owner_name
------------------+------------
 native_linear_reg | dbadmin
 linear_reg_spark1 | dbadmin
(2 rows)
```

As a user with the MLSUPERVISOR role enabled, you can then change the status of both models to 'staging' with the CHANGE_MODEL_STATUS function, which accepts a *registered_name* and *registered_version* :

```
=> SELECT CHANGE_MODEL_STATUS('linear_reg_app', 2, 'staging');
                    CHANGE_MODEL_STATUS
-------------------------------------------------------------------------------
The status of model [linear_reg_app] - version [2] is changed to [staging]
(1 row)

=> SELECT CHANGE_MODEL_STATUS('linear_reg_app', 3, 'staging');
                    CHANGE_MODEL_STATUS
-------------------------------------------------------------------------------
The status of model [linear_reg_app] - version [3] is changed to [staging]
(1 row)
```

After comparing the evaluation metrics of the two staged models against the model currently in production, you can put the better performing model into production. In this case, the linear_reg_spark1 model is moved into production and the linear_reg_spark1 model is declined:

```
=> SELECT CHANGE_MODEL_STATUS('linear_reg_app', 2, 'declined');
CHANGE_MODEL_STATUS
-------------------------------------------------------------------------------
The status of model [linear_reg_app] - version [2] is changed to [declined]
(1 row)

=> SELECT CHANGE_MODEL_STATUS('linear_reg_app', 3, 'production');
CHANGE_MODEL_STATUS
-------------------------------------------------------------------------------
The status of model [linear_reg_app] - version [3] is changed to [production]
(1 row)
```

You can then query the REGISTERED_MODELS system table to confirm that the linear_reg_spark1 model is now in 'production', the native_linear_reg model has been set to 'declined', and the previously in production linear_reg_spark1 model has been automatically moved to 'archived':

```
=> SELECT * FROM REGISTERED_MODELS;
 registered_name | registered_version |    status    |       registered_time        |     model_id     | schema_name |    model_name     |      model_type       |
category
-----------------+--------------------+--------------+------------------------------+------------------+-------------+-------------------+-----------------------+----------------
 linear_reg_app  |                  3 | PRODUCTION   | 2023-01-26 09:52:00.082166-04 | 45035996273714020 | public      | linear_reg_spark1 |
PMML_REGRESSION_MODEL | PMML
 linear_reg_app  |                  2 | DECLINED     | 2023-01-26 09:51:04.553102-05 | 45035996273850350 | public      | native_linear_reg |
LINEAR_REGRESSION     | VERTICA_MODELS
 linear_reg_app  |                  1 | ARCHIVED     | 2023-01-24 05:29:25.990626-02 | 45035996273853740 | public      | linear_reg_newton |
LINEAR_REGRESSION     | VERTICA_MODELS
 logistic_reg_app |                  1 | PRODUCTION   | 2023-01-23 08:49:25.990626-02 | 45035996273853740 | public      | log_reg_cgd       |
LOGISTIC_REGRESSION   | VERTICA_MODELS
(4 rows)
```

To remove the declined native_linear_reg model from the versioning environment, you can set the status to 'unregistered':

```
=> SELECT CHANGE_MODEL_STATUS('linear_reg_app', 2, 'unregistered');
                        CHANGE_MODEL_STATUS
-------------------------------------------------------------------------------
The status of model [linear_reg_app] - version [2] is changed to [unregistered]
(1 row)

=> SELECT * FROM REGISTERED_MODELS;
  registered_name | registered_version |    status    |       registered_time       |      model_id     | schema_name |    model_name     |      model_type       |
category
------------------+--------------------+--------------+-----------------------------+-------------------+-------------+-------------------+-----------------------+----------------
 linear_reg_app   |                  3 | PRODUCTION   | 2023-01-26 09:52:00.082166-04 | 45035996273714020 | public      | linear_reg_spark1 |
PMML_REGRESSION_MODEL | PMML
 linear_reg_app   |                  1 | ARCHIVED     | 2023-01-24 05:29:25.990626-02 | 45035996273853740 | public      | linear_reg_newton |
LINEAR_REGRESSION     | VERTICA_MODELS
 logistic_reg_app |                  1 | PRODUCTION   | 2023-01-23 08:49:25.990626-02 | 45035996273853740 | public      | log_reg_cgd       |
LOGISTIC_REGRESSION   | VERTICA_MODELS
(3 rows)
```

You can see that the native_linear_reg model no longer appears in the REGISTERED_MODELS system table. However, you can still query the MODEL_STATUS_HISTORY system table to view the status history of native_linear_reg :

```
=> SELECT * FROM MODEL_STATUS_HISTORY WHERE model_id=45035996273850350;
 registered_name | registered_version |  new_status  |  old_status  |       status_change_time      |    operator_id    | operator_name |      model_id     |
schema_name |    model_name
-----------------+--------------------+--------------+------------- +-------------------------------+-------------------+---------------+-------------------+-------------+------------------
linear_reg_app   |                  2 | UNDER_REVIEW | UNREGISTERED | 2023-01-26 09:51:04.553102-05 | 45035996273964824 | user1         |
45035996273850350 | public      | native_linear_reg
linear_reg_app   |                  2 | STAGING      | UNDER_REVIEW | 2023-01-29 11:33:02.052464-05 | 45035996273704962 | supervisor1   |
45035996273850350 | public      | native_linear_reg
linear_reg_app   |                  2 | DECLINED     | STAGING      | 2023-01-30 04:12:30.481136-05 | 45035996273704962 | supervisor1   | 45035996273850350 |
public      | native_linear_reg
linear_reg_app   |                  2 | UNREGISTERED | DECLINED     | 2023-02-02 03:25:32.332132-05 | 45035996273704962 | supervisor1   |
45035996273850350 | public      | native_linear_reg
(4 rows)
```

See also

- REGISTERED_MODELS
- MODEL_STATUS_HISTORY
- REGISTER_MODEL
- CHANGE_MODEL_STATUS
- MLSUPERVISOR

Altering models

You can modify a model using ALTER MODEL , in response to your model's needs. You can alter a model by renaming the model, changing the owner, and changing the schema.

You can drop or alter any model that you create.

In this section

- Changing model ownership
- Moving models to another schema
- Renaming a model

Changing model ownership

As a superuser or model owner, you can reassign model ownership with ALTER MODEL as follows:

ALTER MODEL *model-name* OWNER TO *owner-name*

Changing model ownership is useful when a model owner leaves or changes responsibilities. Because you can change the owner, the models do not need to be rewritten.

Example

The following example shows how you can use ALTER_MODEL to change the model owner:

1. Find the model you want to alter. As the dbadmin, you own the model.

```
=> SELECT * FROM V_CATALOG.MODELS WHERE model_name='mykmeansmodel';
-[ RECORD 1 ]--+-----------------------------------------
model_id       | 45035996273816618
model_name     | mykmeansmodel
schema_id      | 45035996273704978
schema_name    | public
owner_id       | 45035996273704962
owner_name     | dbadmin
category       | VERTICA_MODELS
model_type     | kmeans
is_complete    | t
create_time    | 2017-03-02 11:16:04.990626-05
size           | 964
```

2. Change the model owner from dbadmin to user1.

```
=> ALTER MODEL mykmeansmodel OWNER TO user1;
   ALTER MODEL
```

3. Review V_CATALOG.MODELS to verify that the owner was changed.

```
=> SELECT * FROM V_CATALOG.MODELS WHERE model_name='mykmeansmodel';

-[ RECORD 1 ]--+-----------------------------------------
model_id       | 45035996273816618
model_name     | mykmeansmodel
schema_id      | 45035996273704978
schema_name    | public
owner_id       | 45035996273704962
owner_name     | user1
category       | VERTICA_MODELS
model_type     | kmeans
is_complete    | t
create_time    | 2017-03-02 11:16:04.990626-05
size           | 964
```

Moving models to another schema

You can move a model from one schema to another with ALTER MODEL . You can move the model as a superuser or user with USAGE privileges on the current schema and CREATE privileges on the destination schema.

Example

The following example shows how you can use ALTER MODEL to change the model schema:

1. Find the model you want to alter.

```
=> SELECT * FROM V_CATALOG.MODELS WHERE model_name='mykmeansmodel';
-[ RECORD 1 ]--+-----------------------------------------
model_id       | 45035996273816618
model_name     | mykmeansmodel
schema_id      | 45035996273704978
schema_name    | public
owner_id       | 45035996273704962
owner_name     | dbadmin
category       | VERTICA_MODELS
model_type     | kmeans
is_complete    | t
create_time    | 2017-03-02 11:16:04.990626-05
size           | 964
```

2. Change the model schema.

```
=> ALTER MODEL mykmeansmodel SET SCHEMA test;
   ALTER MODEL
```

3. Review V_CATALOG.MODELS to verify that the owner was changed.

```
=> SELECT * FROM V_CATALOG.MODELS WHERE model_name='mykmeansmodel';

-[ RECORD 1 ]--+-----------------------------------------
model_id       | 45035996273816618
model_name     | mykmeansmodel
schema_id      | 45035996273704978
schema_name    | test
owner_id       | 45035996273704962
owner_name     | dbadmin
category       | VERTICA_MODELS
model_type     | kmeans
is_complete    | t
create_time    | 2017-03-02 11:16:04.990626-05
size           | 964
```

Renaming a model

ALTER MODEL lets you rename models. For example:

1. Find the model you want to alter.

```
=> SELECT * FROM V_CATALOG.MODELS WHERE model_name='mymodel';
-[ RECORD 1 ]--+-----------------------------------------
model_id       | 45035996273816618
model_name     | mymodel
schema_id      | 45035996273704978
schema_name    | public
owner_id       | 45035996273704962
owner_name     | dbadmin
category       | VERTICA_MODELS
model_type     | kmeans
is_complete    | t
create_time    | 2017-03-02 11:16:04.990626-05
size           | 964
```

2. Rename the model.

```
=> ALTER MODEL mymodel RENAME TO mykmeansmodel;
   ALTER MODEL
```

3. Review V_CATALOG.MODELS to verify that the model name was changed.

```
=> SELECT * FROM V_CATALOG.MODELS WHERE model_name='mykmeansmodel';

-[ RECORD 1 ]--+-----------------------------------------
model_id       | 45035996273816618
model_name     | mykmeansmodel
schema_id      | 45035996273704978
schema_name    | public
owner_id       | 45035996273704962
owner_name     | dbadmin
category       | VERTICA_MODELS
model_type     | kmeans
is_complete    | t
create_time    | 2017-03-02 11:16:04.990626-05
size           | 964
```

Dropping models

DROP MODEL removes one or more models from the database. For example:

1. Find the model you want to drop.

```
=> SELECT * FROM V_CATALOG.MODELS WHERE model_name='mySvmClassModel';
-[ RECORD 1 ]--+---------------------------------
model_id       | 45035996273765414
model_name     | mySvmClassModel
schema_id      | 45035996273704978
schema_name    | public
owner_id       | 45035996273704962
owner_name     | dbadmin
category       | VERTICA_MODELS
model_type     | SVM_CLASSIFIER
is_complete    | t
create_time    | 2017-02-14 10:30:44.903946-05
size           | 525
```

2. Drop the model.

```
=> DROP MODEL mySvmClassModel;
   DROP MODEL
```

3. Review V_CATALOG.MODELS to verify that the model was dropped.

```
=> SELECT * FROM V_CATALOG.MODELS WHERE model_name='mySvmClassModel';
(0 rows)
```

Managing model security

You can manage the security privileges on your models by using the GRANT and REVOKE statements. The following examples show how you can change privileges on user1 and user2 using the faithful table and the linearReg model.

- In the following example, the dbadmin grants the SELECT privilege to user1:

```
=> GRANT SELECT ON TABLE faithful TO user1;
GRANT PRIVILEGE
```

- Then, the dbadmin grants the CREATE privilege on the public schema to user1:

```
=> GRANT CREATE ON SCHEMA public TO user1;
GRANT PRIVILEGE
```

- Connect to the database as user1:

```
=> \c - user1
```

- As user1, build the linearReg model:

```
=> SELECT LINEAR_REG('linearReg', 'faithful', 'waiting', 'eruptions');
LINEAR_REG
---------------------------
Finished in 1 iterations
(1 row)
```

- As user1, grant USAGE privileges to user2:

```
=> GRANT USAGE ON MODEL linearReg TO user2;
GRANT PRIVILEGE
```

- Connect to the database as user2:

```
=> \c - user2
```

- To confirm privileges were granted to user2, run the GET_MODEL_SUMMARY function. A user with the USAGE privilege on a model can run GET_MODEL_SUMMARY on that model:

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='linearReg');

=======
details
=======
predictor|coefficient|std_err |t_value |p_value
---------+-----------+--------+--------+--------
Intercept| 33.47440  | 1.15487|28.98533| 0.00000
eruptions| 10.72964  | 0.31475|34.08903| 0.00000

==============
regularization
==============
type| lambda
----+--------
none| 1.00000

============
call_string
============
linear_reg('public.linearReg', 'faithful', '"waiting"', 'eruptions'
USING PARAMETERS optimizer='newton', epsilon=1e-06, max_iterations=100, regularization='none', lambda=1)

===============
Additional Info
===============
Name              |Value
------------------+-----
iteration_count   |  1
rejected_row_count|  0
accepted_row_count| 272
(1 row)
```

- Connect to the database as user1:

```
=> \c - user1
```

- Then, you can use the REVOKE statement to revoke privileges from user2:

```
=> REVOKE USAGE ON MODEL linearReg FROM user2;
REVOKE PRIVILEGE
```

- To confirm the privileges were revoked, connect as user 2 and run the GET_MODEL_SUMMARY function:

```
=> \c - user2
=>SELECT GET_MODEL_SUMMARY('linearReg');
ERROR 7523:  Problem in get_model_summary.
Detail: Permission denied for model linearReg
```

See also

- GRANT (model)
- REVOKE (model)

Viewing model attributes

The following topics explain the model attributes for the Vertica machine learning algorithms. These attributes describe the internal structure of a particular model:

- Cross validation model attributes
- K-means model attributes
- Naive Bayes model attributes
- Normalization model attributes
- One Hot Encoder model attributes

- Random forest model attributes
- Regression model attributes
- SVM model attributes

Summarizing models

1. Find the model you want to summarize.

```
=> SELECT * FROM v_catalog.models WHERE model_name='svm_class';
model_id      | model_name |     schema_id   | schema_name |     owner_id      | owner_name |    category    |
model_type   | is_complete |     create_time          | size
------------------+------------+------------------+-------------+------------------+------------+--------
--------+----------------+-------------+-------------------------------+------
45035996273715226 | svm_class  | 45035996273704980 | public      | 45035996273704962 | dbadmin    | VERTICA_MODELS
| SVM_CLASSIFIER | t           | 2017-08-28 09:49:00.082166-04 | 1427
(1 row)
```

2. View the model summary.

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='svm_class');
-------------------------------------------------------------------------------


=======
details
=======
predictor|coefficient
---------+-----------
Intercept| -0.02006
cyl      |  0.15367
mpg      |  0.15698
wt       | -1.78157
hp       |  0.00957

===========
call_string
===========
SELECT svm_classifier('public.svm_class', 'mtcars_train', '"am"', 'cyl, mpg, wt, hp, gear'
USING PARAMETERS exclude_columns='gear', C=1, max_iterations=100, epsilon=0.001);

===============
Additional Info
===============
Name              |Value
------------------+-----
accepted_row_count| 20
rejected_row_count|  0
iteration_count   | 12
(1 row)
```

See also

- GET_MODEL_SUMMARY

Viewing models

Vertica stores the models you create in the V_CATALOG.MODELS system table.

You can query V_CATALOG.MODELS to view information about the models you have created:

```
=> SELECT * FROM V_CATALOG.MODELS;
-[ RECORD 1 ]--+-----------------------------------------
model_id       | 45035996273765414
model_name     | mySvmClassModel
schema_id      | 45035996273704978
schema_name    | public
owner_id       | 45035996273704962
owner_name     | dbadmin
category       | VERTICA_MODELS
model_type     | SVM_CLASSIFIER
is_complete    | t
create_time    | 2017-02-14 10:30:44.903946-05
size           | 525
-[ RECORD 2 ]--+-----------------------------------------

model_id       | 45035996273711466
model_name     | mtcars_normfit
schema_id      | 45035996273704978
schema_name    | public
owner_id       | 45035996273704962
owner_name     | dbadmin
category       | VERTICA_MODELS
model_type     | SVM_CLASSIFIER
is_complete    | t
create_time    | 2017-02-06 15:03:05.651941-05
size           | 288
```

See also

- MODELS
- V_CATALOG schema

Using external models with Vertica

To give you the utmost in machine learning flexibility and scalability, Vertica supports importing, exporting, and predicting with PMML and TensorFlow models.

The machine learning configuration parameter MaxModelSizeKB sets the maximum size of a model that can be imported into Vertica.

Support for PMML models

Vertica supports the import and export of K-means, linear regression, and logistic regression machine learning models in Predictive Model Markup Language (PMML) format. Support for this platform-independent model format allows you to use models trained on other platforms to predict on data stored in your Vertica database. You can also use Vertica as your model repository. Vertica supports PMML version 4.4 .

With the PREDICT_PMML function, you can use a PMML model archived in Vertica to run prediction on data stored in the Vertica database.

For more information, see Using PMML models .

For details on the PMML attributes that Vertica does and does not currently support, see PMML features and attributes .

Support for TensorFlow models

Vertica now supports importing trained TensorFlow models, and using those models to do prediction in Vertica on data stored in the Vertica database. Vertica supports TensorFlow models trained in TensorFlow version 1.15.

The PREDICT_TENSORFLOW function lets you predict on data in Vertica with any TensorFlow model.

For additional information, see TensorFlow models .

Additional functions that support external models

The following functions support both PMML and TensorFlow models:

IMPORT_MODELS EXPORT_MODELS GET_MODEL_ATTRIBUTE GET_MODEL_SUMMARY

In this section

- TensorFlow models
- Using PMML models

TensorFlow models

Tensorflow is a framework for creating neural networks. It implements basic linear algebra and multi-variable calculus operations in a scalable fashion, and allows users to easily chain these operations into a computation graph.

Vertica supports importing, exporting, and making predictions with TensorFlow 1.x and 2.x models trained outside of Vertica.

In-database TensorFlow integration with Vertica offers several advantages:

- Your models live inside your database, so you never have to move your data to make predictions.
- The volume of data you can handle is limited only by the size of your Vertica database, which makes Vertica particularly well-suited for machine learning on Big Data.
- Vertica offers in-database model management, so you can store as many models as you want.
- Imported models are portable and can be exported for use elsewhere.

When you run a TensorFlow model to predict on data in the database, Vertica calls a TensorFlow process to run the model. This allows Vertica to support any model you can create and train using TensorFlow. Vertica just provides the inputs - your data in the Vertica database - and stores the outputs.

In this section

- TensorFlow integration and directory structure
- TensorFlow example
- tf_model_desc.json overview

TensorFlow integration and directory structure

This page covers importing Tensorflow models into Vertica, making predictions on data in the Vertica database, and exporting the model for use on another Vertica cluster or a third-party platform.

For a start-to-finish example through each operation, see TensorFlow example .

Vertica supports models created with either TensorFlow 1.x and 2.x, but 2.x is strongly recommended.

To use TensorFlow with Vertica, install the TFIntegration UDX package on any node. You only need to do this once:

```
$ /opt/vertica/bin/admintools -t install_package -d database_name -p 'password' --package TFIntegration
```

Directory and file structure for TensorFlow models

Before importing your models, you should have a separate directory for each model that contains one of each of the following. Note that Vertica uses the directory name as the model name when you import it:

- *model_name* .pb : a trained model in frozen graph format
- tf_model_desc.json : a description of the model

You can generate both files for a given TensorFlow 2 model with the script freeze_tf2_model.py included in Machine-Learning-Examples/TensorFlow repository (or in opt/vertica/packages/TFIntegration/examples), though you may need to make modifications to the description based on your use-case and how you want your Vertica database to interact with your model:

```
$ python3 Tensorflow/freeze_tf2_model.py your/model/directory
```

For example, the tf_models directory contains two models, tf_mnist_estimator and tf_mnist_keras :

```
tf_models/
├─── tf_mnist_estimator
|    ├─── mnist_estimator.pb
|    └─── tf_model_desc.json
└─── tf_mnist_keras
    ├─── mnist_keras.pb
    └─── tf_model_desc.json
```

tf_model_desc.json

The tf_model_desc.json file forms the bridge between TensorFlow and Vertica. It describes the structure of the model so that Vertica can correctly match up its inputs and outputs to input/output tables.

Notice that the freeze_tf2_model.py script automatically generates this file for your model, and this generated file can often be used as-is. For more complex models or use cases, you might have to edit this file. For a detailed breakdown of each field, see tf_model_desc.json overview .

Importing TensorFlow models into Vertica

To import TensorFlow models, use IMPORT_MODELS with the category 'TENSORFLOW' .

Import a single model. Keep in mind that the Vertica database uses the directory name as the model name:

```
select IMPORT_MODELS ( '/path/tf_models/tf_mnist_keras' USING PARAMETERS category='TENSORFLOW');
 import_models
---------------
 Success
(1 row)
```

Import all models in the directory (where each model has its own directory) with a wildcard (*):

```
select IMPORT_MODELS ('/path/tf_models/*' USING PARAMETERS category='TENSORFLOW');
 import_models
---------------
 Success
(1 row)
```

Predicting with an imported TensorFlow model

After importing your TensorFlow model, you can use the model to predict on data in a Vertica table.

The PREDICT_TENSORFLOW function is different from the other predict functions in that it does not accept any parameters that affect the input columns such as "exclude_columns" or "id_column"; rather, the function always predicts on all the input columns provided. However, it does accept a num_passthru_cols parameter which allows the user to "skip" some number of input columns, as shown below.

The OVER(PARTITION BEST) clause tells Vertica to parallelize the operation across multiple nodes. See Window partition clause for details:

```
=> select PREDICT_TENSORFLOW (*
                USING PARAMETERS model_name='tf_mnist_keras', num_passthru_cols=1)
                OVER(PARTITION BEST) FROM tf_mnist_test_images;

--example output, the skipped columns are displayed as the first columns of the output
 ID | col0 | col1 | col2 | col3 | col4 | col5 | col6 | col7 | col8 | col9
----+------+------+------+------+------+------+------+------+------+------
  1 |    0 |    0 |    1 |    0 |    0 |    0 |    0 |    0 |    0 |    0
  3 |    1 |    0 |    0 |    0 |    0 |    0 |    0 |    0 |    0 |    0
  6 |    0 |    0 |    0 |    0 |    0 |    1 |    0 |    0 |    0 |    0
...
```

Exporting TensorFlow models

Vertica exports the model as a frozen graph, which can then be re-imported at any time. Keep in mind that models that are saved as a frozen graph cannot be trained further.

Use EXPORT_MODELS to export TensorFlow models. For example, to export the tf_mnist_keras model to the /path/to/export/to directory:

```
=> SELECT EXPORT_MODELS ('/path/to/export/to', 'tf_mnist_keras');
 export_models
---------------
 Success
(1 row)
```

When you export a TensorFlow model, the Vertica database creates and uses the specified directory to store files that describe the model:

```
$ ls tf_mnist_keras/
crc.json  metadata.json  mnist_keras.pb  model.json  tf_model_desc.json
```

The .pb and tf_model_desc.json files describe the model, and the rest are organizational files created by the Vertica database.

File Name	Purpose

model_name.pb	Frozen graph of the model.
tf_model_desc.json	Describes the model.
crc.json	Keeps track of files in this directory and their sizes. It is used for importing models.
metadata.json	Contains Vertica version, model type, and other information.
model.json	More verbose version of tf_model_desc.json .

See also

- TensorFlow models
- TensorFlow example
- IMPORT_MODELS
- EXPORT_MODELS
- PREDICT_TENSORFLOW

TensorFlow example

Vertica uses the TFIntegration UDX package to integrate with TensorFlow. You can train your models outside of your Vertica database, then import them to Vertica and make predictions on your data.

TensorFlow scripts and datasets are included in the GitHub repository under Machine-Learning-Examples/TensorFlow .

The example below creates a Keras (a TensorFlow API) neural network model trained on the MNIST handwritten digit classification dataset , the layers of which are shown below.

The data is fed through each layer from top to bottom, and each layer modifies the input before returning a score. In this example, the data passed in is a set of images of handwritten Arabic numerals and the output would be the probability of the input image being a particular digit:

```
inputs = keras.Input(shape=(28, 28, 1), name="image")
x = layers.Conv2D(32, 5, activation="relu")(inputs)
x = layers.MaxPooling2D(2)(x)
x = layers.Conv2D(64, 5, activation="relu")(x)
x = layers.MaxPooling2D(2)(x)
x = layers.Flatten()(x)
x = layers.Dense(10, activation='softmax', name='OUTPUT')(x)
tfmodel = keras.Model(inputs, x)
```

For more information on how TensorFlow interacts with your Vertica database and how to import more complex models, see TensorFlow integration and directory structure .

Prepare a TensorFlow model for Vertica

The following procedures take place outside of Vertica.

Train and save a TensorFlow model

1. Install TensorFlow 2 .
2. Train your model. For this particular example, you can run train_simple_model.py to train and save the model. Otherwise, and more generally, you can manually train your model in Python and then save it :
   ```
   $ mymodel.save('my_saved_model_dir')
   ```
3. Run the freeze_tf2_model.py script included in the Machine-Learning-Examples repository or in opt/vertica/packages/TFIntegration/examples , specifying your model and an output directory (optional, defaults to frozen_tfmodel).
   This script transforms your saved model into the Vertica-compatible frozen graph format and creates the tf_model_desc.json file, which describes how Vertica should translate its tables to TensorFlow tensors:
   ```
   $ ./freeze_tf2_model.py path/to/tf/model frozen_model_dir
   ```

Import TensorFlow models and make predictions in Vertica

1. If you haven't already, as dbadmin , install the TFIntegration UDX package on any node. You only need to do this once.
   ```
   $ /opt/vertica/bin/admintools -t install_package -d database_name -p 'password' --package TFIntegration
   ```
2. Copy the the directory to any node in your Vertica cluster and import the model:

```
=> SELECT IMPORT_MODELS('path/to/frozen_model_dir' USING PARAMETERS category='TENSORFLOW');
```

3. Import the dataset you want to make a prediction on. For this example:
   1. Copy the Machine-Learning-Examples/TensorFlow/data directory to any node on your Vertica cluster.
   2. From that data directory, run the SQL script load_tf_data.sql to load the MNIST dataset:

      ```
      $ vsql -f load_tf_data.sql
      ```

4. Make a prediction with your model on your dataset with PREDICT_TENSORFLOW . In this example, the model is used to classify the images of handwritten numbers in the MNIST dataset:

```
=> SELECT PREDICT_TENSORFLOW (*
                USING PARAMETERS model_name='tf_mnist_keras', num_passthru_cols=1)
                OVER(PARTITION BEST) FROM tf_mnist_test_images;

--example output, the skipped columns are displayed as the first columns of the output
 ID | col0 | col1 | col2 | col3 | col4 | col5 | col6 | col7 | col8 | col9
----+------+------+------+------+------+------+------+------+------+------
 1 |   0 |   0 |   1 |   0 |   0 |   0 |   0 |   0 |   0 |   0
 3 |   1 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
 6 |   0 |   0 |   0 |   0 |   1 |   0 |   0 |   0 |   0 |   0
...
```

5. To export the model with EXPORT_MODELS :

```
=> SELECT EXPORT_MODELS('/path/to/export/to', 'tf_mnist_keras');
 EXPORT_MODELS
---------------
 Success
(1 row)
```

> **Note**
> While you cannot continue training a TensorFlow model after you export it from Vertica, you can use it to predict on data in another Vertica cluster, or outside Vertica on another platform.

TensorFlow 1 (deprecated)

1. Install TensorFlow 1.15 with Python 3.7 or below.
2. Run train_save_model.py in Machine-Learning-Examples/TensorFlow/tf1 to train and save the model in TensorFlow and frozen graph formats.

See also

- TensorFlow models
- User-defined extensions

tf_model_desc.json overview

Before importing your externally trained TensorFlow models, you must:

- save the model in frozen graph (.pb) format
- create tf_model_desc.json , which describes to your Vertica database how to map its inputs and outputs to input/output tables

Conveniently, the script freeze_tf2_model.py included in the TensorFlow directory of the Machine-Learning-Examples repository (and in opt/vertica/packages/TFIntegration/examples) will do both of these automatically. In most cases, the generated tf_model_desc.json can be used as-is, but for more complex datasets and use cases, you might need to edit it.

tf_model_desc.json for an example dataset

The following tf_model_desc.json is generated from the MNIST handwriting dataset used by the TensorFlow example .

```
{
   "frozen_graph": "mnist_keras.pb",
   "input_desc": [
      {
         "op_name": "image_input",
         "tensor_map": [
            {
               "idx": 0,
               "dim": [
                  1,
                  28,
                  28,
                  1
               ],
               "col_start": 0
            }
         ]
      }
   ],
   "output_desc": [
      {
         "op_name": "OUTPUT/Softmax",
         "tensor_map": [
            {
               "idx": 0,
               "dim": [
                  1,
                  10
               ],
               "col_start": 0
            }
         ]
      }
   ]
}
```

This file describes the structure of the model's inputs and outputs. It must contain a frozen_graph field that matches the filename of the .pb model, an **input_desc** field, and an *output_desc* field.

- **input_desc** and **output_desc** : the descriptions of the input and output nodes in the TensorFlow graph. Each of these include the following fields:
  - **op_name** : the name of the operation node which is set when creating and training the model. You can typically retrieve the names of these parameters from *tfmodel* .inputs and *tfmodel* .outputs . For example:

    ```
    $ print({t.name:t for t in tfmodel.inputs})
    {'image_input:0': <tf.Tensor 'image_input:0' shape=(?, 28, 28, 1) dtype=float32>}
    ```

    ```
    $ print({t.name:t for t in tfmodel.outputs})
    {'OUTPUT/Softmax:0': <tf.Tensor 'OUTPUT/Softmax:0' shape=(?, 10) dtype=float32>}
    ```

    In this case, the respective values for **op_name** would be the following.
    - input_desc : image_input
    - output_desc : OUTPUT/Softmax

    For a more detailed example of this process, review the code for freeze_tf2_model.py .
  - *tensor_map* : how to map the tensor to Vertica columns, which can be specified with the following:
    - **idx** : the index of the tensor (should be 0 for the first input/output, 1 for the second input/output, etc.)
    - **dim** : the vector holding the dimensions of the tensor; it provides the number of columns
    - **col_start** (only used if **col_idx** is not specified): the starting column index. When used with **dim** , it specifies a range of indices of Vertica columns starting at **col_start** and ending at **col_start** + **dim** ; Vertica starts at the column specified by the index **col_start** and gets the next **dim** columns
    - **col_idx** : the indices in the Vertica columns corresponding to the flattened tensors. This allows you explicitly specify the indices of the Vertica columns that couldn't otherwise be specified as a simple range with **col_start** and **dim** (e.g. 1, 3, 5, 7)

- **data_type** (not shown): the data type of the input or output, one of the following:
  - TF_FLOAT (default)
  - TF_DOUBLE
  - TF_INT8
  - TF_INT16
  - TF_INT32
  - TF_INT64

Complex tf_model_desc.json example

Below is a more complex example that includes multiple inputs and outputs:

```
{
   "input_desc": [
      {
         "op_name": "input1",
         "tensor_map": [
            {
               "idx": 0,
               "dim": [
                  4
               ],
               "col_idx": [
                  0,
                  1,
                  2,
                  3
               ]
            },
            {
               "idx": 1,
               "dim": [
                  2,
                  2
               ],
               "col_start": 4
            }
         ]
      },
      {
         "op_name": "input2",
         "tensor_map": [
            {
               "idx": 0,
               "dim": [],
               "col_idx": [
                  8
               ]
            },
            {
               "idx": 1,
               "dim": [
                  2
               ],
               "col_start": 9
            }
         ]
      }
   ],
   "output_desc": [
      {
         "op_name": "output",
         "tensor_map": [
```

```
            {
               "idx": 0,
               "dim": [
                  2
               ],
               "col_start": 0
            }
         ]
      }
   ]
}
```

See also

- TensorFlow models
- TensorFlow example

Using PMML models

Vertica can import, export, and make predictions with PMML models of version 4.4 and below.

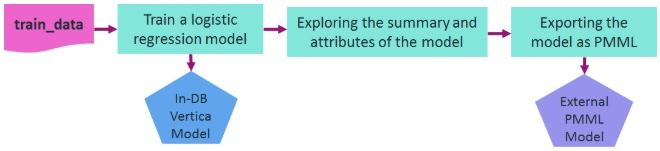
In this section

- Exporting Vertica models in PMML format
- Importing and predicting with PMML models
- PMML features and attributes

Exporting Vertica models in PMML format

You can take advantage of the built-in distributed algorithms in Vertica to train machine learning models. There might be cases in which you want to use these models for prediction outside Vertica, for example on an edge node. Now, you can export your model in PMML format and use it for prediction using a library or platform that supports reading and evaluating PMML models.

Here is an example for training a model in Vertica and then exporting it in PMML format. The following diagram shows the workflow of the example. We use vsql to run this example.



Let's assume that you want to train a logistic regression model on the data in a relation named 'patients' in order to predict the second attack of patients given their treatment and trait anxiety.

After training, the model is shown in a system table named V_CATALOG.MODELS which lists the archived ML models in Vertica.

```
=> -- Training a logistic regression model on a training_data
=> SELECT logistic_reg('myModel', 'patients', 'second_attack', 'treatment, trait_anxiety');
      logistic_reg
---------------------------
 Finished in 5 iterations

(1 row)

=> -- Looking at the models table
=> SELECT model_name, schema_name, category, model_type, create_time, size FROM models;
 model_name | schema_name |    category    |     model_type      |          create_time          | size
------------+-------------+----------------+---------------------+-------------------------------+------
 myModel    | public      | VERTICA_MODELS | LOGISTIC_REGRESSION | 2020-07-28 00:05:18.441958-04 | 1845
(1 row)
```

You can look at the summary of the model using the GET_MODEL_SUMMARY function.

```
=> -- Looking at the summary of the model
=> \t
Showing only tuples.
=> SELECT get_model_summary(USING PARAMETERS model_name='myModel');


=======
details
=======
  predictor  |coefficient|std_err |z_value |p_value
-------------+-----------+--------+--------+--------
  Intercept  | -6.36347  | 3.21390|-1.97998| 0.04771
  treatment  | -1.02411  | 1.17108|-0.87450| 0.38185
trait_anxiety|  0.11904  | 0.05498| 2.16527| 0.03037


==============
regularization
==============
type| lambda
----+--------
none| 1.00000


===========
call_string
===========
logistic_reg('public.myModel', 'patients', '"second_attack"', 'treatment, trait_anxiety'
USING PARAMETERS optimizer='newton', epsilon=1e-06, max_iterations=100, regularization='none', lambda=1, alpha=0.5)

===============
Additional Info
===============
       Name       |Value
------------------+-----
 iteration_count  |  5
rejected_row_count|  0
accepted_row_count| 20
```

You can also retrieve the model's attributes using the GET_MODEL_ATTRIBUTE function.

```
=> \t
Tuples only is off.
=> -- The list of the attributes of the model
=> SELECT get_model_attribute(USING PARAMETERS model_name='myModel');
     attr_name      |                    attr_fields                    | #_of_rows
--------------------+---------------------------------------------------+-----------
 details            | predictor, coefficient, std_err, z_value, p_value |         3
 regularization     | type, lambda                                      |         1
 iteration_count    | iteration_count                                   |         1
 rejected_row_count | rejected_row_count                                |         1
 accepted_row_count | accepted_row_count                                |         1
 call_string        | call_string                                       |         1
(6 rows)

=> -- Returning the coefficients of the model in a tabular format
=> SELECT get_model_attribute(USING PARAMETERS model_name='myModel', attr_name='details');
   predictor   |    coefficient    |      std_err       |      z_value       |      p_value
---------------+-------------------+--------------------+--------------------+--------------------
 Intercept     | -6.36346994178182 |   3.21390452471434 |  -1.97998101463435 | 0.0477056620380991
 treatment     | -1.02410605239327 |    1.1710801464903 | -0.874496980810833 |  0.381847663704613
 trait_anxiety | 0.119044916668605 | 0.0549791755747139 |   2.16527285875412 | 0.0303667955962211
(3 rows)
```

You can use the EXPORT_MODELS function in a simple statement to export the model in PMML format, as shown below.

```
=> -- Exporting the model as PMML
=> SELECT export_models('/data/username/temp', 'myModel' USING PARAMETERS category='PMML');
 export_models
---------------
 Success
(1 row)
```

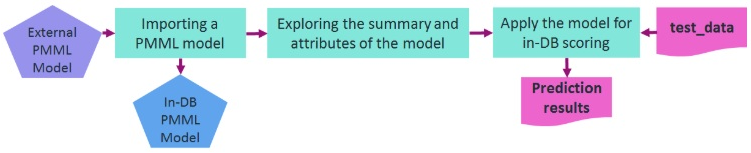
See also

- MODELS
- V_CATALOG schema

Importing and predicting with PMML models

As a Vertica user, you can train ML models in other platforms, convert them to standard PMML format, and then import them into Vertica for in-database prediction on data stored in Vertica relations.

Here is an example of how to import a PMML model trained in Spark. The following diagram shows the workflow of the example.



You can use the IMPORT_MODELS function in a simple statement to import the PMML model. The imported model then appears in a system table named V_CATALOG.MODELS which lists the archived ML models in Vertica.

```
=> -- importing the PMML model trained and generated in Spark
=> SELECT import_models('/data/username/temp/spark_logistic_reg' USING PARAMETERS category='PMML');
 import_models
---------------
 Success
(1 row)

=> -- Looking at the models table=> SELECT model_name, schema_name, category, model_type, create_time, size FROM models;
    model_name     | schema_name | category |      model_type       |          create_time          | size
-------------------+-------------+----------+-----------------------+-------------------------------+------
 spark_logistic_reg | public      | PMML     | PMML_REGRESSION_MODEL | 2020-07-28 00:12:29.389709-04 | 5831
(1 row)
```

You can look at the summary of the model using GET_MODEL_SUMMARY function.

```
=> \t
Showing only tuples.
=> SELECT get_model_summary(USING PARAMETERS model_name='spark_logistic_reg');

==============
function_name
==============
classification

============
data_fields
============
 name  |dataType|  optype
-------+--------+-----------
field_0| double |continuous
field_1| double |continuous
field_2| double |continuous
field_3| double |continuous
field_4| double |continuous
field_5| double |continuous
field_6| double |continuous
field_7| double |continuous
field_8| double |continuous
target | string |categorical


===========
predictors
===========
 name  |exponent|coefficient
-------+--------+-----------
field_0|   1    | -0.23318
field_1|   1    |  0.73623
field_2|   1    |  0.29964
field_3|   1    |  0.12809
field_4|   1    | -0.66857
field_5|   1    |  0.51675
field_6|   1    | -0.41026
field_7|   1    |  0.30829
field_8|   1    | -0.17788


================
Additional Info
================
    Name     | Value
-------------+--------
is_supervised|   1
  intercept  |-1.20173
```

You can also retrieve the model's attributes using the GET_MODEL_ATTRIBUTE function.

```
=> \t
Tuples only is off.
=> -- The list of the attributes of the PMML model
=> SELECT get_model_attribute(USING PARAMETERS model_name='spark_logistic_reg');
   attr_name   |         attr_fields          | #_of_rows
---------------+------------------------------+-----------
 is_supervised | is_supervised                |         1
 function_name | function_name                |         1
 data_fields   | name, dataType, optype       |        10
 intercept     | intercept                    |         1
 predictors    | name, exponent, coefficient |         9
(5 rows)

=> -- The coefficients of the PMML model
=> SELECT get_model_attribute(USING PARAMETERS model_name='spark_logistic_reg', attr_name='predictors');
  name   | exponent |    coefficient
---------+----------+--------------------
 field_0 |        1 |   -0.2331769167607
 field_1 |        1 |  0.736227459496199
 field_2 |        1 |   0.29963728232024
 field_3 |        1 |  0.128085369856188
 field_4 |        1 | -0.668573096260048
 field_5 |        1 |  0.516750679584637
 field_6 |        1 |  -0.41025989394959
 field_7 |        1 |  0.308289533913736
 field_8 |        1 | -0.177878773139411
(9 rows)
```

You can then use the PREDICT_PMML function to apply the imported model on a relation for in-database prediction. The internal parameters of the model can be matched to the column names of the input relation by their names or their listed position. Direct input values can also be fed to the function as displayed below.

```
=> -- Using the imported PMML model for scoring direct input values
=> SELECT predict_pmml(1.5,0.5,2,1,0.75,4.2,3.1,0.9,1.1
username(>                    USING PARAMETERS model_name='spark_logistic_reg', match_by_pos=true);
 predict_pmml
--------------
 1
(1 row)

=> -- Using the imported PMML model for scoring samples in a table
=> SELECT predict_pmml(* USING PARAMETERS model_name='spark_logistic_reg') AS prediction
=>     FROM test_data;
 prediction
------------
 1
 0
(2 rows)
```

See also

- MODELS
- V_CATALOG schema

PMML features and attributes

Using external models with Vertica gives an overview of the features Vertica supports for working with external models. This topic provides additional details on limitations in how Vertica supports working with PMML models.

In order to be supported by Vertica, PMML models:

- Must not contain a data preprocessing step.

- Must encode only these model types: k-means, linear regression, logistic regression, random forest (classifier and regressor), XGBoost (classifier and regressor), GeneralRegressionModel, and TreeModel.

Supported PMML tags and attributes

The following table lists supported PMML tags and their attributes.

XML-tag name	Ignored attributes	Supported attributes	Unsupported attributes	Ignored sub-ta
Categories	-	•	-	-
Category	-	value (required)	-	-
CategoricalPredictor	-	• name (required) • value (required) • coefficient (required)	-	-
Cluster	size	• id • name	-	• KohonenMa • Covariances
ClusteringField	-	• field (required) • isCenterField (only "true" is supported) • compareFunction	• fieldWeight • similarityScale	-
ClusteringModel	modelName	• functionName (required, only "clustering" is supported) • algorithmName • modelClass (required, only "centerBased" is supported) • numberOfClusters(required) • isScorable (only "true" is supported)	-	ModelVerificatic
ComparisonMeasure	• minimum • maximum	• kind (required, only "distance" is supported) • compareFunction	-	-
CompoundPredicate	-	booleanOperator (required)	-	-
CovariateList	-	-	-	-
DataDictionary	-	numberOfFields	-	-

DataField	displayName	• name (required) • optype (required) • dataType (required)	• taxonomy • isCyclic	-
FactorList	-	-	-	-
False	-	-	-	-
GeneralRegressionModel	• modelName • targetVariableName • startTimeVariable • subjectIDVariable	• modelType (required) • functionName (required) • algorithmName • targetReferenceCategory • cumulativeLink • linkFunction • linkParameter • trialsVariable • trialsValue • distribution • distParameter • offsetVariable • offsetValue • modelDF • isScoreable (only "true" is supported)	• endTimeVariable • statusVariable • baselineStrataVariable	• ModelVerifi • PCovMatrix
Header	• copyright • description • modelVersion	-	-	• Extension • Application • Annotation • Timestamp
MiningField	• importance • missingValueTreatment	• name(required) • usageType • optype	• outliers • lowValue • highValue • missingValueReplacement • invalidValueTreatment	-
MiningModel	• modelName • algorithmName	• functionName (required) • isScoreable (only "true" is supported)	-	ModelVerificatic
MiningSchema	-	-	-	-
Node	-	• id • score • recordCount • defaultChild	-	-

NumericPredictor	-	• name (required) • exponent • coefficient (required)	-	-
Output	-	-	-	-
OutputField	• displayName • opType	• name (required) • dataType (required) • feature • value • isFinalResult	• targetField • ruleFeature • algorithm • rankBasis • segmentId • rank • rankOrder • isMultiValued	-
Parameter	-	• name (required) • label	referencePoint	-
ParameterList	-	-	-	-
ParamMatrix	-	-	-	-
PCell	-	• parameterName (required) • targetCategory • beta (required) • df	-	-
PPCell	-	• parameterName (required) • predictorName (required) • parameterName (required) • targetCategory	-	-
PPMatrix	-	-	-	-
PMML	-	• version (required) • xmlns	-	MiningBuildTas
Predictor	-	• name (required) • contrastMatrixType	-	-
RegressionModel	• modelName • targetFieldName • modelType	• functionName (required) • algorithmName • normalizationMethod • isScorable (only "true" is supported)	-	ModelVerificatic
RegressionTable	-	• intercept (required) • targetCategory	-	-

Segment	-	• id • weight	•	-
Segmentation	• missingPredictionTreatment • missingThreshold	multipleModelMethod (required)	-	-
SimplePredicate	-	• field (required) • operator (required) • value	-	-
SimpleSetPredicate	-	• field (required) • booleanOperator (required)	-	-
Target	-	• field • optype • rescaleConstant • rescaleFactor	• castInteger • min • max	-
Targets	-	-	-	-
TreeModel	• modelName • missingValueStrategy • missingValuePenalty	• functionName (required) • algorithmName • noTrueChildStrategy • splitCharacteristic • isScorable (only "true" is supported)	-	ModelVerificatic
True	-	-	-	-
ScoreDistribution	-	• value (required) • recordCount (required)	-	-
Value	displayValue	• value (required) • property	-	-

Geospatial analytics

Vertica provides functions that allows you to manipulate complex two- and three-dimensional spatial objects. These functions follow the Open Geospatial Consortium (OGC) standards. Vertica also provides data types and SQL functions that allow you to specify and store spatial objects in a database according to OGC standards.

Convert well-known text (WKT) and well-known binary (WKB)

Convert WKT and WKB.

Optimized spatial joins

Perform fast spatial joins using ST_Intersects and STV_Intersects.

Load and export spatial data from shapefiles

Easily load and export shapefiles.

Store and retrieve objects

Determine if:

- An object contains self-intersection or self-tangency points.
- One object is entirely within another object, such as a point within a polygon.

Test the relationships between objects

For example, if they intersect or touch:

- Identify the boundary of an object.
- Identify vertices of an object.

Calculate

- Shortest distance between two objects.
- Size of an object (length, area).
- Centroid for one or more objects.
- Buffer around one or more objects.

In this section

- Best practices for geospatial analytics
- Spatial objects
- Working with spatial objects in tables
- Working with spatial objects from client applications
- OGC spatial definitions
- Spatial data type support limitations

Best practices for geospatial analytics

Vertica recommends the following best practices when performing geospatial analytics in Vertica.

Performance optimization

Recommendation	Details
Use the minimum column size for spatial data.	Performance degrades as column widths increase. When creating columns for your spatial data, use the smallest size column that can accommodate your data. For example, use GEOMETRY(85) for point data.
Use GEOMETRY types where possible.	Performance of functions on GEOGRAPHY types is slower than functions that support GEOMETRY types. Use GEOMETRY types where possible.
To improve the performance of the following functions, sort projections on spatial columns: • STV_Intersect scalar function • ST_Distance • ST_Area • ST_Length	You may improve the pruning efficiency of these functions by sorting the projection on the GEOMETRY column. However, sorting on a large GEOMETRY column may slow down data load.

Spatial joins with points and polygons

Vertica provides two ways to identify whether a set of points intersect with a set of polygons. Depending on the size of your data set, choose the approach that gives the best performance.

For a detailed example of best practices with spatial joins, see Best practices for spatial joins .

Recommendation	Details
Use ST_Intersects to intersect a constant geometry with a set of geometries.	When you intersect a set of geometries with a geometry, use the ST_Intersects function. Express the constant geometry argument as a WKT or WKB. For example: `ST_Intersects(geom,ST_GeomFromText( 'POLYGON((43.1 50.1,43.1 59.0, 48.9 59.0,43.1 50.1))'))` For more information, see Performing spatial joins with ST_Intersects .
Create a spatial index only when performing spatial joins with STV_Intersect.	Spatial indexes should only be used with STV_Intersect. Creating a spatial index and then performing spatial joins with ST_Intersects will not improve performance.

Use the STV_Intersect function when you intersect a set of points with a set of polygons.	Determine if a set of points intersects with a set of polygons in a medium to large data set. First, create a spatial index using STV_Create_Index. Then, use one of the STV_Intersect functions to return the set of pairs that intersect. Spatial indexes provide the best performance for accessing a large number of polygons.
When using the STV_Intersect transform function, partition the data and use an OVER(PARTITION BEST) clause.	The STV_Intersect transform function does not require that you partition the data. However, you may improve performance by partitioning the data and using an OVER(PARTITION BEST) clause.

Spatial indexes

The STV_Create_Index function can consume large amounts of processing time and memory. When you index new data for the first time, monitor memory usage to be sure it stays within safe limits. Memory usage depends on:

- Number of polygons
- Number of vertices
- Amount of overlap among polygons

Recommendation	**Details**
Segment polygon data when a table contains a large number of polygons.	Segmenting the data allows the index creation process to run in parallel. This is advantageous because sometimes STV_Create_Index tasks cannot be completed when large tables that are not segmented prior to index creation.
Adjust STV_Create_Index parameters as needed for memory allocation and CPU usage.	The max_mem_mb parameter can affect the resource usage of STV_Create_Index. max_mem_mb assigns a limit to the amount of memory that STV_Create_Index can allocate. **Default:** 256 **Valid values:** Any value less than or equal to the amount of memory in the GENERAL resource pool. Assigning a higher value results in an error.
Make changes if STV_Create_Index cannot allocate 300 MB memory.	Before STV_Create_Index starts creating the index, it tries to allocate about 300 MB of memory. If that much memory is not available, the function fails. If you get a failure message, try these solutions: • Create the index at a time of less load on the system. • Avoid concurrent index creation. • Add more memory to your system.
Create misplaced indexes again, if needed.	When you back up your Vertica database, spatial index files are not included. If you misplace an index, use STV_Create_Index to re-create it.
Use STV_Refresh_Index to add new or updated polygons to an existing index.	Instead of rebuilding your spatial index each time you add new or updated polygons to a table, you can use STV_Refresh_Index to append the polygons to your existing spatial index.

Checking polygon validity

Recommendation	**Details**
Run ST_IsValid to check if polygons are valid.	Many spatial functions do not check the validity of polygons. • Run ST_IsValid on all polygons to determine if they are valid. • If your object is not valid, run STV_IsValidReason to get information about the location of the invalid polygon. For more information, see Ensuring polygon validity before creating or refreshing an index.

Spatial objects

Vertica implements several data types for storing spatial objects, Well-Known Text (WKT) strings, and Well-Known Binary (WKB) representations. These data types include:

- Supported spatial objects
- Spatial reference identifiers (SRIDs)

In this section

- Supported spatial objects
- Spatial reference identifiers (SRIDs)

Supported spatial objects

Vertica supports two spatial data types. These data types store two- and three-dimensional spatial objects in a table column:

- GEOMETRY : Spatial object with coordinates expressed as (*x* , *y*) pairs, defined in the Cartesian plane. All calculations use Cartesian coordinates.
- GEOGRAPHY : Spatial object defined as on the surface of a perfect sphere, or a spatial object in the WGS84 coordinate system. Coordinates are expressed in longitude/latitude angular values, measured in degrees. All calculations are in meters. For perfect sphere calculations, the sphere has a radius of 6371 kilometers, which approximates the shape of the earth.

> **Note**
> Some spatial programs use an ellipsoid to model the earth, resulting in slightly different data.

The maximum size of a GEOMETRY or GEOGRAPHY data type is 10,000,000 bytes (10 MB). You cannot use either data type as a table's primary key.

Spatial reference identifiers (SRIDs)

A *spatial reference identifier* (SRID) is an integer value that represents a method for projecting coordinates on the plane. A SRID is metadata that indicates the coordinate system in which a spatial object is defined.

Geospatial functions using Geometry arguments must contain the same SRID. If the functions do not contain the same SRID, then the query returns an error.

For example, in this query the two points have different SRIDs. As a result the query returns an error:

```
=> SELECT ST_Distance(ST_GeomFromText('POINT(34 9)',2749), ST_GeomFromText('POINT(70 12)', 3359));
ERROR 5861:  Error calling processBlock() in User Function ST_Distance at [src/Distance.cpp:65],
error code: 0, message: Geometries with different SRIDs found: 2749, 3359
```

Supported SRIDs

Vertica supports SRIDs derived from the EPSG standards. Geospatial functions using Geometry arguments must use supported SRIDs when performing calculations. SRID values of 0 to 2^{32-1} are valid. Queries with SRID values outside of this range will return an error.

Working with spatial objects in tables

- Defining table columns for spatial data
- Exporting spatial data from a table
- Identifying null spatial objects
- Loading spatial data from shapefiles
- Loading spatial data into tables using COPY
- Retrieving spatial data from a table as well-known text (WKT)

In this section

- Defining table columns for spatial data
- Exporting spatial data from a table
- Identifying null spatial objects
- Loading spatial data from shapefiles
- Loading spatial data into tables using COPY
- Retrieving spatial data from a table as well-known text (WKT)

- Working with GeoHash data
- Spatial joins with ST_Intersects and STV_Intersect

Defining table columns for spatial data

To define columns to contain GEOMETRY and GEOGRAPHY data, use this command:

```
=> CREATE TABLE [[db-name.]schema.]table-name (
  column-name GEOMETRY[(length)],
  column-name GEOGRAPHY[(length)]);
```

If you omit the length specification, the default column size is 1 MB. The maximum column size is 10 MB. The upper limit is not enforced, but the geospatial functions can only accept or return spatial data up to 10 MB.

You cannot modify the size or data type of a GEOMETRY or GEOGRAPHY column after creation. If the column size you created is not sufficient, create a new column with the desired size. Then copy the data from the old column, and drop the old column from the table.

You cannot import data to or export data from tables that contain spatial data from another Vertica database.

Important
A column width that is too large could impact performance. Use a column width that fits the data without being excessively large. See STV_MemSize .

Exporting spatial data from a table

You can export spatial data from a table in your Vertica database to a shapefile.

To export spatial data from a table to a shapefile:

1. As the superuser ., set the shapefile export directory.
   ```
   => SELECT STV_SetExportShapefileDirectory(USING PARAMETERS path = '/home/geo/temp');
                STV_SetExportShapefileDirectory
   ------------------------------------------------------------------
    SUCCESS. Set shapefile export directory: [/home/geo/temp]
   (1 row)
   ```
2. Export your spatial data to a shapefile.
   ```
   => SELECT STV_Export2Shapefile(*
              USING PARAMETERS shapefile = 'visualizations/city-data.shp',
                           shape = 'Polygon') OVER() FROM spatial_data;
    Rows Exported |                          File Path
   ---------------+-------------------------------------------------------------
          185873 | v_geo-db_node0001: /home/geo/temp/visualizations/city-data.shp
   (1 row)
   ```
   - The value asterisk (*) is the equivalent to listing all columns in the FROM clause.
   - You can specify sub-directories when exporting your shapefile.
   - Your shapefile must end with the file extension .shp.
3. Verify that three files now appear in the shapefile export directory.
   ```
   $ ls
   city-data.dbf  city-data.shp   city-data.shx
   ```

Identifying null spatial objects

You can identify null GEOMETRY and GEOGRAPHY objects using the Vertica IS NULL and IS NOT NULL constructs.

This example uses the following table, where the row with id =2 has a null value in the geog field.

```
=> SELECT id, ST_AsText(geom), ST_AsText(geog) FROM locations
   ORDER BY 1 ASC;
 id |          ST_AsText             |                ST_AsText
----+--------------------------------+-----------------------------------
  1 | POINT (2 3)                    | POINT (-85 15)
  2 | POINT (4 5)                    |
  3 | POLYGON ((-1 2, 0 3, 1 2, -1 2)) | POLYGON ((-24 12, -15 23, -20 27, -24 12))
  4 | LINESTRING (-1 2, 1 5)         | LINESTRING (-42.74 23.98, -62.19 23.78)
(4 rows)
```

Identify all the rows that have a null geog value:

```
=> SELECT id, ST_AsText(geom), (ST_AsText(geog) IS NULL) FROM locations
   ORDER BY 1 ASC;
 id |          ST_AsText            | ?column?
----+-----------------------------------+----------
  1 | POINT (2 3)                   | f
  2 | POINT (4 5)                   | t
  3 | POLYGON ((-1 2, 0 3, 1 2, -1 2)) | f
  4 | LINESTRING (-1 2, 1 5)        | f
(4 rows)
```

Identify the rows where the geog value is not null:

```
=> SELECT id, ST_AsText(geom), (ST_AsText(geog) IS NOT NULL) FROM locations
   ORDER BY 1 ASC;
 id |          st_astext            | ?column?
----+-----------------------------------+----------
  1 | POINT (2 3)                   | t
  2 | POINT (4 5)                   | f
  3 | LINESTRING (-1 2, 1 5)        | t
  4 | POLYGON ((-1 2, 0 3, 1 2, -1 2)) | t
(4 rows)
```

Loading spatial data from shapefiles

Vertica provides the capability to load and parse spatial data that is stored in shapefiles. Shapefiles describe points, lines, and polygons. A shapefile is made up of three required files; all three files must be present and in the same directory to define the geometries:

- .shp—Contains the geometry data.
- .shx—Contains the positional index of the geometry.
- .dbf—Contains the attributes for each geometry.

To load spatial data from a shapefile:

1. Use STV_ShpCreateTable to generate a CREATE TABLE statement.

   ```
   => SELECT STV_ShpCreateTable ( USING PARAMETERS file = '/home/geo/temp/shp-files/spatial_data.shp')
                               OVER() AS spatial_data;
           spatial_data
   -----------------------------------
    CREATE TABLE spatial_data(
      gid IDENTITY(64) PRIMARY KEY,
      uniq_id INT8,
      geom GEOMETRY(85)
   );
   (5 rows)
   ```

2. Create the table.

   ```
   => CREATE TABLE spatial_data(
      gid IDENTITY(64) PRIMARY KEY,
      uniq_id INT8,
      geom GEOMETRY(85));
   ```

3. Load the shapefile.

   ```
   => COPY spatial_data WITH SOURCE STV_ShpSource(file='/home/geo/temp/shp-files/spatial_data.shp')
       PARSER STV_ShpParser();
    Rows Loaded
   -------------
             10
   (1 row)
   ```

Supported shapefile shape types

The following table lists the shapefile shape types that Vertica supports.

Shapefile Shape Type	Supported
Null shape	Yes
Point	Yes
Polyline	Yes
Polygon	Yes
MultiPoint	Yes
PointZ	No
PolylineZ	No
PolygonZ	No
MultiPointZ	No
PointM	No
PolylineM	No
PolygonM	No
MultiPointM	No
MultiPatch	No

Loading spatial data into tables using COPY

You can load spatial data into a table in Vertica using a COPY statement.

To load data into Vertica using a COPY statement:

1. Create a table.

```
=> CREATE TABLE spatial_data (id INTEGER, geom GEOMETRY(200));
CREATE TABLE
```

2. Create a text file named spatial.dat with the following data.

```
1|POINT(2 3)
2|LINESTRING(-1 2, 1 5)
3|POLYGON((-1 2, 0 3, 1 2, -1 2))
```

3. Use COPY to load the data into the table.

```
=> COPY spatial_data (id, gx FILLER LONG VARCHAR(605), geom AS ST_GeomFromText(gx)) FROM LOCAL 'spatial.dat';
 Rows Loaded
-------------
           3
(1 row)
```

   The statement specifies a LONG VARCHAR(32000000) filler, which is the maximum size of WKT. You must specify a filler value large enough to hold the largest WKT you want to insert into the table.

Retrieving spatial data from a table as well-known text (WKT)

GEOMETRY and GEOGRAPHY data is stored in Vertica tables as LONG VARBINARY, which isn't human readable. You can use ST_AsText to return the spatial data as Well-Known Text (WKT).

To return spatial data as WKT:

```
=> SELECT id, ST_AsText(geom) AS WKT FROM spatial_data;
 id |              WKT
----+-----------------------------------
  1 | POINT (2 3)
  2 | LINESTRING (-1 2, 1 5)
  3 | POLYGON ((-1 2, 0 3, 1 2, -1 2))
(3 rows)
```

Working with GeoHash data

Vertica supports GeoHashes . A GeoHash is a geocoding system for hierarchically encoding increasingly granular spatial references. Each additional character in a GeoHash drills down to a smaller section of a map.

You can use Vertica to generate spatial data from GeoHashes and GeoHashes from spatial data. Vertica supports the following functions for use with GeoHashes:

- ST_GeoHash - Returns a GeoHash in the shape of the specified geometry.
- ST_GeomFromGeoHash - Returns a polygon in the shape of the specified GeoHash.
- ST_PointFromGeoHash - Returns the center point of the specified GeoHash.

For example, to generate a full precision and partial precision GeoHash from a single point.

```
=> SELECT ST_GeoHash(ST_GeographyFromText('POINT(3.14 -1.34)')), LENGTH(ST_GeoHash(ST_GeographyFromText('POINT(3.14 -1.34)'))),
                ST_GeoHash(ST_GeographyFromText('POINT(3.14 -1.34)') USING PARAMETERS numchars=5) partial_hash;
    ST_GeoHash      | LENGTH | partial_hash
---------------------+--------+--------------
 kpf0rkn3zmcswks75010 |     20 | kpf0r
(1 row)
```

This example shows how to generate a GeoHash from a multipoint point object. The returned polygon is a geometry object of the smallest tile that encloses that GeoHash.

```
=> SELECT ST_AsText(ST_GeomFromGeoHash(ST_GeoHash(ST_GeomFromText('MULTIPOINT(0 0, 0.0002 0.0001)')))) AS region_1,
                ST_AsText(ST_GeomFromGeoHash(ST_GeoHash(ST_GeomFromText('MULTIPOINT(0.0001 0.0001, 0.0003 0.0002)')))) AS region_2;
-[ RECORD 1 ]-----------------------------------------------------------------------------------------------
  region_1 | POLYGON ((0 0, 0.00137329101562 0, 0.00137329101562 0.00137329101562, 0 0.00137329101562, 0 0))
  region_2 | POLYGON ((0 0, 0.010986328125 0, 0.010986328125 0.0054931640625, 0 0.0054931640625, 0 0))
```

Spatial joins with ST_Intersects and STV_Intersect

Spatial joins allow you to identify spatial relationships between two sets of spatial data. For example, you can use spatial joins to:

- Calculate the density of mobile calls in various regions to determine the location of a new cell phone tower.
- Identify homes that fall within the impact zone of a hurricane.
- Calculate the number of users who live within a certain ZIP code.
- Calculate the number of customers in a retail store at any given time.

In this section

- Best practices for spatial joins
- Ensuring polygon validity before creating or refreshing an index
- STV_Intersect: scalar function vs. transform function
- Performing spatial joins with STV_Intersect functions
- When to use ST_Intersects vs. STV_Intersect

Best practices for spatial joins

Use these best practices to improve overall performance and optimize your spatial queries.

Best practices for using spatial joins in Vertica include:

- Table segmentation to speed up index creation
- Adequately sizing a geometry column to store point data
- Loading Well-Known Text (WKT) directly into a Geometry column, using STV_GeometryPoint in a COPY statement
- Using OVER (PARTITION BEST) with STV_Intersect transform queries

Best practices example

> **Note**
> The following example was originally published in a Vertica blog post about using spatial data in museums. To read the entire blog, see Using Location Data with Vertica Place

Before performing the steps in the following example, download place_output.csv.zip from the Vertica Place GitHub repository (https://github.com/vertica/Vertica-Geospatial). You need to use the data set from this repository.

1. Create the table for the polygons. Use a GEOMETRY column width that fits your data without being excessively large. A good column-width fit improves performance. In addition, segmenting the table by HASH provides the advantages of parallel computation.

```
=> CREATE TABLE artworks (gid int, g GEOMETRY(700)) SEGMENTED BY HASH(gid) ALL NODES;
```

2. Use a copy statement with ST_Buffer to create and load the polygons on which to run the intersect. By using ST_Buffer in your copy statement, you can use that function to create the polygons.

```
=> COPY artworks(gid, gx FILLER LONG VARCHAR, g AS ST_Buffer(ST_GeomFromText(gx),8)) FROM STDIN DELIMITER ',';
>> 1, POINT(10 45)
>> 2, POINT(25 45)
>> 3, POINT(35 45)
>> 4, POINT(35 15)
>> 5, POINT(30 5)
>> 6, POINT(15 5)
>> \.
```

3. Create a table for the location data, represented by points. You can store point data in a GEOMETRY column of 100 bytes. Avoid over-fitting your GEOMETRY column. Doing so can significantly degrade spatial intersection performance. Also, segment this table by HASH, to take advantage of parallel computation.

```
=> CREATE TABLE usr_data (gid identity, usr_id int, date_time timestamp, g GEOMETRY(100))
   SEGMENTED BY HASH(gid) ALL NODES;
```

4. During the copy statement, transform the raw location data to GEOMETRY data. You must perform this transformation because your location data needs to use the GEOMETRY data type. Use the function STV_GeometryPoint to transform the *x* and *y* columns of the source table.

```
=> COPY usr_data (usr_id, date_time, x FILLER LONG VARCHAR,
                  y FILLER LONG VARCHAR, g AS STV_GeometryPoint(x, y))
  FROM LOCAL 'place_output.csv' DELIMITER ',' ENCLOSED BY '';
```

5. Create the spatial index for the polygons. This index helps you speed up intersection calculations.

```
=> SELECT STV_Create_Index(gid, g USING PARAMETERS index='art_index', overwrite=true) OVER() FROM artworks;
```

6. Write an analytic query that returns the number of intersections per polygon. Specify that Vertica ignore any usr_id that intersects less than 20 times with a given polygon.

```
=> SELECT pol_gid,
     COUNT(DISTINCT(usr_id)) AS count_user_visit
  FROM
    (SELECT pol_gid,
      usr_id,
      COUNT(usr_id) AS user_points_in
    FROM
      (SELECT STV_Intersect(usr_id, g USING PARAMETERS INDEX='art_index') OVER(PARTITION BEST) AS (usr_id,
                                          pol_gid)
    FROM usr_data
     WHERE date_time BETWEEN '2014-07-02 09:30:20' AND '2014-07-02 17:05:00') AS c
    GROUP BY pol_gid,
    usr_id HAVING COUNT(usr_id) > 20) AS real_visits
    GROUP BY pol_gid
    ORDER BY count_user_visit DESC;
```

Optimizations in the example query

This query has the following optimizations:

- The time predicated appears in the subquery.

- Using the location data table avoids the need for an expensive join.
- The query uses OVER (PARTITION BEST), to improve performance by partitioning the data.
- The user_points_in provides an estimate of the combined time spent intersecting with the artwork by all visitors.

Ensuring polygon validity before creating or refreshing an index

When Vertica creates or updates a spatial index it does not check polygon validity. To prevent getting invalid results when you query your spatial index, you should check the validity of your polygons prior to creating or updating your spatial index.

The following example shows you how to check the validity of polygons.

1. Create a table and load spatial data.

```
=> CREATE TABLE polygon_validity_test (gid INT, geom GEOMETRY);
CREATE TABLE
=> COPY polygon_validity_test (gid, gx FILLER LONG VARCHAR, geom AS St_GeomFromText(gx)) FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 2|POLYGON((-31 74,8 70,8 50,-36 53,-31 74))
>> 3|POLYGON((-38 50,4 13,11 45,0 65,-38 50))
>> 4|POLYGON((-12 42,-12 42,27 48,14 26,-12 42))
>> 5|POLYGON((0 0,1 1,0 0,2 1,1 1,0 0))
>> 6|POLYGON((3 3,2 2,2 1,2 3,3 3))
>> \.
```

2. Use ST_IsValid and STV_IsValidReason to find any invalid polygons.

```
=> SELECT gid, ST_IsValid(geom), STV_IsValidReason(geom) FROM polygon_validity_test;
 gid | ST_IsValid |            STV_IsValidReason
-----+------------+-----------------------------------------
   4 | t          |
   6 | f          | Self-intersection at or near POINT (2 1)
   2 | t          |
   3 | t          |
   5 | f          | Self-intersection at or near POINT (0 0)
(5 rows)
```

Now that we have identifed the invalid polygons in our table, there are a couple different ways you can handle the invalid polygons when creating or refreshing a spatial index.

Filtering invalid polygons using a WHERE clause

This method is slower than filtering before creating an index because it checks the validity of each polygon at execution time.

The following example shows you how to exclude invalid polygons using a WHERE clause.

```
=> SELECT STV_Create_Index(gid, geom USING PARAMETERS index = 'valid_polygons') OVER()
   FROM polygon_validity_test
   WHERE ST_IsValid(geom) = 't';
```

Filtering invalid polygons before creating or refreshing an index

This method is faster than filtering using a WHERE clause because you incur the performance cost prior to building the index.

The following example shows you how to exclude invalid polygons by creating a new table excluding invalid polygons.

```
=> CREATE TABLE polygon_validity_clean AS
   SELECT *
   FROM polygon_validity_test
   WHERE ST_IsValid(geom) = 't';
CREATE TABLE
=> SELECT STV_Create_Index(gid, geom USING PARAMETERS index = 'valid_polygons') OVER()
   FROM polygon_validity_clean;
```

STV_Intersect: scalar function vs. transform function

The STV_Intersect functions are similar in purpose, but you use them differently.

STV_Intersect Function Type	Description	Performance
Scalar	Matches a point to a polygon. If several polygons contain the point, this function returns a gid value. The result is a polygon gid or, if no polygon contains the point, the result is NULL.	Eliminates points that do not intersect with any indexed polygons, avoiding unnecessary comparisons.
Transform	Matches a point to all the polygons that contain it. When a point does not intersect with any polygon in the index, the function returns no rows.	Processes all input points regardless of whether or not they intersect with the indexed polygons.

In the following example, the STV_Intersect scalar function compares the points in the points table to the polygons in a spatial index named my_polygons . STV_Intersect returns all points and polygons that match exactly:

```
=> SELECT gid AS pt_gid
  STV_Intersect(geom USING PARAMETERS index='my_polygons') AS pol_gid
  FROM points ORDER BY pt_gid;
 pt_gid | pol_gid
--------+---------
   100 |       2
   101 |
   102 |       2
   103 |
   104 |
   105 |       3
   106 |
   107 |
(8 rows)
```

The following example shows how to use the STV_Intersect transform function to return information about the three point-polygon pairs that match and each of the polygons they match:

```
=> SELECT STV_Intersect(gid, geom
  USING PARAMETERS index='my_polygons')
  OVER (PARTITION BEST) AS (pt_gid, pol_id)
  FROM points;
 pt_gid | pol_id
--------+--------
   100 |      1
   100 |      2
   100 |      3
   102 |      2
   105 |      3
(3 rows)
```

See also

- STV_Intersect scalar function
- STV_Intersect transform function

Performing spatial joins with STV_Intersect functions

Suppose you want to process a medium-to-large spatial data set and determine which points intersect with which polygons. In that case, first create a spatial index using STV_Create_Index . A spatial index provides efficient access to the set of polygons.

Then, use the STV_Intersect scalar or transform function to identify which point-polygon pairs match.

In this section

- Spatial indexes and STV_Intersect

Spatial indexes and STV_Intersect

Before performing a spatial join using one of the STV_Intersect functions, you must first run STV_Create_Index to create a database object that contains information about polygons. This object is called a *spatial index* of the set of polygons. The spatial index improves the time it takes for the STV_Intersect functions to access the polygon data.

Vertica creates spatial indexes in a global space. Thus, any user with access to the STV_*_Index functions can describe, rename, or drop indexes created by any other user.

Vertica provides functions that work with spatial indexes:

- STV_Create_Index —Stores information about polygons in an index to improve performance.
- STV_Describe_Index —Retrieves information about an index.
- STV_Drop_Index —Deletes a spatial index.
- STV_Refresh_Index —Refreshes a spatial index.
- STV_Rename_Index —Renames a spatial index.

When to use ST_Intersects vs. STV_Intersect

Vertica provides two capabilities to identify whether a set of points intersect with a set of polygons. Depending on the size of your data set, choose the approach that gives the best performance:

- When comparing a set of geometries to a single geometry to see if they intersect, use the ST_Intersects function.
- To determine if a set of points intersects with a set of polygons in a medium-to-large data set, first create a spatial index using STV_Create_Index . Then, use one of the STV_Intersect functions to return the set of pairs that intersect.

> **Note**
> You can only perform spatial joins on GEOMETRY data.

In this section

- Performing spatial joins with ST_Intersects

Performing spatial joins with ST_Intersects

The ST_Intersects function determines if two GEOMETRY objects intersect or touch at a single point.

Use ST_Intersects when you want to identify if a small set of geometries in a column intersect with a given geometry.

Example

The following example uses ST_Intersects to compare a column of point geometries to a single polygon. The table that contains the points has 1 million rows.

ST_Intersects returns only the points that intersect with the polygon. Those points represent about 0.01% of the points in the table:

```
=> CREATE TABLE points_1m(gid IDENTITY, g GEOMETRY(100)) ORDER BY g;
=> COPY points_1m(wkt FILLER LONG VARCHAR(100), g AS ST_GeomFromText(wkt))
   FROM LOCAL '/data/points.dat';
 Rows Loaded
-------------
     1000000
(1 row)
=> SELECT ST_AsText(g) FROM points_1m WHERE
   ST_Intersects
     (
      g,
      ST_GeomFromText('POLYGON((-71 42, -70.9 42, -70.9 42.1, -71 42.1, -71 42))')
      );
         st_astext
----------------------------
 POINT (-70.97532 42.03538)
 POINT (-70.97421 42.0376)
 POINT (-70.99004 42.07538)
 POINT (-70.99477 42.08454)
 POINT (-70.99088 42.08177)
 POINT (-70.98643 42.07593)
 POINT (-70.98032 42.07982)
 POINT (-70.95921 42.00982)
 POINT (-70.95115 42.02177)
...
(116 rows)
```

Vertica recommends that you test the intersections of two columns of geometries by creating a spatial index. Use one of the STV_Intersect functions as described in STV_Intersect: scalar function vs. transform function .

Working with spatial objects from client applications

The Vertica client driver libraries provide interfaces for connecting your client applications to your Vertica database. The drivers simplify exchanging data for loading, report generation, and other common database tasks.

There are three separate client drivers:

- Open Database Connectivity (ODBC)—The most commonly-used interface for third-party applications and clients written in C, Python, PHP, Perl, and most other languages.
- Java Database Connectivity (JDBC)—Used by clients written in the Java programming language.
- ActiveX Data Objects for .NET (ADO.NET)—Used by clients developed using Microsoft's .NET Framework and written in C#, Visual Basic .NET, and other .NET languages.

Vertica Place supports the following new data types:

- LONG VARCHAR
- LONG VARBINARY
- GEOMETRY
- GEOGRAPHY

The client driver libraries support these data types; the following sections describe that support and provide examples.

In this section

- Using LONG VARCHAR and LONG VARBINARY data types with ODBC
- Using LONG VARCHAR and LONG VARBINARY data types with JDBC
- Using GEOMETRY and GEOGRAPHY data types in ODBC
- Using GEOMETRY and GEOGRAPHY data types in JDBC
- Using GEOMETRY and GEOGRAPHY data types in ADO.NET

Using LONG VARCHAR and LONG VARBINARY data types with ODBC

The ODBC drivers support the LONG VARCHAR and LONG VARBINARY data types similarly to VARCHAR and VARBINARY data types. When binding input or output parameters to a LONG VARCHAR or LONG VARBINARY column in a query, use the SQL_LONGVARCHAR and SQL_LONGVARBINARY constants to set the column's data type. For example, to bind an input parameter to a LONG VARCHAR column, you would use a statement that looks like this:

```
rc = SQLBindParameter(hdlStmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_LONGVARCHAR,
        80000, 0, (SQLPOINTER)myLongString, sizeof(myLongString), NULL);
```

Note

Do not use inefficient encoding formats for LONG VARBINARY and LONG VARCHAR values. Vertica cannot load encoded values larger than 32MB, even if the decoded value is less than 32 MB in size. For example, Vertica returns an error if you attempt to load a 32MB LONG VARBINARY value encoded in octal format, since the octal encoding quadruples the size of the value (each byte is converted into a backslash followed by three digits).

Using LONG VARCHAR and LONG VARBINARY data types with JDBC

Using LONG VARCHAR and LONG VARBINARY data types in a JDBC client application is similar to using VARCHAR and VARBINARY data types. The JDBC driver transparently handles the conversion (for example, between a Java String object and a LONG VARCHAR).

The following example code demonstrates inserting and retrieving a LONG VARCHAR string. It uses the JDBC Types class to determine the data type of the string returned by Vertica, although it does not actually need to know whether the database column is a LONG VARCHAR or just a VARCHAR in order to retrieve the value.

```
import java.sql.*;
import java.util.Properties;

public class LongVarcharExample {
    public static void main(String[] args) {
        try {
            Class.forName("com.vertica.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Could not find the JDBC driver class.");
            e.printStackTrace();
            return;
        }
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                        "jdbc:vertica://VerticaHost:5433/ExampleDB",
                        myProp);
            // establish connection and make a table for the data.
            Statement stmt = conn.createStatement();

            // How long we want the example string to be. This is
            // larger than can fit into a traditional VARCHAR (which is limited
            // to 65000.
            int length = 100000;

            // Create a table with a LONG VARCHAR column that can store
            // the string we want to insert.
            stmt.execute("DROP TABLE IF EXISTS longtable CASCADE");
            stmt.execute("CREATE TABLE longtable (text LONG VARCHAR(" + length
                    + "))");
            // Build a long string by appending an integer to a string builder
            // until we hit the size limit. Will result in a string
            // containing 01234567890123....
            StringBuilder sb = new StringBuilder(length);
```

```
        for (int i = 0; i < length; i++)
        {
            sb.append(i % 10);
        }
        String value = sb.toString();

        System.out.println("String value is " + value.length() +
                           " characters long.");

        // Create the prepared statement
        PreparedStatement pstmt = conn.prepareStatement(
                    "INSERT INTO longtable (text)" +
                    " VALUES(?)");
        try {
            // Insert LONG VARCHAR value
            System.out.println("Inserting LONG VARCHAR value");
            pstmt.setString(1, value);
            pstmt.addBatch();
            pstmt.executeBatch();

            // Query the table we created to get the value back.
            ResultSet rs = null;
            rs = stmt.executeQuery("SELECT * FROM longtable");

            // Get metadata about the result set.
            ResultSetMetaData rsmd = rs.getMetaData();
            // Print the type of the first column. Should be
            // LONG VARCHAR. Also check it against the Types class, to
            // recognize it programmatically.
            System.out.println("Column #1 data type is: " +
                               rsmd.getColumnTypeName(1));
            if (rsmd.getColumnType(1) == Types.LONGVARCHAR) {
                System.out.println("It is a LONG VARCHAR");
            } else {
                System.out.println("It is NOT a LONG VARCHAR");
            }

            // Print out the string length of the returned value.
            while (rs.next()) {
                // Use the same getString method to get the value that you
                // use to get the value of a VARCHAR.
                System.out.println("Returned string length: " +
                                   rs.getString(1).length());
            }
        } catch (SQLException e) {
            System.out.println("Error message: " + e.getMessage());
            return; // Exit if there was an error
        }
        // Cleanup
        conn.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Note

Do not use inefficient encoding formats for LONG VARBINARY and LONG VARCHAR values. Vertica cannot load encoded values larger than 32MB, even if the decoded value is less than 32 MB in size. For example, Vertica returns an error if you attempt to load a 32MB LONG VARBINARY value encoded in octal format, since the octal encoding quadruples the size of the value (each byte is converted into a backslash followed by three

digits).

Using GEOMETRY and GEOGRAPHY data types in ODBC

Vertica GEOMETRY and GEOGRAPHY data types are backed by LONG VARBINARY native types and ODBC client applications treat them as binary data. However, these data types have a format that is unique to Vertica. To manipulate this data in your C++ application, you must use the functions in Vertica that convert them to a recognized format.

To convert a WKT or WKB to the GEOMETRY or GEOGRAPHY format, use one of the following SQL functions:

- ST_GeographyFromText —Converts a WKT to a GEOGRAPHY type.
- ST_GeographyFromWKB —Converts a WKB to a GEOGRAPHY type.
- ST_GeomFromText —Converts a WKT to a GEOMETRY type.
- ST_GeomFromWKB —Converts a WKB to GEOMETRY type.

To convert a GEOMETRY or GEOGRAPHY object to its corresponding WKT or WKB, use one of the following SQL functions:

- ST_AsText —Converts a GEOMETRY or GEOGRAPHY object to a WKT, returns a LONGVARCHAR.
- ST_AsBinary —Converts a GEOMETRY or GEOGRAPHY object to a WKB, returns a LONG VARBINARY.

The following code example converts WKT data into GEOMETRY data using ST_GeomFromText and stores it in a table. Later, this example retrieves the GEOMETRY data from the table and converts it to WKT and WKB format using ST_AsText and ST_AsBinary .

```
// Compile on Linux using:
//  g++ -g -I/opt/vertica/include -L/opt/vertica/lib64 -lodbc -o SpatialData SpatialData.cpp
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <sstream>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
// Helper function to print SQL error messages.
template <typename HandleT>
void reportError(int handleTypeEnum, HandleT hdl)
{
   // Get the status records.
   SQLSMALLINT   i, MsgLen;
   SQLRETURN     ret2;
   SQLCHAR       SqlState[6], Msg[SQL_MAX_MESSAGE_LENGTH];
   SQLINTEGER    NativeError;
   i = 1;
   printf("\n");
   while ((ret2 = SQLGetDiagRec(handleTypeEnum, hdl, i, SqlState, &NativeError,
                         Msg, sizeof(Msg), &MsgLen)) != SQL_NO_DATA) {
      printf("error record %d\n", i);
      printf("sqlstate: %s\n", SqlState);
      printf("detailed msg: %s\n", Msg);
      printf("native error code: %d\n\n", NativeError);
      i++;
   }
   exit(EXIT_FAILURE); // bad form... but Ok for this demo
}
int main()
{
   // Set up the ODBC environment
   SQLRETURN ret;
```

```
SQLRETURN ret;
SQLHENV hdlEnv;
ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
assert(SQL_SUCCEEDED(ret));
// Tell ODBC that the application uses ODBC 3.
ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
    (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
assert(SQL_SUCCEEDED(ret));
// Allocate a database handle.
SQLHDBC hdlDbc;
ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
assert(SQL_SUCCEEDED(ret));
// Connect to the database
printf("Connecting to database.\n");
const char *dsnName = "ExampleDB";
const char* userID = "dbadmin";
const char* passwd = "password123";
ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
    SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
    (SQLCHAR*)passwd, SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not connect to database.\n");
    reportError<SQLHDBC>(SQL_HANDLE_DBC, hdlDbc);

} else {
    printf("Connected to database.\n");
}

// Disable AUTOCOMMIT
ret = SQLSetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF,
    SQL_NTS);

// Set up a statement handle
SQLHSTMT hdlStmt;
SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);

// Drop any previously defined table.
ret = SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS polygons",
    SQL_NTS);
if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);}

// Run query to create a table to hold a geometry.
ret = SQLExecDirect(hdlStmt,
    (SQLCHAR*)"CREATE TABLE polygons(id INTEGER PRIMARY KEY, poly GEOMETRY);",
    SQL_NTS);
if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);}

// Create the prepared statement. This will insert data into the
// table we created above. It uses the ST_GeomFromText function to convert the
// string-formatted polygon definition to a GEOMETRY datat type.
printf("Creating prepared statement\n");
ret = SQLPrepare (hdlStmt,
    (SQLTCHAR*)"INSERT INTO polygons(id, poly) VALUES(?, ST_GeomFromText(?))",
    SQL_NTS) ;
if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);}

SQLINTEGER id = 0;
int numBatches = 5;
int rowsPerBatch = 10;

// Polygon definition as a string.
char polygon[] = "polygon((1 1, 1 2, 2 2, 2 1, 1 1))";
```

```
char polygon[] = "polygon((1 1, 1 2, 2 2, 2 1, 1 1))";
// Bind variables to the parameters in the prepared SQL statement
ret = SQLBindParameter(hdlStmt, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
    0, 0, &id, 0 , NULL);
if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT,hdlStmt);}
// Bind polygon string to the geometry column
SQLBindParameter(hdlStmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_LONGVARCHAR,
    strlen(polygon), 0, (SQLPOINTER)polygon, strlen(polygon), NULL);
 if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT,hdlStmt);}
// Execute the insert
ret = SQLExecute(hdlStmt);
if(!SQL_SUCCEEDED(ret)) {
   reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);
}  else {
    printf("Executed batch.\n");
}

// Commit the transaction
printf("Committing transaction\n");
ret = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);
}  else {
    printf("Committed transaction\n");
}
// Now, create a query to retrieve the geometry.
ret = SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);}
printf("Getting data from table.\n");
// Execute a query to get the id, raw geometry data, and
// the geometry data as a string. Uses the ST_AsText SQL function to
// format raw data back into a string polygon definition
ret = SQLExecDirect(hdlStmt,
    (SQLCHAR*)"select id,ST_AsBinary(poly),ST_AsText(poly) from polygons ORDER BY id;",
    SQL_NTS);
if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT,hdlStmt);}

SQLINTEGER idval;
// 10MB buffer to hold the raw data from the geometry (10Mb is the maximum
// length of a GEOMETRY)
SQLCHAR* polygonval = (SQLCHAR*)malloc(10485760);
SQLLEN polygonlen, polygonstrlen;
// Buffer to hold a LONGVARCHAR that can result from converting the
// geometry to a string.
SQLTCHAR* polygonstr = (SQLTCHAR*)malloc(33554432);

// Get the results of the query and print each row.
do {
    ret = SQLFetch(hdlStmt);
    if (SQL_SUCCEEDED(ret)) {
        // ID column
        ret = SQLGetData(hdlStmt, 1, SQL_C_LONG, &idval, 0, NULL);
        if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);}
        printf("id: %d\n",idval);
        // The WKB format geometry data
        ret = SQLGetData(hdlStmt, 2, SQL_C_BINARY, polygonval, 10485760,
            &polygonlen);
        if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);}
        printf("Polygon in WKB format: ");
        // Print each byte of polygonval buffer in hex format.
        for (int z = 0; z < polygonlen; z++)
            printf("%02x ",polygonval[z]);
```

```
            printf("\n");
            // Geometry data formatted as a string.
            ret = SQLGetData(hdlStmt, 3, SQL_C_TCHAR, polygonstr, 33554432, &polygonstrlen);
            if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);}
            printf("Polygon in WKT format: %s\n", polygonstr);
        }
    } while(SQL_SUCCEEDED(ret));


    free(polygonval);
    free(polygonstr);
    // Clean up
    printf("Free handles.\n");
    ret = SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);}
    ret = SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);}
    ret = SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    if (!SQL_SUCCEEDED(ret)) {reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);}
    exit(EXIT_SUCCESS);
}
```

The output of running the above example is:

```
Connecting to database.
Connected to database.
Creating prepared statement
Executed batch.
Committing transaction
Committed transaction
Getting data from table.
id: 0
Polygon in WKB format: 01 03 00 00 00 01 00 00 00 05 00 00 00 00 00 00 00 00 f0 3f 00 00 00 00 00 00 f0 3f 00 00 00 00 00 00 f0 3f 00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 f0 3f 00 00 00 00 00 00 f0 3f 00 00 00 00 00 00 f0 3f
Polygon in WKT format: POLYGON ((1 1, 1 2, 2 2, 2 1, 1 1))
Free handles.
```

> **Note**
> Do not use inefficient encoding formats for LONG VARBINARY and LONG VARCHAR values. Vertica cannot load encoded values larger than 32 MB, even if the decoded value is less than 32 MB in size. For example, Vertica returns an error if you attempt to load a 32 MB LONG VARBINARY value encoded in octal format, since the octal encoding quadruples the size of the value (each byte is converted into a backslash followed by three digits).

Using GEOMETRY and GEOGRAPHY data types in JDBC

Vertica GEOMETRY and GEOGRAPHY data types are backed by LONG VARBINARY native types and JDBC client applications treat them as binary data. However, these data types have a format that is unique to Vertica. To manipulate this data in your Java application, you must use the functions in Vertica that convert them to a recognized format.

To convert a WKT or WKB to the GEOMETRY or GEOGRAPHY format, use one of the following SQL functions:

- ST_GeographyFromText —Converts a WKT to a GEOGRAPHY type.
- ST_GeographyFromWKB —Converts a WKB to a GEOGRAPHY type.
- ST_GeomFromText —Converts a WKT to a GEOMETRY type.
- ST_GeomFromWKB —Converts a WKB to GEOMETRY type.

To convert a GEOMETRY or GEOGRAPHY object to its corresponding WKT or WKB, use one of the following SQL functions:

- ST_AsText —Converts a GEOMETRY or GEOGRAPHY object to a WKT, returns a LONGVARCHAR.
- ST_AsBinary —Converts a GEOMETRY or GEOGRAPHY object to a WKB, returns a LONG VARBINARY.

The following code example converts WKT and WKB data into GEOMETRY data using ST_GeomFromText and ST_GeomFromWKB and stores it in a table. Later, this example retrieves the GEOMETRY data from the table and converts it to WKT and WKB format using ST_AsText and ST_AsBinary .

```
import java.io.InputStream;
import java.io.Reader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
public class GeospatialDemo
{
   public static void main(String [] args) throws Exception
   {
      Class.forName("com.vertica.jdbc.Driver");
      Connection conn =
          DriverManager.getConnection("jdbc:vertica://localhost:5433/db",
                                      "user", "password");
      conn.setAutoCommit(false);

      Statement stmt = conn.createStatement();
      stmt.execute("CREATE TABLE polygons(id INTEGER PRIMARY KEY, poly GEOMETRY)");

      int id = 0;
      int numBatches = 5;
      int rowsPerBatch = 10;

      //batch inserting WKT data
      PreparedStatement pstmt = conn.prepareStatement("INSERT INTO polygons
                              (id, poly) VALUES(?, ST_GeomFromText(?))");
      for(int i = 0; i < numBatches; i++)
      {

         for(int j = 0; j < rowsPerBatch; j++)
         {
            //Insert your own WKT data here
            pstmt.setInt(1, id++);
            pstmt.setString(2, "polygon((1 1, 1 2, 2 2, 2 1, 1 1))");
            pstmt.addBatch();
         }
         pstmt.executeBatch();
      }

      conn.commit();
      pstmt.close();
      //batch insert WKB data
      pstmt = conn.prepareStatement("INSERT INTO polygons(id, poly)
                                 VALUES(?, ST_GeomFromWKB(?))");
      for(int i = 0; i < numBatches; i++)
      {
         for(int j = 0; j < rowsPerBatch; j++)
         {
            //Insert your own WKB data here
            byte [] wkb = getWKB();
            pstmt.setInt(1, id++);
            pstmt.setBytes(2, wkb);
            pstmt.addBatch();
         }
         pstmt.executeBatch();
      }
```

```
        conn.commit();
        pstmt.close();
        //selecting data as WKT
        ResultSet rs = stmt.executeQuery("select ST_AsText(poly) from polygons");
        while(rs.next())
        {
            String wkt = rs.getString(1);
            Reader wktReader = rs.getCharacterStream(1);
            //process the wkt as necessary
        }
        rs.close();

        //selecting data as WKB
        rs = stmt.executeQuery("select ST_AsBinary(poly) from polygons");
        while(rs.next())
        {
            byte [] wkb = rs.getBytes(1);
            InputStream wkbStream = rs.getBinaryStream(1);
            //process the wkb as necessary
        }
        rs.close();

        //binding parameters in predicates
        pstmt = conn.prepareStatement("SELECT id FROM polygons WHERE
                                ST_Contains(ST_GeomFromText(?), poly)");
        pstmt.setString(1, "polygon((1 1, 1 2, 2 2, 2 1, 1 1))");
        rs = pstmt.executeQuery();
        while(rs.next())
        {
            int pk = rs.getInt(1);
            //process the results as necessary
        }
        rs.close();

        conn.close();
    }
}
```

Using GEOMETRY and GEOGRAPHY data types in ADO.NET

Vertica GEOMETRY and GEOGRAPHY data types are backed by LONG VARBINARY native types and ADO.NET client applications treat them as binary data. However, these data types have a format that is unique to Vertica. To manipulate this data in your C# application, you must use the functions in Vertica that convert them to a recognized format.

To convert a WKT or WKB to the GEOMETRY or GEOGRAPHY format, use one of the following SQL functions:

- ST_GeographyFromText —Converts a WKT to a GEOGRAPHY type.
- ST_GeographyFromWKB —Converts a WKB to a GEOGRAPHY type.
- ST_GeomFromText —Converts a WKT to a GEOMETRY type.
- ST_GeomFromWKB —Converts a WKB to GEOMETRY type.

To convert a GEOMETRY or GEOGRAPHY object to its corresponding WKT or WKB, use one of the following SQL functions:

- ST_AsText —Converts a GEOMETRY or GEOGRAPHY object to a WKT, returns a LONGVARCHAR.
- ST_AsBinary —Converts a GEOMETRY or GEOGRAPHY object to a WKB, returns a LONG VARBINARY.

The following C# code example converts WKT data into GEOMETRY data using ST_GeomFromText and stores it in a table. Later, this example retrieves the GEOMETRY data from the table and converts it to WKT and WKB format using ST_AsText and ST_AsBinary .

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         VerticaConnectionStringBuilder builder =
                        new VerticaConnectionStringBuilder();
         builder.Host = "VerticaHost";
         builder.Database = "VMart";
         builder.User = "ExampleUser";
         builder.Password = "password123";
         VerticaConnection _conn = new
                         VerticaConnection(builder.ToString());
         _conn.Open();

         VerticaCommand command = _conn.CreateCommand();
         command.CommandText = "DROP TABLE IF EXISTS polygons";
         command.ExecuteNonQuery();
         command.CommandText =
           "CREATE TABLE polygons (id INTEGER PRIMARY KEY, poly GEOMETRY)";
         command.ExecuteNonQuery();
         // Prepare to insert a polygon using a prepared statement. Use the
         // ST_GeomFromtText SQl function to convert from WKT to GEOMETRY.
         VerticaTransaction txn = _conn.BeginTransaction();
         command.CommandText =
          "INSERT into polygons VALUES(@id, ST_GeomFromText(@polygon))";
         command.Parameters.Add(new
                        VerticaParameter("id", VerticaType.BigInt));
         command.Parameters.Add(new
                        VerticaParameter("polygon", VerticaType.VarChar));
         command.Prepare();
         // Set the values for the parameters
         command.Parameters["id"].Value = 0;
         //
         command.Parameters["polygon"].Value =
                         "polygon((1 1, 1 2, 2 2, 2 1, 1 1))";
         // Execute the query to insert the value
         command.ExecuteNonQuery();

         // Now query the table
         VerticaCommand query = _conn.CreateCommand();
         query.CommandText =
           "SELECT id, ST_AsText(poly), ST_AsBinary(poly) FROM polygons;";
         VerticaDataReader dr = query.ExecuteReader();
         while (dr.Read())
         {
            Console.WriteLine("ID: " + dr[0]);
            Console.WriteLine("Polygon WKT format data type: "
               + dr.GetDataTypeName(1) +
               " Value: " + dr[1]);
            // Get the WKB format of the polygon and print it out as hex.
            Console.Write("Polygon WKB format data type: "
                        + dr.GetDataTypeName(2));
            Console.WriteLine(" Value: "
                        + BitConverter.ToString((byte[])dr[2]));
         }
         _conn.Close();
      }
   }
```

```
    }
}
```

The example code prints the following on the system console:

```
ID: 0
Polygon WKT format data type: LONG VARCHAR Value: POLYGON ((1 1, 1 2,
2 2, 2 1,1 1))
Polygon WKB format data type: LONG VARBINARY Value: 01-03-00-00-00-01
-00-00-00-05-00-00-00-00-00-00-00-00-00-F0-3F-00-00-00-00-00-00-F0-3F
-00-00-00-00-00-00-F0-3F-00-00-00-00-00-00-00-40-00-00-00-00-00-00-00
-40-00-00-00-00-00-00-00-40-00-00-00-00-00-00-00-40-00-00-00-00-00-00
-F0-3F-00-00-00-00-00-00-F0-3F-00-00-00-00-00-00-F0-3F
```

OGC spatial definitions

Using Vertica requires an understanding of the Open Geospatial Consortium (OGC) concepts and capabilities. For more information, see the OGC Simple Feature Access Part 1 - Common Architecture specification.

In this section

- Spatial classes
- Spatial object representations
- Spatial definitions

Spatial classes

Vertica supports several classes of objects, as defined in the OGC standards.

In this section

- Point
- Multipoint
- Linestring
- Multilinestring
- Polygon
- Multipolygon

Point

A location in two-dimensional space that is identified by one of the following:

- X and Y coordinates
- Longitude and latitude values

A point has dimension 0 and no boundary.

Examples

The following example uses a GEOMETRY point:

```
=> CREATE TABLE point_geo (gid int, geom GEOMETRY(100));
CREATE TABLE
=> COPY point_geo(gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter ',';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1, POINT(3 5)
>>\.
=> SELECT gid, ST_AsText(geom) FROM point_geo;
 gid |  ST_AsText
-----+-------------
   1 | POINT (3 5)
(1 row)
```

The following example uses a GEOGRAPHY point:

```
=> CREATE TABLE point_geog (gid int, geog geography(100));
CREATE TABLE
=> COPY point_geog(gid, gx filler LONG VARCHAR, geog AS ST_GeographyFromText(gx)) FROM stdin delimiter ',';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1, POINT(42 71)
>>\.
=> SELECT gid, ST_AsText(geog) FROM point_geog;
 gid |   ST_AsText
-----+---------------
   1 | POINT (42 71)
(1 row)
```

Multipoint

A set of one or more points. A multipoint object has dimension 0 and no boundary.

Examples

The following example uses a GEOMETRY multipoint:

```
=> CREATE TABLE mpoint_geo (gid int, geom GEOMETRY(1000));
CREATE TABLE
=> COPY mpoint_geo(gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1|MULTIPOINT(4 7, 8 10)
>>\.
=> SELECT gid, ST_AsText(geom) FROM mpoint_geo;
 gid |       st_astext
-----+------------------------
   1 | MULTIPOINT (7 8, 6 9)
(1 row)
```

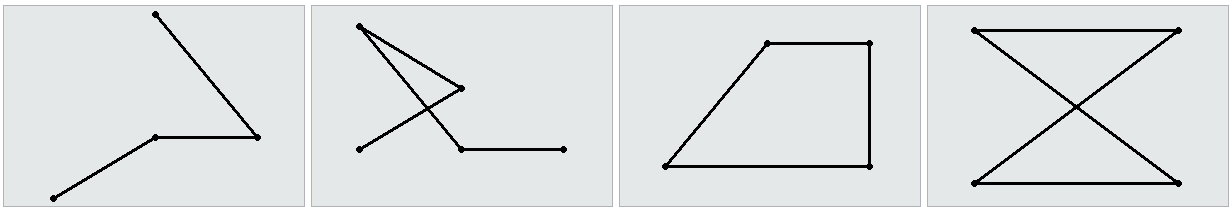
The following example uses a GEOGRAPHY multipoint:

```
=> CREATE TABLE mpoint_geog (gid int, geog GEOGRAPHY(1000));
CREATE TABLE
=> COPY mpoint_geog(gid, gx filler LONG VARCHAR, geog AS ST_GeographyFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1|MULTIPOINT(42 71, 41.4 70)
>>\.
=> SELECT gid, ST_AsText(geom) FROM mpoint_geo;
 gid |       st_astext
-----+------------------------
   1 | MULTIPOINT (42 71, 41.4 70)
(1 row)
```

Linestring

One or more connected lines, identified by pairs of consecutive points. A linestring has dimension 1. The boundary of a linestring is a multipoint object containing its start and end points.

The following are examples of linestrings:



Examples

The following example uses the GEOMETRY type to create a table, use copy to load a linestring to the table, and then queries the table to view the linestring:

```
=> CREATE TABLE linestring_geom (gid int, geom GEOMETRY(1000));
CREATE TABLE
=> COPY linestring_geom(gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1|LINESTRING(0 0, 1 1, 2 2, 3 4, 2 4, 1 5)
>>\.
=> SELECT gid, ST_AsText(geom) FROM linestring_geom;
 gid |               ST_AsText
-----+-----------------------------------------
   1 | LINESTRING (0 0, 1 1, 2 2, 3 4, 2 4, 1 5)
(1 row)
```

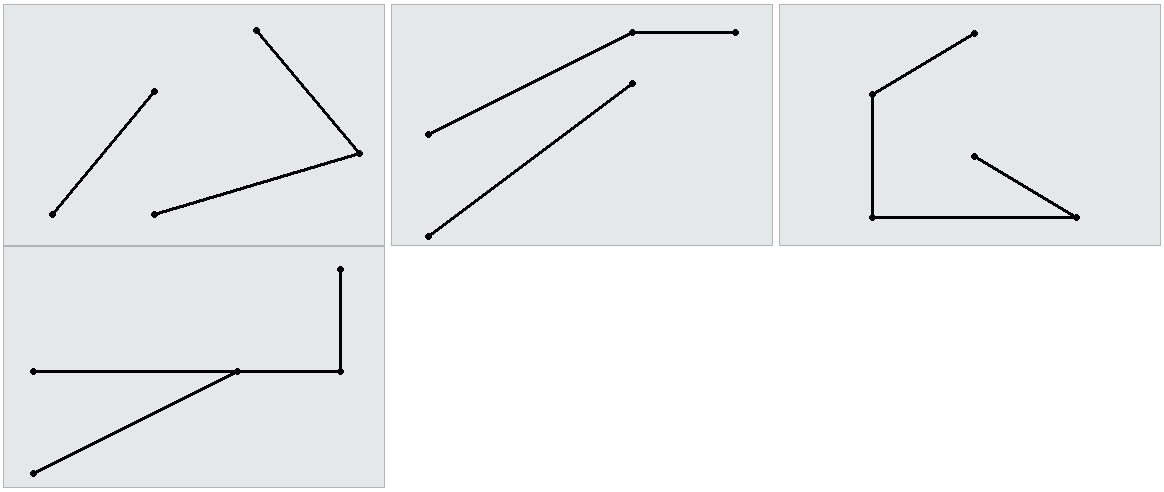
The following example uses the GEOGRAPHY type to create a table, use copy to load a linestring to the table, and then queries the table to view the linestring:

```
=> CREATE TABLE linestring_geog (gid int, geog GEOGRAPHY(1000));
CREATE TABLE
=> COPY linestring_geog(gid, gx filler LONG VARCHAR, geog AS ST_GeographyFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1|LINESTRING(42.1 71, 41.4 70, 41.3 72.9, 42.99 71.46, 44.47 73.21)
>>\.
=> SELECT gid, ST_AsText(geog) FROM linestring_geog;
 gid |                          ST_AsText
-----+-------------------------------------------------------------------
   1 | LINESTRING (42.1 71, 41.4 70, 41.3 72.9, 42.99 71.46, 44.47 73.21)
(1 row)
```

Multilinestring

A collection of zero or more linestrings. A multilinestring has no dimension. The boundary of a multilinestring is a multipoint object containing the start and end points of all the linestrings.

The following are examples of multilinestrings:



Examples

The following example uses the GEOMETRY type to create a table, use copy to load a multilinestring to the table, and then queries the table to view the multilinestring:

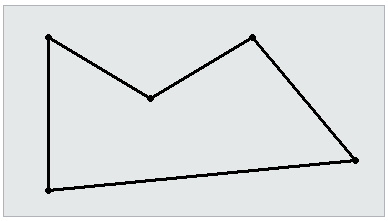
```
=> CREATE TABLE multilinestring_geom (gid int, geom GEOMETRY(1000));
CREATE TABLE
=> COPY multilinestring_geom(gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1|MULTILINESTRING((1 5, 2 4, 5 3, 6 6),(3 5, 3 7))
>>\.
=> SELECT gid, ST_AsText(geom) FROM multilinestring_geom;
 gid |                    ST_AsText
-----+-----------------------------------------------------
   1 | MULTILINESTRING ((1 5, 2 4, 5 3, 6 6), (3 5, 3 7))
(1 row)
```

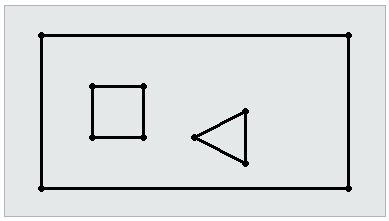
The following example uses the GEOGRAPHY type to create a table, use copy to load a multilinestring to the table, and then queries the table to view the multilinestring:

```
=> CREATE TABLE multilinestring_geog (gid int, geog GEOGRAPHY(1000));
CREATE TABLE
=> COPY multilinestring_geog(gid, gx filler LONG VARCHAR, geog AS ST_GeographyFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1|MULTILINESTRING((42.1 71, 41.4 70, 41.3 72.9), (42.99 71.46, 44.47 73.21))
>>\.
=> SELECT gid, ST_AsText(geog) FROM multilinestring_geog;
 gid |                              ST_AsText
-----+-------------------------------------------------------------------------
   1 | MULTILINESTRING((42.1 71, 41.4 70, 41.3 72.9), (42.99 71.46, 44.47 73.21))
(1 row)
```

Polygon

An object identified by a set of closed linestrings. A polygon can have one or more holes, as defined by interior boundaries, but all points must be connected. Two examples of polygons are:





Inclusive and exclusive polygons

Polygons that include their points in clockwise order include all space inside the perimeter of the polygon and exclude all space outside that perimeter. Polygons that include their points in counterclockwise order exclude all space inside the perimeter and include all space outside that perimeter.

Examples

The following example uses the GEOMETRY type to create a table, use copy to load a polygon into the table, and then queries the table to view the polygon:

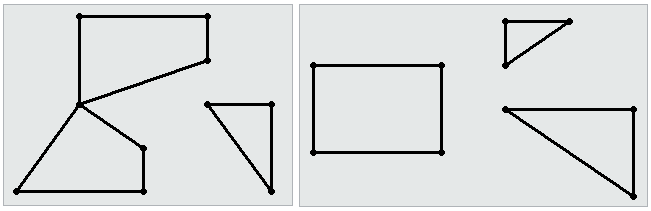
```
=> CREATE TABLE polygon_geom (gid int, geom GEOMETRY(1000));
CREATE TABLE
=> COPY polygon_geom(gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1|POLYGON(( 2 6, 2 9, 6 9, 7 7, 4 6, 2 6))
>>\.
=> SELECT gid, ST_AsText(geom) FROM polygon_geom;
 gid |               ST_AsText
-----+-----------------------------------------
   1 | POLYGON((2 6, 2 9, 6 9, 7 7, 4 6, 2 6))
(1 row)
```

The following example uses the GEOGRAPHY type to create a table, use copy to load a polygon into the table, and then queries the table to view the polygon:

```
=> CREATE TABLE polygon_geog (gid int, geog GEOGRAPHY(1000));
CREATE TABLE
=> COPY polygon_geog(gid, gx filler LONG VARCHAR, geog AS ST_GeographyFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1|POLYGON((42.1 71, 41.4 70, 41.3 72.9, 44.47 73.21, 42.99 71.46, 42.1 71))
>>\.
=> SELECT gid, ST_AsText(geog) FROM polygon_geog;
 gid |                            ST_AsText
-----+-------------------------------------------------------------------------
   1 | POLYGON((42.1 71, 41.4 70, 41.3 72.9, 44.47 73.21, 42.99 71.46, 42.1 71))
(1 row)
```

Multipolygon

A collection of zero or more polygons that do not overlap.



Examples

The following example uses the GEOMETRY type to create a table, use copy to load a multipolygon into the table, and then queries the table to view the polygon:

```
=> CREATE TABLE multipolygon_geom (gid int, geom GEOMETRY(1000));
CREATE TABLE
=> COPY multipolygon_geom(gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>9|MULTIPOLYGON(((2 6, 2 9, 6 9, 7 7, 4 6, 2 6)),((0 0, 0 5, 1 0, 0 0)),((0 2, 2 5, 4 5, 0 2)))
>>\.
=> SELECT gid, ST_AsText(geom) FROM polygon_geom;
 gid |                                    ST_AsText
-----+-----------------------------------------------------------------------------------------------
   9 | MULTIPOLYGON(((2 6, 2 9, 6 9, 7 7, 4 6, 2 6)),((0 0, 0 5, 1 0, 0 0)),((0 2, 2 5, 4 5, 0 2)))
(1 row)
```

The following example uses the GEOGRAPHY type to create a table, use copy to load a multipolygon into the table, and then queries the table to view the polygon:

```
=> CREATE TABLE multipolygon_geog (gid int, geog GEOGRAPHY(1000));
CREATE TABLE
=> COPY polygon_geog(gid, gx filler LONG VARCHAR, geog AS ST_GeographyFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1|POLYGON((42.1 71, 41.4 70, 41.3 72.9, 44.47 73.21, 42.99 71.46, 42.1 71))
>>\.
=> SELECT gid, ST_AsText(geog) FROM polygon_geog;
 gid |                            ST_AsText
-----+-------------------------------------------------------------------------
   1 | POLYGON(((42.1 71, 41.4 70, 41.3 72.9, 42.1 71)),((44.47 73.21, 42.99 71.46, 42.1 71, 44.47 73.21)))
(1 row)
```

Spatial object representations

The OGC defines two ways to represent spatial objects:

- Well-known text (WKT)
- Well-known binary (WKB)

In this section

- Well-known text (WKT)
- Well-known binary (WKB)

Well-known text (WKT)

Well-Known Text (WKT) is an ASCII representation of a spatial object.

WKTs are not case sensitive; Vertica recognizes any combination of lowercase and uppercase letters.

Some examples of valid WKTs are:

WKT Example	**Description**
POINT(1 2)	The point (1,2)
MULTIPOINT(0 0,1 1)	A set made up of the points (0,0) and (1,1)
LINESTRING(1.5 2.45,3.21 4)	The line from the point (1.5,2.45) to the point (3.21,4)
MULTILINESTRING((0 0,–1 –2,–3 –4),(2 3,3 4,6 7))	Two linestrings, one that passes through (0,0), (–1,–2), and (–3,–4), and one that passes through (2,3), (3,4), and (6,7).
POLYGON((1 2,1 4,3 4,3 2,1 2))	The rectangle whose four corners are indicated by (1,2), (1,4), (3,4), and (3,2). A polygon must be closed, so the first and last points in the WKT must match.
POLYGON((0.5 0.5,5 0,5 5,0 5,0.5 0.5), (1.5 1,4 3,4 1,1.5 1))	A polygon (0.5 0.5,5 0,5 5,0 5,0.5 0.5) with a hole in it (1.5 1,4 3,4 1,1.5 1).
MULTIPOLYGON(((0 1,3 0,4 3,0 4,0 1)), ((3 4,6 3,5 5,3 4)), ((0 0,–1 –2,–3 –2,–2 –1,0 0)))	A set of three polygons
GEOMETRYCOLLECTION(POINT(5 8), LINESTRING(–1 3,1 4))	A set containing the point (5,8) and the line from (–1,3) to (1,4)
POINT EMPTY MULTIPOINT EMPTY LINESTRING EMPTY MULTILINESTRING EMPTY MULTILINESTRING(EMPTY) POLYGON EMPTY POLYGON(EMPTY) MULTIPOLYGON EMPTY MULTIPOLYGON(EMPTY)	Empty spatial objects; empty objects have no points.

Invalid WKTs are:

- POINT(1 NAN), POINT(1 INF)—Coordinates must be numbers.
- POLYGON((1 2, 1 4, 3 4, 3 2))—A polygon must be closed.
- POLYGON((1 4, 2 4))—A linestring is not a valid polygon.

Well-known binary (WKB)

Well-Known Binary (WKB) is a binary representation of a spatial object. This format is primarily used to port spatial data between applications.

Spatial definitions

The OGC defines properties that describe

- Characteristics of spatial objects
- Spatial relationships that can exist among objects

Vertica provides functions that test for and analyze the following properties and relationships.

In this section

- Boundary
- Buffer
- Contains
- Convex hull
- Crosses
- Disjoint
- Envelope
- Equals
- Exterior
- GeometryCollection
- Interior
- Intersection
- Overlaps
- Relates
- Simple
- Symmetric difference
- Union
- Validity
- Within

Boundary

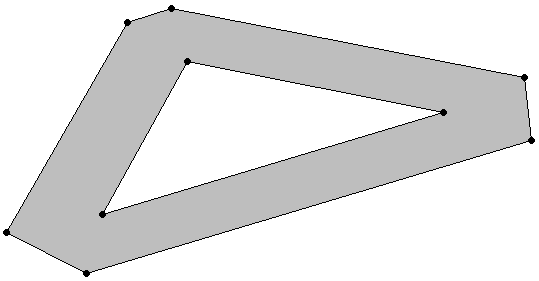
The set of points that define the limit of a spatial object:

- Points, multipoints, and geometrycollections do not have boundaries.
- The boundary of a linestring is a multipoint object. This object contains its start and end points.
- The boundary of a multilinestring is a multipoint object. This object contains the start and end points of all the linestrings that make up the multilinestring.
- The boundary of a polygon is a linestring that begins and ends at the same point. If the polygon has one or more holes, the boundary is a multilinestring that contains the boundaries of the exterior polygon and any interior polygons.
- The boundary of a multipolygon is a multilinestring that contains the boundaries of all the polygons that make up the multipolygon.

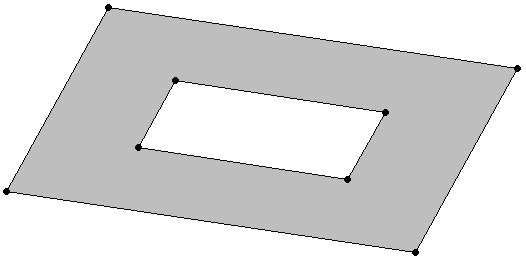
Buffer

The set of all points that are within or equal to a specified distance from the boundary of a spatial object. The distance can be positive or negative.

Positive buffer:



Negative buffer:



Contains

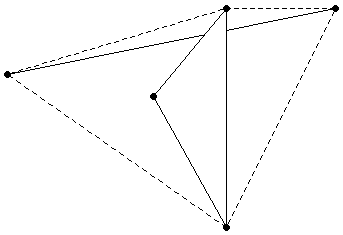
One spatial object contains another spatial object if its interior includes all points of the other object. If an object such as a point or linestring only exists along a polygon's boundary, the polygon does not contain it. If a point is on a linestring, the linestring contains it; the interior of a linestring is all the points on the linestring *except* the start and end points.

Contains(a, b) is spatially equivalent to within(b, a).

Convex hull

The smallest convex polygon that contains one or more spatial objects.

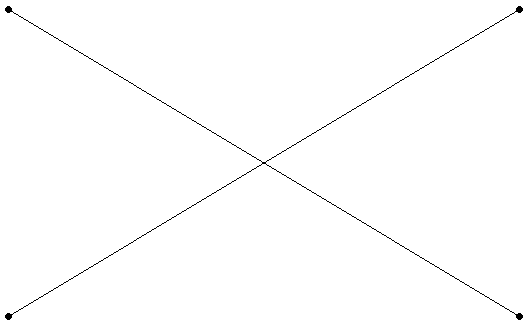
In the following figure, the dotted lines represent the convex hull for a linestring and a triangle.



Crosses

Two spatial objects cross if both of the following are true:

- The two objects have some but not all interior points in common.
- The dimension of the result of their intersection is less than the maximum dimension of the two objects.



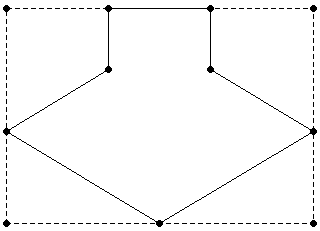
Disjoint

Two spatial objects have no points in common; they do not intersect or touch.

Envelope

The minimum bounding rectangle that contains a spatial object.

The envelope for the following polygon is represented by the dotted lines in the following figure.



Equals

Two spatial objects are equal when their coordinates match exactly. Synonymous with *spatially equivalent* .

The order of the points do not matter in determining spatial equivalence:

- LINESTRING(1 2, 4 3) equals LINESTRING(4 3, 1 2).
- POLYGON ((0 0, 1 1, 1 2, 2 2, 2 1, 3 0, 1.5 –1.5, 0 0)) equals POLYGON((1 1 , 1 2, 2 2, 2 1, 3 0, 1.5 –1.5, 0 0, 1 1)).
- MULTILINESTRING((1 2, 4 3),(0 0, –1 –4)) equals MULTILINESTRING((0 0, –1 –4),(1 2, 4 3)).

Exterior

The set of points not contained within a spatial object nor on its boundary.

GeometryCollection

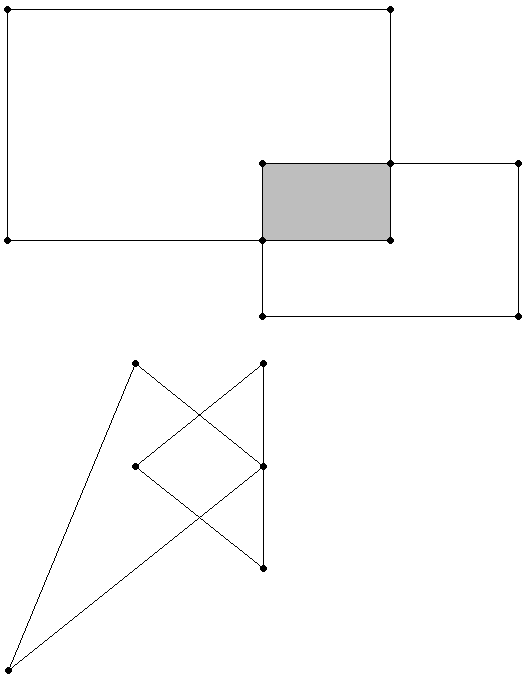
A set of zero or more objects from any of the supported classes of spatial objects.

Interior

The set of points contained in a spatial object, excluding its boundary.

Intersection

The set of points that two or more spatial objects have in common.



Overlaps

If a spatial object shares space with another object, but is not contained within that object, the objects overlap. The objects must overlap at their interiors; if two objects touch at a single point or intersect only along a boundary, they do not overlap.

Relates

When a spatial object is spatially related to another object as defined by a DE-9IM pattern matrix string.

A DE-9IM pattern matrix string identifies how two spatial objects are spatially related to each other. For more information about the DE-9IM standard, see Understanding Spatial Relations .

Simple

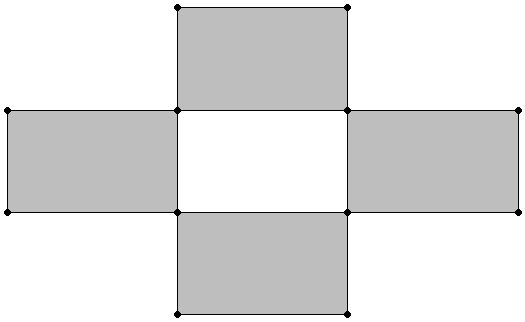
For points, multipoints, linestrings, or multilinestrings, a spatial object is simple if it does not intersect itself or has no self-tangency points.

Polygons, multipolygons, and geometrycollections are always simple.

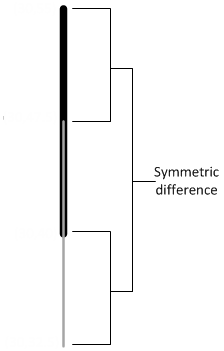
Symmetric difference

The set of all points of a pair of spatial objects where the objects do not intersect. This difference is spatially equivalent to the union of the two objects less their intersection. The symmetric difference contains the boundaries of the intersections.

In the following figure, the shaded areas represent the symmetric difference of the two rectangles.

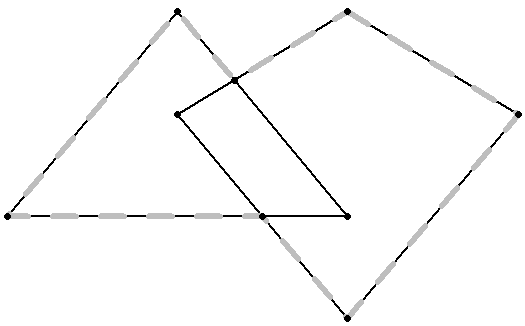


The following figure shows the symmetric difference of two overlapping linestrings.



Union

For two or more spatial objects, the set of all points in all the objects.



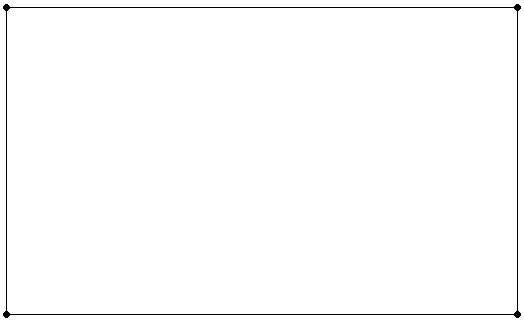
Validity

For a polygon or multipolygon, when all of the following are true:

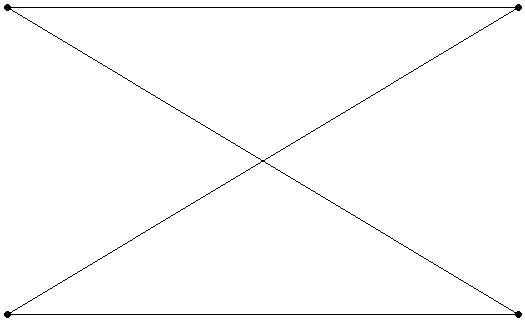
- It is closed; its start point is the same as its end point.
- Its boundary is a set of linestrings.
- No two linestrings in the boundary cross. The linestrings in the boundary may touch at a point but they cannot cross.

- Any polygons in the interior must be completely contained; they cannot touch the boundary of the exterior polygon *except* at a vertex.

Valid polygons:



Invalid polygon:

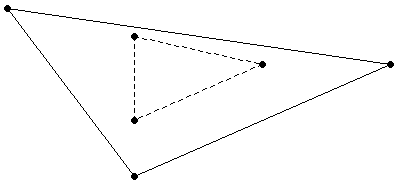


Within

A spatial object is considered within another spatial object when all its points are inside the other object's interior. Thus, if a point or linestring only exists along a polygon's boundary, it is not considered within the polygon. The polygon boundary is not part of its interior.

If a point is on a linestring, it is considered within the linestring. The interior of a linestring is all the points along the linestring, except the start and end points.

Within(a, b) is spatially equivalent to contains(b, a).



Spatial data type support limitations

Vertica does not support all types of GEOMETRY and GEOGRAPHY objects. See the respective function page for a list of objects that function supports. Spherical geometry is generally more complex than Euclidean geometry. Thus, there are fewer spatial functions that support the GEOGRAPHY data type.

Limitations of spatial data type support:

- A non-WGS84 GEOGRAPHY object is a spatial object defined on the surface of a perfect sphere of radius 6371 kilometers. This sphere approximates the shape of the earth. Other spatial programs may use an ellipsoid to model the earth, resulting in slightly different data.
- You cannot modify the size or data type of a GEOMETRY or GEOGRAPHY column after creation.
- You cannot import data to or export data from tables that contain spatial data from another Vertica database.
- You can only use the STV_Intersect functions with points and polygons.
- GEOGRAPHY objects of type GEOMETRYCOLLECTION are not supported.
- Values for longitude must be between -180 and +180 degrees. Values for latitude must be between –90 and +90 degrees. The Vertica geospatial functions do not validate these values.
- GEOMETRYCOLLECTION objects cannot contain empty objects. For example, you cannot specify GEOMETRYCOLLECTION (LINESTRING(1 2, 3 4), POINT(5 6) , POINT EMPTY) .
- If you pass a spatial function a NULL geometry, the function returns NULL, unless otherwise specified. A result of NULL has no value.

- Polymorphic functions, such as NVL and GREATEST , do not accept GEOMETRY and GEOGRAPHY arguments.

Time series analytics

Time series analytics evaluate the values of a given set of variables over time and group those values into a window (based on a time interval) for analysis and aggregation. Common scenarios for using time series analytics include: stock market trades and portfolio performance changes over time, and charting trend lines over data.

Since both time and the state of data within a time series are continuous, it can be challenging to evaluate SQL queries over time. Input records often occur at non-uniform intervals, which can create gaps. To solve this problem Vertica provides:

- Gap-filling functionality, which fills in missing data points
- Interpolation scheme, which constructs new data points within the range of a discrete set of known data points.

Vertica interpolates the non-time series columns in the data (such as analytic function results computed over time slices) and adds the missing data points to the output. This section describes gap filling and interpolation in detail.

You can use event-based windows to break time series data into windows that border on significant events within the data. This is especially relevant in financial data, where analysis might focus on specific events as triggers to other activity.

Sessionization is a special case of event-based windows that is frequently used to analyze click streams, such as identifying web browsing sessions from recorded web clicks.

Vertica provides additional support for time series analytics with the following SQL extensions:

- TIMESERIES clause in a SELECT statement supports gap-filling and interpolation (GFI) computation.
- TS_FIRST_VALUE and TS_LAST_VALUE are time series aggregate functions that return the value at the start or end of a time slice, respectively, which is determined by the interpolation scheme.
- TIME_SLICE is a (SQL extension) date/time function that aggregates data by different fixed-time intervals and returns a rounded-up input TIMESTAMP value to a value that corresponds with the start or end of the time slice interval.

See also

- SQL analytics
- Event-based windows
- Sessionization with event-based windows

In this section

- Gap filling and interpolation (GFI)
- Null values in time series data

Gap filling and interpolation (GFI)

The examples and graphics that explain the concepts in this topic use the following simple schema:

```
CREATE TABLE TickStore (ts TIMESTAMP, symbol VARCHAR(8), bid FLOAT);
INSERT INTO TickStore VALUES ('2009-01-01 03:00:00', 'XYZ', 10.0);
INSERT INTO TickStore VALUES ('2009-01-01 03:00:05', 'XYZ', 10.5);
COMMIT;
```

In Vertica, time series data is represented by a sequence of rows that conforms to a particular table schema, where one of the columns stores the time information.

Both time and the state of data within a time series are continuous. Thus, evaluating SQL queries over time can be challenging because input records usually occur at non-uniform intervals and can contain gaps.

For example, the following table contains two input rows five seconds apart: 3:00:00 and 3:00:05.

```
=> SELECT * FROM TickStore;
        ts         | symbol | bid
---------------------+--------+------
 2009-01-01 03:00:00 | XYZ    |   10
 2009-01-01 03:00:05 | XYZ    | 10.5
(2 rows)
```

Given those two inputs, how can you determine a bid price that falls between the two points, such as at 3:00:03 PM? The TIME_SLICE function normalizes timestamps into corresponding time slices; however, TIME_SLICE does not solve the problem of missing inputs (time slices) in the data. Instead, Vertica provides gap-filling and interpolation (GFI) functionality, which fills in missing data points and adds new (missing) data points within a range of known data points to the output. It accomplishes these tasks with the time series aggregate functions (TS_FIRST_VALUE and TS_LAST_VALUE) and the SQL TIMESERIES clause . But first, we'll illustrate the components that make up gap filling and interpolation in Vertica, starting with Constant interpolation . The images in the following topics use the following legend:

- The x-axis represents the timestamp (ts) column
- The y-axis represents the bid column.
- The vertical blue lines delimit the time slices.
- The red dots represent the input records in the table, $10.0 and $10.5.
- The blue stars represent the output values, including interpolated values.

In this section

- Constant interpolation
- TIMESERIES clause and aggregates
- Time series rounding
- Linear interpolation
- GFI examples

Constant interpolation

Given known input timestamps at 03:00:00 and 03:00:05 in the sample TickStore schema , how might you determine the bid price at 03:00:03?

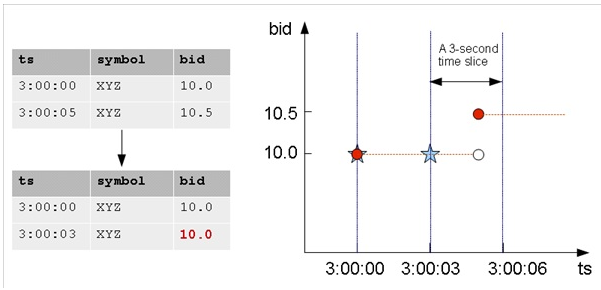
A common interpolation scheme used on financial data is to set the bid price to *the last seen value so far* . This scheme is referred to as **constant interpolation** , in which Vertica computes a new value based on the previous input records.

> **Note**
> Constant is Vertica's default interpolation scheme. Another interpolation scheme, linear , is discussed in an upcoming topic.

Returning to the problem query, here is the table output, which shows a 5-second lag between bids at 03:00:00 and 03:00:05:

```
=> SELECT * FROM TickStore;
        ts         | symbol | bid
---------------------+--------+------
 2009-01-01 03:00:00 | XYZ    |   10
 2009-01-01 03:00:05 | XYZ    | 10.5
(2 rows)
```

Using constant interpolation, the interpolated bid price of XYZ remains at $10.0 at 3:00:03, which falls between the two known data inputs (3:00:00 PM and 3:00:05). At 3:00:05, the value changes to $10.5. The known data points are represented by a red dot, and the interpolated value at 3:00:03 is represented by the blue star.



In order to write a query that makes the input rows more uniform, you first need to understand the TIMESERIES clause and time series aggregate functions .

TIMESERIES clause and aggregates

The SELECT..TIMESERIES clause and time series aggregates help solve the problem of gaps in input records by normalizing the data into 3-second time slices and interpolating the bid price when it finds gaps.

TIMESERIES clause

The TIMESERIES clause is an important component of time series analytics computation. It performs gap filling and interpolation (GFI) to generate time slices missing from the input records. The clause applies to the timestamp columns/expressions in the data, and takes the following form:

```
TIMESERIES slice_time AS 'length_and_time_unit_expression'
OVER ( ... [ window-partition-clause[ , ... ] ]
... ORDER BY time_expression )
... [ ORDER BY table_column [ , ... ] ]
```

> **Note**
> The TIMESERIES clause requires an ORDER BY operation on the timestamp column.

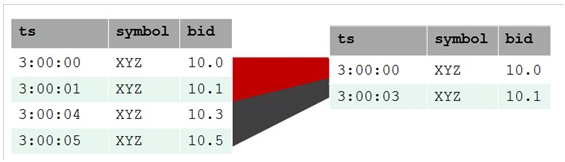
Time series aggregate functions

The Timeseries Aggregate (TSA) functions TS_FIRST_VALUE and TS_LAST_VALUE evaluate the values of a given set of variables over time and group those values into a window for analysis and aggregation.

TSA functions process the data that belongs to each time slice. One output row is produced per time slice or per partition per time slice if a partition expression is present.

The following table shows 3-second time slices where:

- The first two rows fall within the first time slice, which runs from 3:00:00 to 3:00:02. These are the input rows for the TSA function's output for the time slice starting at 3:00:00.
- The second two rows fall within the second time slice, which runs from 3:00:03 to 3:00:05. These are the input rows for the TSA function's output for the time slice starting at 3:00:03.
  The result is the start of each time slice.



ts	symbol	bid
3:00:00	XYZ	10.0
3:00:01	XYZ	10.1
3:00:04	XYZ	10.3
3:00:05	XYZ	10.5

ts	symbol	bid
3:00:00	XYZ	10.0
3:00:03	XYZ	10.1

Examples

The following examples compare the values returned with and without the TS_FIRST_VALUE TSA function.

This example shows the TIMESERIES clause without the TS_FIRST_VALUE TSA function.

```
=> SELECT slice_time, bid FROM TickStore TIMESERIES slice_time AS '3 seconds' OVER(PARTITION by TickStore.bid ORDER BY ts);
```

This example shows both the TIMESERIES clause and the TS_FIRST_VALUE TSA function. The query returns the values of the bid column, as determined by the specified constant interpolation scheme.

```
=> SELECT slice_time, TS_FIRST_VALUE(bid, 'CONST') bid FROM TickStore
  TIMESERIES slice_time AS '3 seconds' OVER(PARTITION by symbol ORDER BY ts);
```

Vertica interpolates the last known value and fills in the missing datapoint, returning 10 at 3:00:03:

First query		Interpolated value
slice_time | bid ---+--- 2009-01-01 03:00:00 | 10 2009-01-01 03:00:03 | 10.5 (2 rows)	==>	slice_time | bid ---+--- 2009-01-01 03:00:00 | 10 2009-01-01 03:00:03 | **10** (2 rows)

Time series rounding

Vertica calculates all time series as equal intervals relative to the timestamp 2000-01-01 00:00:00 . Vertica rounds time series timestamps as needed, to conform with this baseline. Start times are also rounded down to the nearest whole unit for the specified interval.

Given this logic, the TIMESERIES clause generates series of timestamps as described in the following sections.

Minutes

Time series of minutes are rounded down to full minutes. For example, the following statement specifies a time span of 00:00:03 - 00:05:50 :

```
=> SELECT ts FROM (
   SELECT '2015-01-04 00:00:03'::TIMESTAMP AS tm
     UNION
   SELECT '2015-01-04 00:05:50'::TIMESTAMP AS tm
  ) t
  TIMESERIES ts AS '1 minute' OVER (ORDER BY tm);
```

Vertica rounds down the time series start and end times to full minutes, 00:00:00 and 00:05:00 , respectively:

```
        ts
---------------------
 2015-01-04 00:00:00
 2015-01-04 00:01:00
 2015-01-04 00:02:00
 2015-01-04 00:03:00
 2015-01-04 00:04:00
 2015-01-04 00:05:00
(6 rows)
```

Weeks

Because the baseline timestamp 2000-01-01 00:00:00 is a Saturday, all time series of weeks start on Saturday. Vertica rounds down the series start and end timestamps accordingly. For example, the following statement specifies a time span of 12/10/99 - 01/10/00:

```
=> SELECT ts FROM (
    SELECT '1999-12-10 00:00:00'::TIMESTAMP AS tm
      UNION
    SELECT '2000-01-10 23:59:59'::TIMESTAMP AS tm
    ) t
    TIMESERIES ts AS '1 week' OVER (ORDER BY tm);
```

The specified time span starts on Friday (12/10/99), so Vertica starts the time series on the preceding Saturday, 12/04/99. The time series ends on the last Saturday within the time span, 01/08/00:

```
        ts
---------------------
 1999-12-04 00:00:00
 1999-12-11 00:00:00
 1999-12-18 00:00:00
 1999-12-25 00:00:00
 2000-01-01 00:00:00
 2000-01-08 00:00:00
(6 rows)
```

Months

Time series of months are divided into equal 30-day intervals, relative to the baseline timestamp 2000-01-01 00:00:00 . For example, the following statement specifies a time span of 09/01/99 - 12/31/00:

```
=> SELECT ts FROM (
    SELECT '1999-09-01 00:00:00'::TIMESTAMP AS tm
      UNION
    SELECT '2000-12-31 23:59:59'::TIMESTAMP AS tm
  ) t
  TIMESERIES ts AS '1 month' OVER (ORDER BY tm);
```

Vertica generates a series of 30-day intervals, where each timestamp is rounded up or down, relative to the baseline timestamp:

```
         ts
---------------------
 1999-08-04 00:00:00
 1999-09-03 00:00:00
 1999-10-03 00:00:00
 1999-11-02 00:00:00
 1999-12-02 00:00:00
 2000-01-01 00:00:00
 2000-01-31 00:00:00
 2000-03-01 00:00:00
 2000-03-31 00:00:00
 2000-04-30 00:00:00
 2000-05-30 00:00:00
 2000-06-29 00:00:00
 2000-07-29 00:00:00
 2000-08-28 00:00:00
 2000-09-27 00:00:00
 2000-10-27 00:00:00
 2000-11-26 00:00:00
 2000-12-26 00:00:00
(18 rows)
```

Years

Time series of years are divided into equal 365-day intervals. If a time span overlaps leap years since or before the baseline timestamp 2000-01-01 00:00:00 , Vertica rounds the series timestamps accordingly.

For example, the following statement specifies a time span of 01/01/95 - 05/08/09, which overlaps four leap years, including the baseline timestamp:

```
=> SELECT ts FROM (
     SELECT '1995-01-01 00:00:00'::TIMESTAMP AS tm
       UNION
     SELECT '2009-05-08'::TIMESTAMP AS tm
   ) t timeseries ts AS '1 year' over (ORDER BY tm);
```

Vertica generates a series of timestamps that are rounded up or down, relative to the baseline timestamp:

```
         ts
---------------------
 1994-01-02 00:00:00
 1995-01-02 00:00:00
 1996-01-02 00:00:00
 1997-01-01 00:00:00
 1998-01-01 00:00:00
 1999-01-01 00:00:00
 2000-01-01 00:00:00
 2000-12-31 00:00:00
 2001-12-31 00:00:00
 2002-12-31 00:00:00
 2003-12-31 00:00:00
 2004-12-30 00:00:00
 2005-12-30 00:00:00
 2006-12-30 00:00:00
 2007-12-30 00:00:00
 2008-12-29 00:00:00
(16 rows)
```

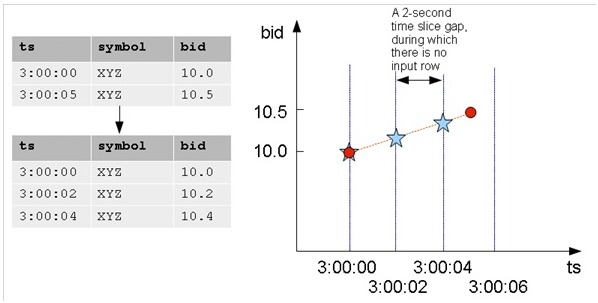
Linear interpolation

Instead of interpolating data points based on the last seen value (Constant interpolation), linear interpolation is where Vertica interpolates values in a linear slope based on the specified time slice.

The query that follows uses linear interpolation to place the input records in 2-second time slices and return the first bid value for each symbol/time slice combination (the value at the start of the time slice):

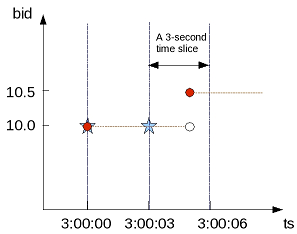
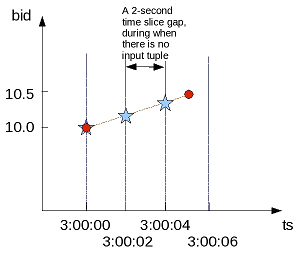
```
=> SELECT slice_time, TS_FIRST_VALUE(bid, 'LINEAR') bid FROM Tickstore
   TIMESERIES slice_time AS '2 seconds' OVER(PARTITION BY symbol ORDER BY ts);
    slice_time      | bid
--------------------+------
 2009-01-01 03:00:00 |   10
 2009-01-01 03:00:02 | 10.2
 2009-01-01 03:00:04 | 10.4
(3 rows)
```

The following figure illustrates the previous query results, showing the 2-second time gaps (3:00:02 and 3:00:04) in which no input record occurs. Note that the interpolated bid price of XYZ changes to 10.2 at 3:00:02 and 10.3 at 3:00:03 and 10.4 at 3:00:04, all of which fall between the two known data inputs (3:00:00 and 3:00:05). At 3:00:05, the value would change to 10.5.



> **Note**
> The known data points above are represented by a red dot, and the interpolated values are represented by blue stars.

The following is a side-by-side comparison of constant and linear interpolation schemes.

CONST interpolation	LINEAR interpolation
	

GFI examples

This topic illustrates some of the queries you can write using the constant and linear interpolation schemes.

Constant interpolation

The first query uses TS_FIRST_VALUE() and the TIMESERIES clause to place the input records in 3-second time slices and return the first bid value for each symbol/time slice combination (the value at the start of the time slice).

> **Note**
> The TIMESERIES clause requires an ORDER BY operation on the TIMESTAMP column.

```
=> SELECT slice_time, symbol, TS_FIRST_VALUE(bid) AS first_bid FROM TickStore
   TIMESERIES slice_time AS '3 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

Because the bid price of stock XYZ is 10.0 at 3:00:03, the first_bid value of the second time slice, which starts at 3:00:03 is till 10.0 (instead of 10.5) because the input value of 10.5 does not occur until 3:00:05. In this case, the interpolated value is inferred from the last value seen on stock XYZ for time 3:00:03:

```
    slice_time      | symbol | first_bid
--------------------+--------+-----------
 2009-01-01 03:00:00 | XYZ    |        10
 2009-01-01 03:00:03 | XYZ    |        10
(2 rows)
```

The next example places the input records in 2-second time slices to return the first bid value for each symbol/time slice combination:

```
=> SELECT slice_time, symbol, TS_FIRST_VALUE(bid) AS first_bid FROM TickStore
   TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

The result now contains three records in 2-second increments, all of which occur between the first input row at 03:00:00 and the second input row at 3:00:05. Note that the second and third output record correspond to a time slice where there is no input record:

```
    slice_time      | symbol | first_bid
--------------------+--------+-----------
 2009-01-01 03:00:00 | XYZ    |        10
2009-01-01 03:00:02 | XYZ    |        10
2009-01-01 03:00:04 | XYZ    |        10
(3 rows)
```

Using the same table schema, the next query uses TS_LAST_VALUE() , with the TIMESERIES clause to return the last values of each time slice (the values at the end of the time slices).

> **Note**
> Time series aggregate functions process the data that belongs to each time slice. One output row is produced per time slice or per partition per time slice if a partition expression is present.

```
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid) AS last_bid FROM TickStore
   TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

Notice that the last value output row is 10.5 because the value 10.5 at time 3:00:05 was the last point inside the 2-second time slice that started at 3:00:04:

```
    slice_time      | symbol | last_bid
--------------------+--------+----------
 2009-01-01 03:00:00 | XYZ    |       10
 2009-01-01 03:00:02 | XYZ    |       10
 2009-01-01 03:00:04 | XYZ    |     10.5
(3 rows)
```

Remember that because constant interpolation is the default, the same results are returned if you write the query using the CONST parameter as follows:

```
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid, 'CONST') AS last_bid FROM TickStore
   TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

Linear interpolation

Based on the same input records described in the constant interpolation examples, which specify 2-second time slices, the result of TS_LAST_VALUE with linear interpolation is as follows:

```
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid, 'linear') AS last_bid   FROM TickStore
   TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

In the results, no last_bid value is returned for the last row because the query specified TS_LAST_VALUE, and there is no data point after the 3:00:04 time slice to interpolate.

```
    slice_time      | symbol | last_bid
---------------------+--------+----------
 2009-01-01 03:00:00 | XYZ    |     10.2
 2009-01-01 03:00:02 | XYZ    |     10.4
 2009-01-01 03:00:04 | XYZ    |
(3 rows)
```

Using multiple time series aggregate functions

Multiple time series aggregate functions can exists in the same query. They share the same *gap-filling* policy as defined in the TIMESERIES clause; however, each time series aggregate function can specify its own interpolation policy. In the following example, there are two constant and one linear interpolation schemes, but all three functions use a three-second time slice:

```
=> SELECT slice_time, symbol,
      TS_FIRST_VALUE(bid, 'const') fv_c,
      TS_FIRST_VALUE(bid, 'linear') fv_l,
      TS_LAST_VALUE(bid, 'const') lv_c
   FROM TickStore
   TIMESERIES slice_time AS '3 seconds' OVER(PARTITION BY symbol ORDER BY ts);
```

In the following output, the original output is compared to output returned by multiple time series aggregate functions.

```
    ts     | symbol | bid
-----------+--------+------
 03:00:00 | XYZ    |   10
 03:00:05 | XYZ    | 10.5
(2 rows)
```

==>

```
    slice_time      | symbol | fv_c | fv_l | lv_c
---------------------+--------+------+------+------
 2009-01-01 03:00:00 | XYZ    |   10 |   10 |   10
 2009-01-01 03:00:03 | XYZ    |   10 | 10.3 | 10.5
(2 rows)
```

Using the analytic LAST_VALUE function

Here's an example using LAST_VALUE(), so you can see the difference between it and the GFI syntax.

```
=> SELECT *, LAST_VALUE(bid) OVER(PARTITION by symbol ORDER BY ts)
   AS "last bid" FROM TickStore;
```

There is no gap filling and interpolation to the output values.

```
        ts          | symbol | bid  | last bid
---------------------+--------+------+----------
 2009-01-01 03:00:00 | XYZ    |   10 |       10
 2009-01-01 03:00:05 | XYZ    | 10.5 |     10.5
(2 rows)
```

Using slice_time

In a TIMESERIES query, you cannot use the column slice_time in the WHERE clause because the WHERE clause is evaluated before the TIMESERIES clause, and the slice_time column is not generated until the TIMESERIES clause is evaluated. For example, Vertica does not support the following query:

```
=> SELECT symbol, slice_time, TS_FIRST_VALUE(bid IGNORE NULLS) AS fv
   FROM TickStore
   WHERE slice_time = '2009-01-01 03:00:00'
   TIMESERIES slice_time as '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
ERROR:  Time Series timestamp alias/Time Series Aggregate Functions not allowed in WHERE clause
```

Instead, you could write a subquery and put the predicate on slice_time in the outer query:

```
=> SELECT * FROM (
     SELECT symbol, slice_time,
       TS_FIRST_VALUE(bid IGNORE NULLS) AS fv
     FROM TickStore
     TIMESERIES slice_time AS '2 seconds'
     OVER (PARTITION BY symbol ORDER BY ts) ) sq
   WHERE slice_time = '2009-01-01 03:00:00';
 symbol |     slice_time      | fv
--------+---------------------+----
 XYZ    | 2009-01-01 03:00:00 | 10
(1 row)
```

Creating a dense time series

The TIMESERIES clause provides a convenient way to create a dense time series for use in an outer join with fact data. The results represent every time point, rather than just the time points for which data exists.

The examples that follow use the same TickStore schema described in Gap filling and interpolation (GFI) , along with the addition of a new inner table for the purpose of creating a join:

```
=> CREATE TABLE inner_table (
      ts TIMESTAMP,
      bid FLOAT
   );
=> CREATE PROJECTION inner_p (ts, bid) as SELECT * FROM inner_table
   ORDER BY ts, bid UNSEGMENTED ALL NODES;
=> INSERT INTO inner_table VALUES ('2009-01-01 03:00:02', 1);
=> INSERT INTO inner_table VALUES ('2009-01-01 03:00:04', 2);
=> COMMIT;
```

You can create a simple union between the start and end range of the timeframe of interest in order to return every time point. This example uses a 1-second time slice:

```
=> SELECT ts FROM (
     SELECT '2009-01-01 03:00:00'::TIMESTAMP AS time FROM TickStore
     UNION
     SELECT '2009-01-01 03:00:05'::TIMESTAMP FROM TickStore) t
   TIMESERIES ts AS '1 seconds' OVER(ORDER BY time);
         ts
---------------------
 2009-01-01 03:00:00
 2009-01-01 03:00:01
 2009-01-01 03:00:02
 2009-01-01 03:00:03
 2009-01-01 03:00:04
 2009-01-01 03:00:05
(6 rows)
```

The next query creates a union between the start and end range of the timeframe using 500-millisecond time slices:

```
=> SELECT ts FROM (
    SELECT '2009-01-01 03:00:00'::TIMESTAMP AS time
    FROM TickStore
    UNION
    SELECT '2009-01-01 03:00:05'::TIMESTAMP FROM TickStore) t
  TIMESERIES ts AS '500 milliseconds' OVER(ORDER BY time);
         ts
-----------------------
 2009-01-01 03:00:00
 2009-01-01 03:00:00.5
 2009-01-01 03:00:01
 2009-01-01 03:00:01.5
 2009-01-01 03:00:02
 2009-01-01 03:00:02.5
 2009-01-01 03:00:03
 2009-01-01 03:00:03.5
 2009-01-01 03:00:04
 2009-01-01 03:00:04.5
 2009-01-01 03:00:05
(11 rows)
```

The following query creates a union between the start- and end-range of the timeframe of interest using 1-second time slices:

```
=> SELECT * FROM (
    SELECT ts FROM (
      SELECT '2009-01-01 03:00:00'::timestamp AS time FROM TickStore
      UNION
      SELECT '2009-01-01 03:00:05'::timestamp FROM TickStore) t
      TIMESERIES ts AS '1 seconds' OVER(ORDER BY time) ) AS outer_table
  LEFT OUTER JOIN inner_table ON outer_table.ts = inner_table.ts;
```

The union returns a complete set of records from the left-joined table with the matched records in the right-joined table. Where the query found no match, it extends the right side column with null values:

```
        ts         |         ts          | bid
---------------------+---------------------+-----
 2009-01-01 03:00:00 |                     |
 2009-01-01 03:00:01 |                     |
 2009-01-01 03:00:02 | 2009-01-01 03:00:02 |   1
 2009-01-01 03:00:03 |                     |
 2009-01-01 03:00:04 | 2009-01-01 03:00:04 |   2
 2009-01-01 03:00:05 |                     |
(6 rows)
```

Null values in time series data

Null values are uncommon inputs for gap-filling and interpolation (GFI) computation. When null values exist, you can use time series aggregate (TSA) functions TS_FIRST_VALUE and TS_LAST_VALUE with IGNORE NULLS to affect output of the interpolated values. TSA functions are treated like their analytic counterparts FIRST_VALUE and LAST_VALUE : if the timestamp itself is null, Vertica filters out those rows before gap filling and interpolation occurs.

Constant interpolation with null values

Figure 1 illustrates a default (constant) interpolation result on four input rows where none of the inputs contains a NULL value:

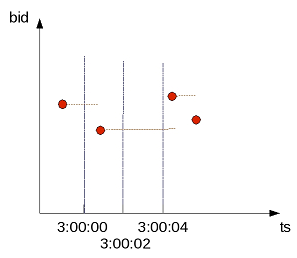
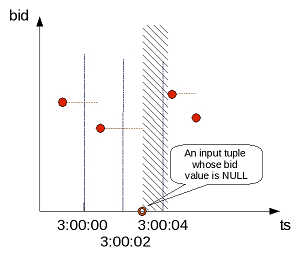


Figure 2 shows the same input rows with the addition of another input record whose bid value is NULL, and whose timestamp (ts) value is 3:00:03:



For constant interpolation, the bid value starting at 3:00:03 is null until the next non-null bid value appears in time. In Figure 2, the presence of the null row makes the interpolated bid value null in the time interval denoted by the shaded region. If TS_FIRST_VALUE(bid) is evaluated with constant interpolation on the time slice that begins at 3:00:02, its output is non-null. However, TS_FIRST_VALUE(bid) on the next time slice produces null. If the last value of the 3:00:02 time slice is null, the first value for the next time slice (3:00:04) is null. However, if you use a TSA function with IGNORE NULLS , then the value at 3:00:04 is the same value as it was at 3:00:02.

To illustrate, insert a new row into the TickStore table at 03:00:03 with a null bid value, Vertica outputs a row for the 03:00:02 record with a null value but no row for the 03:00:03 input:

```
=> INSERT INTO tickstore VALUES('2009-01-01 03:00:03', 'XYZ', NULL);
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid) AS last_bid FROM TickStore
-> TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
    slice_time      | symbol | last_bid
---------------------+--------+----------
 2009-01-01 03:00:00 | XYZ    |       10
 2009-01-01 03:00:02 | XYZ    |
 2009-01-01 03:00:04 | XYZ    |     10.5
(3 rows)
```

If you specify IGNORE NULLS, Vertica fills in the missing data point using a constant interpolation scheme. Here, the bid price at 03:00:02 is interpolated to the last known input record for bid, which was $10 at 03:00:00:

```
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid IGNORE NULLS) AS last_bid FROM TickStore
    TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
    slice_time      | symbol | last_bid
---------------------+--------+----------
 2009-01-01 03:00:00 | XYZ    |       10
 2009-01-01 03:00:02 | XYZ    |       10
 2009-01-01 03:00:04 | XYZ    |     10.5
(3 rows)
```

Now, if you insert a row where the timestamp column contains a null value, Vertica filters out that row before gap filling and interpolation occurred.

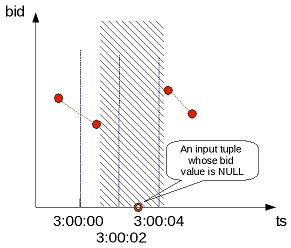
```
=> INSERT INTO tickstore VALUES(NULL, 'XYZ', 11.2);
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid) AS last_bid FROM TickStore
    TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

Notice there is no output for the 11.2 bid row:

```
    slice_time      | symbol | last_bid
---------------------+--------+----------
 2009-01-01 03:00:00 | XYZ    |       10
 2009-01-01 03:00:02 | XYZ    |
 2009-01-01 03:00:04 | XYZ    |     10.5
(3 rows)
```

Linear interpolation with null values

For linear interpolation, the interpolated bid value becomes null in the time interval, represented by the shaded region in Figure 3:



In the presence of an input null value at 3:00:03, Vertica cannot linearly interpolate the bid value around that time point.

Vertica takes the closest non null value on either side of the time slice and uses that value. For example, if you use a linear interpolation scheme and do not specify IGNORE NULLS , and your data has one real value and one null, the result is null. If the value on either side is null, the result is null. Therefore, to evaluate TS_FIRST_VALUE(bid) with linear interpolation on the time slice that begins at 3:00:02, its output is null. TS_FIRST_VALUE(bid) on the next time slice remains null.

```
=> SELECT slice_time, symbol, TS_FIRST_VALUE(bid, 'linear') AS fv_l FROM TickStore
    TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
    slice_time      | symbol | fv_l
---------------------+--------+------
 2009-01-01 03:00:00 | XYZ    |   10
 2009-01-01 03:00:02 | XYZ    |
 2009-01-01 03:00:04 | XYZ    |
(3 rows)
```

Data aggregation

You can use functions such as SUM and COUNT to aggregate the results of GROUP BY queries at one or more levels.

In this section

- Single-level aggregation
- Multi-level aggregation
- Aggregates and functions for multilevel grouping
- Aggregate expressions for GROUP BY
- Pre-aggregating data in projections

Single-level aggregation

The simplest GROUP BY queries aggregate data at a single level. For example, a table might contain the following information about family expenses:

- Category
- Amount spent on that category during the year
- Year

Table data might look like this:

```
=> SELECT * FROM expenses ORDER BY Category;
 Year |  Category  | Amount
------+------------+--------
2005  | Books      |  39.98
2007  | Books      |  29.99
2008  | Books      |  29.99
2006  | Electrical | 109.99
2005  | Electrical | 109.99
2007  | Electrical | 229.98
```

You can use aggregate functions to get the total expenses per category or per year:

```
=> SELECT SUM(Amount), Category FROM expenses GROUP BY Category;
 SUM     | Category
---------+------------
  99.96  | Books
 449.96  | Electrical
=> SELECT SUM(Amount), Year FROM expenses GROUP BY Year;
 SUM    | Year
--------+------
 149.97 | 2005
 109.99 | 2006
  29.99 | 2008
 259.97 | 2007
```

Multi-level aggregation

Over time, tables that are updated frequently can contain large amounts of data. Using the simple table shown earlier, suppose you want a multilevel query, like the number of expenses per category per year.

The following query uses the ROLLUP aggregation with the SUM function to calculate the total expenses by category and the overall expenses total. The NULL fields indicate subtotal values in the aggregation.

- When only the Year column is NULL , the subtotal is for all the Category values.
- When both the Year and Category columns are NULL , the subtotal is for all Amount values for both columns.

Using the ORDER BY clause orders the results by expense category, the year the expenses took place, and the GROUP BY level that the GROUPING_ID function creates:

```
=> SELECT Category, Year, SUM(Amount) FROM expenses
   GROUP BY ROLLUP(Category, Year) ORDER BY Category, Year, GROUPING_ID();
 Category   | Year |  SUM
------------+------+--------
 Books      | 2005 |  39.98
 Books      | 2007 |  29.99
 Books      | 2008 |  29.99
 Books      |      |  99.96
 Electrical | 2005 | 109.99
 Electrical | 2006 | 109.99
 Electrical | 2007 | 229.98
 Electrical |      | 449.96
            |      | 549.92
```

Similarly, the following query calculates the total sales by year and the overall sales total and then uses the ORDER BY clause to sort the results:

```
=> SELECT Category, Year, SUM(Amount) FROM expenses
   GROUP BY ROLLUP(Year, Category) ORDER BY 2, 1, GROUPING_ID();
  Category  | Year |  SUM
------------+------+--------
 Books      | 2005 |  39.98
 Electrical | 2005 | 109.99
            | 2005 | 149.97
 Electrical | 2006 | 109.99
            | 2006 | 109.99
 Books      | 2007 |  29.99
 Electrical | 2007 | 229.98
            | 2007 | 259.97
 Books      | 2008 |  29.99
            | 2008 |  29.99
            |      | 549.92
(11 rows)
```

You can use the CUBE aggregate to perform all possible groupings of the category and year expenses. The following query returns all possible groupings, ordered by grouping:

```
=> SELECT Category, Year, SUM(Amount) FROM expenses
   GROUP BY CUBE(Category, Year) ORDER BY 1, 2, GROUPING_ID();
 Category   | Year |  SUM
------------+------+--------
 Books      | 2005 |  39.98
 Books      | 2007 |  29.99
 Books      | 2008 |  29.99
 Books      |      |  99.96
 Electrical | 2005 | 109.99
 Electrical | 2006 | 109.99
 Electrical | 2007 | 229.98
 Electrical |      | 449.96
            | 2005 | 149.97
            | 2006 | 109.99
            | 2007 | 259.97
            | 2008 |  29.99
            |      | 549.92
```

The results include subtotals for each category and each year and a total ($549.92) for all transactions, regardless of year or category.

ROLLUP , CUBE , and GROUPING SETS generate NULL values in grouping columns to identify subtotals. If table data includes NULL values, differentiating these from NULL values in subtotals can sometimes be challenging.

In the preceding output, the NULL values in the Year column indicate that the row was grouped on the Category column, rather than on both columns. In this case, ROLLUP added the NULL value to indicate the subtotal row.

Aggregates and functions for multilevel grouping

Vertica provides several aggregates and functions that group the results of a GROUP BY query at multiple levels.

Aggregates for multilevel grouping

Use the following aggregates for multilevel grouping:

- ROLLUP automatically performs subtotal aggregations. ROLLUP performs one or more aggregations across multiple dimensions, at different levels.
- CUBE performs the aggregation for all permutations of the CUBE expression that you specify.
- GROUPING SETS let you specify which groupings of aggregations you need.

You can use CUBE or ROLLUP expressions inside GROUPING SETS expressions. Otherwise, you cannot nest multilevel aggregate expressions.

Grouping functions

You use one of the following three grouping functions with ROLLUP, CUBE, and GROUPING SETS:

- GROUP_ID returns one or more numbers, starting with zero (0), to uniquely identify duplicate sets.
- GROUPING_ID produces a unique ID for each grouping combination.
- GROUPING identifies for each grouping combination whether a column is a part of this grouping. This function also differentiates NULL values in the data from NULL grouping subtotals.

These functions are typically used with multilevel aggregates.

Aggregate expressions for GROUP BY

You can include CUBE and ROLLUP aggregates within a GROUPING SETS aggregate. Be aware that the CUBE and ROLLUP aggregates can result in a large amount of output. However, you can avoid large outputs by using GROUPING SETS to return only specified results.

```
...GROUP BY a,b,c,d,ROLLUP(a,b)...
...GROUP BY a,b,c,d,CUBE((a,b),c,d)...
```

You cannot include any aggregates in a CUBE or ROLLUP aggregate expression.

You can append multiple GROUPING SETS, CUBE, or ROLLUP aggregates in the same query.

```
...GROUP BY a,b,c,d,CUBE(a,b),ROLLUP (c,d)...
...GROUP BY a,b,c,d,GROUPING SETS ((a,d),(b,c),CUBE(a,b));...
...GROUP BY a,b,c,d,GROUPING SETS ((a,d),(b,c),(a,b),(a),(b),())...
```

Pre-aggregating data in projections

Queries that use aggregate functions such as SUM and COUNT can perform more efficiently when they use projections that already contain the aggregated data. This improved efficiency is especially true for queries on large quantities of data.

For example, a power grid company reads 30 million smart meters that provide data at five-minute intervals. The company records each reading in a database table. Over a given year, three trillion records are added to this table.

The power grid company can analyze these records with queries that include aggregate functions to perform the following tasks:

- Establish usage patterns.
- Detect fraud.
- Measure correlation to external events such as weather patterns or pricing changes.

To optimize query response time, you can create an aggregate projection, which stores the data is stored after it is aggregated.

Aggregate projections

Vertica provides several types of projections for storing data that is returned from aggregate functions or expressions:

- Live aggregate projection: Projection that contains columns with values that are aggregated from columns in its anchor table. You can also define live aggregate projections that include user-defined transform functions .
- Top-K projection : Type of live aggregate projection that returns the top *k* rows from a partition of selected rows. Create a Top-K projection that satisfies the criteria for a Top-K query.
- Projection that pre-aggregates UDTF results : Live aggregate projection that invokes user-defined transform functions (UDTFs). To minimize overhead when you query those projections of this type, Vertica processes the UDTF functions in the background and stores their results on disk.
- Projection that contains expressions : Projection with columns whose values are calculated from anchor table columns.

Recommended use

- Aggregate projections are most useful for queries against large sets of data.
- For optimal query performance, the size of LAP projections should be a small subset of the anchor table—ideally, between 1 and 10 percent of the anchor table, or smaller, if possible.

Restrictions

- MERGE operations must be optimized if they are performed on target tables that have live aggregate projections.
- You cannot update or delete data in temporary tables with live aggregate projections.

Requirements

In the event of manual recovery from an unclean database shutdown, live aggregate projections might require some time to refresh.

In this section

- Live aggregate projections
- Top-k projections
- Pre-aggregating UDTF results
- Aggregating data through expressions
- Aggregation information in system tables

Live aggregate projections

A live aggregate projection contains columns with values that are aggregated from columns in its anchor table. When you load data into the table, Vertica aggregates the data before loading it into the live aggregate projection. On subsequent loads—for example, through INSERT or COPY —Vertica recalculates aggregations with the new data and updates the projection.

In this section

- Functions supported for live aggregate projections
- Creating live aggregate projections
- Live aggregate projection example

Functions supported for live aggregate projections

Vertica can aggregate results in live aggregate projections from the following aggregate functions:

- SUM
- MAX
- MIN
- COUNT

Aggregate functions with DISTINCT

Live aggregate projections can support queries that include aggregate functions qualified with the keyword DISTINCT . The following requirements apply:

- The aggregated expression must evaluate to a non-constant.
- The projection's GROUP BY clause must specify the aggregated expression.

For example, the following query uses SUM(DISTINCT) to calculate the total of all unique salaries in a given region:

```
SELECT customer_region, SUM(DISTINCT annual_income)::INT
  FROM customer_dimension GROUP BY customer_region;
```

This query can use the following live aggregate projection, which specifies the aggregated column (annual_income) in its GROUP BY clause:

```
CREATE PROJECTION public.TotalRegionalIncome
(
 customer_region,
 annual_income,
 Count
)
AS
 SELECT customer_dimension.customer_region,
        customer_dimension.annual_income,
        count(*) AS Count
 FROM public.customer_dimension
 GROUP BY customer_dimension.customer_region,
          customer_dimension.annual_income
;
```

> **Note**
> This projection includes the aggregate function COUNT , which here serves no logical objective; it is included only because live aggregate projections require at least one aggregate function.

Creating live aggregate projections

You define a live aggregate projection with the following syntax:

```
=> CREATE PROJECTION proj-name AS
    SELECT select-expression FROM table
    GROUP BY group-expression;
```

For full syntax options, see CREATE PROJECTION .

For example:

```
=> CREATE PROJECTION clicks_agg AS
  SELECT page_id, click_time::DATE click_date, COUNT(*) num_clicks FROM clicks
  GROUP BY page_id, click_time::DATE KSAFE 1;
```

For an extended discussion, see Live aggregate projection example .

Requirements

The following requirements apply to live aggregate projections:

- The projection cannot be unsegmented. Unless the CREATE PROJECTION statement or its anchor table DDL specifies otherwise , all projections are segmented by default.

- SELECT and GROUP BY columns must be in the same order. GROUP BY expressions must be at the beginning of the SELECT list.

Restrictions

The following restrictions apply to live aggregate projections:

- MERGE operations must be optimized if they are performed on target tables that have live aggregate projections.
- Live aggregate projections can reference only one table.
- Live aggregate projections cannot use row access policies.
- Vertica does not regard live aggregate projections as superprojections, even those that include all table columns.
- You cannot modify the anchor table metadata of columns that are included in live aggregate projections—for example, a column's data type or default value. You also cannot drop these columns. To make these changes, first drop all live aggregate and Top-K projections that are associated with the table.

> **Note**
> One exception applies: You can set and drop NOT NULL on columns that are included in a live aggregate projection.

Live aggregate projection example

This example shows how you can track user clicks on a given web page using the following clicks table:

```
=> CREATE TABLE clicks(
  user_id INTEGER,
  page_id INTEGER,
  click_time TIMESTAMP NOT NULL);
```

You can aggregate user-specific activity with the following query:

```
=> SELECT page_id, click_time::DATE click_date, COUNT(*) num_clicks FROM clicks
  WHERE click_time::DATE = '2015-04-30'
  GROUP BY page_id, click_time::DATE ORDER BY num_clicks DESC;
```

To facilitate performance of this query, create a live aggregate projection that counts the number of clicks per user:

```
=> CREATE PROJECTION clicks_agg AS
  SELECT page_id, click_time::DATE click_date, COUNT(*) num_clicks FROM clicks
  GROUP BY page_id, click_time::DATE KSAFE 1;
```

When you query the clicks table on user clicks, Vertica typically directs the query to the live aggregate projection clicks_agg . As additional data is loaded into clicks , Vertica pre-aggregates the new data and updates clicks_agg , so queries always return with the latest data.

For example:

```
=> SELECT page_id, click_time::DATE click_date, COUNT(*) num_clicks FROM clicks
   WHERE click_time::DATE = '2015-04-30' GROUP BY page_id, click_time::DATE
   ORDER BY num_clicks DESC;
 page_id | click_date | num_clicks
---------+------------+------------
   2002 | 2015-04-30 |         10
   3003 | 2015-04-30 |          3
   2003 | 2015-04-30 |          1
   2035 | 2015-04-30 |          1
  12034 | 2015-04-30 |          1
(5 rows)
```

Top-k projections

A Top-K query returns the top *k* rows from partitions of selected rows. Top-K projections can significantly improve performance of Top-K queries. For example, you can define a table that stores gas meter readings with three columns: gas meter ID, time of meter reading, and the read value:

```
=> CREATE TABLE readings (
   meter_id INT,
   reading_date TIMESTAMP,
   reading_value FLOAT);
```

Given this table, the following Top-K query returns the five most recent meter readings for a given meter:

```
SELECT meter_id, reading_date, reading_value FROM readings
   LIMIT 5 OVER (PARTITION BY meter_id ORDER BY reading_date DESC);
```

To improve the performance of this query, you can create a Top-K projection, which is a special type of live aggregate projection:

```
=> CREATE PROJECTION readings_topk (meter_id, recent_date, recent_value)
   AS SELECT meter_id, reading_date, reading_value FROM readings
   LIMIT 5 OVER (PARTITION BY meter_id ORDER BY reading_date DESC);
```

After you create this Top-K projection and load its data (through START_REFRESH or REFRESH), Vertica typically redirects the query to the projection and returns with the pre-aggregated data.

In this section

- Creating top-k projections
- Top-k projection examples

Creating top-k projections

You define a Top-K projection with the following syntax:

```
CREATE PROJECTION proj-name [(proj-column-spec)]
   AS SELECT select-expression FROM table
   LIMIT num-rows OVER (PARTITION BY expression ORDER BY column-expr);
```

For full syntax options, see CREATE PROJECTION .

For example:

```
=> CREATE PROJECTION readings_topk (meter_id, recent_date, recent_value)
   AS SELECT meter_id, reading_date, reading_value FROM readings
   LIMIT 5 OVER (PARTITION BY meter_id ORDER BY reading_date DESC);
```

For an extended discussion, see Top-k projection examples .

Requirements

The following requirements apply to Top-K projections:

- The projection cannot be unsegmented.
- The window partition clause must use PARTITION BY .
- Columns in PARTITION BY and ORDER BY clauses must be the first columns specified in the SELECT list.
- You must use the LIMIT option to create a Top-K projection, instead of subqueries. For example, the following SELECT statements are equivalent:

  ```
  => SELECT symbol, trade_time last_trade, price last_price FROM (
     SELECT symbol, trade_time, price, ROW_NUMBER()
     OVER(PARTITION BY symbol ORDER BY trade_time DESC) rn FROM trades) trds WHERE rn <=1;

  => SELECT symbol, trade_time last_trade, price last_price FROM trades
     LIMIT 1 OVER(PARTITION BY symbol ORDER BY trade_time DESC);
  ```

  Both return the same results:

```
   symbol     |      last_trade       | last_price
------------------+-----------------------+------------
 AAPL             | 2011-11-10 10:10:20.5 |   108.4000
 HPQ              | 2012-10-10 10:10:10.4 |    42.0500
(2 rows)
```

A Top-K projection that pre-aggregates data for use by both queries must include the LIMIT option:

```
=> CREATE PROJECTION trades_topk AS
   SELECT symbol, trade_time last_trade, price last_price FROM trades
   LIMIT 1 OVER(PARTITION BY symbol ORDER BY trade_time DESC);
```

Restrictions

The following restrictions apply to Top-K projections:

- Top-K projections can reference only one table.
- Vertica does not regard Top-K projections as superprojections, even those that include all table columns.
- You cannot modify the anchor table metadata of columns that are included in Top-K projections—for example, a column's data type or default value. You also cannot drop these columns. To make these changes, first drop all live aggregate and Top-K projections that are associated with the table.

> **Note**
> One exception applies: You can set and drop NOT NULL on columns that are included in a live aggregate projection.

Top-k projection examples

The following examples show how to query a table with two Top-K projections for the most-recent trade and last trade of the day for each stock symbol.

1. Create a table that contains information about individual stock trades:
   - Stock symbol
   - Timestamp
   - Price per share
   - Number of shares

```
=> CREATE TABLE trades(
   symbol CHAR(16) NOT NULL,
   trade_time TIMESTAMP NOT NULL,
   price NUMERIC(12,4),
   volume INT )
   PARTITION BY (EXTRACT(year from trade_time) * 100 +
   EXTRACT(month from trade_time));
```

2. Load data into the table:

```
INSERT INTO trades VALUES('AAPL','2010-10-10 10:10:10'::TIMESTAMP,100.00,100);
INSERT INTO trades VALUES('AAPL','2010-10-10 10:10:10.3'::TIMESTAMP,101.00,100);
INSERT INTO trades VALUES ('AAPL','2011-10-10 10:10:10.5'::TIMESTAMP,106.1,1000);
INSERT INTO trades VALUES ('AAPL','2011-10-10 10:10:10.2'::TIMESTAMP,105.2,500);
INSERT INTO trades VALUES ('HPQ','2012-10-10 10:10:10.2'::TIMESTAMP,42.01,400);
INSERT INTO trades VALUES ('HPQ','2012-10-10 10:10:10.3'::TIMESTAMP,42.02,1000);
INSERT INTO trades VALUES ('HPQ','2012-10-10 10:10:10.4'::TIMESTAMP,42.05,100);
COMMIT;
```

3. Create two Top-K projections that obtain the following information from the trades table:
   - Return the most recent trades for each stock symbol .
   - Return the last trade on each trading day .

   For each stock symbol, return the most recent trade.

```
=> CREATE PROJECTION trades_topk_a AS SELECT symbol, trade_time last_trade, price last_price
     FROM trades LIMIT 1 OVER(PARTITION BY symbol ORDER BY trade_time DESC);

=> SELECT symbol, trade_time last_trade, price last_price FROM trades
  LIMIT 1 OVER(PARTITION BY symbol ORDER BY trade_time DESC);

    symbol     |      last_trade       | last_price
---------------+-----------------------+------------
 HPQ           | 2012-10-10 10:10:10.4 |    42.0500
 AAPL          | 2011-10-10 10:10:10.5 |   106.1000
(2 rows)
```

For each stock symbol, return the last trade on each trading day.

```
=> CREATE PROJECTION trades_topk_b
   AS SELECT symbol, trade_time::DATE trade_date, trade_time, price close_price, volume
   FROM trades LIMIT 1 OVER(PARTITION BY symbol, trade_time::DATE ORDER BY trade_time DESC);

=> SELECT symbol, trade_time::DATE trade_date, trade_time, price close_price, volume
   FROM trades LIMIT 1 OVER(PARTITION BY symbol, trade_time::DATE ORDER BY trade_time DESC);

    symbol     | trade_date |      trade_time       | close_price | volume
---------------+------------+-----------------------+-------------+--------
 HPQ           | 2012-10-10 | 2012-10-10 10:10:10.4 |     42.0500 |    100
 AAPL          | 2011-10-10 | 2011-10-10 10:10:10.5 |    106.1000 |   1000
 AAPL          | 2010-10-10 | 2010-10-10 10:10:10.3 |    101.0000 |    100
(3 rows)
```

In each scenario, Vertica redirects queries on the trades table to the appropriate Top-K projection and returns the aggregated data from them. As additional data is loaded into this table, Vertica pre-aggregates the new data and updates the Top-K projections, so queries always return with the latest data.

Pre-aggregating UDTF results

CREATE PROJECTION can define live aggregate projections that invoke user-defined transform functions (UDTFs). To minimize overhead when you query those projections, Vertica processes these functions in the background and stores their results on disk.

Important
Currently, live aggregate projections can only reference UDTFs that are developed in C++.

Defining projections with UDTFs

The projection definition characterizes UDTFs in one of two ways:

- Identifies the UDTF as a *pre-pass UDTF* , which transforms newly loaded data before it is stored in the projection ROS containers.
- Identifies the UDTF as a *batch UDTF* , which aggregates and stores projection data.

The projection definition identifies a UDTF as a pre-pass UDTF or batch UDTF in its window partition clause , through the keywords PREPASS or BATCH . A projection can specify one pre-pass or batch UDTF or include both (see UDTF Specification Options).

In all cases, the projection is implicitly segmented and ordered on the PARTITION BY columns.

UDTF specification options

Projections can invoke batch and pre-pass UDTFs singly or in combination.

Single pre-pass UDTF

Vertica invokes the pre-pass UDTF when you load data into the projection's anchor table—for example through COPY or INSERT statements. A pre-pass UDTF transforms the new data and then stores the transformed data in the projection's ROS containers.

Use the following syntax:

```
CREATE PROJECTION [ IF NOT EXISTS ] [[database.]schema.]projection
[ (
  { projection-column | grouped-clause
  [ ENCODING encoding-type ]
  [ ACCESSRANK integer ] }[,...]
) ]

AS SELECT { table-column | expr-with-table-columns }[,...], prepass-udtf(prepass-args)
   OVER (PARTITION PREPASS BY partition-column-expr[,...])
   [ AS (prepass-output-columns) ] FROM table [[AS] alias]
```

Single batch UDTF

When invoked singly, a batch UDTF transforms and aggregates projection data on mergeout, data load, and query operations. The UDTF stores aggregated results in the projection's ROS containers. Aggregation is cumulative across mergeout and load operations, and is completed (if necessary) on query execution.

Use the following syntax:

```
CREATE PROJECTION [ IF NOT EXISTS ] [[database.]schema.]projection
[ (
  { projection-column | grouped-clause
  [ ENCODING encoding-type ]
  [ ACCESSRANK integer ]  }[,...]
) ]
AS SELECT { table-column | expr-with-table-columns }[,...], batch-udtf(batch-args)
  OVER (PARTITION BATCH BY partition-column-expr[,...])
  [ AS (batch-output-columns) FROM table [ [AS] alias ]
```

Combined pre-pass and batch UDTFs

You can define a projection with a subquery that invokes a pre-pass UDTF. The pre-pass UDTF returns transformed data to the outer batch query. The batch UDTF then iteratively aggregates results across mergeout operations. It completes aggregation (if necessary) on query execution.

Use the following syntax:

```
CREATE PROJECTION [ IF NOT EXISTS ] [[database.]schema.]projection
[ (
  { projection-column | grouped-clause
  [ ENCODING encoding-type ]
  [ ACCESSRANK integer ] }[,...]
) ]
AS SELECT { table-column | expr-with-table-columns }[,...], batch-udtf(batch-args)
  OVER (PARTITION BATCH BY partition-column-expr[,...]) [ AS (batch-output-columns) ] FROM (
    SELECT { table-column | expr-with-table-columns }[,...], prepass-udtf (prepass-args)
    OVER (PARTITION PREPASS BY partition-column-expr[,...]) [ AS (prepass-output-columns) ] FROM table ) sq-ref
```

Important

The outer batch UDTF arguments *batch-args* must exactly match the output columns returned by the pre-pass UDTF, in name and order.

Examples

Single pre-pass UDTF

The following example shows how to use the UDTF text_index , which extracts from a text document strings that occur more than once.

The following projection specifies to invoke text_index as a pre-pass UDTF:

```
=> CREATE TABLE documents ( doc_id INT PRIMARY KEY, text VARCHAR(140));

=> CREATE PROJECTION index_proj (doc_id, text)
    AS SELECT doc_id, text_index(doc_id, text)
    OVER (PARTITION PREPASS BY doc_id) FROM documents;
```

The UDTF is invoked whenever data is loaded into the anchor table documents . text_index transforms the newly loaded data, and Vertica stores the transformed data in the live aggregate projection ROS containers.

So, if you load the following data into documents :

```
=> INSERT INTO documents VALUES
(100, 'A SQL Query walks into a bar. In one corner of the bar are two tables.
 The Query walks up to the tables and asks - Mind if I join you?');
 OUTPUT
--------
      1
(1 row)
```

text_index transforms the newly loaded data and stores it in the projection ROS containers. When you query the projection, it returns with the following results:

```
doc_id | frequency |     term
-------+-----------+--------------
100    | 2         | bar
100    | 2         | Query
100    | 2         | tables
100    | 2         | the
100    | 2         | walks
```

Combined Pre-Pass and Batch UDTFs

The following projection specifies pre-pass and batch UDTFs stv_intersect and aggregate_classified_points , respectively:

```
CREATE TABLE points( point_id INTEGER, point_type VARCHAR(10), coordinates GEOMETRY(100));

CREATE PROJECTION aggregated_proj
  AS SELECT point_type, aggregate_classified_points( sq.point_id, sq.polygon_id)
  OVER (PARTITION BATCH BY point_type) AS some_alias
  FROM
    (SELECT point_type, stv_intersect(
       point_id, coordinates USING PARAMETERS index='polygons' )
     OVER (PARTITION PREPASS BY point_type) AS (point_id, polygon_id) FROM points) sq;
```

The pre-pass query UDTF stv_intersect returns its results (a set of point and matching polygon IDs) to the outer batch query. The outer batch query then invokes the UDTF aggregate_classified_points . Vertica aggregates the result set that is returned by aggregate_classified_points whenever a mergeout operation consolidates projection data. Final aggregation (if necessary) occurs when the projection is queried.

The batch UDTF arguments must exactly match the output columns returned by the pre-pass UDTF stv_intersect , in name and order. In this example, the pre-pass subquery explicitly names the pre-pass UDTF output columns point_id and polygon_id . Accordingly, the batch UDTF arguments match them in name and order: sq.point_id and sq.polygon_id .

Aggregating data through expressions

You can create projections where one or more columns are defined by expressions. An expression can reference one or more anchor table columns. For example, the following table contains two integer columns, a and b :

```
=> CREATE TABLE values (a INT, b INT);
```

You can create a projection with an expression that calculates the value of column c as the product of a and b :

```
=> CREATE PROJECTION values_product (a, b, c)
  AS SELECT a, b, a*b FROM values SEGMENTED BY HASH(a) ALL NODES KSAFE;
```

When you load data into this projection, Vertica resolves the expression a*b in column c. You can then query the projection instead of the anchor table. Vertica returns the pre-calculated data and avoids the overhead otherwise incurred by resource-intensive computations.

Using expressions in projections also lets you sort or segment data on the calculated results of an expression instead of sorting on single column values.

> **Note**
> If a projection with expressions also includes aggregate functions such as SUM or COUNT , Vertica treats it like a live aggregate projection .

Support for user-defined scalar functions

Important
Currently, support for pre-aggregating UDSF results is limited to C++.

Vertica treats user-defined scalar functions (UDSFs) like other expressions. On each load operation, the UDSF is invoked and returns its results. Vertica stores these results on disk, and returns them when you query the projection directly.

In the following example, the projection points_p1 specifies the UDSF zorder , which is invoked whenever data is loaded in the anchor table points . When data is loaded into the projection, Vertica invokes this function and stores its results for fast access by future queries.

```
=> CREATE TABLE points(point_id INTEGER, lat NUMERIC(12,9), long NUMERIC(12,9));

=> CREATE PROJECTION points_p1
   AS SELECT point_id, lat, long, zorder(lat, long) zorder FROM points
   ORDER BY zorder(lat, long) SEGMENTED BY hash(point_id) ALL NODES;
```

Requirements

- Any ORDER BY expression must be in the SELECT list.
- All projection columns must be named.

Restrictions

- MERGE operations must be optimized if they are performed on target tables that have live aggregate projections.
- Unlike live aggregate projections, Vertica does not redirect queries with expressions to an equivalent existing projection.
- Projection expressions must be immutable—that is, they must always return the same result. For example, a projection cannot include expressions that use TO CHAR (depends on locale) or RANDOM (returns different value at each invocation).
- Projection expressions cannot include Vertica meta-functions such as ADVANCE_EPOCH , ANALYZE_STATISTICS , EXPORT_TABLES , or START_REFRESH .

In this section

- Querying data through expressions example

Querying data through expressions example

The following example uses a table that contains two integer columns, a and b :

```
=> CREATE TABLE values (a INT, b INT);
```

You can create a projection with an expression that calculates the value of column c as the product of a and b :

```
=> CREATE PROJECTION values_product (a, b, c)
   AS SELECT a, b, a*b FROM values SEGMENTED BY HASH(a) ALL NODES KSAFE;
=> COPY values FROM STDIN DELIMITER ',' DIRECT;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 3,11
>> 3,55
>> 8,9
>> 8,23
>> 16,41
>> 22,111
>> \.
=>
```

To query this projection, use the name that Vertica assigns to it or to its buddy projections. For example, the following queries target different instances of the projection defined earlier, and return the same results:

```
=> SELECT * FROM values_product_b0;
=> SELECT * FROM values_product_b1;
```

The following example queries the anchor table:

```
=> SELECT * FROM values;
 a  |  b
----+-----
 3 |  11
 3 |  55
 8 |   9
 8 |  23
16 |  41
22 | 111
```

Given the projection created earlier, querying that projection returns the following values:

```
VMart=> SELECT * FROM values_product_b0;
 a  |  b  | product
----+-----+---------
 3 |  11 |      33
 3 |  55 |     165
 8 |   9 |      72
 8 |  23 |     184
16 |  41 |     656
22 | 111 |    2442
```

Aggregation information in system tables

You can query the following system table fields for information about live aggregate projections, Top-K projections, and projections with expressions:

Table	Fields
TABLES	TABLE_HAS_AGGREGATE_PROJECTION
PROJECTIONS	AGGREGATE_TYPE
	HAS_EXPRESSIONS
	AGGREGATE_TYPE
PROJECTION_COLUMNS	COLUMN_EXPRESSION
	IS_AGGREGATE

	IS_EXPRESSION
	ORDER_BY_POSITION
	ORDER_BY_TYPE
	PARTITION_BY_POSITION

Flex tables

Flex tables are a different kind of database table designed for loading and querying unstructured data, also called *semi-structured* data. Flex tables allow you to load data with different or evolving schemas into a single table. You can then explore this data and materialize real columns from it.

If you know some of the columns in the data, you can define them. A flex table that also has real columns is called a hybrid table. Both flex and hybrid tables are fully supported tables, stored as projections and with the same K-safety as your database.

After you create a flex table, you can quickly load data without specifying its schema. This allows you to load arbitrary JSON data, log files, and other semi-structured data and immediately begin querying it.

Creating flex tables is similar to creating other tables, except column definitions are optional. When you create flex tables, with or without column definitions, Vertica implicitly adds a real column to your table, called __raw__ . This column stores loaded data. The __raw__ column is a LONG VARBINARY column with a NOT NULL constraint. It contains the documented limits for its data type (see Long data types). The __raw__ column's default maximum width is 130,000 bytes (with an absolute maximum of 32,000,000 bytes). You can change the default width with the FlexTableRawSize configuration parameter.

If you create a flex table without other column definitions, the table includes a second default column, __identity__ , declared as an auto-incrementing IDENTITY (1,1) column. When no other columns are defined, flex tables use the __identity__ column for segmentation and sort order.

Loading data into a flex table encodes the record into a VMap type and populates the __raw__ column. The VMap is a standard dictionary type, pairing keys with string values as virtual columns.

Flex table terms

Flex tables have the following special vocabulary:

- VMap: An internal map data format.
- Virtual Columns: Key-value pairs contained in a flex table __raw__ column.
- Real Columns: Explicit columns defined in addition to the flex (VMap) column.
- Promoted Columns: Virtual columns that have been materialized to real columns.
- Map Keys: Virtual column names within VMap data.

In this section

- Getting started
- Understanding flex tables
- Creating flex tables
- Bulk loading data into flex tables
- Inserting data into flex tables
- Using flex tables for IDOL data
- Using flex table parsers
- Computing flex table keys
- Materializing flex tables
- Updating flex table views
- Querying flex tables
- Querying nested data
- Querying flex views
- Listing flex tables

Getting started

The following tutorial demonstrates the basics of creating, exploring, and using flex tables.

These examples use a data file named mountains.json with the following contents:

```
{"name": "Everest", "type":"mountain", "height":29029, "hike_safety": 34.1}
{"name": "Mt St Helens", "type":"volcano", "height":29029, "hike_safety": 15.4}
{"name": "Denali", "type":"mountain", "height":17000, "hike_safety": 12.2}
{"name": "Kilimanjaro", "type":"mountain", "height":14000 }
{"name": "Mt Washington", "type":"mountain", "hike_safety": 50.6}
```

Create a flex table

Create a flex table and load the sample data as follows:

```
=> CREATE FLEX TABLE mountains();

=> COPY mountains FROM '/home/dbadmin/data/mountains.json'
   PARSER FJSONPARSER();
 Rows Loaded
-------------
           5
(1 row)
```

You can now query this table, just as if you had created the table with explicit columns:

```
=> SELECT name, type, height FROM mountains;
     name      |   type   | height
---------------+----------+--------
 Everest       | mountain | 29029
 Mt St Helens  | volcano  | 29029
 Denali        | mountain | 17000
 Kilimanjaro   | mountain | 14000
 Mt Washington | mountain |
(5 rows)
```

The __raw__ column

Data loaded into a flex table is added to a special column named __raw__ , which holds a map of all the keys and values. The following example shows this column, with carriage returns added for readability:

```
=> \x
Expanded display is on.
=> SELECT * FROM mountains;
[ RECORD 1 ]+-------------------------------------------------------------------------------
__identity__ | 1
__raw__      | \001\000\000\000,\000\000\000\004\000\000\000\024\000\000\000\031\000\000\000\
035\000\000\000$\000\000\0002902934.1Everestmountain\004\000\000\000\024\000\000\000\032\000\
000\000%\000\000\000)\000\000\000heighthike_safetynametype
[ RECORD 2 ]+-------------------------------------------------------------------------------
__identity__ | 2
__raw__      | \001\000\000\0000\000\000\000\004\000\000\000\024\000\000\000\031\000\000\000\
035\000\000\000)\000\000\0002902915.4Mt St Helensvolcano\004\000\000\000\024\000\000\000\032\000\
000\000%\000\000\000)\000\000\000heighthike_safetynametype
[ RECORD 3 ]+-------------------------------------------------------------------------------
__identity__ | 3
__raw__      | \001\000\000\000+\000\000\000\004\000\000\000\024\000\000\000\031\000\000\000\
035\000\000\000#\000\000\0001700012.2Denalimountain\004\000\000\000\024\000\000\000\032\000\000
\000%\000\000\000)\000\000\000heighthike_safetynametype
[ RECORD 4 ]+-------------------------------------------------------------------------------
__identity__ | 4
__raw__      | \001\000\000\000(\000\000\000\003\000\000\000\020\000\000\000\025\000\000\000\
000\000\00014000Kilimanjaromountain\003\000\000\000\020\000\000\000\026\000\000\000\032\000\
000\000heightnametype
[ RECORD 5 ]+-------------------------------------------------------------------------------
__identity__ | 5
__raw__      | \001\000\000\000)\000\000\000\003\000\000\000\020\000\000\000\024\000\000\000\
000\000\00050.6Mt Washingtonmountain\003\000\000\000\020\000\000\000\033\000\000\000\037\000\
000\000hike_safetynametype
```

As this example demonstrates, querying __raw__ directly is not generally helpful. Instead, you can use functions to extract values from it. Use the MAPTOSTRING function to see this column in a more readable JSON format:

```
=> SELECT MAPTOSTRING(__raw__) FROM mountains;
                                    MAPTOSTRING
-------------------------------------------------------------------------------------------
 {
   "height" : "29029",
   "hike_safety" : "34.1",
   "name" : "Everest",
   "type" : "mountain"
}

 {
   "height" : "29029",
   "hike_safety" : "15.4",
   "name" : "Mt St Helens",
   "type" : "volcano"
}

 {
   "height" : "17000",
   "hike_safety" : "12.2",
   "name" : "Denali",
   "type" : "mountain"
}

 {
   "height" : "14000",
   "name" : "Kilimanjaro",
   "type" : "mountain"
}

 {
   "hike_safety" : "50.6",
   "name" : "Mt Washington",
   "type" : "mountain"
}
```

You can use the COMPUTE_FLEXTABLE_KEYS function to extract information about the keys in the __raw__ column into a new table. The new table shows the key names, frequencies, and inferred data types:

```
=> SELECT COMPUTE_FLEXTABLE_KEYS('mountains');
    COMPUTE_FLEXTABLE_KEYS
---------------------------------------------------
Please see public.mountains_keys for updated keys
(1 row)

=> SELECT * FROM public.mountains_keys;
  key_name   | frequency | data_type_guess
-------------+-----------+-----------------
 height      |         4 | Integer
 hike_safety |         4 | Numeric(6,2)
 name        |         5 | Varchar(26)
 type        |         5 | Varchar(20)
(4 rows)
```

Vertica generates this table automatically when you create a flex table.

Review data types

Vertica infers data types from the raw data. You can adjust these types, either to be more specific or more general. The example that follows changes the NUMERIC value to FLOAT. Note also that VARCHAR sizes are based on the strings in the data and might not be large enough for future data. You should review and adjust data types before loading more data into a new flex table.

When you create a flex table, you can create a corresponding view. If you make changes in the data types in the keys table, regenerate the view. The following example demonstrates this process.

First, create the view:

```
=> SELECT BUILD_FLEXTABLE_VIEW('mountains');
                BUILD_FLEXTABLE_VIEW
-----------------------------------------------------------
 The view public.mountains_view is ready for querying
(1 row)

=> SELECT * FROM public.mountains_view;
     name      |   type   | height | hike_safety
---------------+----------+--------+-------------
 Everest       | mountain |  29029 |       34.10
 Mt St Helens  | volcano  |  29029 |       15.40
 Denali        | mountain |  17000 |       12.20
 Kilimanjaro   | mountain |  14000 |
 Mt Washington | mountain |        |       50.60
(5 rows)
```

Query the VIEW_COLUMNS system table to review the columns and data types:

```
=> SELECT column_name, data_type FROM VIEW_COLUMNS
   WHERE table_name = 'mountains_view';
 column_name |  data_type
-------------+-------------
 name        | varchar(26)
 type        | varchar(20)
 height      | int
 hike_safety | numeric(6,2)
(4 rows)
```

The view has the same data types as the keys table. Now update one of these data types in the keys table. After making the change, rebuild the view:

```
=> UPDATE mountains_keys SET data_type_guess = 'float' WHERE key_name = 'hike_safety';
 OUTPUT
--------
      1
(1 row)

=> COMMIT;

=> SELECT BUILD_FLEXTABLE_VIEW('mountains');
                BUILD_FLEXTABLE_VIEW
-----------------------------------------------------------
 The view public.mountains_view is ready for querying
(1 row)
```

Now, repeat the VIEW_COLUMNS query and note the changed type:

```
=> SELECT column_name, data_type FROM VIEW_COLUMNS WHERE table_name = 'mountains_view';
 column_name |  data_type
-------------+-------------
 name        | varchar(26)
 type        | varchar(20)
 height      | int
 hike_safety | float
(4 rows)
```

Create a hybrid flex table

If you already know that some of the data you load and query regularly needs full Vertica performance and support, you can create a hybrid flex table. A hybrid flex table has one or more real columns that you define, as well as the __raw__ column that holds all of the data. Querying real columns is faster than querying the __raw__ column. You can define default values for the columns.

When creating a hybrid flex table, specify real columns as you would for other tables:

```
=> CREATE FLEX TABLE mountains_hybrid(
   name VARCHAR(41) DEFAULT name::VARCHAR(41),
   hike_safety FLOAT DEFAULT hike_safety::float);

=> COPY mountains_hybrid FROM '/home/dbadmin/Downloads/mountains.json'
   PARSER FJSONPARSER();
 Rows Loaded
-------------
           5
(1 row)
```

You can compute keys and build the view in a single step:

```
=> SELECT COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW('mountains_hybrid');
                                COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW
-----------------------------------------------------------------------------------------------------
 Please see public.mountains_hybrid_keys for updated keys
The view public.mountains_hybrid_view is ready for querying
(1 row)
```

The keys table shows the data types you declared for the real columns and inferred types for other fields:

```
=> SELECT * FROM mountains_hybrid_keys;
  key_name   | frequency | data_type_guess
-------------+-----------+-----------------
 height      |         4 | Integer
 hike_safety |         4 | float
 name        |         5 | varchar(41)
 type        |         5 | Varchar(20)
(4 rows)
```

If you create a basic flex table, you can later promote one or more virtual columns to real columns by using ALTER TABLE . See Materializing flex tables .

Materialize virtual columns in a hybrid flex table

After you explore your flex table data, you can promote one or more virtual columns in your flex table to real columns. You do not need to create a separate columnar table.

The MATERIALIZE_FLEXTABLE_COLUMNS function creates real columns for a specified number of virtual columns, starting with the most-used:

```
=> SELECT MATERIALIZE_FLEXTABLE_COLUMNS('mountains_hybrid', 3);
                             MATERIALIZE_FLEXTABLE_COLUMNS
-----------------------------------------------------------------------------------------------
 The following columns were added to the table public.mountains_hybrid:
   type
   height
For more details, run the following query:
SELECT * FROM v_catalog.materialize_flextable_columns_results WHERE table_schema = 'public' and table_name = 'mountains_hybrid';

(1 row)
```

The call specified three columns to materialize, but the table was created with two real columns, leaving only two more virtual columns. After materializing columns, you can check the results in the MATERIALIZE_FLEXTABLE_COLUMNS_RESULTS system table. The following output shows two columns with a status of ADDED and one with a status of EXISTS:

```
=> \x
Expanded display is on.
=> SELECT table_id, table_schema, table_name, key_name, status, message FROM MATERIALIZE_FLEXTABLE_COLUMNS_RESULTS
WHERE table_name = 'mountains_hybrid';
-[ RECORD 1 ]+-----------------------------------------------------
table_id     | 45035996273708192
table_schema | public
table_name   | mountains_hybrid
key_name     | type
status       | ADDED
message      | Added successfully
-[ RECORD 2 ]+-----------------------------------------------------
table_id     | 45035996273708192
table_schema | public
table_name   | mountains_hybrid
key_name     | height
status       | ADDED
message      | Added successfully
-[ RECORD 3 ]+-----------------------------------------------------
table_id     | 45035996273708192
table_schema | public
table_name   | mountains_hybrid
key_name     | name
status       | EXISTS
message      | Column of same name already exists in table definition
```

View the definition of the hybrid table after these changes:

```
=> \d mountains_hybrid
List of Fields by Tables
-[ RECORD 1 ]----------------------------------------------------------------
Schema      | public
Table       | mountains_hybrid
Column      | __raw__
Type        | long varbinary(130000)
Size        | 130000
Default     |
Not Null    | t
Primary Key | f
Foreign Key |
-[ RECORD 2 ]----------------------------------------------------------------
Schema      | public
Table       | mountains_hybrid
Column      | name
Type        | varchar(41)
Size        | 41
Default     | (public.MapLookup(mountains_hybrid.__raw__, 'name'))::varchar(41)
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 3 ]----------------------------------------------------------------
Schema      | public
Table       | mountains_hybrid
Column      | hike_safety
Type        | float
Size        | 8
Default     | (public.MapLookup(mountains_hybrid.__raw__, 'hike_safety'))::float
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 4 ]----------------------------------------------------------------
Schema      | public
Table       | mountains_hybrid
Column      | type
Type        | varchar(20)
Size        | 20
Default     | (public.MapLookup(mountains_hybrid.__raw__, 'type'))::!varchar(20)
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 5 ]----------------------------------------------------------------
Schema      | public
Table       | mountains_hybrid
Column      | height
Type        | int
Size        | 8
Default     | (public.MapLookup(mountains_hybrid.__raw__, 'height'))::!int
Not Null    | f
Primary Key | f
Foreign Key |
```

Understanding flex tables

The term *unstructured data* (sometimes called *semi-structured* or *dark* data) does not indicate that the data you load into flex tables is entirely without structure. However, you may not know the data's composition or the inconsistencies of its design. In some cases, the data might not be relational.

Your data might have some structure (like JSON and delimited data). Data might be semi-structured or stringently structured, but in ways that you either do not know about or do not expect. The term *flexible data* encompasses these and other kinds of data. You can load your flexible data directly into a flex table and query its contents.

To summarize, you can load data first, without knowing its structure, and then query its content after a few simple transformations. In some cases, you already know the data structure, such as some keys in the data. If so, you can query these values explicitly as soon as you load the data.

Storing flex table data

While you can store unstructured data in a flex table __raw__ column, that column is implemented as a real column.

Vertica compresses __raw__ column data by about one half (1/2). While this factor is less than the compression rate for real columns, the reduction is significant for large amounts (more than 1TB) of unstructured data. After compression is complete, Vertica writes the data to disk. This approach maintains K-safety in your cluster and supports standard recovery processes should node failures occur. Flex tables are included in full backups (or, if you choose, in object-level backups).

What happens when you create flex tables?

Whenever you execute a CREATE FLEX TABLE statement, Vertica creates three objects, as follows:

- The flexible table (*flex_table*)
- An associated keys table (*flex_table* _keys)
- A default view for the main table (*flex_table* _view)

The _keys and _view objects are dependents of the parent, *flex_table* . Dropping the flex table also removes its dependents, although you can drop the _keys or _view objects independently.

You can create a flex table without specifying any column definitions. When you do so, Vertica automatically creates two tables, the named flex table (such as darkdata) and its associated keys table, darkdata_keys :

```
=> CREATE flex table darkdata();
CREATE TABLE
=> \dt dark*
              List of tables
 Schema |     Name      | Kind  |  Owner  | Comment
--------+---------------+-------+---------+---------
 public | darkdata      | table | dbadmin |
 public | darkdata_keys | table | dbadmin |
(2 rows)
```

Each flex table has two default columns, __raw__ and __identity__ . The __raw__ column exists in every flex table to hold the data you load. The __identity__ column is auto-incrementing. Vertica uses the __identity__ column for segmentation and sort order when no other column definitions exist. The flex keys table has three columns:

```
=> SELECT * FROM darkdata_keys;
 key_name | frequency | data_type_guess
----------+-----------+-----------------
(0 rows)
```

If you define columns when creating a flex table, Vertica still creates the __raw__ column. However, the table has no __identity__ column because columns are specified for segmentation and sort order. Two tables are created automatically, as shown in the following example:

```
=> CREATE FLEX TABLE darkdata1 (name VARCHAR);
CREATE TABLE

=> SELECT * FROM darkdata1;
 __raw__ | name
---------+------
(0 rows)

=> \d darkdata1*
                                    List of Fields by Tables
 Schema |   Table   | Column |          Type          |  Size  | Default | Not Null | Primary Key | Foreign Key
--------+-----------+--------+------------------------+--------+---------+----------+-------------+-------------
 public | darkdata1 | __raw__ | long varbinary(130000) | 130000 |         | t        | f           |
 public | darkdata1 | name    | varchar(80)            |     80 |         | f        | f           |
(2 rows)

=> \dt darkdata1*
                 List of tables
 Schema |      Name      | Kind  |  Owner  | Comment
--------+----------------+-------+---------+---------
 public | darkdata1      | table | dbadmin |
 public | darkdata1_keys | table | dbadmin |
(2 rows)
```

For more examples, see Creating flex tables .

Creating superprojections automatically

In addition to creating two tables for each flex table, Vertica creates superprojections for both the main flex table and its associated keys table. Using the \dj command, you can display the projections created automatically for the darkdata and darkdata1 tables in this set of examples:

```
=> \dj darkdata*
                        List of projections
 Schema |          Name           |  Owner  |       Node       | Comment
--------+-------------------------+---------+------------------+---------
 public | darkdata1_b0            | dbadmin |                  |
 public | darkdata1_b1            | dbadmin |                  |
 public | darkdata1_keys_super    | dbadmin | v_vmart_node0001 |
 public | darkdata1_keys_super    | dbadmin | v_vmart_node0003 |
 public | darkdata1_keys_super    | dbadmin | v_vmart_node0004 |
 public | darkdata_b0             | dbadmin |                  |
 public | darkdata_b1             | dbadmin |                  |
 public | darkdata_keys__super    | dbadmin | v_vmart_node0001 |
 public | darkdata_keys_super     | dbadmin | v_vmart_node0003 |
 public | darkdata_keys_super     | dbadmin | v_vmart_node0004 |
(10 rows)
```

Default flex table view

When you create a flex table, you also create a default view. This default view uses the table name with a _view suffix. If you query the default view, you are prompted to use the COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW function. This view enables you to update the view after you load data so that it includes all keys and values:

```
=> \dv darkdata*
                List of View Fields
 Schema |      View      | Column |     Type     | Size
--------+----------------+--------+--------------+------
 public | darkdata_view  | status | varchar(124) |  124
 public | darkdata1_view | status | varchar(124) |  124
(2 rows)
```

For more information, see Updating flex table views .

Flex functions

Three sets of functions support flex tables and extracting data into VMaps:

- Data (helper) functions
- Extractor functions
- Map functions

Using clients with flex tables

You can use the Vertica-supported client drivers with flex tables as follows:

- To load data into a flex table, you can use the INSERT or COPY LOCAL statement with the appropriate flex parser.
- The driver metadata APIs return only real columns. For example, if you select * from a flex table with a single materialized column, the statement returns that column and the __raw__ . However, it does not return virtual columns from within __raw__ . To access virtual columns and their values, query the associated _keys table, just as you would in vsql .

Creating flex tables

You can create a flex table or an external flex table without column definitions or other parameters. You can use any CREATE TABLE statement parameters you prefer, as usual.

Creating basic flex tables

Here's how to create the table:

```
=> CREATE FLEX TABLE darkdata();
CREATE TABLE
```

Selecting from the table before loading any data into it reveals its two real columns, __identity__ and __raw__ :

```
=> SELECT * FROM darkdata;
__identity__    | __raw__
--------------+---------
(0 rows)
```

Below is an example of creating a flex table with a column definition:

```
=> CREATE FLEX TABLE darkdata1(name VARCHAR);
CREATE TABLE
```

When flex tables exist, you can add new columns (including those with default derived expressions), as described in Materializing Flex Tables .

Materializing flex table virtual columns

After you create your flex table and load data, you compute keys from virtual columns. After completing those tasks, you can materialize some keys by promoting virtual columns to real table columns. By promoting virtual columns, you query real columns rather than the raw data.

You can promote one or more virtual columns — materializing those keys from within the __raw__ data to real columns. Vertica recommends this approach to get the best query performance for all important keys. You don't need to create new columnar tables from your flex table.

Materializing flex table columns results in a hybrid table. Hybrid tables:

- Maintain the convenience of a flex table for loading unstructured data
- Improve query performance for any real columns

If you have only a few columns to materialize, try altering your flex table progressively, adding columns whenever necessary. You can use the ALTER TABLE...ADD COLUMN statement to do so, just as you would with a columnar table. See Materializing flex tables for ideas about adding columns.

If you want to materialize columns automatically, use the helper function MATERIALIZE_FLEXTABLE_COLUMNS .

Creating columnar tables from flex tables

You can create a regular Vertica table from a flex table, but you cannot use one flex table to create another.

Typically, you create a columnar table from a flex table after loading data. Then, you specify the virtual column data you want in a regular table, casting virtual columns to regular data types.

To create a columnar table from a flex table, darkdata , select two virtual columns, (user.lang and user.name), for the new table. Use a command such as the following, which casts both columns to varchars for the new table:

```
=> CREATE TABLE darkdata_full AS SELECT "user.lang"::VARCHAR, "user.name"::VARCHAR FROM darkdata;
CREATE TABLE
=> SELECT * FROM darkdata_full;
 user.lang |      user.name
-----------+---------------------
 en        | Frederick Danjou
 en        | The End
 en        | Uptown gentleman.
 en        | ~G A B R I E L A â¿
 es        | Flu Beach
 es        | I'm Toasterâ¥
 it        | laughing at clouds.
 tr        | seydo shi
           |
           |
           |
           |
(12 rows)
```

Creating temporary flex tables

Before you create temporary global and local flex tables, be aware of the following considerations:

- GLOBAL TEMP flex tables are supported. Creating a temporary global flex table results in the flextable_keys table and the flextable_view having temporary table restrictions for their content.
- LOCAL TEMP flex tables must include at least one column definition. The reason for this requirement is that local temp tables do not support automatically-incrementing data (such as the flex table default __identity__ column). Creating a temporary local flex table results in the flextable_keys table and the flextable_view existing in the local temporary object scope.
- LOCAL TEMP views are supported for flex and columnar temporary tables.

For global or local temp flex tables to function correctly, you must also specify the ON COMMIT PRESERVE ROWS clause. You must use the ON COMMIT clause for the flex table helper functions, which rely on commits. Create a local temp table as follows:

```
=> CREATE FLEX LOCAL TEMP TABLE good(x int) ON COMMIT PRESERVE ROWS;
CREATE TABLE
```

After creating a local temporary flex table using this approach, you can then load data into the table, create table keys, and a flex table view:

```
=> COPY good FROM '/home/release/KData/bake.json' PARSER fjsonparser();
 Rows Loaded
-------------
           1
(1 row)

=> select compute_flextable_keys_and_build_view('good');
                          compute_flextable_keys_and_build_view
-----------------------------------------------------------------------------------------
 Please see v_temp_schema.good_keys for updated keys
The view good_view is ready for querying
(1 row)
```

Similarly, you can create global temp tables as follows:

```
=> CREATE FLEX GLOBAL TEMP TABLE good_global(x int) ON COMMIT PRESERVE ROWS;
```

After creating a global temporary flex table using this approach, you can then load data into the table, create table keys, and a flex table view:

```
=> COPY good_global FROM '/home/dbadmin/data/flex/bake_single.json' PARSER fjsonparser();
Rows Loaded
-------------
5
(1 row)

=> SELECT compute_flextable_keys_and_build_view('good_global');
                          compute_flextable_keys_and_build_view
-----------------------------------------------------------------------------------------------
 Please see v_temp_schema.good_keys for updated keys
The view good_view is ready for querying
(1 row)
```

Creating external flex tables

To create an external flex table:

```
=> CREATE flex external table mountains() AS COPY FROM 'home/release/KData/kmm_ountains.json' PARSER fjsonparser();
CREATE TABLE
```

As with other flex tables, creating an external flex table produces two regular tables: the named table and its associated _keys table. The keys table is not an external table:

```
=> \dt mountains
                List of tables
 Schema |   Name    | Kind  |  Owner  | Comment
--------+-----------+-------+---------+---------
 public | mountains | table | release |
(1 row)
```

You can use the helper function, COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW , to compute keys and create a view for the external table:

```
=> SELECT compute_flextable_keys_and_build_view ('appLog');

                    compute_flextable_keys_and_build_view
-----------------------------------------------------------------------------------------------
Please see public.appLog_keys for updated keys
The view public.appLog_view is ready for querying
(1 row)
```

Check the keys from the _keys table for the results of running the helper application:

```
=> SELECT * FROM appLog_keys;
                         key_name                         | frequency |   data_type_guess
----------------------------------------------------------+-----------+-------------------
contributors                                              |         8 | varchar(20)
coordinates                                               |         8 | varchar(20)
created_at                                                |         8 | varchar(60)
entities.hashtags                                         |         8 | long varbinary(186)
.
.
.
retweeted_status.user.time_zone                           |         1 | varchar(20)
retweeted_status.user.url                                 |         1 | varchar(68)
retweeted_status.user.utc_offset                          |         1 | varchar(20)
retweeted_status.user.verified                            |         1 | varchar(20)
(125 rows)
```

You can query the view:

```
=> SELECT "user.lang" FROM appLog_view;
 user.lang
-----------
it
en
es
en
en
es
tr
en
(12 rows)
```

> **Note**
> External tables are fully supported for both flex and columnar tables. However, using external flex (or columnar) tables is less efficient than using flex tables whose data is stored in the Vertica database. Data that is maintained externally requires reloading each time you query.

Creating a flex table from query results

You can use the CREATE FLEX TABLE AS statement to create a flex table from the results of a query.

You can use this statement to create three types of flex tables:

- Flex table with no materialized columns
- Flex table with some materialized columns
- Flex table with all materialized columns

When a flex __raw__ column is present in the CTAS query, the entire source VMap is carried to the flex table. If the query has matching column names, the key values are overridden.

> **Note**
> ORDER BY and segmentation clauses are only passed to the new flex table if the relevant columns are materialized.

Examples

Creating a flex table with no materialized columns from a regular table causes the results of the query to be stored in the __raw__ column as a VMap.

1. Create a regular table named pets with two columns:
   ```
   => CREATE TABLE pets(age INT, name VARCHAR);
   CREATE TABLE
   ```
2. Create a flex table named family_pets by using the CTAS statement to copy the columns age and name from the pets :
   ```
   => CREATE FLEX TABLE family_pets() AS SELECT age, name FROM pets;
   CREATE TABLE
   ```
3. View the new flex table to confirm the operation has been successful and that the columns age and name have not been materialized.
   ```
   => \d family_pets;
      List of Fields by Tables
   Schema | Table       |    Column    |          Type          |  Size  | Default | Not Null | Primary Key | Foreign Key
   --------+-------------+--------------+------------------------+--------+---------+----------+-------------+-------------
   public | family_pets | __identity__ | int                    |      8 |         | t        | f           |
   public | family_pets | __raw__      | long varbinary(130000) | 130000 |         | t        | f           |
   (2 rows)
   ```

   You can create a flex table with no materialized columns from the results of a query of another flex table. This inserts all the VMaps from the source flex table into the target. This creates a flex table segmented and ordered by the __identity__ column.
4. Create a flex table named city_pets by using the CTAS statement to copy the age and __raw__ columns from family_pets :
   ```
   => CREATE FLEX TABLE city_pets() AS SELECT age, __raw__ FROM family_pets;
   CREATE TABLE
   ```

5. View the new flex table to confirm that the operation has been successful and the columns age and __raw__ have not been materialized.

```
=> SELECT * FROM city_pets;
   List of Fields by Tables
Schema | Table     |    Column    |          Type          |  Size  | Default | Not Null | Primary Key | Foreign Key
--------+----------+--------------+------------------------+--------+---------+----------+-------------+-------------
public | city_pets | __identity__ | int                    |      8 |         | t        | f           |
public | city_pets | __raw__      | long varbinary(130000) | 130000 |         | t        | f           |
(2 rows)
```

You can create a flex table with some materialized columns. This uses a syntax similar to the syntax for creating columnar tables with some materialized columns. Unlike columnar tables, however, you need to match the number of columns with the columns that are returned by the query. In the following example, our query returns three columns (amount , type , and available), but Vertica only materializes the first two.

6. Create a table named animals with three columns, amount , type , and available :

```
=> CREATE TABLE animals(amount INT, type VARCHAR, available DATE);
```

7. Create a flex table named inventory with columns animal_amount and animal_type using the CTAS statement to copy columns amount , type , and available from animals .

```
=> CREATE FLEX TABLE inventory(animal_amount, animal_type) AS SELECT amount, type, available FROM animals;
CREATE TABLE
```

8. View the table data to confirm that columns amount and type have been materialized under the column names animal_amount and animal_type . Column available from animals has also been copied over but was not materialized:

```
=> \d inventory
   List of Fields by Tables
Schema | Table | Column          |          Type          |  Size  | Default | Not Null | Primary Key | Foreign Key
--------+-------+-----------------+------------------------+--------+---------+----------+-------------+-------------
public | flex3 | __raw__         | long varbinary(130000) | 130000 |         | t        | f           |
public | flex3 | animal_amount   | int                    |      8 |         | f        | f           |
public | flex3 | animal_type     | varchar(80)            |     80 |         | f        | f           |
(3 rows)
```

Notice that including empty parentheses in the statement results in a flex table with no materialized columns:

9. Create a flex table named animals_for_sale using the CTAS statement with empty parentheses to copy columns amount , type , and available from animals into a pure flex table:

```
=> CREATE FLEX TABLE animals__for_sale() AS SELECT amount, type, available FROM animals;
CREATE TABLE
```

10. View the table data to confirm that no columns were materialized:

```
=>\d animals_for_sale;
   List of Fields by Tables
Schema | Table           |    Column    |          Type          |  Size  | Default | Not Null | Primary Key | Foreign Key
--------+-----------------+--------------+------------------------+--------+---------+----------+-------------+-------------
public | animals_for_sale | __identity__ | int                    |      8 |         | t        | f           |
public | animals_for_sale | __raw__      | long varbinary(130000) | 130000 |         | t        | f           |
(2 rows)
```

Omitting any parentheses in the statement causes all columns to be materialized:

11. Create a flex table named animals_sold using the CTAS statement without parentheses. This copies columns amount , type , and available from animals and materialize all columns:

```
=> CREATE FLEX TABLE animals_sold AS SELECT amount, type, available FROM animals;
CREATE TABLE
```

12. View the table data to confirm that all columns were materialized:

```
=> \d animals_sold;
   List of Fields by Tables
Schema | Table        | Column    |          Type          |  Size  | Default | Not Null | Primary Key | Foreign Key
--------+-------------+-----------+------------------------+--------+---------+----------+-------------+-------------
public | animals_sold | __raw__   | long varbinary(130000) | 130000 |         | t        | f           |
public | animals_sold | amount    | int                    |      8 |         | f        | f           |
public | animals_sold | type      | varchar(80)            |     80 |         | f        | f           |
public | animals_sold | available | date                   |      8 |         | f        | f           |
(4 rows)
```

Bulk loading data into flex tables

You bulk load data into a flex table with a COPY statement, specifying one of the flex parsers:

- FAVROPARSER
- FCEFPARSER
- FCSVPARSER
- FDELIMITEDPAIRPARSER
- FDELIMITEDPARSER
- FJSONPARSER
- FREGEXPARSER

All flex parsers store the data as a single-value VMap. They reside in the VARBINARY __raw__ column, which is a real column with a NOT NULL constraint. The VMap is encoded into a single binary value for storage in the __raw__ column. The encoding places the value strings in a contiguous block, followed by the key strings. Vertica supports null values within the VMap for keys with NULL -specified columns. The key and value strings represent the virtual columns and their values in your flex table.

If a flex table data row is too large to fit in the __raw__ column, it is rejected. By default, the rejected data and exceptions files are stored in the standard CopyErrorLogs location, a subdirectory of the catalog directory:

```
v_mart_node003_catalog/CopyErrorLogs/trans-STDIN-copy-from-exceptions.1
v_mart_node003_catalog/CopyErrorLogs/trans-STDIN-copy-rejections.1
```

Flex tables do not copy any rejected data, due to disk space considerations. The rejected data file exists, but it contains only a new line character for every rejected record. The corresponding exceptions file lists the reason why each record was rejected.

You can specify a different path and file for the rejected data and exceptions files. To do so, use the COPY options REJECTED DATA and EXCEPTIONS , respectively. You can also save load rejections and exceptions in a table. For more information, see Data load .

Basic flex table load and query

Loading data into a flex table is similar to loading data into a regular columnar table. The difference is that you must use the PARSER option with one of the flex parsers:

```
=> COPY darkdata FROM '/home/dbadmin/data/tweets_12.json' PARSER fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

> **Note**
> You can use many additional COPY parameters as required but not all are supported.

Loading data into flex table real columns

If you create a hybrid flex table with one or more real column definitions, COPY evaluates each virtual column key name during data load and automatically populates real columns with the values from their virtual column counterparts. For columns of scalar types, COPY also loads the keys and values as part of the VMap data in the __raw__ column. For columns of complex types , COPY does not add the values to the __raw__ column.

> **Note**
> Over time, storing values in both column types can impact your licensed data limits. For more information about Vertica licenses, see Managing licenses .

For example, continuing with the JSON example:

1. Create a flex table with a column definition of one of the keys in the data you will load:

```
=> CREATE FLEX TABLE darkdata1 ("user.lang" VARCHAR);
CREATE TABLE
```

2. Load data into the table:

```
=> COPY darkdata1 FROM '/test/vertica/flextable/DATA/tweets_12.json' PARSER fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

3. Query the real column directly:

```
=> SELECT "user.lang" FROM darkdata1;
 user.lang
-----------
 es
 es
 tr
 it
 en
 en
 en
 en
(12 rows)
```

   Empty column rows indicate NULL values. For more information about how NULLs are handled in flex tables, see NULL value .

4. You can query for other virtual columns with similar results:

```
=> SELECT "user.name" FROM darkdata1;
      user.name
---------------------
 I'm Toasterâ¥
 Flu Beach
 seydo shi
 The End
 Uptown gentleman.
 ~G A B R I E L A â¿
 Frederick Danjou
 laughing at clouds.
(12 rows)
```

> **Note**
> While the results for these two queries are similar, the difference in accessing the keys and their values is significant. Data for user.lang has been materialized into a real table column, while user.name remains a virtual column. For production-level data usage (rather than test data sets), materializing flex table data improves query performance significantly.

Handling default values during loading

You can create a flex table with a real column, named for a virtual column that exists in your incoming data. For example, if the data you load has a user.lang virtual column, define the flex table with that column. You can also specify a default column value when creating the flex table with real columns. The next example shows how to define a real column (user.lang), which has a default value from a virtual column (user.name):

```
=> CREATE FLEX TABLE darkdata1 ("user.lang" LONG VARCHAR default "user.name");
```

When you load data into a flex table, COPY uses values from the flex table data, ignoring the default column definition. Loading data into a flex table requires MAPLOOKUP to find keys that match any real column names. A match exists when the incoming data has a virtual column with the same name as a real column. When COPY detects a match, it populates the column with values. COPY returns either a value or NULL for each row, so real columns always have values.

For example, after creating the flex table described in the previous example, load data with COPY :

```
=> COPY darkdata1 FROM '/test/vertica/flextable/DATA/tweets_12.json' PARSER fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

If you query the table after loading, the data shows that the values for the user.lang column were extracted:

- From the data being loaded — values for the user.lang virtual column
- With NULL — rows without values

In this case, the table column default value for user.lang was ignored:

```
=> SELECT "user.lang" FROM darkdata1;
 user.lang
-----------
 it
 en
 es
 en
 en
 es
 tr
 en
(12 rows)
```

Using COPY to specify default column values

You can add an expression to a COPY statement to specify default column values when loading data. For flex tables, specifying any column information requires that you list the __raw__ column explicitly. The following example shows how to use an expression for the default column value. In this case, loading populates the defined user.lang column with data from the input data user.name values:

```
=> COPY darkdata1(__raw__, "user.lang" as "user.name"::VARCHAR)
   FROM '/test/vertica/flextable/DATA/tweets_12.json' PARSER fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
=> SELECT "user.lang" FROM darkdata1;
      user.lang
---------------------
 laughing at clouds.
 Avita Desai
 I'm Toasterâ¥
 Uptown gentleman.
 ~G A B R I E L A â¿
 Flu Beach
 seydo shi
 The End
(12 rows)
```

You can specify default values when adding columns, as described in Altering Flex Tables . When you do so, a different behavior results. For more information about using COPY, its expressions and parameters, see Getting Data into Vertica and the COPY reference page.

Inserting data into flex tables

You can load data into a Vertica flex table using a standard INSERT statement, specifying data for one or more columns. When you use INSERT, Vertica populates any materialized columns and stores the VMap data in the __raw__ column.

Vertica provides two ways to use INSERT with flex tables:

- INSERT ... VALUES
- INSERT ... SELECT

Inserting values into flex tables

To insert data values into a flex table, use an INSERT ... VALUES statement. If you do not specify any columns in your INSERT ... VALUES statement, Vertica positionally assigns values to the real columns of the flex table.

This example shows two ways to insert values into a simple flex table. For both statements, Vertica assigns the values 1 and 'x' to columns a and b, respectively.This example inserts values into the two real columns defined in the flex table:

```
=> CREATE FLEX TABLE flex0 (a INT, b VARCHAR);
CREATE TABLE
=> INSERT INTO flex0 VALUES (1, 'x');
 OUTPUT
--------
      1
(1 row)
```

This example inserts values into a flex table without any real columns:

```
=> CREATE FLEX TABLE flex1();
 CREATE TABLE
=> INSERT INTO flex1(a,b) VALUES (1, 'x');
 OUTPUT
--------
      1
(1 row)
```

For the preceding example, the __raw__ column contains the inserted data:

```
=> SELECT MapToString(__raw__) FROM flex1;
          MapToString
-----------------------------------
{
"a" : "1",
"b" : "x"
}
(1 row)
```

Using INSERT ... SELECT with flex tables

Using an INSERT ... SELECT statement with a flex table is similar to using INSERT ... SELECT with a regular table. The SELECT statement returns the data to insert into the target table.

However, Vertica does *not* require that you balance the number of columns and values. If you do not specify a value for a column, Vertica inserts NULL.

In the next example, Vertica copies the a and b values from the flex1 table, and creates columns c, d, e, and f. Because the statement does not specify a value for f, Vertica assigns it a NULL.

```
=> CREATE FLEX TABLE flex2();
CREATE TABLE
=> INSERT INTO flex2(a,b) SELECT a,b, '2016-08-10 11:10' c, 'Hello' d, 3.1415 e, f from flex1;
 OUTPUT
--------
      1
(1 row)
=> SELECT MapToString(__raw__) FROM flex2;
                          MapToString
-----------------------------------------------------------------------------------------------------------
{
"a" : "1",
"b" : "x",
"c" : "2016-08-10 11:10",

"d" : "Hello",
"e" : "3.1415",
"f" : null
}
(1 row)
```

Inserting __raw__ columns into a flex table

Inserting a __raw__ column into a flex table inserts the entire source VMap into the target table. Vertica does not assign the __raw__ column to any target column. Its position in the SELECT statement does not matter.

The following two INSERT statements are equivalent.

```
=> INSERT INTO flex4(a,b) SELECT a, __raw__, b FROM flex3;
=> INSERT INTO flex4(a,b) SELECT a, b, __raw__ FROM flex3;
```

Error handling

Type coercion errors occur only with real columns. The insert operation fails as follows:

```
=> CREATE FLEX TABLE my_table(a INT, b VARCHAR);
 CREATE TABLE
=> INSERT INTO my_table(a, b) VALUES ('xyz', '5');
ERROR: Invalid input syntax for integer: "xyz"
```

If you try to insert values into the __raw__ column, the insert fails as follows:

```
=> CREATE FLEX TABLE my_table(a INT, b VARCHAR);
 CREATE TABLE
=> INSERT INTO my_table(a,b,__raw__) VALUES (1,'x','abcdef');
ERROR 7372: Cannot assign value to "__raw__" column
```

See also

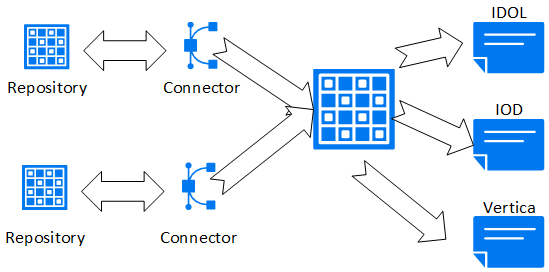
- INSERT
- Bulk loading data into flex tables
- Data type coercion

Using flex tables for IDOL data

You can create flex tables to use with the IDOL Connector Framework Server (CFS) and an ODBC client. The CFS VerticaIndexer module uses the connector to retrieve data. CFS then indexes the data into your Vertica database.

CFS supports many connectors for interfacing to different unstructured file types stored in repositories. Examples of repositories include Microsoft Exchange (email), file systems (including Word documents, images, and videos), Microsoft SharePoint, and Twitter (containing Tweets).

Connectors retrieve and aggregate data from repositories. CFS indexes the data, sending it to IDOL, IDOL OnDemand, or Vertica. The following figure illustrates a basic setup with a repository and a connector.



After you configure CFS and connect it to your Vertica database, the connector monitors the repository for changes and deletions to loaded documents, and for new files not previously added to the server. CFS then updates its server destinations automatically.

To achieve the best query results with ongoing CFS updates and deletes, Vertica recommends using live aggregate projections and top-K projections. For more information about how these projections work, and for examples of using them, see Projections .

ODBC connection string for CFS

There are several steps to setting up the CFS VerticaIndexer to load IDOL metadata into your database.

One of the first steps is to add information to the CFS configuration file. To do so, add an Indexing section to the configuration file that specifies the ODBC ConnectionString details.

Successfully loading data requires a valid database user with write permissions to the destination table. Two ODBC connection parameters (UID and PWD) specify the Vertica user and password. The following example shows a sample CFS Indexing section. The section includes a ConnectionString with the basic parameters, including a sample user (UID=fjones) and password (PWD=fjones_password):

```
[Indexing]
IndexerSections=vertica
IndexTimeInterval=30
[vertica]
IndexerType = Library
ConnectionString=Driver=Vertica;Server=123.456.478.900;Database=myDB;UID=fjones;PWD=fjones_password
TableName = marcomm.myFlexTable
LibraryDirectory = ./shared_library_indexers
LibraryName = verticaIndexer
```

For more information about ODBC connection parameters, see [ODBC Configuration Parameters.](http://vertica.com/docs/7.1.x/HTML/index.htm#Authoring/ConnectingToVertica/ClientODBC/DSNParameters.htm?TocPath=Connecting to HP Vertica|Client Libraries|Creating an ODBC Data Source Name (DSN)|____4)

CFS COPY LOCAL statement

CFS first indexes and processes metadata from a document repository to add to your database. Then, CFS uses the Indexing information you added to the configuration file to create an ODBC connection. After establishing a connection, CFS generates a standard COPY LOCAL statement, specifying the fjsonparser . CFS loads data directly into your pre-existing flex table with a statement such as the following:

```
=> COPY myFlexTable FROM LOCAL path_to_compressed_temporary_json_file PARSER fjsonparser();
```

```
=> SELECT * FROM myavro;
 __identity__ | __raw__
--------------+---------
(0 rows)
```

When your IDOL metadata appears in a flex table, you can optionally add new table columns, or materialize other data, as described in Altering Flex Tables .

Using flex table parsers

You can load flex tables with one of several parsers, using the options that these parsers support.

In addition to the parsers listed in this section, the following parsers described in Data formats in Data load support flex tables:

- JSON data
- Avro data
- Matches from regular expressions

- Common event format (CEF) data

In this section
- Loading columnar tables with flex parsers
- Loading CSV data
- Loading delimited data

Loading columnar tables with flex parsers

You can use any of the flex parsers to load data into columnar tables. Using the flex table parsers to load columnar tables gives you the capability to mix data loads in one table. For example, you can load JSON data in one session and delimited data in another.

> **Note**
> For Avro data, you can load only data into a columnar table, not the schema. For flex tables, Avro schema information is required to be embedded in the data.

The following basic examples illustrate how you can use flex parsers with columnar tables.

1. Create a columnar table, `super` , with two columns, `age` and `name` :

```
=> CREATE TABLE super(age INT, name VARCHAR);
CREATE TABLE
```

2. Enter JSON values from STDIN, using the `fjsonparser()` :

```
=> COPY super FROM stdin PARSER fjsonparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"age": 5, "name": "Tim"}
>>  {"age": 3}
>>  {"name": "Fred"}
>>  {"name": "Bob", "age": 10}
>> \.
```

3. Query the table to see the values you entered:

```
=> SELECT * FROM super;
 age | name
-----+------
     | Fred
  10 | Bob
   5 | Tim
   3 |
(4 rows)
```

4. Enter some delimited values from STDIN, using the `fdelimitedparser()` :

```
=> COPY super FROM stdin PARSER fdelimitedparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> name |age
>> Tim|50
>> |30
>> Fred|
>> Bob|100
>> \.
```

5. Query the flex table. Both JSON and delimited data are saved in the same columnar table, `super` .

```
=> SELECT * FROM super;
 age | name
-----+------
  50 | Tim
  30 |
   3 |
   5 | Tim
 100 | Bob
     | Fred
  10 | Bob
     | Fred
(8 rows)
```

Use the reject_on_materialized_type_error parameter to avoid loading data with type mismatch. If reject_on_materialized_type_error is set to false , the flex parser will accept the data with type mismatch. Consider the following example:

Assume that the CSV file to be loaded has the following sample contents:

```
$ cat json.dat
{"created_by":"system","site_source":"flipkart_india_kol","updated_by":"system1","invoice_id":"INVDPKOL100",
"vendor_id":"VEN15731","total_quantity":12,"created_at":"2012-01-09 23:15:52.0"}
{"created_by":"system","site_source":"flipkart_india_kol","updated_by":"system2","invoice_id":"INVDPKOL101",
"vendor_id":"VEN15732","total_quantity":14,"created_at":"hello"}
```

1. Create a columnar table.

```
=> CREATE TABLE hdfs_test (
site_source VARCHAR(200),
total_quantity int ,
vendor_id varchar(200),
invoice_id varchar(200),
updated_by varchar(200),
created_by varchar(200),
created_at timestamp
);
```

2. Load JSON data.

```
=>COPY hdfs_test FROM '/home/dbadmin/json.dat' PARSER fjsonparser() ABORT ON ERROR;
Rows Loaded
-------------
2
(1 row)
```

3. View the contents.

```
=> SELECT * FROM hdfs_test;
site_source | total_quantity | vendor_id | invoice_id | updated_by | created_by | created_at
--------------------+----------------+-----------+-------------+------------+------------+---------------------
flipkart_india_kol | 12 | VEN15731 | INVDPKOL100 | system1 | system | 2012-01-09 23:15:52
flipkart_india_kol | 14 | VEN15732 | INVDPKOL101 | system2 | system |
(2 rows)
```

4. If reject_on_materialized_type_error parameter is set to true , you will receive errors when loading the sample JSON data.

```
=> COPY hdfs_test FROM '/home/dbadmin/data/flex/json.dat' PARSER fjsonparser(reject_on_materialized_type_error=true) ABORT ON ERROR;
ERROR 2035:  COPY: Input record 2 has been rejected (Rejected by user-defined parser)
```

Loading CSV data

Use the fcsvparser to load data in CSV format (comma-separated values). Because no formal CSV standard exists, Vertica supports the RFC 4180 standard as the default behavior for fcsvparser . Other parser parameters simplify various combinations of CSV options into columnar or flex tables. Using fcsvparser parses the following CSV data formats:

- **RFC 4180:** The RFC4180 CSV format parser for Vertica flex tables. The parameters for this format are fixed and cannot be changed.
- **Traditional:** The traditional CSV parser lets you specify the parameter values such as delimiter or record terminator. For a detailed list of parameters, see FCSVPARSER .

Using default parser settings

These fixed parameter settings apply to the RCF4180 format.

You may use the same value for enclosed_by and escape . Other values must be unique.

Parameter	Data Type	Fixed Value (RCF4180)	Default Value (Traditional)
delimiter	CHAR	,	,
enclosed_by	CHAR	"	"
escape	CHAR	"	\
record_terminator	CHAR	\n or \r\n	\n or \r\n

Use the type parameter to indicate either an RFC 4180-compliant file or a traditional-compliant file. You can specify type as RCF4180 . However, you must first verify that the data is compatible with the preceding fixed values for parameters of the RFC4180 format. The default value of the type parameter is RFC4180.

Loading CSV data (RFC4180)

Follow these steps to use fcsvparser to load data in the RFC4180 CSV data format.

To perform this task, assume that the CSV file to be loaded has the following sample contents:

```
$ more /home/dbadmin/flex/flexData1.csv
sno,name,age,gender
1,John,14,male
2,Mary,23,female
3,Mark,35,male
```

1. Create a flex table:
   ```
   => CREATE FLEX TABLE csv_basic();
   CREATE TABLE
   ```
2. Load the data from the CSV file using fcsvparser :
   ```
   => COPY csv_basic FROM '/home/dbadmin/flex/flexData1.csv' PARSER fcsvparser();
   Rows Loaded
   -------------
   3
   (1 row)
   ```
3. View the data loaded in the flex table:

```
=> SELECT maptostring(__raw__) FROM csv_basic;
maptostring
-------------------------------------------------------------------------------
{
"age" : "14",
"gender" : "male",
"name" : "John",
"sno" : "1"
}
{
"age" : "23",
"gender" : "female",
"name" : "Mary",
"sno" : "2"
}
{
"age" : "35",
"gender" : "male",
"name" : "Mark",
"sno" : "3"
}
(3 rows)
```

Loading CSV data (traditional)

Follow these steps to use fcsvparser to load data in traditional CSV data format using fcsvparser .

In this example, the CSV file uses $ as a delimiter and # as a record_terminator . The sample CSV file to load has the following contents:

```
$ more /home/dbadmin/flex/flexData1.csv
sno$name$age$gender#
1$John$14$male#
2$Mary$23$female#
3$Mark$35$male#
```

1. Create a flex table:

```
=> CREATE FLEX TABLE csv_basic();
CREATE TABLE
```

2. Load the data in flex table using fscvparser with parameters type='traditional' , delimiter='$' and record_terminator='#' :

```
=> COPY csv_basic FROM '/home/dbadmin/flex/flexData2.csv' PARSER fcsvparser(type='traditional',
delimiter='$', record_terminator='#');
Rows Loaded
-------------
3
(1 row)
```

3. View the data loaded in the flex table:

```
=> SELECT maptostring(__raw__) FROM csv_basic;
maptostring
-------------------------------------------------------------------------------
{
"age" : "14",
"gender" : "male",
"name" : "John",
"sno" : "1"
}
{
"age" : "23",
"gender" : "female",
"name" : "Mary",
"sno" : "2"
}
{
"age" : "35",
"gender" : "male",
"name" : "Mark",
"sno" : "3"
}
(3 rows)
```

Rejecting duplicate values

You can reject duplicate values using the reject_on_duplicate=true option with the fcsvparser . The load continues after it rejects a duplicate value. The next example shows how to use this parameter and then displays the specified exception and rejected data files. Saving rejected data to a table, rather than a file, includes both the data and its exception.

```
=> CREATE FLEX TABLE csv_basic();
CREATE TABLE
=> COPY csv_basic FROM stdin PARSER fcsvparser(reject_on_duplicate=true)
exceptions '/home/dbadmin/load_errors/except.out' rejected data '/home/dbadmin/load_errors/reject.out';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> A|A
>> 1|2
>> \.
=> \! cat /home/dbadmin/load_errors/reject.out
A|A
=> \! cat /home/dbadmin/load_errors/except.out
COPY: Input record 1 has been rejected (Processed a header row with duplicate keys with
reject_on_duplicate specified; rejecting.).  Please see /home/dbadmin/load_errors/reject.out,
record 1 for the rejected record.
COPY: Loaded 0 rows, rejected 1 rows.
```

Rejecting data on materialized column type errors

The fcsvparser parser has a Boolean parameter, reject_on_materialized_type_error . Setting this parameter to true causes rows to be rejected if *both* the following conditions exist in the input data:

- Includes keys matching an existing materialized column
- Has a key value that cannot be coerced into the materialized column's data type

The following examples illustrate setting this parameter.

1. Create a table, reject_true_false , with two real columns:

   ```
   => CREATE FLEX TABLE reject_true_false(one int, two int);
   CREATE TABLE
   ```

2. Load CSV data into the table (from STDIN), using the fcsvparser with reject_on_materialized_type_error=false . While false is the default value, you can specify it explicitly, as shown. Additionally, set the parameter header=true to specify the columns for input values:

```
=> COPY reject_true_false FROM stdin PARSER fcsvparser(reject_on_materialized_type_error=false,header=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> one,two
>> 1,2
>> "3","four"
>> "five",6
>> 7,8
>> \.
```

3. Invoke maptostring to display the table values after loading data:

```
=> SELECT maptostring(__raw__), one, two FROM reject_true_false;
maptostring               | one | two
------------------------------------------+-----+-----
{
"one" : "1",
"two" : "2"
}
|   1 |   2
{
"one" : "3",
"two" : "four"
}
|   3 |
{
"one" : "five",
"two" : "6"
}
|     |   6
{
"one" : "7",
"two" : "8"
}
|   7 |   8
(4 rows)
```

4. Truncate the table to empty the data stored in the table:

```
=> TRUNCATE TABLE reject_true_false;
TRUNCATE TABLE
```

5. Reload the same data again, but this time, set reject_on_materialized_type_error=true :

```
=> COPY reject_true_false FROM stdin PARSER fcsvparser(reject_on_materialized_type_error=true,header=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> one,two
>> 1,2
>> "3","four"
>> "five",6
>> 7,8
>> \.
```

6. Call maptostring to display the table contents. Only two rows are currently loaded, whereas the previous results had four rows. The rows having input values with incorrect data type have been rejected:

```
=> SELECT maptostring(__raw__), one, two FROM reject_true_false;
maptostring             | one | two
-------------------------------------+-----+-----
{
"one" : "1",
"two" : "2"
}
|   1 |   2
{
"one" : "7",
"two" : "8"
}
|   7 |   8
(2 rows)
```

Note

The parser fcsvparser uses null values if there is a type mismatch and you set the reject_on_materialized_type_error parameter to false .

Rejecting or omitting empty rows

Valid CSV files can include empty key and value pairs. Such rows are invalid for SQL. You can control the behavior for empty rows by either rejecting or omitting them, using two boolean FCSVPARSER parameters:

- reject_on_empty_key
- omit_empty_keys

The following example illustrates how to set these parameters:

1. Create a flex table:

```
 => CREATE FLEX TABLE csv_basic();
CREATE TABLE
```

2. Load CSV data into the table (from STDIN), using the fcsvparser with reject_on_empty_key=false . While false is the default value, you can specify it explicitly, as shown. Additionally, set the parameter header=true to specify the columns for input values:

```
=> COPY csv_basic FROM stdin PARSER fcsvparser(reject_on_empty_key=false,header=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> ,num
>> 1,2
>> \.
```

3. Invoke maptostring to display the table values after loading data:

```
=>SELECT maptostring(__raw__) FROM csv_basic;
maptostring
----------------------------------
{
"" : "1",
"num" : "2"
}

(1 row)
```

4. Truncate the table to empty the data stored in the table:

```
=> TRUNCATE TABLE csv_basic;
TRUNCATE TABLE
```

5. Reload the same data again, but this time, set reject_on_empty_key=true :

```
=> COPY csv_basic FROM stdin PARSER fcsvparser(reject_on_empty_key=true,header=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> ,num
>> 1,2
>> \.
```

6. Call maptostring to display the table contents. No rows are loaded because one of the keys is empty:

```
=>SELECT maptostring(__raw__) FROM csv_basic;
maptostring
-------------
(0 rows)
```

7. Truncate the table to empty the data stored in the table:

```
=> TRUNCATE TABLE csv_basic;
TRUNCATE TABLE
```

8. Reload the same data again, but this time, set omit_empty_keys=true :

```
=> COPY csv_basic FROM stdin PARSER fcsvparser(omit_empty_keys=true,header=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> ,num
>> 1,2
>> \.
```

9. Call maptostring to display the table contents. One row is now loaded, and the rows with empty keys are omitted:

```
=> SELECT maptostring(__raw__) FROM csv_basic;
maptostring
---------------------
{
"num" : "2"
}
(1 row)
```

> **Note**
> If no header names exist, fcsvparser uses a default header of ucol *n* , where *n* is the column offset number. If a table header name and key name match, the parser loads the column with values associated with the matching key name.

Using the NULL parameter

Use the COPY NULL metadata parameter with fcsvparser to load NULL values into a flex table.

The next example uses this parameter:

1. Create a flex table:

```
=> CREATE FLEX TABLE fcsv(c1 int);
CREATE TABLE
```

2. Load CSV data in the flex table using STDIN and the NULL parameter:

```
=> COPY fcsv FROM STDIN PARSER fcsvparser() NULL 'NULL' ;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> a,b,c1
>> 10,20,NULL
>> 20,30,50
>> 20,30,40
>> \.
```

3. Use the compute_flextable_keys_and_build_view function to compute keys and build the flex view:

```
=> SELECT compute_flextable_keys_and_build_view('fcsv');
compute_flextable_keys_and_build_view
---------------------------------------------------------------------------------
Please see public.fcsv_keys for updated keys
The view public.fcsv_view is ready for querying
(1 row)
```

4. View the flex view and replace the NULL values:

```
=> SELECT * FROM public.fcsv_view;
a  | b  | c1
----+----+----
20 | 30 | 50
10 | 20 |
20 | 30 | 40
(3 rows)
=> SELECT a,b, ISNULL(c1,-1) from public.fcsv_view;
a  | b  | ISNULL
----+----+--------
20 | 30 |     50
10 | 20 |     -1
20 | 30 |     40
(3 rows)
```

Handling column headings

The fcsvparser lets you specify your own column headings with the HEADER_NAMES= parameter. This parameter entirely replaces column names in the CSV source header row.

For example, to use these six column headings for a CSV file you are loading, use the fcsvparser parameter as follows:

```
HEADER_NAMES='FIRST, LAST, SOCIAL_SECURITY, TOWN, STATE, COUNTRY'
```

Supplying fewer header names than existing data columns causes fcsvparser to use default names after those you supply. Default header names consist of ucol *n* , where *n* is the column offset number, starting at 0 for the first column. For example, if you supply four header names for a 6-column table, fcsvparser supplies the default names ucol4 and ucol5 , following the fourth header name you provide.

If you supply more headings than the existing table columns, any additional headings remain unused.

Loading delimited data

You can load flex tables with one of two delimited parsers, fdelimitedparser or fdelimitedpairparser .

- Use fdelimitedpairparser when the data specifies column names with the data in each row.
- Use fdelimitedparser when the data does not specify column names or has a header row for column names.

This section describes using some options that fdelimitedpairparser and fdelimitedparser support.

Rejecting duplicate values

You can reject duplicate values using the reject_on_duplicate=true option with the fdelimitedparser . The load continues after it rejects a duplicate value. The next example shows how to use this parameter and then displays the specified exception and rejected data files. Saving rejected data to a table, rather than a file, includes both the data and its exception.

```
=> CREATE FLEX TABLE delim_dupes();
CREATE TABLE
=> COPY delim_dupes FROM stdin PARSER fdelimitedparser(reject_on_duplicate=true)
exceptions '/home/dbadmin/load_errors/except.out' rejected data '/home/dbadmin/load_errors/reject.out';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> A|A
>> 1|2
>> \.
=> \! cat /home/dbadmin/load_errors/reject.out
A|A
=> \! cat /home/dbadmin/load_errors/except.out
COPY: Input record 1 has been rejected (Processed a header row with duplicate keys with
reject_on_duplicate specified; rejecting.).  Please see /home/dbadmin/load_errors/reject.out,
record 1 for the rejected record.
COPY: Loaded 0 rows, rejected 1 rows.
```

Rejecting materialized column type errors

Both the fjsonparser and fdelimitedparser parsers have a boolean parameter, reject_on_materialized_type_error . Setting this parameter to true causes rows to be rejected if *both* the following conditions exist in the input data:

- Includes keys matching an existing materialized column
- Has a value that cannot be coerced into the materialized column's data type

Suppose the flex table has a materialized column, OwnerPercent , declared as a FLOAT . Trying to load a row with an OwnerPercent key that has a VARCHAR value causes fdelimitedparser to reject the data row.

The following examples illustrate setting this parameter.

1. Create a table, reject_true_false , with two real columns:
   ```
   => CREATE FLEX TABLE reject_true_false(one VARCHAR, two INT);
   CREATE TABLE
   ```
2. Load JSON data into the table (from STDIN), using the fjsonparser with reject_on_materialized_type_error=false . While false is the default value, the following example specifies it explicitly for illustration:
   ```
   => COPY reject_true_false FROM stdin PARSER fjsonparser(reject_on_materialized_type_error=false);
   Enter data to be copied followed by a newline.
   End with a backslash and a period on a line by itself.
   >> {"one": 1, "two": 2}
   >> {"one": "one", "two": "two"}
   >> {"one": "one", "two": 2}
   >> \.
   ```
3. Invoke maptostring to display the table values after loading data:

```
=>  SELECT maptostring(__raw__), one, two FROM reject_true_false;
               maptostring        | one | two
----------------------------------+-----+-----
 {
   "one" : "one",
   "two" : "2"
}
   | one |   2
{
   "one" : "1",
   "two" : "2"
}
   | 1   |   2
 {
   "one" : "one",
   "two" : "two"
}
   | one |
(3 rows)
```

4. Truncate the table:

```
=> TRUNCATE TABLE reject_true_false;
```

5. Reload the same data again, but this time, set reject_on_materialized_type_error=true :

```
=> COPY reject_true_false FROM stdin PARSER fjsonparser(reject_on_materialized_type_error=true);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"one": 1, "two": 2}
>> {"one": "one", "two": "two"}
>> {"one": "one", "two": 2}
>> \.
```

6. Call maptostring to display the table contents. Only two rows were loaded, whereas the previous results had three rows:

```
=> SELECT maptostring(__raw__), one, two FROM reject_true_false;
               maptostring               | one | two
-----------------------------------------+-----+-----
 {
   "one" : "1",
   "two" : "2"
}
   | 1   |   2
 {
   "one" : "one",
   "two" : "2"
}
 | one |   2
(2 rows)
```

Computing flex table keys

After loading data into a flex table, you can determine the set of keys that exist in the __raw__ column (the map data). Two helper functions compute keys from flex table map data:

- COMPUTE_FLEXTABLE_KEYS determines which keys exist as virtual columns in the flex map.
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW performs the same functionality as COMPUTE_FLEXTABLE_KEYS, additionally building a new view. See Updating Flex Table Views .

Using COMPUTE_FLEXTABLE_KEYS

During execution, this function calculates the following information for the flex keys table columns:

Column	Description

KEY_NAME	The name of the virtual column (key). Keys larger than 65,000 bytes are truncated.
FREQUENCY	The number of times the key occurs in the VMap.
DATA_TYPE_GUESS	Estimate of the data type for the key based on the non-null values found in the VMap. The function determines the type of each non-string value, depending on the length of the key, and whether the key includes nested maps. If the EnableBetterFlexTypeGuessing configuration parameter is 0 (OFF), this function instead treats all flex table keys as string types ([LONG] VARCHAR or [LONG] VARBINARY).

COMPUTE_FLEXTABLE_KEYS sets the column width for keys to the length of the largest value for each key multiplied by the FlexTableDataTypeGuessMultiplier factor.

Assigning flex key data types

When the EnableBetterFlexTypeGuessing configuration parameter is ON (the default), COMPUTE_FLEXTABLE_KEYS produces more specific results.

Use the sample CSV data in this section to compare the results of using or not using the parameter:

```
Year,Quarter,Region,Species,Grade,Pond Value,Number of Quotes,Available
2015,1,2 - Northwest Oregon & Willamette,Douglas-fir,1P,$615.12 ,12,No
2015,1,2 - Northwest Oregon & Willamette,Douglas-fir,SM,$610.78 ,12,Yes
2015,1,2 - Northwest Oregon & Willamette,Douglas-fir,2S,$596.00 ,20,Yes
2015,1,2 - Northwest Oregon & Willamette,Hemlock,P,$520.00 ,6,Yes
2015,1,2 - Northwest Oregon & Willamette,Hemlock,SM,$510.00 ,6,No
2015,1,2 - Northwest Oregon & Willamette,Hemlock,2S,$490.00 ,14,No
```

Create a flex table and load this data:

```
=> CREATE FLEX TABLE trees();
=> COPY trees FROM '/home/dbadmin/tempdat/trees.csv' PARSER fcsvparser();
```

Compute keys and check the results with the default setting:

```
=> SELECT COMPUTE_FLEXTABLE_KEYS('trees');
           COMPUTE_FLEXTABLE_KEYS
------------------------------------------------
 Please see public.trees_keys for updated keys
(1 row)

=> SELECT * FROM trees_keys;
    key_name     | frequency | data_type_guess
-----------------+-----------+-----------------
 Year            |         6 | Integer
 Quarter         |         6 | Integer
 Region          |         6 | Varchar(66)
 Available       |         6 | Boolean
 Number of Quotes |         6 | Integer
 Grade           |         6 | Varchar(20)
 Species         |         6 | Varchar(22)
 Pond Value      |         6 | Numeric(8,3)
(8 rows)
```

Set EnableBetterFlexTypeGuessing to 0 (OFF), compute keys again, and compare the results:

```
=> SELECT * FROM trees_keys;
   key_name     | frequency | data_type_guess
----------------+-----------+-----------------
 Year           |         6 | varchar(20)
 Quarter        |         6 | varchar(20)
 Region         |         6 | varchar(66)
 Available      |         6 | varchar(20)
 Grade          |         6 | varchar(20)
 Number of Quotes |        6 | varchar(20)
 Pond Value     |         6 | varchar(20)
 Species        |         6 | varchar(22)
(8 rows)
```

Materializing flex tables

After creating flex tables, you can change the table structure to promote virtual columns to materialized (real) columns. If your table is already a hybrid table, you can change existing real columns and promote other important virtual columns. This section describes some key aspects of promoting columns, adding columns, specifying constraints, and declaring default values. It also presents some differences when loading flex or hybrid tables, compared with columnar tables.

> **Note**
> Materializing virtual columns by promoting them to real columns can significantly improve query performance. Vertica recommends that you materialize important virtual columns before running large and complex queries. Promoted columns cause a small decrease in load performance.

Adding columns to flex tables

Add columns to your flex tables to promote virtual columns:

1. Add a real column with the same name as a virtual column (user.name):

```
=> ALTER TABLE darkdata1 ADD COLUMN "user.name" VARCHAR;
ALTER TABLE
```

2. Load some data into the table.

```
 => COPY darkdata1 FROM '/vertica/flextable/DATA/tweets_12.json' PARSER fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

3. Query the materialized column. Notice that loading data populates the column automatically. Blank rows indicate no values or NULLs:

```
=> SELECT "user.name" FROM darkdata1;
     user.name
----------------------
 I'm Toasterâ¥

 Flu Beach
 seydo shi

 The End
 Uptown gentleman.
 ~G A B R I E L A â¿
 Avita Desai

 laughing at clouds.
(12 rows)
```

Adding columns with default values

The section Bulk loading data into flex tables describes the use of default values, and how Vertica evaluates them during loading. As with all tables, using COPY to load data ignores any column default values.

> **Note**
> Adding a table column default expression to a flex table requires casting the column to an explicit data type.

1. Create a darkdata1 table with a column definition. The following example uses a column name (talker) that does not correspond to a virtual column name. Assign a default value with a virtual column name. In this example, the default value for the column talker is (" user.lang "). Since user.lang is a virtual column in the LONG VARBINARY __raw__ column, you must cast its value to VARCHAR to match the talker column definition:

```
=> CREATE FLEX TABLE darkdata1(talker VARCHAR default "user.lang"::VARCHAR);
CREATE TABLE
```

2. Load some JSON data, specifying the __raw__ column:

```
=> COPY darkdata1 (__raw__) FROM '/test/vertica/flextable/DATA/tweets_12.json'
   PARSER fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

3. Query the talker column. Notice that Vertica used the default column value (" user.lang "), because you specified __raw__ . Blank rows indicate no values or NULLs:

```
=> SELECT "talker" FROM darkdata1;
 talker
--------
 it
 en
 es
 en
 en
 es
 tr
 en
(12 rows)
```

4. Alter the table to add a column with a known virtual column name (user.name), assigning the key name as the default value (recommended), and casting it to a VARCHAR:

```
=> ALTER TABLE darkdata1 ADD COLUMN "user.name" VARCHAR default "user.name"::VARCHAR;
ALTER TABLE
```

5. Load data again, this time without __raw__ :

```
=> COPY darkdata1 FROM '/test/vertica/flextable/DATA/tweets_12.json' PARSER fjsonparser();
```

6. Query the two real columns. Notice that talker has no values, because you did not specifiy the __raw__ column. The user.lang column contains values from the user.name virtual column:

```
=> SELECT "talker", "user.name" FROM darkdata1;
 talker |      user.name
--------+---------------------
        | laughing at clouds.
        | Avita Desai
        | I'm Toasterâ¥
        |
        |
        |
        | Uptown gentleman.
        | ~G A B R I E L A â¿
        | Flu Beach
        |
        | seydo shi
        | The End
(12 rows)
```

7. Load data once more, this time specifying a COPY statement with a default value expression for user.name :

```
=> COPY darkdata1 (__raw__, "user.name" as 'QueenElizabeth'::varchar) FROM
'/test/vertica/flextable/DATA/tweets_12.json' PARSER fjsonparser();
 Rows Loaded
-------------
          12
(1 row)
```

8. Query once more. Notice that the real column talker has its default values (you used __raw__). As specified in COPY , the "user.name" as 'QueenElizabeth' expression overrode the user.name default column value:

```
=> SELECT "talker", "user.name" FROM darkdata1;
 talker |   user.name
--------+---------------
 it     | QueenElizabeth
 en     | QueenElizabeth
 es     | QueenElizabeth
        | QueenElizabeth
        | QueenElizabeth
        | QueenElizabeth
 en     | QueenElizabeth
 en     | QueenElizabeth
 es     | QueenElizabeth
        | QueenElizabeth
 tr     | QueenElizabeth
 en     | QueenElizabeth
(12 rows)
```

To summarize, you can set a default column value as part of the ALTER TABLE...ADD COLUMN operation. For materializing columns, the default value should reference the key name of the virtual column (as in "user.lang"). Subsequently loading data with a COPY value expression overrides the default value of the column definition.

Changing the __raw__ column size

You can change the default size of the __raw__ column for flex tables you plan to create, the current size of an existing flex table, or both.

To change the default size for the __raw__ column in new flex tables, set the FlexTableRawSize configuration parameter:

```
=> ALTER DATABASE DEFAULT SET FlexTableRawSize = 120000;
```

Changing the configuration parameter affects all flex tables you create after making this change.

To change the size of the _raw_ column in an existing flex table, use the ALTER TABLE statement to change the definition of the __raw__ column:

```
=> ALTER TABLE tester ALTER COLUMN __raw__ SET DATA TYPE LONG VARBINARY(120000);
ALTER TABLE
```

Note
An error occurs if you try to reduce the __raw__ column size to a value smaller than any data the column contains.

Changing flex table real columns

You can make the following changes to the flex table real columns (__raw__ and __identity__), but not to any virtual columns:

Actions	raw	identity
Change NOT NULL constraints (default)	Yes	Yes
Add primary key and foreign key (PK/FK) constraints	No	Yes
Create projections	No	Yes
Segment	No	Yes
Partition	No	Yes
Specify a user-defined scalar function (UDSF) as a default column expression in ALTER TABLE x ADD COLUMN y statement	No	No

Note
While segmenting and partitioning the __raw__ column is permitted, it is not recommended due to its long data type. By default, if you not define any real columns, flex tables are segmented on the __identity__ column.

Dropping flex table columns

There are two considerations about dropping columns:

- You cannot drop the last column in your flex table's sort order.
- If you have not created a flex table with any real columns, or materialized any columns, you cannot drop the __identity__ column.

Updating flex table views

Adding new columns to a flex table that has an associated view does not update the view's result set, even if the view uses a wildcard (*) to represent all table columns. To incorporate new columns, you must re-create the view .

Updating flex table views

Creating a flex table also creates a default view to accompany the table. The view has the name of the flex table with an underscore (_view) suffix. When you perform a SELECT query from the default view, Vertica prompts you to call the helper function COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW:

```
=> \dv dark*
                List of View Fields
 Schema |      View      | Column |     Type     | Size
--------+----------------+--------+--------------+------
 public | darkdata_view  | status | varchar(124) |  124
 public | darkdata1_view | status | varchar(124) |  124
(2 rows)

=> SELECT * FROM darkdata_view;
                                  status
-------------------------------------------------------------------------------------------
 Please run compute_flextable_keys_and_build_view() to update this view to reflect real and
virtual columns in the flex table
(1 row)
```

There are two helper functions that create views:

- BUILD_FLEXTABLE_VIEW : Creates or re-creates a view for a default or user-defined keys table, ignoring any empty keys.
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW : Computes virtual columns (keys) from the VMap data of a flex table and constructs a view (see also Using COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW).

Using BUILD_FLEXTABLE_VIEW

After computing keys for a flex table (Computing flex table keys), call this function with one or more arguments. The records under the key_name column of the *table_name* _keys table are used as view columns, along with any values for the key. If no values exist, the column value is NULL.

Regardless of the number of arguments, calling this function replaces the contents of the existing view as follows:

- BUILD_FLEXTABLE_VIEW (' *table* ') : Changes the table's existing view with the current contents of the *table* _keys table.
- BUILD_FLEXTABLE_VIEW (' *table* ', ' *view_name* ') : Changes the named view with the current contents of the *table* _keys table.
- BUILD_FLEXTABLE_VIEW (' *table* ', ' *view_name* ', ' *table_keys* ') : Changes the named view using the named keys table. Use this option if you created a custom keys table from the flex table map data, rather than from the default keys table.

If you do not specify a view name, the default name is the flex table name with a _view suffix. For example, if you specify the table darkdata as the sole argument to this function, the default view is called darkdata_view .

You cannot specify a custom view name with the same name as the default view (*table_name* _view), unless you first drop the default-named view and then create your own view of the same name.

Creating a view stores a definition of the column structure at the time of creation. Thus, if you create a flex table view and then promote virtual columns to real columns, you must rebuild the view. Querying a rebuilt flex table view that has newly promoted real columns produces two results. These results reflect values from both virtual columns in the map data and real columns.

Handling JSON duplicate key names in views

SQL is a case-insensitive language, so the names TEST , test , and TeSt are identical. JSON data is case sensitive, so that it can validly contain key names of different cases with separate values.

When you build a flex table view, the function generates a warning if it detects same-name keys with different cases in the *table* _keys table. For example, calling BUILD_FLEXTABLE_VIEW or COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW on a flex table with duplicate key names results in these warnings:

```
=> SELECT COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW('dupe');
WARNING 5821:  Detected keys sharing the same case-insensitive key name
WARNING 5909:  Found and ignored keys with names longer than the maximum column-name length limit


                        compute_flextable_keys_and_build_view
-------------------------------------------------------------------------------------------
 Please see public.dupe_keys for updated keys
The view public.dupe_view is ready for querying
(1 row)
```

While a keys table can include duplicate key names with different cases, a view cannot. Creating a flex table view with either of the helper functions consolidates any duplicate key names to one column name, consisting of all lowercase characters. All duplicate key values for that column are saved. For example, if a flex table contains keys named test , Test , and tESt , the view includes a virtual column test with values from all three keys.

The following examples include added line breaks for readability.

Consider the following query, showing the duplicate test key names:

```
=> \x
Expanded display is on.
dbt=> select * from dupe_keys;
-[ RECORD 1 ]---+----------------------------------------------------------------------------------------------------
key_name        | TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttest
TesttestTesttestTesttestTesttest
frequency       | 2
data_type_guess | varchar(20)
-[ RECORD 2 ]---+----------------------------------------------------------------------------------------------------
key_name        | TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttest
TesttestTesttestTesttestTest12345
frequency       | 2
data_type_guess | varchar(20)
-[ RECORD 3 ]---+----------------------------------------------------------------------------------------------------
key_name        | test
frequency       | 8
data_type_guess | varchar(20)
-[ RECORD 4 ]---+----------------------------------------------------------------------------------------------------
key_name        | TEst
frequency       | 8
data_type_guess | varchar(20)
-[ RECORD 5 ]---+----------------------------------------------------------------------------------------------------
key_name        | TEST
frequency       | 8
data_type_guess | varchar(20)
```

The following query displays the dupe flex table (dupe_view). It shows the consolidated virtual columns, with all values in the test column:

```
=> SELECT * FROM dupe_view;
  test  | testtesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttesttest
testtesttesttesttesttesttest
--------+----------------------------------------------------------------------------------------------------
 upper2 |
 half4  |
 lower1 |
 upper1 |
 half1  |
 half4  |
        |
 lower1 |
 half1  |
 upper2 |
        |
 lower2 |
 lower3 |
 upper1 |
 lower2 |
 lower3 |
(16 rows)
```

Creating a flex table view

The following example shows how to create and query a view for a flex table that contains JSON data:

```
=> CREATE VIEW dd_view AS SELECT "user.lang"::VARCHAR, "user.name"::VARCHAR FROM darkdata;
CREATE VIEW

=> SELECT * FROM dd_view;
 user.lang |      user.name
-----------+---------------------
 en        | Uptown gentleman.
 en        | The End
 it        | laughing at clouds.
 es        | I'm Toasterâ¥
           |
 en        | ~G A B R I E L A â¿
           |
 en        | Avita Desai
 tr        | seydo shi
           |
           |
 es        | Flu Beach
(12 rows)
```

This example shows how to call BUILD_FLEXTABLE_VIEW with the original table and the view you previously created:

```
=> SELECT BUILD_FLEXTABLE_VIEW ('darkdata', 'dd_view');
            build_flextable_view
-------------------------------------------------
 The view public.dd_view is ready for querying
(1 row)
```

Query the view again. You can se that the function populated the view with the contents of the darkdata_keys table. Next, review a snippet from the results, with the key columns and their values:

```
=> \x
Expanded display is on.

=> SELECT * FROM dd_view;
.
.
.
user.following                    |
user.friends_count                |  791
user.geo_enabled                  |  F
user.id                           |  164464905
user.id_str                       |  164464905
user.is_translator                |  F
user.lang                         |  en
user.listed_count                 |  4
user.location                     |  Uptown..
user.name                         |  Uptown gentleman.
.
.
.
```

When building views, be aware that creating a view stores a definition of the column structure at the time the view is created. If you promote virtual columns to real columns after building a view, the existing view definition is not changed. Querying this view with a select statement such as the following, returns values from only the __raw__ column:

```
=> SELECT * FROM myflextable_view;
```

Also understand that rebuilding the view after promoting virtual columns changes the resulting value. Future queries return values from both virtual columns in the map data and from real columns.

Using COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW

Call this function with a flex table to compute Flex table keys (see Computing flex table keys) and create a view in one step.

Querying flex tables

After you create your flex table (with or without additional columns) and load data, you can use SELECT, COPY, TRUNCATE, and DELETE as with other tables. You can use SELECT queries for virtual columns that exist in the __raw__ column and other real columns in your flex tables. Field names in the __raw__ column are case-insensitive, as are the names of real columns.

Unsupported DDL and DML statements

You cannot use the following DDL and DML statements with flex tables:

- CREATE TABLE...AS...
- CREATE TABLE...LIKE...
- SELECT INTO
- UPDATE
- MERGE

Determining flex table data contents

If you do not know what your flex table contains, two helper functions explore the VMap data to determine its contents. Use these functions to compute the keys in the flex table __raw__ column and, optionally, build a view based on those keys:

- COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW

For more information about these and other helper functions, see Flex data functions .

To determine what key value pairs exist as virtual columns:

1. Call the function as follows:

```
=> SELECT compute_flextable_keys('darkdata');
         compute_flextable_keys
-----------------------------------------------------
 Please see public.darkdata_keys for updated keys(1 row)
```

2. View the key names by querying the darkdata_keys table:

```
=> SELECT * FROM darkdata_keys;
```

```
                    key_name                    | frequency |   data_type_guess
----------------------------------------------------+-----------+-------------
 contributors                                      |         8 | varchar(20)
 coordinates                                       |         8 | varchar(20)
 created_at                                        |         8 | varchar(60)
 entities.hashtags                                 |         8 | long varbinary(186)
 .
 .
 retweeted_status.user.time_zone                   |         1 | varchar(20)
 retweeted_status.user.url                         |         1 | varchar(68)
 retweeted_status.user.utc_offset                  |         1 | varchar(20)
 retweeted_status.user.verified                    |         1 | varchar(20)
(125 rows)
```

Querying virtual columns

Continuing with the JSON data example, use a SELECT statement query to explore content from the virtual columns. Then, analyze what is most important to you in case you want to materialize any virtual columns. This example shows how to query some common virtual columns in the VMap data:

```
=> SELECT "user.name", "user.lang", "user.geo_enabled" FROM darkdata1;
     user.name      | user.lang | user.geo_enabled
--------------------+-----------+------------------
 laughing at clouds. | it        | T
 Avita Desai         | en        | F
 I'm Toasterâ¥       | es        | T
                     |           |
                     |           |
                     |           |
 Uptown gentleman.   | en        | F
 ~G A B R I E L A â¿ | en        | F
 Flu Beach           | es        | F
                     |           |
 seydo shi           | tr        | T
 The End             | en        | F
(12 rows)
```

Querying flex table keys

If you reference an undefined column ('which_column') in a flex table query, Vertica converts the query to a call to the MAPLOOKUP function as follows:

```
MAPLOOKUP(__raw__, 'which_column')
```

The MAPLOOKUP function searches the VMap data for the requested key and returns the following information:

- String values associated with the key for a row.
- NULL if the key is not found.

For more information about handling NULL values, see MAPCONTAINSKEY ().

Using functions and casting in flex table queries

You can cast the virtual columns as required and use functions in your SELECT statement queries. The next example uses a SELECT statement to query the created_at and retweet_count virtual columns, and to cast their values in the process:

```
=> SELECT "created_at"::TIMESTAMP, "retweet_count"::INT FROM darkdata1 ORDER BY 1 DESC;
     created_at      | retweet_count
---------------------+---------------
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:05 |             0
 2012-10-15 14:41:04 |             1
                     |
                     |
                     |
                     |
(12 rows)
```

The following query uses the COUNT and AVG functions to determine the average length of text in different languages:

```
=> SELECT "user.lang", count (*), avg(length("text"))::int
   FROM darkdata1 GROUP BY 1 ORDER BY 2 DESC;
 user.lang | count | avg
-----------+-------+-----
 en        |     4 |  42
           |     4 |
 es        |     2 |  96
 it        |     1 |  50
 tr        |     1 |  16
(5 rows)
```

Casting data types in a query

The following query requests the values of the created_at virtual column, without casting to a specific data type:

```
=> SELECT "created_at" FROM darkdata1;
          created_at
-------------------------------
 Mon Oct 15 18:41:04 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
 Mon Oct 15 18:41:05 +0000 2012
(12 rows)
```

The next example queries the same virtual column, casting created_at to a TIMESTAMP . Casting results in different output and the regional time:

```
=> SELECT "created_at"::TIMESTAMP FROM darkdata1 ORDER BY 1 DESC;
    created_at
---------------------
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:05
 2012-10-15 14:41:04
```

Accessing an epoch key

The term *EPOCH* (all uppercase letters) is reserved in Vertica for internal use.

If your JSON data includes a virtual column called epoch , you can query it within your flex table. However, use the MAPLOOKUP function to do so.

Querying nested data

If you load JSON or Avro data with flatten_arrays=FALSE (the default), VMap data in the __raw__ column can contain multiple nested structures. In fact, any VMap JSON or Avro data can contain nested structures. This section describes how best to query such data.

Query VMap nested values

To query a nested structure, you can use multiple maplookup() functions, one for each level. However, the most efficient method is to use bracket ([]) operators.

When parsing or extracting VMap data, the default behavior is to flatten data. Flattened VMap data concatenates key names into one long name, delimiting elements with either the default delimiter (.), or a user-defined delimiter character.

To use bracket operators for nested structures in your VMap data, the data must not be flattened. Further, you cannot use bracket operators on any existing, flattened VMap data.

To load or extract VMap data correctly, specify flatten_maps=FALSE for fjsonparser , favroparser , and the mapjsonextractor() function.

> **Note**
> While bracket operator values look similar to array element specifications, they are strings, not integers. You must enter each nested structure as a string, even if the value is an integer. For example, if the value is 2, specify it as ['2'] , not [2] .

Bracket operators for nested JSON

This example uses the following JSON data as an example of nested data. Save this data as restaurant.json :

```
{
  "restaurant" : {
     "_name_" : "Bob's pizzeria",
     "cuisine" : "Italian",
     "location" : {"city" : "Cambridge", "zip" : "02140"},
     "menu" : [{"item" : "cheese pizza", "price" : "$8.25"},
               {"item" : "chicken pizza", "price" : "$11.99"},
               {"item" : "spinach pizza", "price" : "$10.50"}]
  }
}
```

Create a flex table, rests, and load it with the restaurant.json file:

```
=> COPY rests FROM '/home/dbadmin/tempdat/restaurant.json' PARSER fjsonparser (flatten_maps=false);
 Rows Loaded
-------------
           1
(1 row)
```

After loading your data into a flex table, there are two ways to access nested data using brackets:

- Beginning with the __raw__ column, followed by the character values in brackets
- Starting with the name of the top-most element, followed by the character values in brackets

Both methods are equally efficient. Here are examples of both:

```
=> SELECT __raw__['restaurant']['location']['city'] FROM rests;
  __raw__
-----------
 Cambridge
(1 row)
=> SELECT restaurant['location']['city'] from rests;
 restaurant
------------
 Cambridge
(1 row)
```

Bracket operators for twitter data

This example shows how to extract some basic information from Twitter data.

After creating a flex table, tweets , and loading in some data, the flex table has a block of tweets.

In the following SELECT statement, notice how to specificy the __raw__ column of table tweets , followed by the bracket operators to define the virtual columns of interest (['delete']['status']['user_id']). This query uses the COUNT() function to calculate the number of deleted tweets and outputs 10 results:

```
=> SELECT  __raw__['delete']['status']['user_id'] as UserId, COUNT(*) as TweetsDelete from tweets
-> WHERE mapcontainskey(__raw__, 'delete')
-> GROUP BY __raw__['delete']['status']['user_id']
-> ORDER BY TweetsDelete DESC, UserID ASC LIMIT 10;
  UserId   | TweetsDelete
-----------+--------------
 106079547 |            4
 403474369 |            4
 181188657 |            3
 223136123 |            3
 770139481 |            3
 154602299 |            2
 192127653 |            2
 215011332 |            2
 23321883  |            2
 242173898 |            2
(10 rows)
```

Querying flex views

Flex tables offer the ability of dynamic schema through the application of query rewriting. Use flex views to support restricted access to flex tables. As with flex tables, each time you use a select query on a flex table view, internally, Vertica invokes the maplookup() function, to return information on all virtual columns. This query behavior occurs for any flex or columnar table that includes a __raw__ column.

This example illustrates querying a flex view:

1. Create a flex table.
```
=> CREATE FLEX TABLE twitter();
```
2. Load JSON data into flex table using fjsonparser .
```
=> COPY twitter FROM '/home/dbadmin/data/flex/tweets_10000.json' PARSER fjsonparser();
Rows Loaded
-------------
10000
(1 row)
```
3. Create a flex view on top of flex table twitter with constraint retweet_count>0 .
```
=> CREATE VIEW flex_view AS SELECT __raw__ FROM twitter WHERE retweet_count::int > 0;
CREATE VIEW
```
4. Query the view. First 5 rows are displayed.
```
=> SELECT retweeted,retweet_count,source FROM (select __raw__ from flex_view) t1 limit 5;
retweeted | retweet_count |                                    source
-----------+---------------+-----------------------------------------------------------------------------------
F         | 1             | <a href="http://blackberry.com/twitter" rel="nofollow">Twitter for BlackBerry®</a>
F         | 1             | web
F         | 1             | <a href="http://twitter.com/download/iphone" rel="nofollow">Twitter for iPhone</a>
F         | 23            | <a href="http://twitter.com/download/android" rel="nofollow">Twitter for Android</a>
F         | 7             | <a href="http://twitter.com/download/iphone" rel="nofollow">Twitter for iPhone</a>
(5 rows)
```

Listing flex tables

You can determine which tables in your database are flex tables by querying the is_flextable column of the v_catalog.tables system table. For example, use a query such as the following to see all tables with a true (t) value in the is_flextable column:

```
=> SELECT table_name, table_schema, is_flextable FROM v_catalog.tables;
    table_name      | table_schema | is_flextable
--------------------+--------------+-----------------
 bake1              | public       | t
 bake1_keys         | public       | f
 del                | public       | t
 del_keys           | public       | f
 delicious          | public       | t
 delicious_keys     | public       | f
 bake               | public       | t
 bake_keys          | public       | f
 appLog             | public       | t
 appLog_keys        | public       | f
 darkdata           | public       | t
 darkdata_keys      | public       | f
(12 rows)
```

Data export

You can export data from Vertica, which you might do for the following reasons:

- To instead use the data in external tables; see Working with external data .
- To share data with other clients or consumers in an ecosystem.

- To copy data to another Vertica cluster.

Vertica provides two ways to export data. With the data exporters, you can export the results of a SELECT query to files in Parquet, ORC, or delimited format. You can use partitioning in the export, which can decrease output file size and improve performance when reading the data in external tables. For more information, see File export .

Vertica also provides a way to move data directly between Vertica clusters, without having to export from one and load in the other. You can perform the operation as either an import from another database or an export to another database; aside from the direction of data travel, these two operations are quivalent. For more information, see Database export and import .

In this section

- File export
- Database export and import

File export

You might want to export a table or other query results from Vertica, either to share itwith other applications or to move lower-priority data fromROS to less-expensive storage. Vertica supports several EXPORT TO statements for different file formats: EXPORT TO PARQUET , EXPORT TO ORC , EXPORT TO JSON , and EXPORT TO DELIMITED .

You can export data to HDFS, S3, Google Cloud Storage (GCS), Azure Blob Storage, or the Linux file system. Be careful to avoid concurrent exports to the same output destination. Doing so is an error on any file system and can produce incorrect results.

Not all exporters support all complex types. See the Data Types section on individual reference pages.

You can export ROS data or data that is readable through external tables. After exporting ROS data, you can drop affected ROS partitions to reclaim storage space.

After exporting data, you can define external tables to read that data in Vertica. Parquet and ORC are columnar formats that Vertica can take advantage of in query optimization, so consider using one of those formats for external tables. See Working with external data .

In this section

- Syntax
- Partitioning and sorting data
- Exporting to object stores
- Exporting to the Linux file system
- Monitoring exports

Syntax

> **Note**
> EXPORT TO PARQUET , EXPORT TO ORC , EXPORT TO JSON , and EXPORT TO DELIMITED have the same general syntax. This documentation refers to them collectively as EXPORT statements, not to be confused with EXPORT TO VERTICA .

Use the EXPORT statements to export data specified by a SELECT statement, as in the following example:

```
=> EXPORT TO PARQUET(directory='webhdfs:///data/sales_data')
   AS SELECT * FROM public.sales;
 Rows Exported
---------------
         14336
(1 row)
```

The directory argument specifies where to write the files and is required. You must have permission to write to the output directory. The directory must not already exist. You can export to HDFS, S3, GCS, Azure, or the Linux file system. For additional considerations specific to object stores, see Exporting to object stores . If you export to the local Linux file system (not a shared NFS mount) you must take some additional steps; see Exporting to the Linux file system .

> **Note**
> You can only perform one export per output directory. If you perform more than one concurrent export to the same directory, only one will succeed.

You can use EXPORT statements to write queries across multiple tables in Vertica and export the results. With this approach you can take advantage of powerful, fast query execution in Vertica while making the results available to other clients:

```
=> EXPORT TO ORC(directory='webhdfs:///data/sales_by_region')
   AS SELECT sale.price, sale.date, store.region
   FROM public.sales sale
   JOIN public.vendor store ON sale.distribID = store.ID;
 Rows Exported
---------------
         23301
(1 row)
```

Data types

All exporters can export scalar types. All except EXPORT TO DELIMITED also support primitive arrays . The JSON and Parquet exporters also support the ARRAY and ROW types in any combination:

```
=> SELECT * FROM restaurants;
       name        | cuisine |       location_city        |                                menu
-------------------+---------+----------------------------+-------------------------------------------------------------------
 Bob's pizzeria    | Italian | ["Cambridge","Pittsburgh"] | [{"item":"cheese pizza","price":null},{"item":"spinach pizza","price":10.5}]
 Bakersfield Tacos | Mexican | ["Pittsburgh"]             | [{"item":"veggie taco","price":9.95},{"item":"steak taco","price":10.95}]
(2 rows)

=> EXPORT TO JSON (directory='/output/json') AS SELECT * FROM restaurants;
 Rows Exported
---------------
             2
(1 row)

=> \! cat /output/json/*.json
{"name":"Bob's pizzeria","cuisine":"Italian","location_city":["Cambridge","Pittsburgh"],"menu":[{"item":"cheese pizza","price":null},{"item":"spinach
pizza","price":10.5}]}
{"name":"Bakersfield Tacos","cuisine":"Mexican","location_city":["Pittsburgh"],"menu":[{"item":"veggie taco","price":9.95},{"item":"steak taco","price":10.95}]}
```

Query

EXPORT statements rewrite the query you specify, because the export is done by a user-defined transform function (UDTF). Because of this rewrite, there are some restrictions on the query you supply.

The query can contain only a single outer SELECT statement. For example, you cannot use UNION:

```
=> EXPORT TO PARQUET(directory = '/mnt/shared_nfs/accounts/rm')
   OVER(PARTITION BY hash)
   AS
   SELECT 1 as account_id, '{}' as json, 0 hash
   UNION ALL
   SELECT 2 as account_id, '{}' as json, 1 hash;
ERROR 8975:  Only a single outer SELECT statement is supported
HINT:  Please use a subquery for multiple outer SELECT statements
```

Instead, rewrite the query to use a subquery:

```
=> EXPORT TO PARQUET(directory = '/mnt/shared_nfs/accounts/rm')
  OVER(PARTITION BY hash)
  AS
  SELECT
   account_id,
   json
  FROM
  (
    SELECT 1 as account_id, '{}' as json, 0 hash
    UNION ALL
    SELECT 2 as account_id, '{}' as json, 1 hash
  ) a;
 Rows Exported
---------------
             2
(1 row)
```

To use composite statements such as UNION, INTERSECT, and EXCEPT, rewrite them as subqueries.

When exporting data you can use an OVER() clause to partition and sort the data as described in Partitioning and sorting data . Partitioning and sorting can improve query performance. If you partition the data, you cannot specify schema and table names in the SELECT statement. In this case, specify only the column name.

All exporters have a required directory parameter. All allow you to specify a compression type, and each exporter has format-specific parameters. See the parameter descriptions on the individual reference pages: EXPORT TO PARQUET , EXPORT TO ORC , EXPORT TO JSON , and EXPORT TO DELIMITED .

Partitioning and sorting data

When exporting, you can use the optional OVER clause in the SELECT statement to specify how to partition and/or sort the exported data. Partitioning reduces the sizes of the output data files and canimprove performance when Vertica queries external tables containing thisdata. (See Partitioned file paths .) If you do not specify how to partition the data, Vertica optimizes the export for maximum parallelism.

To specify partition columns, use PARTITION BY in theOVER clause as in the following example:

```
=> EXPORT TO PARQUET(directory = 'webhdfs:///data/export')
  OVER(PARTITION BY date) AS SELECT transactionID, price FROM public.sales;
 Rows Exported
---------------
         28337
(1 row)
```

You can both partition by a column and include that column in the SELECT clause. Including the column allows you to sort it. Vertica still takes advantage of the partitioning during query execution.

You can sort values within a partition for a further performance improvement. Sort table columns based on the likelihood of their occurrence in query predicates; columns that most frequently occur in comparison or range predicates should be sorted first. You can sort values within each partition using ORDER BY in the OVER clause:

```
=> EXPORT TO PARQUET(directory = 'webhdfs:///data/export')
  OVER(PARTITION BY date ORDER BY price) AS SELECT date, price FROM public.sales;
 Rows Exported
---------------
         28337
(1 row)
```

You can use ORDER BY even without partitioning. Storing data in sorted order can improve data access and predicate evaluation performance.

Targets in the OVER clause must be column references; they cannot beexpressions.For more information about OVER, see SQL analytics .

If you are exporting data to a local file system, you might want to force a single node to write all of the files. To do so, use an empty OVER clause.

Exporting to object stores

Object-store file systems (S3, Google Cloud Storage, and Azure Blob Storage) have some differences from other file systems that affect data export. You must set some additional configuration parameters for authentication and region, and there are some restrictions on the output.

URI formats and configuration parameters are described on the following reference pages:

- S3 object store
- Google Cloud Storage (GCS) object store
- Azure Blob Storage object store

Configuration parameters affect all access to the corresponding object stores, both reads and writes. Instead of setting them globally, you can limit the effects of your settings by setting them at the session level before exporting data.

Output restrictions

Object-store file systems do not support renaming files in place; they implement a rename as a copy followed by a delete. On other file systems, the exporters support atomicity by writing output into a temporary directory and renaming it when complete. Such an approach is impractical for object stores, so the exporters write directly to the destination path. It is therefore possible to begin reading the exported data before the export has finished, which could lead to errors. Be careful to wait for the export to finish before using the data.

Vertica does not support simultaneous exports to the same path in object stores. The results are undefined.

S3 limits buckets to 5TB. You might need to divide very large exports.

Exporting to the Linux file system

Exports to the local file system can be to an NFS mount (shared) or to the Linux file system on each node (non-shared). If you export to an NFS mount on the Linux file system, the exporters behave the same way as for any other shared location: all of the exported files are written to the same (shared) destination.

If you export to the local Linux file system, each Vertica node writes its portion of the export to its local (non-shared) file system. Exports to non-shared local file systems have the following restrictions:

- The output directory must not exist on any node.
- You must have a USER storage location or superuser privileges.
- You cannot override the permissions mode of 700 for directories and 600 for files.

To define an external table to read data exported to a non-shared local file system, use COPY...ON EACH NODE, as in the following example:

```
=> CREATE EXTERNAL TABLE sales (...)
  AS COPY FROM '/data/sales/*.parquet' ON EACH NODE PARQUET;
```

The path is the same path that you specified in the export statement and is the same on all nodes.

If you omit the ON clause and just specify a path, COPY only loads the data it finds on the initiator node.

Monitoring exports

You can review information about exports, including numbers of row groups, file sizes, and file names.

The export statements are UDxs. The UDX_EVENTS system table records events logged during UDx execution, including timestamps, node names, and session IDs. This table contains a column (**RAW**), which holds a VMap of whatever additional data an individual UDx logged. The export statements log details about the exported files in this table. While you can work with this table directly and materialize values from the VMap column, you might prefer to define a view to simplify your access.

The following statement defines a view showing only the events from EXPORT TO PARQUET , materializing the VMap values.

```
=> CREATE VIEW parquet_export_events AS
  SELECT
   report_time,
   node_name,
   session_id,
   user_id,
   user_name,
   transaction_id,
   statement_id,
   request_id,
   udx_name,
   file,
   created,
   closed,
   rows,
   row_groups,
   size_mb
  FROM
   v_monitor.udx_events
  WHERE
   udx_name ilike 'ParquetExport%';
```

The exporters report the following UDx-specific columns:

Column Name	Data Type	Description
FILE	VARCHAR	Name of the output file.
CREATED	TIMESTAMPTZ	When the file was created.
CLOSED	TIMESTAMPTZ	When the file was closed after writing.
ROWS	INTEGER	The total number of rows in the file.
ROW_GROUPS	INTEGER	The number of row groups, for formats that use them (Parquet and ORC).
SIZE_MB	FLOAT	File size.

The following example shows the results of a single export.

```
=> SELECT file,rows,row_groups,size_mb FROM PARQUET_EXPORT_EVENTS;
                                file                                | rows  | row_groups | size_mb
--------------------------------------------------------------------------------+-------+------------+----------
 /data/outgZxN3irt/450c4213-v_vmart_node0001-139770732459776-0.parquet | 29696 | 1          | 0.667203
 /data/outgZxN3irt/9df1c797-v_vmart_node0001-139770860660480-0.parquet | 29364 | 1          | 0.660922
(2 rows)
```

In this table, the output directory name (/data/out) is appended with a generated string (gZxN3irt). For exports to HDFS or to local file systems (including NFS), EXPORT TO PARQUET first writes data into a scratch directory and then renames it at the end of the operation. The events are logged during export and so show the temporary name. Some output destinations, such as AWS S3, do not support rename operations, so in those cases this table does not show generated names.

Database export and import

Vertica can easily import data from and export data to other Vertica databases. Importing and exporting data is useful for common tasks such as moving data back and forth between a development or test database and a production database, or between databases that have different purposes but need to share data on a regular basis.

Moving data directly between databases

To move data between databases you first establish a connection using CONNECT TO VERTICA and then use one of the following statements to move data:

- COPY FROM VERTICA
- EXPORT TO VERTICA

These statements are symmetric; copying from cluster A to cluster B is the same as exporting from cluster B to cluster A. The difference is only in which cluster drives the operation.

To configure TLS settings for the connection, see Configuring connection security between clusters .

Creating SQL scripts to export data

Three functions return a SQL script you can use to export database objects to recreate elsewhere:

- EXPORT_CATALOG
- EXPORT_OBJECTS
- EXPORT_TABLES

While copying and exporting data is similar to Backing up and restoring the database , you should use them for different purposes, outlined below:

Task	Backup and Restore	COPY and EXPORT Statements
Back up or restore an entire database, or incremental changes	YES	NO
Manage database objects (a single table or selected table rows)	YES	YES
Use external locations to back up and restore your database	YES	NO
Use direct connections between two databases	OBJECT RESTORE ONLY	YES
Use external shell scripts to back up and restore your database	YES	NO
Use SQL commands to incorporate copy and export tasks into DB operations	NO	YES

The following sections explain how you import and export data between Vertica databases.

When importing from or exporting to a Vertica database, you can connect only to a database that uses trusted (username only) or password-based authentication, as described in Security and authentication . SSL, Kerberos, and OAuth authentication methods are not supported.

Other exports

This section is about exporting data to another Vertica database. For information about exporting data to files, which can then be used in external tables or COPY statements, see File export .

In this section

- Configuring connection security between clusters
- Exporting data to another database
- Copying data from another Vertica database
- Copy and export data on AWS
- Changing node export addresses
- Using public and private IP networks
- Handling node failure during copy/export
- Using EXPORT functions

Configuring connection security between clusters

When copying data between clusters, Vertica can encrypt both data and plan metadata.

Data is encrypted if you configure internode encryption (see Internode TLS).

For metadata, by default Vertica tries TLS first and falls back to plaintext. You can configure Vertica to require TLS and to fail if the connection cannot be made. You can also have Vertica verify the certificate and hostname before connecting.

Enabling TLS between clusters

To use TLS between clusters, you must first configure TLS between nodes:

1. Set the EncryptSpreadComms parameter.
2. Configure the data_channel TLS Configuration.

3. Set the ImportExportTLSMode parameter.

To specify the level of strictness when connecting to another cluster, set the ImportExportTLSMode configuration parameter. This parameter applies for both importing and exporting data. The possible values are:

- PREFER : Try TLS but fall back to plaintext if TLS fails.
- REQUIRE : Use TLS and fail if the server does not support TLS.
- VERIFY_CA : Require TLS (as with REQUIRE), and also validate the other server's certificate using the CA specified by the "server" TLS Configuration's CA certificates (in this case, "ca_cert" and "ica_cert"):

```
=> SELECT name, certificate, ca_certificate, mode FROM tls_configurations WHERE name = 'server';
 name |  certificate   |  ca_certificate   |  mode
--------+------------------+-------------------+-----------
 server | server_cert      | ca_cert,ica_cert  | VERIFY_CA
(1 row)
```

- VERIFY_FULL : Require TLS and validate the certificate (as with VERIFY_CA), and also validate the server certificate's hostname.
- REQUIRE_FORCE , VERIFY_CA_FORCE , and VERIFY_FULL_FORCE : Same behavior as REQUIRE , VERIFY_CA , and VERIFY_FULL , respectively, and cannot be overridden by CONNECT TO VERTICA .

ImportExportTLSMode is a global parameter that applies to all import and export connections you make using CONNECT TO VERTICA . You can override it for an individual connection.

For more information about these and other configuration parameters, see Security parameters .

Exporting data to another database

EXPORT TO VERTICA exports table data from one Vertica database to another. The following requirements apply:

- You already opened a connection to the target database with CONNECT TO VERTICA .
- The source database is no more than one major release behind the target database.
- The table in the target database must exist.
- Source and target table columns must have the same or compatible data types.

Each EXPORT TO VERTICA statement exports data from only one table at a time. You can use the same database connection for multiple export operations.

Export process

Exporting is a three-step process:

1. Connect to the target database with CONNECT TO VERTICA .
   For example:

```
=> CONNECT TO VERTICA testdb USER dbadmin PASSWORD '' ON 'VertTest01', 5433;
CONNECT
```

2. Export the desired data with EXPORT TO VERTICA. For example, the following statement exports all table data in customer_dimension to a table of the same name in target database testdb :

```
=> EXPORT TO VERTICA testdb.customer_dimension FROM customer_dimension;
Rows Exported
---------------
        23416
(1 row)
```

3. DISCONNECT disconnects from the target database when all export and import operations are complete:

```
=> DISCONNECT testdb;
DISCONNECT
```

> **Note**
> Closing your session also closes the database connection. However, it is a good practice to explicitly close the connection to the other database, both to free up resources and to prevent issues with other SQL scripts that might be running in your session. Always closing the connection prevents potential errors if you run a script in the same session that attempts to open a connection to the same database, since each session can only have one connection to a given database at a time.

Mapping between source and target columns

If you export all table data from one database to another as in the previous example, EXPORT TO VERTICA can omit specifying column lists. This is possible only if column definitions in both tables comply with the following conditions:

- Same number of columns
- Identical column names
- Same sequence of columns
- Matching or compatible column data types

If any of these conditions is not true, the EXPORT TO VERTICA statement must include column lists that explicitly map source and target columns to each other, as follows:

- Contain the same number of columns.
- List source and target columns in the same order.
- Pair columns with the same (or compatible) data types.

For example:

```
=> EXPORT TO VERTICA testdb.people (name, gender, age)
   FROM customer_dimension (customer_name, customer_gender, customer_age);
```

Exporting subsets of table data

In general, you can export a subset of table data in two ways:

- Export data of specific source table columns.
- Export the result set of a query (including historical queries) on the source table.

In both cases, the EXPORT TO VERTICA statement typically must specify column lists for the source and target tables.

The following example exports data from three columns in the source table to three columns in the target table. Accordingly, the EXPORT TO VERTICA statement specifies a column list for each table. The order of columns in each list determines how Vertica maps target columns to source columns. In this case, target columns name , gender , and age map to source columns customer_name , customer_gender , and customer_age , respectively:

```
=> EXPORT TO VERTICA testdb.people (name, gender, age) FROM customer_dimension
(customer_name, customer_gender, customer_age);
Rows Exported
---------------
        23416
(1 row)
```

The next example queries source table customer_dimension , and exports the result set to table ma_customers in target database testdb :

```
=> EXPORT TO VERTICA testdb.ma_customers(customer_key, customer_name, annual_income)
   AS SELECT customer_key, customer_name, annual_income FROM customer_dimension WHERE customer_state = 'MA';
Rows Exported
---------------
         3429
(1 row)
```

> **Note**
> In this example, the source and target column names are identical, so specifying a columns list for target table ma_customers is optional. If one or more of the queried source columns did not have a match in the target table, the statement would be required to include a columns list for the target table.

Exporting IDENTITY columns

You can export tables (or columns) that contain IDENTITY values, but the sequence values are not incremented automatically at the target table. You must use ALTER SEQUENCE to make updates.

Export IDENTITY columns as follows:

- If both source and destination tables have an IDENTITY column and configuration parameter CopyFromVerticaWithIdentity is set to true (1), you do not need to list them.

- If source table has an IDENTITY column, but target table does not, you must explicitly list the source and target columns.

> **Caution**
> Failure to list which IDENTITY columns to export can cause an error, because the IDENTITY column will be interpreted as missing in the destination table.

By default, EXPORT TO VERTICA exports all IDENTITY columns . To disable this behavior globally, set the CopyFromVerticaWithIdentity configuration parameter.

Copying data from another Vertica database

COPY FROM VERTICA imports table data from one Vertica database to another. The following requirements apply:

- You already opened a connection to the target database with CONNECT TO VERTICA .
- The source database is no more than one major release behind the target database.
- The table in the target database must exist.
- Source and target table columns must have the same or compatible data types.

Import process

Importing is a three-step process:

1. Connect to the source database with CONNECT TO VERTICA . For example:

```
=> CONNECT TO VERTICA vmart USER dbadmin PASSWORD '' ON 'VertTest01',5433;
CONNECT
```

2. Import the desired data with COPY FROM VERTICA. For example, the following statement imports all table data in customer_dimension to a table of the same name:

```
=> COPY customer_dimension FROM  VERTICA vmart.customer_dimension;
 Rows Loaded
-------------
    500000
(1 row)
=> DISCONNECT vmart;
DISCONNECT
```

> **Note**
> Successive COPY FROM VERTICA statements in the same session can import data from multiple tables over the same connection.

3. DISCONNECT disconnects from the source database when all import and export operations are complete:

```
=> DISCONNECT vmart;
DISCONNECT
```

> **Note**
> Closing your session also closes the database connection. However, it is a good practice to explicitly close the connection to the other database, both to free up resources and to prevent issues with other SQL scripts that might be running in your session. Always closing the connection prevents potential errors if you run a script in the same session that attempts to open a connection to the same database, since each session can only have one connection to a given database at a time.

Importing IDENTITY columns

You can import IDENTITY columns as follows:

- If both source and destination tables have an IDENTITY column and configuration parameter CopyFromVerticaWithIdentity is set to true (1), you do not need to list them.
- If source table has an IDENTITY column, but target table does not, you must explicitly list the source and target columns.

> **Caution**
> Failure to list which IDENTITY columns to export can cause an error, because the IDENTITY column will be interpreted as missing in the destination table.

After importing the columns, the IDENTITY column values do not increment automatically. Use ALTER SEQUENCE to make updates.

The default behavior for this statement is to import IDENTITY columns by specifying them directly in the source table. To disable this behavior globally, set the CopyFromVerticaWithIdentity configuration parameter.

Copy and export data on AWS

There are common issues that occur when exporting or copying on AWS clusters, as described below. Except for these specific issues as they relate to AWS, copying and exporting data works as documented in Database export and import .

To copy or export data on AWS:

1. **Verify that all nodes in source and destination clusters have their own elastic IPs (or public IPs) assigned.**
   If your destination cluster is located within the same VPC as your source cluster, proceed to step 3. Each node in one cluster must be able to communicate with each node in the other cluster. Thus, each source and destination node needs an elastic IP (or public IP) assigned.
2. (For non-CloudFormation Template installs) Create an S3 gateway endpoint.
   If you aren't using a CloudFormation Template (CFT) to install Vertica, you must create an S3 gateway endpoint in your VPC. For more information, see the AWS documentation .
   For example, the Vertica CFT has the following VPC endpoint:

```
"S3Enpoint" : {
   "Type" : "AWS::EC2::VPCEndpoint",
   "Properties" : {
   "PolicyDocument" : {
      "Version":"2012-10-17",
      "Statement":[{
      "Effect":"Allow",
      "Principal": "*",
      "Action":["*"],
      "Resource":["*"]
      }]
   },
   "RouteTableIds" : [ {"Ref" : "RouteTable"} ],
   "ServiceName" : { "Fn::Join": [ "", [ "com.amazonaws.", { "Ref": "AWS::Region" }, ".s3" ] ] },
   "VpcId" : {"Ref" : "VPC"}
}
```

3. **Verify that your security group allows the AWS clusters to communicate.**
   Check your security groups for both your source and destination AWS clusters. Verify that ports 5433 and 5434 are open. If one of your AWS clusters is on a separate VPC, verify that your network access control list (ACL) allows communication on port 5434.

> **Note**
> **Note:** This communication method exports and copies (imports) data across the Internet. You can alternatively use non-public IPs and gateways, or VPN to connect the source and destination clusters.

4. If there are one or more elastic load balancers (ELBs) between the clusters, verify that port 5433 is open between the ELBs and clusters.
5. If you use the Vertica client to connect to one or more ELBs, the ELBs only distribute incoming connections. The data transmission path occurs between clusters.

Changing node export addresses

You can change the export address for your Vertica cluster. You might need to do so to export data between clusters in different network subnets.

1. Create a subnet for importing and exporting data between Vertica clusters. The CREATE SUBNET statement identifies the public network IP addresses residing on the same subnet.

```
=> CREATE SUBNET kv_subnet with '10.10.10.0';
```

2. Alter the database to specify the subnet name of a public network for import/export.

```
=> ALTER DATABASE DEFAULT EXPORT ON kv_subnet;
```

3. Create network interfaces for importing and exporting data from individual nodes to other Vertica clusters. The CREATE NETWORK INTERFACE statement identifies the public network IP addresses residing on multiple subnets.

```
=> CREATE NETWORK INTERFACE kv_node1 on v_VMartDB_node0001 with '10.10.10.1';
=> CREATE NETWORK INTERFACE kv_node2 on v_VMartDB_node0002 with '10.10.10.2';
=> CREATE NETWORK INTERFACE kv_node3 on v_VMartDB_node0003 with '10.10.10.3';
=> CREATE NETWORK INTERFACE kv_node4 on v_VMartDB_node0004 with '10.10.10.4';
```

   For users on Amazon Web Services (AWS) or using Network Address Translation (NAT), refer to Vertica on Amazon Web Services .

4. Alter the node settings to change the export address. When used with the EXPORT ON clause, the ALTER NODE specifies the network interface of the public network on individual nodes for importing and exporting data.

```
=> ALTER NODE v_VMartDB_node0001 export on kv_node1;
=> ALTER NODE v_VMartDB_node0002 export on kv_node2;
=> ALTER NODE v_VMartDB_node0003 export on kv_node3;
=> ALTER NODE v_VMartDB_node0004 export on kv_node4;
```

5. Verify if the node address and the export address are different on different network subnets of the Vertica cluster.

```
=> SELECT node_name, node_address, export_address FROM nodes;
    node_name     | node_address    | export_address
------------------+-----------------+----------------
v_VMartDB_node0001 | 192.168.100.101 | 10.10.10.1
v_VMartDB_node0002 | 192.168.100.102 | 10.10.10.2
v_VMartDB_node0003 | 192.168.100.103 | 10.10.10.3
v_VMartDB_node0004 | 192.168.100.104 | 10.10.10.4
```

Creating a network interface and altering the node settings to change the export address takes precedence over creating a subnet and altering the database for import/export.

Using public and private IP networks

In many configurations, Vertica cluster hosts use two network IP addresses as follows:

- A private address for communication between the cluster hosts.
- A public IP address for communication with client connections.

By default, importing from and exporting to another Vertica database uses the private network.

> **Note**
> Ensure port 5433 or the port the Vertica database is using is not blocked.

To use the public network address for copy and export activities, as well as moving large amounts of data, configure the system to use the public network to support exporting to or importing from another Vertica cluster:

- Identify the Public Network to Vertica
- Identify the database or nodes used for import/export

Vertica encrypts data during transmission (if you have configured a certificate). Vertica attempts to also encrypt plan metadata but, by default, falls back to plaintext if needed. You can configure Vertica to require encryption for metadata too; see Configuring connection security between clusters .

In certain instances, both public and private addresses exceed the demand capacity of a single Local Area Network (LAN). If you encounter this type of scenario, then configure your Vertica cluster to use two LANs: one for public network traffic and one for private network traffic.

In this section

- Identify the public network to Vertica
- Identify the database or nodes used for import/export

Identify the public network to Vertica

To be able to import to or export from a public network, Vertica needs to be aware of the IP addresses of the nodes or clusters on the public network that will be used for import/export activities. Your public network might be configured in either of these ways:

- Public network IP addresses reside on the same subnet (create a subnet)
- Public network IP addresses are on multiple subnets (create a network interface)

To identify public network IP addresses residing on the same subnet:

- Use the CREATE SUBNET statement provide your subnet with a name and to identify the subnet routing prefix.

To identify public network IP addresses residing on multiple subnets:

- Use the CREATE NETWORK INTERFACE statement to configure import/export from specific nodes in the Vertica cluster.

After you've identified the subnet or network interface to be used for import/export, you must Identify the Database Or Nodes Used For Import/Export.

See also

- CREATE SUBNET
- ALTER SUBNET
- DROP SUBNET
- CREATE NETWORK INTERFACE
- ALTER NETWORK INTERFACE
- DROP NETWORK INTERFACE

Identify the database or nodes used for import/export

After you identify the public network to Vertica, you can configure a database and its nodes to use it for import and export operations:

- Use ALTER DATABASE to specify a subnet on the public network for the database. After doing so, all nodes in the database automatically use the network interface on the subnet for import/export operations.
- On each database node, use ALTER NODE to specify a network interface of the public network.

See also

- CREATE PROCEDURE (external)
- CREATE NETWORK ADDRESS
- NETWORK_INTERFACES

Handling node failure during copy/export

When an export (EXPORT TO VERTICA) or import from Vertica (COPY FROM VERTICA) task is in progress, and a non-initiator node fails, Vertica does not complete the task automatically. A non-initiator node is any node that is not the source or target node in your export or import statement. To complete the task, you must run the statement again.

You address the problem of a non-initiator node failing during an import or export as follows:

> **Note**
> Both Vertica databases must be running in a safe state.

1. You export or import from one cluster to another using the EXPORT TO VERTICA or COPY FROM VERTICA statement.
   During the export or import, a non-initiating node on the target or source cluster fails. Vertica issues an error message that indicates possible node failure, one of the following:
   - ERROR 4534: Receive on v_tpchdb1_node0002: Message receipt from v_tpchdb2_node0005 failed
   - WARNING 4539: Received no response from v_tpchdb1_node0004 in abandon plan
   - ERROR 3322: [tpchdb2] Execution canceled by operator
2. Complete your import or export by running the statement again. The failed node does not need to be up for Vertica to successfully complete the export or import.

Using EXPORT functions

Vertica provides several EXPORT_ functions that let you recreate a database, or specific schemas and tables, in a target database. For example, you can use the EXPORT_ functions to transfer some or all of the designs and objects you create in a development or test environment to a production database.

The EXPORT_ functions create SQL scripts that you can run to generate the exported database designs or objects. These functions serve different purposes to the export statements, COPY FROM VERTICA (pull data) and EXPORT TO VERTICA (push data). These statements transfer data directly from source to target database across a network connection between both. They are dynamic actions and do not generate SQL scripts.

The EXPORT_ functions appear in the following table. Depending on what you need to export, you can use one or more of the functions. EXPORT_CATALOG creates the most comprehensive SQL script, while EXPORT_TABLES and EXPORT_OBJECTS are subsets of that function to narrow the export scope.

Use this function...	**To recreate...**
EXPORT_CATALOG	These catalog items: • An existing schema design, tables, projections, constraints, views, and stored procedures. • The Database Designer-created schema design, tables, projections, constraints, and views • A design on a different cluster.
EXPORT_TABLES	Non-virtual objects up to, and including, the schema of one or more tables.
EXPORT_OBJECTS	Catalog objects in order dependency for replication.

The designs and object definitions that the script creates depend on the EXPORT_ function scope you specify. The following sections give examples of the commands and output for each function and the scopes it supports.

Saving scripts for export functions

All of the examples in this section were generated using the standard Vertica VMART database, with some additional test objects and tables. One output directory was created for all SQL scripts that the functions created:

```
/home/dbadmin/xtest
```

If you specify the destination argument as an empty string (''), the function writes the export results to STDOUT.

> **Note**
> A superuser can export all available database output to a file with the EXPORT_ functions. For a non-superuser, the EXPORT_ functions generate a script containing only the objects to which the user has access.

In this section

- Exporting the catalog
- Exporting tables
- Exporting objects

Exporting the catalog

Vertica function EXPORT_CATALOG generates a SQL script for copying a database design to another cluster. This script replicates the physical schema design of the source database. You call this function as follows:

```
EXPORT_CATALOG ( ['[destination]' [, '[scope]']] )
```

The SQL script conforms to the following requirements:

- Only includes objects to which the user has access.
- Orders CREATE statements according to object dependencies so they can be recreated in the correct sequence. For example, if a table is in a non-PUBLIC schema, the required CREATE SCHEMA statement precedes the CREATE TABLE statement. Similarly, a table's CREATE ACCESS POLICY statement follows the table's CREATE TABLE statement.
- If possible, creates projections with their KSAFE clause, if any, otherwise with their OFFSET clause.

Setting export scope

If you omit specifying a scope, EXPORT_CATALOG exports all objects. You can set the scope of the export operation to one of the following levels:

Scope	**Exports...**

TABLES	Tables, schemas, and table-dependent objects: constraints, and access policies
DESIGN	All catalog objects: schemas, tables, constraints, views, access policies, projections, SQL macros, and stored procedures.
DIRECTED_QUERIES	All directed queries that are stored in the catalog. For details, see Managing directed queries .

Exporting table objects

Use the TABLES scope to generate a script that recreates all tables and the objects that they depend on: schemas, sequences, constraints, and access policies:

```
=> SELECT EXPORT_CATALOG (
    '/home/dbadmin/xtest/sql_cat_tables.sql',
    'TABLES');
        EXPORT_CATALOG
-------------------------------------
 Catalog data exported successfully
(1 row)
```

The SQL script can include the following statements:

- CREATE SCHEMA
- CREATE TABLE
- ALTER TABLE (constraints)
- CREATE SEQUENCE
- CREATE ACCESS POLICY
- CREATE PROCEDURE (Stored)

Exporting all catalog objects

Use the DESIGN scope to export all design elements of a source database in order of their dependencies. This scope exports all catalog objects, including schemas, tables, constraints, projections, views, and access policies.

```
=> SELECT EXPORT_CATALOG(
    '/home/dbadmin/xtest/sql_cat_design.sql',
    'DESIGN' );
        EXPORT_CATALOG
-------------------------------------
 Catalog data exported successfully
(1 row)
```

The SQL script include statements that are required to recreate the database:

- CREATE SCHEMA
- CREATE TABLE
- ALTER TABLE (constraints)
- CREATE VIEW
- CREATE SEQUENCE
- CREATE ACCESS
- CREATE PROJECTION (with ORDER BY and SEGMENTED BY)

Projection considerations

If a projection to export was created with no ORDER BY clause, the SQL script reflects the default behavior for projections. Vertica implicitly creates projections using a sort order based on the SELECT columns in the projection definition.

The EXPORT_CATALOG script is portable if all projections are created using UNSEGMENTED ALL NODES or SEGMENTED ALL NODES.

See also

- Exporting objects
- Exporting tables

Exporting tables

Vertica function EXPORT_TABLES exports DDL for tables and related objects in the current database. The generated SQL includes all non-virtual table objects to which you have access. You can use this SQL to recreate tables and related non-virtual objects on a different cluster.

You execute EXPORT_TABLES as follows:

```
EXPORT_TABLES( ['[destination]' [, '[scope]']] )
```

The SQL script conforms to the following requirements:

- Only includes objects to which the user has access.
- Orders CREATE statements according to object dependencies so they can be recreated in the correct sequence. For example, if a table references a named sequence, a CREATE SEQUENCE statement precedes the CREATE TABLE statement. Similarly, a table's CREATE ACCESS POLICY statement follows the table's CREATE TABLE statement.

Setting export scope

The EXPORT_TABLES *scope* argument specifies the scope of the export operation:

To export...	**Set scope to...**
All tables to which you have access, including constraints	Empty string (' ')
One or more named tables	Comma-delimited list of table objects. For example: myschema.newtable, yourschema.oldtable You can optionally qualify the schema with the name of the current database: mydb.myschema.newtable
A named table object in the current search path: a schema, table, or sequence. If you specify a schema, EXPORT_TABLES exports all table objects in that schema to which you have access.	Table object's name and, optionally, its path: VMart.myschema

> **Note**
> EXPORT_TABLES does not export views. If you specify a view name, Vertica silently ignores it and the view is omitted from the generated script. To export views, use EXPORT_OBJECTS .

Exporting all table objects

If you set the scope parameter to an empty string (''), EXPORT_TABLES exports all tables and their related objects. For example, the following call to EXPORT_TABLES exports all table objects in the VMart database to the specified output file.

```
=> SELECT EXPORT_TABLES(
    '/home/dbadmin/xtest/sql_tables_empty.sql', '');
        EXPORT_TABLES
---------------------------------------
 Catalog data exported successfully
(1 row)
```

The exported SQL includes the following types of statements, depending on what is required to recreate the tables and related objects, such as schemas, sequences and access policies:

- CREATE SCHEMA
- CREATE TABLE
- CREATE SEQUENCE
- CREATE ACCESS POLICY
- ALTER TABLE (to add foreign key constraints)

Exporting individual table objects

EXPORT_TABLES can specify a comma-separated list of tables and table-related objects such as sequences or schemas to export. The generated SQL script includes the CREATE statements for the specified objects and their dependent objects:

- CREATE SCHEMA
- CREATE TABLE
- CREATE SEQUENCE
- CREATE ACCESS POLICY
- ALTER TABLE (to add foreign keys)

For example, the following call to EXPORT_TABLES exports two VMart tables: store.store_sales_fact and store.store_dimension :

```
=> SELECT export_tables('','store.store_sales_fact, store.store_dimension');
                            export_tables
------------------------------------------------------------------------------

CREATE TABLE store.store_dimension
(
    store_key int NOT NULL,
    store_name varchar(64),
    store_number int,
    store_address varchar(256),
    store_city varchar(64),
    store_state char(2),
    store_region varchar(64),
    floor_plan_type varchar(32),
    photo_processing_type varchar(32),
    financial_service_type varchar(32),
    selling_square_footage int,
    total_square_footage int,
    first_open_date date,
    last_remodel_date date,
    number_of_employees int,
    annual_shrinkage int,
    foot_traffic int,
    monthly_rent_cost int,
    CONSTRAINT C_PRIMARY PRIMARY KEY (store_key) DISABLED
);

CREATE TABLE store.store_sales_fact
(
    date_key int NOT NULL,
    product_key int NOT NULL,
    product_version int NOT NULL,
    store_key int NOT NULL,
    promotion_key int NOT NULL,
    customer_key int NOT NULL,
    employee_key int NOT NULL,
    pos_transaction_number int NOT NULL,
    sales_quantity int,
    sales_dollar_amount int,
    cost_dollar_amount int,
    gross_profit_dollar_amount int,
    transaction_type varchar(16),
    transaction_time time,
    tender_type varchar(8)
);

ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_date FOREIGN KEY (date_key) references public.date_dimension (date_key);
ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_product FOREIGN KEY (product_key, product_version) references
public.product_dimension (product_key, product_version);
ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_store FOREIGN KEY (store_key) references store.store_dimension (store_key);
ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_promotion FOREIGN KEY (promotion_key) references
public.promotion_dimension (promotion_key);
ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_customer FOREIGN KEY (customer_key) references public.customer_dimension
(customer_key);
ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_employee FOREIGN KEY (employee_key) references public.employee_dimension
(employee_key);
```

The following call to EXPORT_TABLES specifies to export all tables in the VMart schema store :

```
=> select export_tables('','store');
                        export_tables
-------------------------------------------------------------------------
CREATE SCHEMA store;

CREATE TABLE store.store_dimension
(
 ...
);

CREATE TABLE store.store_sales_fact
(
 ...
);

CREATE TABLE store.store_orders_fact
(
  ...
);

ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_date FOREIGN KEY (date_key) references public.date_dimension (date_key);
ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_product FOREIGN KEY (product_key, product_version) references
public.product_dimension (product_key, product_version);
ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_store FOREIGN KEY (store_key) references store.store_dimension (store_key);
ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_promotion FOREIGN KEY (promotion_key) references
public.promotion_dimension (promotion_key);
ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_customer FOREIGN KEY (customer_key) references public.customer_dimension
(customer_key);
ALTER TABLE store.store_sales_fact ADD CONSTRAINT fk_store_sales_employee FOREIGN KEY (employee_key) references public.employee_dimension
(employee_key);
ALTER TABLE store.store_orders_fact ADD CONSTRAINT fk_store_orders_product FOREIGN KEY (product_key, product_version) references
public.product_dimension (product_key, product_version);
ALTER TABLE store.store_orders_fact ADD CONSTRAINT fk_store_orders_store FOREIGN KEY (store_key) references store.store_dimension (store_key);
ALTER TABLE store.store_orders_fact ADD CONSTRAINT fk_store_orders_vendor FOREIGN KEY (vendor_key) references public.vendor_dimension
(vendor_key);
ALTER TABLE store.store_orders_fact ADD CONSTRAINT fk_store_orders_employee FOREIGN KEY (employee_key) references
public.employee_dimension (employee_key);

(1 row)
```

See also

- Exporting objects
- Exporting the catalog

Exporting objects

The Vertica function EXPORT_OBJECTS generates a SQL script that you can use to recreate non-virtual catalog objects on a different cluster, as follows:

```
EXPORT_OBJECTS( ['[destination]' [, '[scope]'] [, 'mark-ksafe']] )
```

The SQL script conforms to the following requirements:

- Only includes objects to which the user has access.
- Orders CREATE statements according to object dependencies so they can be recreated in the correct sequence. For example, if a table is in a non-PUBLIC schema, the required CREATE SCHEMA statement precedes the CREATE TABLE statement. Similarly, a table's CREATE ACCESS POLICY statement follows the table's CREATE TABLE statement.
- If possible, creates projections with their KSAFE clause, if any, otherwise with their OFFSET clause.

Setting export scope

The EXPORT_OBJECTS *scope* argument specifies the scope of the export operation:

To export...	Set scope to...
All objects to which you have access	Empty string ('')
One or more named database objects and related objects	Comma-delimited list of objects. For example: myschema.newtable, yourschema.my-sequence You can optionally qualify the schema with the name of the current database: mydb.myschema.newtable If you specify a schema, EXPORT_TABLES exports all objects in that schema to which you have access. If you name a table that references a sequence, the generated script shows the sequence, then the table that references the sequence, and finally any projections of that table.

Note
EXPORT_OBJECTS does not export grants. Preserving grants on libraries can be especially important when you upgrade your database: if the prototypes of UDx libraries change, Vertica drops the grants on them. In order to preserve grants on UDx libraries, back up the grants before upgrading, and then restore them in the upgraded database. For details, see Backing up and restoring grants .

Exporting all objects

If you set the scope parameter to an empty string (''), Vertica exports all non-virtual objects from the source database in order of dependencies. Running the generated SQL script on another cluster creates all referenced objects and their dependent objects.

By default, the function's KSAFE argument is set to true. In this case, the generated script calls MARK_DESIGN_KSAFE, which replicates K-safety of the original database.

```
=> SELECT EXPORT_OBJECTS(
    '/home/dbadmin/xtest/sql_objects_all.sql',
    '',
    'true');
       EXPORT_OBJECTS
----------------------------------------
 Catalog data exported successfully
(1 row)
```

The SQL script includes the following types of statements:

- CREATE SCHEMA
- CREATE TABLE
- ALTER TABLE (constraints)
- CREATE PROJECTION
- CREATE VIEW
- CREATE SEQUENCE
- CREATE ACCESS POLICY
- CREATE PROCEDURE (stored)
- CREATE SCHEDULE
- CREATE TRIGGER

The following output includes the start and end of the output SQL file, including the MARK_DESIGN_KSAFE statement:

```
CREATE SCHEMA store;
CREATE SCHEMA online_sales;
CREATE SEQUENCE public.my_seq ;
CREATE TABLE public.customer_dimension
(
    customer_key int NOT NULL,
    customer_type varchar(16),
    customer_name varchar(256),
    customer_gender varchar(8),
    title varchar(8),
    household_id int,
...

);
...
SELECT MARK_DESIGN_KSAFE(1);
```

Exporting individual objects

You can specify one or more objects as the function scope, where multiple objects are specified in a comma-delimited list. The names of objects in any non-PUBLIC schema must include their respective schemas. The objects to export can include schemas, tables, views, and sequences. Accordingly, the SQL script includes the following statements, depending on what objects you list and their dependencies:

- CREATE SCHEMA
- CREATE TABLE
- ALTER TABLE (to add constraints)
- CREATE VIEW
- CREATE SEQUENCE
- CREATE ACCESS POLICY
- CREATE PROCEDURE (Stored)
- CREATE SCHEDULE
- CREATE TRIGGER

If listed objects have overlapping scopes—for example, the list includes a table and one of its projections—EXPORT_OBJECTS exports the projection only once:

```
=> select export_objects ('','customer_dimension, customer_dimension_super');

                            export_objects
------------------------------------------------------------------------------
CREATE TABLE public.customer_dimension
(
    customer_key int NOT NULL,
    customer_type varchar(16),
    customer_name varchar(256),
    ...
    CONSTRAINT C_PRIMARY PRIMARY KEY (customer_key) DISABLED
);

CREATE ACCESS POLICY ON public.customer_dimension FOR COLUMN customer_address CASE WHEN enabled_role('administrator') THEN
customer_address ELSE '***************' END ENABLE;

CREATE PROJECTION public.customer_dimension_super /*+basename(customer_dimension),createtype(L)*/
(
 customer_key,
 customer_type,
 customer_name,
  ...
)
AS
 SELECT customer_dimension.customer_key,
        customer_dimension.customer_type,
        customer_dimension.customer_name,
        ...
 FROM public.customer_dimension
 ORDER BY customer_dimension.customer_key
SEGMENTED BY hash(customer_dimension.customer_key) ALL NODES OFFSET 0;

SELECT MARK_DESIGN_KSAFE(0);
```

You can export stored procedures by specifying their name and the types of their formal parameters. For stored procedures with the same name but different formal parameters, you can export all implementations by exporting its parent schema.

To export a particular implementation, specify either the types or both the names and types of its formal parameters. The following example specifies the types:

```
=> SELECT EXPORT_OBJECTS('','raiseXY(int, int)');
   EXPORT_OBJECTS
----------------------

CREATE PROCEDURE public.raiseXY(x int, y int)
LANGUAGE 'PL/vSQL'
SECURITY INVOKER
AS '
BEGIN
RAISE NOTICE "x = %", x;
RAISE NOTICE "y = %", y;
-- some processing statements
END
';

SELECT MARK_DESIGN_KSAFE(0);

(1 row)
```

To export all implementations of the overloaded stored procedure raiseXY, export its parent schema:

```
=> SELECT EXPORT_OBJECTS('','public');
   EXPORT_OBJECTS
----------------------


...

CREATE PROCEDURE public.raiseXY(x int, y varchar)
LANGUAGE 'PL/vSQL'
SECURITY INVOKER
AS '
BEGIN
RAISE NOTICE "x = %", x;
RAISE NOTICE "y = %", y;
-- some processing statements
END
';

CREATE PROCEDURE public.raiseXY(x int, y int)
LANGUAGE 'PL/vSQL'
SECURITY INVOKER
AS '
BEGIN
RAISE NOTICE "x = %", x;
RAISE NOTICE "y = %", y;
-- some processing statements
END
';

SELECT MARK_DESIGN_KSAFE(0);

(1 row)
```

See also

Exporting tables

Management Console

Management Console (MC) is the Vertica in-browser monitoring and management tool. Its graphical user interface provides a unified view of your Vertica database operations. Through user-friendly, step-by-step screens, you can create, configure, manage, and monitor your Vertica databases and their associated clusters. You can use MC to operate your Vertica database in Eon Mode or in Enterprise Mode. You can use MC to provision and deploy a Vertica Eon Mode database.

To get started using MC, see Getting started with MC .

What you can do with Management Console

Use Management Console to create:

- A database cluster on hosts that do not have Vertica installed.
- Multiple Vertica databases on one or more clusters from a single point of control.
- MC users and grant them access to MC and databases managed by MC

Use Management Console to configure:

- Database parameters and user settings dynamically.
- Resource pools.

Use Management Console to monitor:

- License usage and conformance.
- Dynamic metrics about your database cluster.
- Resource Pools.
- User information and activity on MC.
- Alerts by accessing a single message box of alerts for all managed databases.
- Recent databases and clusters through a quick link.

- Multiple Vertica databases on one or more clusters from a single point of control

You an import and export:

- Export all database messages or log/query details to a file.
- Import multiple Vertica databases on one or more clusters from a single point of control.

Management Console provides some, but not all, the functionality that Administration tools provides. Management Console also includes extended functionality not available in admintools. This additional functionality includes a graphical view of your Vertica database and detailed monitoring charts and graphs. See Administration Tools and Management Console for more information.

Getting Management Console

Download the Vertica server RPM and the MC package from myVertica Portal. You then have two options:

- Install Vertica and MC at the command line and import one or more Vertica database clusters into the MC interface
- Install Vertica directly through MC

In this section

- Installing Management Console
- Management Console architecture
- Configuring Management Console
- Getting started with MC
- Users, roles, and privileges
- Database management
- Cloud platforms
- Monitoring with MC
- Troubleshooting with MC diagnostics
- Uninstalling Management Console

Installing Management Console

Management Console (MC) 12.0.x is compatible with the latest supported versions of Vertica server. Read the following documents for more information:

- Vertica server and Management Console. Lists supported platforms and browsers for Management Console.
- Installation overview and checklist. Make sure you have everything ready for your Vertica configuration.
- Before you install Vertica. Read for required prerequisites for *all* Vertica configurations, including Management Console.

Cloud installations

Vertica supports Management Console for both Enterprise Mode and Eon Mode on supported cloud providers.

Amazon Web Services

Deploy a Management Console instance on AWS with a CloudFormation Template (CFT). After you deploy the MC instance, you can provision an Eon Mode or Enterprise Mode database with AWS resources.

In some environments, you might automate custom MC deployments without the AWS CFT. To automate a custom MC, add the following properties to the /opt/vconsole/config/console.properties file:

```
vertica.database.mode = eon
cloud.authentication = IAM
mc.iaas = AWS
provisioning.service.enabled = true
```

> **Note**
> Set the cloud.authentication property only if you use IAM roles to authenticate requests. You do not need to set this property if you use access keys.

Additionally, complete the following to deploy custom MC instances:

- Set the required ports for Vertica and the Management Console. For details, see General Requirements.
- Set the IAM role for Vertica and the MC instance. For details, see AWS authentication.

Microsoft Azure

Deploy a Management Console instance from the Azure Marketplace . After you deploy the MC instance, you can provision an Eon Mode database, or monitor an Enterprise Mode database.

Google Cloud Platform

Deploy a Management Console instance on GCP . After you deploy the MC instance, you can provision an Eon Mode database on GCP. For details, see Provision an Eon Mode cluster and database on GCP in MC .

Enterprise Mode or Eon Mode on-premises

Before you install Management Console, review the following hardware, software, and network requirements.

General requirements

- **Ports** : Management Console has the following port requirements:
  - Port 22 is the default SSH port. The Create Cluster Wizard uses SSH on port 22 when you use MC to create a Vertica.
  - Port 5444 is the default agent port and must be available for MC-to-node and node-to-node communications.
  - Port 5450 is the default MC port and must be available for node-to-MC communications.

  See Reserved ports for more information about port and firewall considerations.
- **Firewall** : Make sure that a firewall or iptables are not blocking communications between the cluster's database, Management Console, and Management Console agents on each cluster node
- **IP Address** : If you install MC on a server outside the Vertica cluster it will be monitoring, that server must be accessible to at least the public network interfaces on the cluster.
- **TLS Requirements** : The Schannel package must be installed on your Linux environment so TLS can be set up during the MC configuration process. See TLS protocol .
- **File Permissions** : On your local workstation, you must have at least read/write privileges on any files you plan to upload to MC through the Cluster Installation Wizard . These files include the Vertica server package, the license key (if needed), the private key file, and an optional CSV file of IP addresses.
- **Perl** : The MC cluster installer requires Perl 5 .

Hardware requirements

You can install MC on any node in the cluster, or its own dedicated node. When running the MC on a node in the cluster, note that MC shares RAM and time on CPU cores with other Vertica processes.

The following table provides minimum and recommended hardware requirements:

Requirements	CPU	RAM	Disk Space
Minimum	4-core	4G	2G
Recommended	8-core	8G	2G

See Disk Space Requirements for Vertica .

Time synchronization and MC's self-signed certificate

When you connect to MC through a client browser, Vertica assigns each HTTPS request a self-signed certificate, which includes a timestamp. To increase security and protect against password replay attacks, the timestamp is valid for several seconds only, after which it expires.

To avoid being blocked out of MC, synchronize time on the hosts in your Vertica cluster, and on the MC host if it resides on a dedicated server. To recover from loss or lack of synchronization, resync system time and the Network Time Protocol.

Installing Management Console

Install Management Console (MC) to manage a new cluster, or add it to an existing cluster.

1. Download the MC package from the Vertica website :
   vertica-console-*current-version.Linux-distro*)

   Save the package to a location on the target server, such as /tmp .
2. On the target server, log in as root or a user with sudo privileges.
3. Change to the directory where you saved the MC package.
4. Install MC using your local Linux distribution package management system—rpm, yum, zypper, apt, dpkg. For example:
   Red Hat 8
   ```
   # rpm -Uvh vertica-console-current-version.x86_64.rpm
   ```

   Debian and Ubuntu

```
# dpkg -i vertica-console-current-version.deb
```

5. If you stopped the database before upgrading MC, restart the database.
   As the root user, use the following command:
   ```
   /etc/init.d/verticad start
   ```
   For versions of Red Hat 8/CentOS 8 and above, run:
   ```
   # systemctl start verticad
   ```
6. Open a browser and enter the URL of the MC installation, one of the following:
   - IP address:
     ```
     https://ip-address:mc-port/
     ```
   - Server host name:
     ```
     https://hostname:mc-port/
     ```

   By default, *mc-port* is 5450.

If MC was not previously configured, the Configuration Wizard dialog box appears. Configuration steps are described in Configuring MC .

If MC was previously configured, Vertica prompts you to accept the end-user license agreement (EULA) when you first log in to MC after the upgrade.

Additionally, you can choose to provide Vertica with analytic information about your MC usage. For details, see Management Console settings .

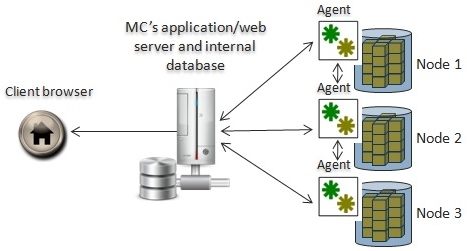
Management Console architecture

MC accepts HTTP requests from a client web browser, gathers information from the Vertica database cluster, and returns that information to the browser for monitoring.

MC components

The primary components that drive Management Console are an application/web server and agents that get installed on each node in the Vertica cluster.

The following diagram is a logical representation of MC, the MC user's interface, and the database cluster nodes.



Application/web server

The application server hosts MC's web application and uses port 5450 for node-to-MC communication and to perform the following:

- Manage one or more Vertica database clusters
- Send rapid updates from MC to the web browser
- Store and report MC metadata, such as alerts and events, current node state, and MC users, on a lightweight, embedded (Derby) database
- Retain workload history

MC agents

MC agents are internal daemon process that run on each Vertica cluster node. The default agent port, 5444, must be available for MC-to-node and node-to-node communications. Agents monitor MC-managed Vertica database clusters and communicate with MC to provide the following functionality:

- Provide local access, command, and control over database instances on a given node, using functionality similar to Administration Tools.
- Report log-level data from the Administration Tools and Vertica log files.
- Cache details from long-running jobs—such as create/start/stop database operations—that you can view through your browser.
- Track changes to data-collection and monitoring utilities and communicate updates to MC .
- Communicate between all cluster nodes and MC through a webhook subscription, which automates information sharing and reports on cluster-specific issues like node state, alerts,and events.

Configuring Management Console

After you complete the steps in Installing Management Console (MC), you need to configure it through a client browser connection. An MC configuration assistant walks you through creating the Linux MC super administrator account, storage locations, and other settings that MC needs to run. Information you provide during the configuration process is stored in the /opt/vconsole/config/console.properties file.

If you need to change settings after the configuration assistant ends, such as port assignments, use the **Home** > **MC Settings** page.

How to configure MC

1. Open a browser session.
2. Enter the IP address or host name of the server on which you installed MC and include the default MC port 5450. For example, enter the following using a IP address:

   ```
   https://10.20.30.40:5450/
   ```

   Or enter the following for a host name:

   ```
   https://hostname:5450/
   ```

3. Follow the configuration assistant.

About authentication for the MC super administrator

In the final step of the configuration process, you choose an authentication method for the MC super administrator. You can decide to have MC authenticate the MC super and complete the process, or you can choose LDAP.

If you choose LDAP, provide the following information for the newly-created MC super administrator:

- Corporate LDAP service host (IP address or host name).
- LDAP server running port (default 389).
- LDAP DN (distinguished name) for base search/lookup/authentication criteria. At a minimum, specify the dc (domain component) field. For example: dc=vertica, dc=com generates a unique identifier of the organization, like the corporate Web URL vertica.com
- Default search path for the organization unit (ou). For example: ou=sales, ou=engineering .
- Search attribute for the user name (uid), common name (cn), and so on. For example, uid=jdoe, cn=Jane Doe .
- Binding DN and password for the MC super administrator.
- In most cases, you provide the "Bind as administrator" fields, information used to establish the LDAP service connection for all LDAP operations, like search. Instead of using the administrator user name and password, the MC administrator could use his or her own LDAP credentials, as long as that user has search privileges.

If you choose bind anonymously

Unless you specifically configure the LDAP server to deny anonymous binds, the underlying LDAP protocol will not cause MC's Configure Authentication process to fail if you choose "Bind anonymously" for the MC administrator. Before you use anonymous bindings for LDAP authentication on MC, be sure that your LDAP server is configured to explicitly disable/enable this option. For more information, see the article on Infusion Technology Solutions and the OpenLDAP documentation on access control.

What happens next

Shortly after you click **Finish** , you should see a status in the browser. However, you might see only an empty page for several seconds. During this brief period, MC runs as the local user 'root' long enough to bind to port number 5450. Then MC switches to the MC super administrator account that you just created, restarts MC, and displays the MC login page.

Managing client connections

Each client session to MC uses a connection from MaxClientSessions , a database configuration parameter. This parameter determines the maximum number of sessions that can run on a single database cluster node. Sometimes multiple MC users, mapped to the same database account, are concurrently monitoring the Overview and Activity pages.

> **Tip**
> You can increase the value for MaxClientSessions on an MC-monitored database to account for extra sessions. See Managing sessions for details.

In this section

- Changing MC or agent ports
- Management Console settings
- Backing up MC
- Connecting securely from MC to a Vertica database

- Upgrading Management Console manually
- Upgrading MC automatically on AWS
- Localizing user interface text

Changing MC or agent ports

When you configure MC, the Configuration Wizard sets up the following default ports:

- 5450—Used to connect a web browser session to MC and allows communication from Vertica cluster nodes to the MC application/web server
- 5444—Provides MC-to-node and node-to-node (agent) communications for database create/import and monitoring activities

If you need to change the MC default ports

A scenario might arise where you need to change the default port assignments for MC or its agents. For example, perhaps one of the default ports is not available on your Vertica cluster, or you encounter connection problems between MC and the agents. The following topics describe how to change port assignments for MC or its agents.

Changing the MC port

Use this procedure to change the default port for MC's application server from 5450 to a different value.

1. Open a web browser and connect to MC as a user with MC ADMIN privileges.
2. On the MC Home page, navigate to **MC Settings > Configuration** and change the **Application server running port** value from 5450 to a new value.
3. In the change-port dialog, click **OK** .
4. Restart MC .
5. Reconnect your browser session using the new port. For example, if you changed the port from 5450 to 5555, use one of the following formats:

   ```
   https://00.00.00.00:5555/
   ```

   OR

   ```
   https://hostname:5555/
   ```

Changing the agent port

Changing the agent port takes place in two steps: at the command line, where you modify the config.py file and through a browser, where you modify MC settings.

Change the agent port in config.py

1. Log in as root on any cluster node and change to the agent directory:

   ```
   $ cd /opt/vertica/oss/python3/lib/python3.9/site-packages/vertica/agent/
   ```

2. Use any text editor to open config.py .
3. Scroll down to the agent_port = 5444 entry and replace 5444 with a different port number.
4. Save and close the file.
5. Copy config.py to the /opt/vertica/oss/python3/lib/python3.9/site-packages/vertica/agent/ directory on all nodes in the cluster.
6. Restart the agent process by running the following command:

   ```
   $ /etc/init.d/vertica_agent restart
   ```

   > **Note**
   > If you are using Red Hat Enterprise Linux/CentOS 7, use the following command instead:

   ```
   $ /opt/vertica/sbin/vertica_agent restart
   ```

7. Repeat (as root) Step 6 on each cluster node where you copied the config.py file.

Change the agent port on MC

1. Open a web browser and connect to MC as a user with MC ADMIN privileges.
2. Navigate to **MC Settings > Configuration** .
3. Change Default Vertica agent port from 5444 to the new value you specified in the config.py file.
4. Click **Apply** and click **Done** .
5. Restart MC so MC can connect to the agent at its new port. See Restarting MC .

See also

- Reserved ports

Management Console settings

The **MC Settings** page allows you to configure properties specific to Management Console. To access **MC Settings** , go to the Management Console home page > **MC Tools** > **MC Settings** .

MC configuration settings

The **Configuration** tab contains the following sections:

System User configurations
View the user name, user group, and user home path for the MC user.

Vertica database configurations
Edit the following database paths:

- License path
- Catalog path
- Data path

MC and Agent Port settings
Configure the server port and the default port the Vertica agent uses.

Application Server JVM settings
Set the initial and maximum heap size for the JVM.

Browser connections settings
Enable and disable username and password auto-complete at Management Console login. After disabling, clear your browser's cache.

MC Password configuration settings
Set password requirements to log into the Management Console. This includes length, expiration, and login attempt settings.

User Analytics and Tracking
Choose if you want to provide Vertica with analytic information about your MC usage. Vertica uses this information to improve the MC in future releases.

> **Note**
> When you accept the End-User License Agreement Terms for the MC, you are given the option to provide Vertica with your anonymous user data. If you want to stop providing data, clear the checkbox in **User Analytics and Tracking** .

Vertica collects the following information:

- Database type (Eon Mode or Enterprise Mode)
- License type (Community Edition, Paid, By the Hour)
- Cloud provider name
- Vertica version
- MC version
- Current page
- Interactions with MC page components, including buttons, drop-down lists, checkboxes, and radio buttons.

To protect your privacy, all collected information is stored and processed anonymously, and in compliance with GDPR regulations. It is stored securely on Vertica servers, and never shared with third-party organizations.

MC monitoring settings

Control the following monitoring settings in Management Console:

- Enable checks and set alert thresholds for spread retransmit rate. This setting is disabled by default. The recommended alert threshold for spread retransmit rate is 10%.
- Set alert thresholds for free Management Console disk space checks. The recommended alert threshold is 500 MB.
- Exclude MC queries from activity charts.
- Set refresh intervals for MC charts and pages.

MC security and authentication settings

- Upload a new SSL certificate or view the current certificate (SSL Certificates tab).
- Use LDAP for user authentication (Authentication tab).

MC user management settings

Create new Management Console users and, with their user credentials, map them to an database managed by Management ConsoleC on the Vertica server. See Creating an MC user and Managing MC users .

MC extended monitoring settings

Configure Extended monitoring , which allows you to monitor more long-term data in Management Console:

- Set up an external storage database for Extended Monitoring. See Managing the storage database .
- Enable or disable Extended Monitoring on your databases. See Managing extended monitoring on a database .

Other MC settings

- View your version of Vertica or upload a new Vertica binary file
- Customize the look and feel of Management Console on the **Themes** tab.
- Configure Management Console to use an alternative data source to monitor your database. See Monitoring external data sources with MC .
- Enable Management Console to send email alerts. See Setting up SMTP for email alerts .

Modifying database-specific settings

To inspect or modify settings related to a database managed by Management Console, go to the **Existing Infrastructure** page. On this page, select a running database to see its Overview page. From the bottom of the Overview page, click the **Settings** tab to make modifications to database-specific settings.

Backing up MC

Before you upgrade MC , Vertica recommends that you back up your MC metadata (configuration and user settings). Use a storage location external to the server on which you installed MC.

1. On the target server (where you want to store MC metadata), log in as root or a user with sudo privileges.
2. Create a backup directory:
   ```
   # mkdir /backups/mc/mc-backup-20130425
   ```
3. Copy the /opt/vconsole directory to the new backup folder:
   ```
   # cp –ar /opt/vconsole /backups/mc/mc-backup-20130425
   ```

Connecting securely from MC to a Vertica database

When you use MC to monitor and manage a Vertica database, MC (running in a browser) connects as the client to the Vertica database server.

MC uses JDBC for most database connections

MC uses Java Database Connectivity (JDBC) for most connections to a Vertica database, including:

- Retrieving database information to display in charts
- Running SQL queries through JDBC
- Configuring and updating database properties
- Configuring the database for extended monitoring

Exception

When MC uses Agents to perform AdminTools tasks, MC does not use JDBC to connect to the database.

Vertica software supports TLS

Vertica databases and Vertica MC support TLS up to version 1.2. This topic and its subtopics describe configuring TLS in MC for JDBC connections to a Vertica database.

About certificate file formats

MC requires that all certificate and key files for upload to MC must be in PEM (Privacy-enhanced Electronic Mail) format.

Vertica database security dictates how MC connects

The TLS/SSL security you configure for a database in MC must be consistent with the security configured on the database itself.

Whether the Vertica database has TLS/SSL configured in server mode or mutual mode, you should configure TLS/SSL for that database in MC to match.

To find out how a Vertica database is configured, see Determining the TLS mode of a Vertica database .

You can configure TLS/SSL in either server mode or mutual mode in MC.

The rest of this topic and related topics use the term TLS, TLS/SSL, and SSL interchangeably.

TLS server mode

When the MC client connects to a Vertica database configured in *server mode* :

- The client requests and verifies the server's credentials.
- The client does not need to present a client certificate and private key file to the server.
- The MC administrator must configure the CA certificate that can verify server's certificate on MC when MC connects to the database over JDBC.

TLS mutual mode

When the MC client connects to a Vertica database configured in *mutual mode* :

- The MC client requests and verifies the database server's credentials.
- The server also requests and verifies the MC client's credentials.
- Each MC user is a separate client, and must present a valid client certificate file and private key file pair (*keypair*), namely a certificate signed by a CA recognized by the Vertica database server as valid.
- The MC administrator must configure:
  - The CA certificate to verify the Vertica database server certificate.
  - A client certificate and private key file (*keypair*) for each MC user. The keypair can be unique for each user, or shared by multiple users, depending on how client authentication is configured on the Vertica database. See Configuring client authentication .
- Each MC user must be configured to map correctly to a user who is configured on the Vertica database server.

For more information on how Vertica supports TLS/SSL security, see TLS protocol .

MC administrator configures MC security

Only MC users having Admin or Super privileges on a database are able to configure TLS certificates and keys on MC for database connections. The topics in this section use "MC administrator" to refer to both of these roles. For more information about MC user roles and privileges, see Users in Management Console .

As the MC administrator, when you first configure security in MC for a Vertica database that requires mutual mode, you configure these certificates for the Vertica database:

- The server certificate and public key of the database.
- Your own client certificate and private key, as the first configured MC user mapped to a Vertica database user.

Configuring TLS/SSL on MC

MC provides the Certificates wizard for configuring TLS certificates for all JDBC connections to the database, to ensure those connections are secure.

In MC, there are three scenarios in which you need to configure TLS security for a Vertica database:

- While you are importing a database to monitor in MC. See Configuring TLS while importing a database on MC .
- When you want to add security for a database that is already monitored by MC. See Configuring TLS for a monitored database in MC .
- When you need to configure client security for an individual MC user who is mapped to a user who has privileges on the Vertica database server, because the database requires mutual authentication. See Configuring mutual TLS for MC users .

Adding certificates to MC for later use

You may want to add multiple CA certificates or client certificates to MC all at one time, to streamline the configuration of security when you are importing databases to MC or creating MC users. For details, see and .

To connect successfully, MC and database security must match

MC Security	**Vertica Database Security**	**Does the connection succeed?**
None	None	Connection succeeds, and it is open and therefore unsecured.
TLS server mode	TLS server mode	Connection succeeds provided MC can verify the server's certificate using the CA certificate configured on MC.

TLS mutual mode	TLS mutual mode	Connection succeeds provided: • MC can verify the server's certificate using the CA certificate configured on MC. • The server can verify the client certificate and private key that MC presents as belonging to a mapped user on the Vertica database.
None	TLS server mode	MC attempts to establish an open connection. The connection fails if the Vertica database requires TLS for client connections. For more information, see: • Client authentication with TLS • CREATE AUTHENTICATION
None	TLS mutual mode	MC attempts to establish an open connection. The connection fails if the Vertica database requires TLS for client connections. The connection fails because MC does not present what the database requires: a valid client certificate and private key that the database can verify as belonging to a mapped database user.
TLS server mode	None	MC attempts to connect to the database securely, however the connection fails as the database is not configured with TLS certificates.
TLS mutual mode	None	MC attempts to connect to the database securely, however the connection fails as the database is not configured with TLS certificates.

In this section

- Management Console security
- Determining the TLS mode of a Vertica database
- Configuring TLS while importing a database on MC
- MC certificates wizard
- Configuring TLS for a monitored database in MC
- Configuring mutual TLS for MC users
- Updating TLS security for MC connections
- Enabling or disabling TLS for a database in MC
- Adding TLS certificates in MC
- Managing TLS certificates in MC
- Updating a TLS certificate in MC
- Removing TLS certificates from MC
- MC icons display database TLS status
- Bulk-configure a group of MC users for TLS

Management Console security

The Management Console (MC) manages multiple Vertica clusters, all of which might have different levels and types of security, such as user names and passwords and LDAP authentication. You can also manage MC users who have varying levels of access across these components.

Open authorization and SSL

Management Console (MC) uses a combination of OAuth (Open Authorization), Secure Socket Layer (SSL), and locally-encrypted passwords to secure HTTPS requests between a user's browser and MC, and between MC and the agents . Authentication occurs through MC and between agents within the cluster. Agents also authenticate and authorize jobs.

The MC configuration process sets up SSL automatically, but you must have the openssl package installed on your Linux environment first.

See the following topics for more information:

- TLS protocol
- Generating certificates and keys for MC
- Importing a new certificate to MC

User authentication and access

MC provides two user authentication methods, LDAP or MC. You can use only one method at a time. For example, if you chose LDAP, all MC users will be authenticated against your organization's LDAP server.

You set up LDAP authentication up through MC Settings > Authentication on the MC interface.

> **Note**
> MC uses LDAP data for authentication purposes only. It does not modify user information in the LDAP repository.

The MC authentication method stores MC user information internally and encrypts passwords. These MC users are not system (Linux) users. They are accounts that have access to MC and, optionally, to one or more MC-managed Vertica databases through the MC interface.

Management Console also has rules for what users can see when they sign in to MC from a client browser. These rules are governed by access levels, each of which is made up of a set of roles.

See also

- Users, roles, and privileges
- Users in Management Console
- Creating an MC user

Determining the TLS mode of a Vertica database

When you configure Vertica for TLS through the Management Console, you must configure the security mode to match what the Vertica database is configured to require: server mode or mutual mode.

To determine the TLS mode for existing sessions, query the SESSIONS system table:

```
=> SELECT session_id, user_name, ssl_state FROM sessions;
          session_id           | user_name | ssl_state
-------------------------------+-----------+-----------
v_vmart_node0001-333611:0x1ab  | dbadmin   | mutual
```

To determine the Vertica database's client-server TLS configuration, query the TLS_CONFIGURATIONS system table for the "server":

```
=> SELECT name, certificate, ca_certificates, mode FROM tls_configurations WHERE name = 'server';
 name  |  certificate   |  ca_certificates  |  mode
--------+----------------+-------------------+-----------
server | server_cert    | ca_cert,ica_cert  | VERIFY_CA
(1 row)
```

The "mode" can be one of the following, in ascending security:

- DISABLE : Disables TLS. All other options for this parameter enable TLS.
- ENABLE : Enables TLS. Vertica does not check client certificates.
- TRY_VERIFY : Establishes a TLS connection if one of the following is true:
  - the other host presents a valid certificate
  - the other host doesn't present a certificate

  If the other host presents an invalid certificate, the connection will use plaintext.
- VERIFY_CA : Connection succeeds if Vertica verifies that the other host's certificate is from a trusted CA. If the other host does not present a certificate, the connection uses plaintext.
- VERIFY_FULL : Connection succeeds if Vertica verifies that the other host's certificate is from a trusted CA and the certificate's cn (Common Name) or subjectAltName attribute matches the hostname or IP address of the other host.
  Note that for client certificates, cn is used for the username, so subjectAltName must match the hostname or IP address of the other host.

Mutual mode corresponds to TRY_VERIFY or higher, which indicates that Vertica is in mutual mode. In mutual mode, Vertica sends its server certificate to the client for verification, and uses the CA certificates (in this case, "ca_cert" and "ica_cert") to verify client certificates.

In contrast, a server mode configuration (which doesn't verify client certificates) might have the following TLS configuration instead:

```
=> SELECT name, certificate, ca_certificates, mode FROM tls_configurations WHERE name = 'server';
 name  |  certificate   |  ca_certificates  |  mode
--------+----------------+-------------------+-----------
server | server_cert    |                   | ENABLE
(1 row)
```

Configuring TLS while importing a database on MC

To configure TLS as you are importing an existing Vertica database on MC:

1. Follow the steps in Importing an existing database into MC .
2. In the **Import Vertica** window, select the database and click the **Use TLS** checkbox.
3. Click **Configure TLS and Import DB** to launch and complete the Certificates wizard.

MC certificates wizard

The MC Certificates wizard lets you configure a CA certificate for the Vertica database server and client certificates for MC to allow secure TLS communication over the JDBC connections between MC and the Vertica database server. Each screen presents options. When you select an option, the wizard displays additional options and details.

1. The first wizard screen provides helpful overview information. Read it, and click **Configure TLS Certificates** to continue.
2. On the **Configure CA Certificates** screen, configure a CA certificate (public key) to add to MC. MC uses this trusted certificate to verify the server's identity during TLS communications over JDBC connections between MC and the Vertica database server.
   Complete one of these options:
   - **Upload a new CA certificate** Browse and select the certificate file and enter an alias for this certificate
     - To add another CA certificate, click **Add More CA Certificates.**
     - Continue adding additional CA certificates until you are finished.
   - **Choose a certificate alias from previously uploaded certificates** Select the alias for the previously uploaded CA certificate you wish to configure for the current database.
3. When you are done adding CA certificates, click **Next.**
4. The **Configure Client Certificate** screen displays the check box **Add Client Certificate and Private Key for Mutual Mode TLS Connection.**
5. If the database is configured for server mode , you do not need a client certificate or key.
   - Leave the **Add Client Certificate** check box *un* checked and click **Review.**
   - Skip to step 10.
6. If the database is configured for mutual mode :
   - Click the **Add Client Certificate** check box.
   - Select one of the options below.
     - **Upload Client Certificate and Private Key files on MC** (shown above.) MC uses its https connection from the browser to MC's host to upload the files.)
       - To add an additional client certificate and create a certificate chain, click **Add Certificate to Chain.** MC reinitializes the Client Certificate file field so you can add another certificate. After you add the last certificate path, click **Next.**
       - To upload an existing certificate chain file, click **Browse** next to the Upload Client Certificate/Certificate chain file field, select the file, and click **Open.**
     - **Manually upload client Certificate and Private Key on MC host and provide paths** Avoids sending the encrypted certificate and private key files over an https connection. To add an additional path for a client certificate and create a certificate chain, click **Add More Certificate Paths.** MC reinitializes the path field so you can add another path. After you add the last certificate path, click **Next.**
     - **Choose Client Certificate and Private Key alias of previously uploaded keypair to use for this database.** (To use existing certificate and key files.)
7. Complete the detail fields for the client certificate and private key option you have chosen above, then click **Next.**
8. The Apply TLS configuration to MC users mapped to database window allows you to configure the client certificate-key pair you have just entered, for use by multiple MC users.

   > **Note**
   > All the MC users you select must be mapped to the same user id on the Vertica database server.

9. Click **Review.** The wizard displays a review window with the TLS options you have configured.
10. Select one of these options:
    - To modify your TLS choices, click **Back.**
    - To confirm your choices:
      - If you are importing a database, click **Configure TLS and Import DB** .
      - If you are configuring TLS for a database already imported to MC, click **Configure TLS for DB.**
      - Click **Close** to complete the wizard.
    - To close the wizard without importing the database and without setting up TLS configuration, click **Cancel.**

MC certificates wizard

The MC Certificates wizard lets you configure a CA certificate for the Vertica database server and client certificates for MC to allow secure TLS communication over the JDBC connections between MC and the Vertica database server. Each screen presents options. When you select an option, the wizard displays additional options and details.

1. The first wizard screen provides helpful overview information. Read it, and click **Configure TLS Certificates** to continue.
2. On the **Configure CA Certificates** screen, configure a CA certificate (public key) to add to MC. MC uses this trusted certificate to verify the server's identity during TLS communications over JDBC connections between MC and the Vertica database server.
   Complete one of these options:
   - **Upload a new CA certificate** Browse and select the certificate file and enter an alias for this certificate
     - To add another CA certificate, click **Add More CA Certificates.**
     - Continue adding additional CA certificates until you are finished.
   - **Choose a certificate alias from previously uploaded certificates** Select the alias for the previously uploaded CA certificate you wish to configure for the current database.
3. When you are done adding CA certificates, click **Next.**
4. The **Configure Client Certificate** screen displays the check box **Add Client Certificate and Private Key for Mutual Mode TLS Connection.**
5. If the database is configured for server mode , you do not need a client certificate or key.
   - Leave the **Add Client Certificate** check box *un* checked and click **Review.**
   - Skip to step 10.
6. If the database is configured for mutual mode :
   - Click the **Add Client Certificate** check box.
   - Select one of the options below.
     - **Upload Client Certificate and Private Key files on MC** (shown above.) MC uses its https connection from the browser to MC's host to upload the files.)
       - To add an additional client certificate and create a certificate chain, click **Add Certificate to Chain.** MC reinitializes the Client Certificate file field so you can add another certificate. After you add the last certificate path, click **Next.**
       - To upload an existing certificate chain file, click **Browse** next to the Upload Client Certificate/Certificate chain file field, select the file, and click **Open.**
     - **Manually upload client Certificate and Private Key on MC host and provide paths** Avoids sending the encrypted certificate and private key files over an https connection. To add an additional path for a client certificate and create a certificate chain, click **Add More Certificate Paths.** MC reinitializes the path field so you can add another path. After you add the last certificate path, click **Next.**
     - **Choose Client Certificate and Private Key alias of previously uploaded keypair to use for this database.** (To use existing certificate and key files.)
7. Complete the detail fields for the client certificate and private key option you have chosen above, then click **Next.**
8. The Apply TLS configuration to MC users mapped to database window allows you to configure the client certificate-key pair you have just entered, for use by multiple MC users.

   > **Note**
   > All the MC users you select must be mapped to the same user id on the Vertica database server.

9. Click **Review.** The wizard displays a review window with the TLS options you have configured.
10. Select one of these options:
    - To modify your TLS choices, click **Back.**
    - To confirm your choices:
      - If you are importing a database, click **Configure TLS and Import DB** .
      - If you are configuring TLS for a database already imported to MC, click **Configure TLS for DB.**
      - Click **Close** to complete the wizard.
    - To close the wizard without importing the database and without setting up TLS configuration, click **Cancel.**

Configuring TLS for a monitored database in MC

This procedure describes how to configure TLS for all JDBC connections to a database that is already being monitored in MC. Note that the Vertica database should already be configured with the TLS certificates required for TLS connections.

1. In MC, navigate to **Databases and Clusters > DB-name > Settings** and click the **Security** tab in the left navigation bar.
2. In the **Configure TLS Connection for Database** section, click **Enabled** in the drop-down beside **Use TLS Connection to database** .
3. Click **Configure TLS Connection** to launch and complete the Certificates wizard.

MC certificates wizard

The MC Certificates wizard lets you configure a CA certificate for the Vertica database server and client certificates for MC to allow secure TLS communication over the JDBC connections between MC and the Vertica database server. Each screen presents options. When you select an option, the wizard displays additional options and details.

1. The first wizard screen provides helpful overview information. Read it, and click **Configure TLS Certificates** to continue.
2. On the **Configure CA Certificates** screen, configure a CA certificate (public key) to add to MC. MC uses this trusted certificate to verify the server's identity during TLS communications over JDBC connections between MC and the Vertica database server.
   Complete one of these options:
   - **Upload a new CA certificate** Browse and select the certificate file and enter an alias for this certificate
     - To add another CA certificate, click **Add More CA Certificates.**
     - Continue adding additional CA certificates until you are finished.
   - **Choose a certificate alias from previously uploaded certificates** Select the alias for the previously uploaded CA certificate you wish to configure for the current database.
3. When you are done adding CA certificates, click **Next.**
4. The **Configure Client Certificate** screen displays the check box **Add Client Certificate and Private Key for Mutual Mode TLS Connection.**
5. If the database is configured for server mode , you do not need a client certificate or key.
   - Leave the **Add Client Certificate** check box *un* checked and click **Review.**
   - Skip to step 10.
6. If the database is configured for mutual mode :
   - Click the **Add Client Certificate** check box.
   - Select one of the options below.
     - **Upload Client Certificate and Private Key files on MC** (shown above.) MC uses its https connection from the browser to MC's host to upload the files.)
       - To add an additional client certificate and create a certificate chain, click **Add Certificate to Chain.** MC reinitializes the Client Certificate file field so you can add another certificate. After you add the last certificate path, click **Next.**
       - To upload an existing certificate chain file, click **Browse** next to the Upload Client Certificate/Certificate chain file field, select the file, and click **Open.**
     - **Manually upload client Certificate and Private Key on MC host and provide paths** Avoids sending the encrypted certificate and private key files over an https connection. To add an additional path for a client certificate and create a certificate chain, click **Add More Certificate Paths.** MC reinitializes the path field so you can add another path. After you add the last certificate path, click **Next.**
     - **Choose Client Certificate and Private Key alias of previously uploaded keypair to use for this database.** (To use existing certificate and key files.)
7. Complete the detail fields for the client certificate and private key option you have chosen above, then click **Next.**
8. The Apply TLS configuration to MC users mapped to database window allows you to configure the client certificate-key pair you have just entered, for use by multiple MC users.

   > **Note**
   > All the MC users you select must be mapped to the same user id on the Vertica database server.

9. Click **Review.** The wizard displays a review window with the TLS options you have configured.
10. Select one of these options:
    - To modify your TLS choices, click **Back.**
    - To confirm your choices:
      - If you are importing a database, click **Configure TLS and Import DB** .
      - If you are configuring TLS for a database already imported to MC, click **Configure TLS for DB.**
      - Click **Close** to complete the wizard.
    - To close the wizard without importing the database and without setting up TLS configuration, click **Cancel.**

Configuring mutual TLS for MC users

You can configure TLS for existing MC users who are already mapped to Vertica database user ids. You would do so if you had just configured TLS in mutual mode on a previously unsecured Vertica database, and needed to configure a client certificate and private key for each MC user who accesses that database.

1. In MC, navigate to **MC Settings** and click the **User Management** tab.
2. Select a user from the list and click **Edit.**
3. In the **Add permissions** window:
   - Choose the database for which you want to edit this MC user's security permissions.
   - MC displays the database username to which this MC user is currently mapped.
   - In the **Restrict Access** drop-down, choose **Admin, Associate, IT,** or **User** to specify the privilege level for this user.

- In the **Use TLS Connection** drop-down, choose **Yes.**
- Click **Configure TLS for user** to launch and complete the Certificates wizard.

MC certificates wizard

The MC Certificates wizard lets you configure a CA certificate for the Vertica database server and client certificates for MC to allow secure TLS communication over the JDBC connections between MC and the Vertica database server. Each screen presents options. When you select an option, the wizard displays additional options and details.

1. The first wizard screen provides helpful overview information. Read it, and click **Configure TLS Certificates** to continue.
2. On the **Configure CA Certificates** screen, configure a CA certificate (public key) to add to MC. MC uses this trusted certificate to verify the server's identity during TLS communications over JDBC connections between MC and the Vertica database server.
   Complete one of these options:
   - **Upload a new CA certificate** Browse and select the certificate file and enter an alias for this certificate
     - To add another CA certificate, click **Add More CA Certificates.**
     - Continue adding additional CA certificates until you are finished.
   - **Choose a certificate alias from previously uploaded certificates** Select the alias for the previously uploaded CA certificate you wish to configure for the current database.
3. When you are done adding CA certificates, click **Next.**
4. The **Configure Client Certificate** screen displays the check box **Add Client Certificate and Private Key for Mutual Mode TLS Connection.**
5. If the database is configured for server mode , you do not need a client certificate or key.
   - Leave the **Add Client Certificate** check box *un* checked and click **Review.**
   - Skip to step 10.
6. If the database is configured for mutual mode :
   - Click the **Add Client Certificate** check box.
   - Select one of the options below.
     - **Upload Client Certificate and Private Key files on MC** (shown above.) MC uses its https connection from the browser to MC's host to upload the files.)
       - To add an additional client certificate and create a certificate chain, click **Add Certificate to Chain.** MC reinitializes the Client Certificate file field so you can add another certificate. After you add the last certificate path, click **Next.**
       - To upload an existing certificate chain file, click **Browse** next to the Upload Client Certificate/Certificate chain file field, select the file, and click **Open.**
     - **Manually upload client Certificate and Private Key on MC host and provide paths** Avoids sending the encrypted certificate and private key files over an https connection. To add an additional path for a client certificate and create a certificate chain, click **Add More Certificate Paths.** MC reinitializes the path field so you can add another path. After you add the last certificate path, click **Next.**
     - **Choose Client Certificate and Private Key alias of previously uploaded keypair to use for this database.** (To use existing certificate and key files.)
7. Complete the detail fields for the client certificate and private key option you have chosen above, then click **Next.**
8. The Apply TLS configuration to MC users mapped to database window allows you to configure the client certificate-key pair you have just entered, for use by multiple MC users.

   > **Note**
   > All the MC users you select must be mapped to the same user id on the Vertica database server.

9. Click **Review.** The wizard displays a review window with the TLS options you have configured.
10. Select one of these options:
    - To modify your TLS choices, click **Back.**
    - To confirm your choices:
      - If you are importing a database, click **Configure TLS and Import DB** .
      - If you are configuring TLS for a database already imported to MC, click **Configure TLS for DB.**
      - Click **Close** to complete the wizard.
    - To close the wizard without importing the database and without setting up TLS configuration, click **Cancel.**

Updating TLS security for MC connections

Maintaining TLS security for MC JDBC connections to a Vertica database is an ongoing process. Initially, you as the MC administrator must configure the appropriate certificates and keys. As time passes, certificates expire or otherwise become invalid. To maintain TLS security in MC, you must configure new certificates to replace any that are about to expire.

If any of the certificates that secure an MC connection to a Vertica database changes or expires, the MC administrator must update the TLS configuration for that database on MC to ensure that unexpired certificates are available so that connections can succeed.

- To update the certificates, simply configure new certificates for the connection between MC and that Vertica database.
- To configure new certificates for a database monitored in MC, see Configuring TLS for a monitored database in MC .
- To configure new client certificates for an MCC user, see Configuring mutual TLS for MC users .
- To replace an expiring or invalid certificate for a database or client, see Updating a TLS certificate in MC .

MC flags the current certificate for a given connection with a "use me" bit. This bit is set only for the current certificate. When you configure a new certificate for a given connection, the new certificate is marked current, and the previous certificate (although still present in the trust store or keystore) is no longer marked as the current certificate.

Enabling or disabling TLS for a database in MC

To enable TLS for all JDBC connections from MC to a Vertica database, configure the certificate and key appropriate for that connection. See:

- Configuring TLS while importing a database on MC
- Configuring TLS for a monitored database in MC
- Configuring mutual TLS for MC users

Disabling a TLS connection

Under some conditions, you as the system administrator might need to disable TLS for JDBC connections from MC to a Vertica database. Here are some examples:

- The TLS certificates are expired and you have not yet obtained new certificates.
- The TLS certificates and keys are revoked and the user does not have new certificates and keys, but you still want to allow that user to connect from MC to the database to show monitoring information and run queries.

To disable TLS for connecting to a Vertica database:

1. In MC, navigate to **Home > Databases and Clusters > DatabaseName > Settings.**
2. Click the **Security** tab in the left navigation bar.
3. In the **Use TLS Connection to database** drop-down, choose **Disabled.**

> **Note**
> To reenable TLS for a database connection after you disable it, you must reconfigure the necessary certificates.
>
> Disabling TLS for a database removes the configuration that tells MC to use the current certificates and keys for a given database, for all users. If it is a mutual mode TLS connection and each user had a separate client certificate and private key configured for that database, to re-enable TLS you must reconfigure the certificate and key for each user individually, for that database.

Re-enabling a disabled TLS connection

1. In MC, navigate to **Home > Databases and Clusters > DatabaseName > Settings.**
2. Click the **Security** tab in the left navigation bar.
3. In the **Use TLS Connection to database** drop-down, choose **Enabled.**
4. MC displays **Configure MC to use secured connection to query Vertica database or modify existing configuration.**
5. To finish re-enabling TLS, click **Configure TLS Connection** to launch the Certificates Wizard.
6. Complete the MC certificates wizard .

Adding TLS certificates in MC

You can add one or more certificates to MC for later use, without immediately associating the certificates with a database. Adding certificates ahead of time makes it easier to configure security for a database or for one or more MC users, because you can just choose a CA or client certificate from a list rather than having to add it to MC during the configuration steps.

Adding CA certificates in MC

To add one or more CA certificates in MC:

1. From the MC home page, navigate to **MC Settings > SSL/TLS Certificates** .
2. Under Manage TLS Certificates for Database Connection, click **Add New CA Certificate** .
3. In the Add new CA certificates for TLS connection window, enter an alias for the certificate, to make it easier to refer to later.
4. Click **Browse** to locate the certficate file you want to add. MC opens an Explorer window.
5. Select the file you want to upload, and click **Open** .

> **Note**
> Make sure the certificate file is unexpired, and is not protected by a password.

6. To add just this one certificate, click **Add New CA** . MC adds the certificate to its list.
7. To add additional CA certificates, click **Add More CA Certificates** . MC adds the certificate to a list, and clears the fields so you can enter the next CA certificate.
8. Repeat the process until you have entered the last certificate you want to add.
9. Click **Add New CA** to add all the CA certificates in the list to the MC:

Adding client certificates and keys in MC

You can add one or more client certificate and private key pairs to MC. In each pair, you can add either a single certificate, a preexisting certificate chain, or a series of client certificates that MC uses to create a new certificate chain.

To add one or more client certificates with their private key files to MC for later use:

1. Navigate to **Home > MC Settings > SSL/TLS Certificates.**
2. Under Manage TLS Certificates for Database Connection, click **Add New Client Certificate** . MC displays the Add new Client Certificate and Private Key for TLS Connection screen.

> **Note**
> When you add a client certificate to MC, you always add it with its private key file. The client certificate and its key are a *key pair* .

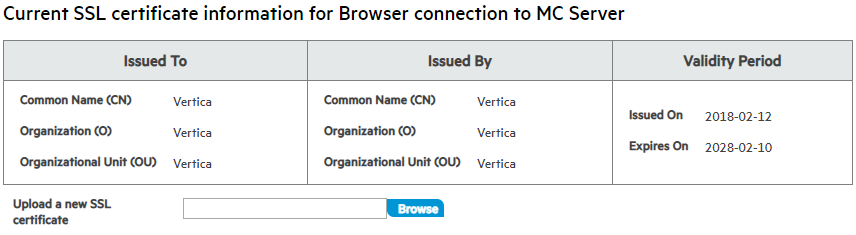
3. Click one of these file upload options:
   - **Upload Client Certificate and Private Key for TLS Connection** . With this option, you paste a certificate and key into browser fields. MC posts the certificate and key from your browser to the MC server via an https connection over the network, secured with TLS/SSL.
   - **Manually upload Client Certificate and Private Key on MC host and provide paths** . Sending the certificates from your browser to the MC server across an https network connection may not be not your preference. If so, you can use this option to specify the paths on the MC server host where you have manually uploaded the client certificate and private key files, instead. The URL of your MC browser shows the IP address of the MC host. Using this option, you must manually handle the transfer of the certificate and the key files to the server.
4. To provide a single client certificate and private key with either input option:
   - Enter a recognizable alias for the key pair.
   - Browse and select the private key file or provide the path.
   - Browse and select the client certificate file or provide the path.
   - Click **Add New Client Certificate.**
   - MC adds the key pair to its list.
5. To upload several certificates and private keys and create a certificate chain:
   - Enter an alias for the key pair.
   - Browse and select the private key file or provide the path.
   - Browse and select the client certificate file or provide the path.
   - Click **Add Certificate to Chain** (or **Add More Certificate Paths**).
   - Repeat the process until you have added the last certificate and key for this certificate chain.
   - Click **Add New Client Certificate.**
   - MC adds the resulting certificate chain to its list.

Adding a new certificate for the browser connection

You can view the existing TLS certificate for the browser connection to the MC server, or add a new certificate to replace it.

To view or replace the current SSL/TLS certificate that MC uses for the user's browser's HTTPS connection to the MC server:

1. From the MC home page, navigate to **MC Settings > SSL/TLS Certificates.**
   The top pane displays the current certificate for the browser connection to the MC server, including the certificate's expiration date:



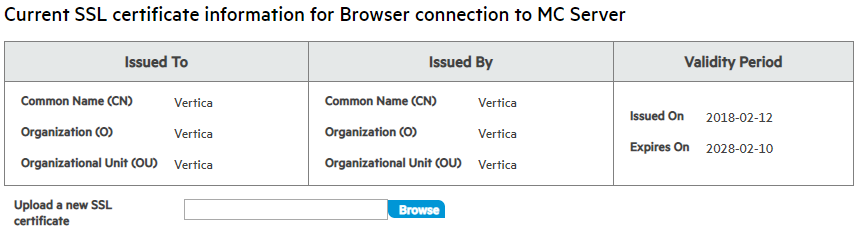
2. To replace the current certificate, click **Browse** next to the **Upload a new SSL certificate** field.
   MC opens an explorer window.
3. Select the certificate file you wish to upload and click **Open.** The certificate file must be in PEM (Privacy-enhanced Email Message) format.
   MC replaces the prior certificate with the new certificate.

Managing TLS certificates in MC

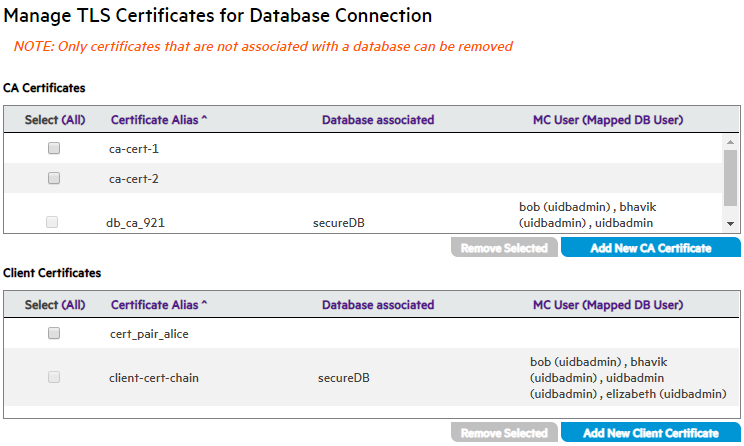
MC maintains a secure list containing all the CA certificates, and the client certificates or certificate chains and their corresponding key files, that you have uploaded into MC.

To manage the certificates already uploaded to MC, navigate to **Home > MC Settings > SSL/TLS Certificates** . This screen controls the TLS security settings for all of MC.

The top pane displays information about the current TLS certificate used to secure the user's browser connection to the MC server. You can add a new certificate to replace it. See .



The middle and lower panes allow you to add and remove CA and client certificates in MC.



You can perform the following tasks to manage your TLS certificates and keys in MC.

- Adding TLS certificates in MC
- Removing TLS certificates from MC

For the security settings for a specific database, open the database in MC and navigate to **Home > Databases and Clusters > DatabaseName > Settings** and click the **Security** tab in the left navigation bar.

Updating a TLS certificate in MC

When a TLS certificate is about to expire, has already expired, or otherwise becomes unusable, it needs to be updated.

This is the method for updating a certificate:

1. In MC, add the new certificate that will replace the expiring or invalid certificate. See Adding TLS certificates in MC .

   > **Note**
   > You can add and configure the new certificate for the database or user whose existing certificate is or will soon be invalid, as a single step, or two steps. Configuring the new certificate for the database dissociates the previously configured certificate from that database. See Connecting securely from MC to a Vertica database .

2. After the old certificate has been disassociated from all databases and users, you can remove it from the MC. See Removing TLS certificates from

MC .

Removing TLS certificates from MC

In some cases, it may be appropriate to disable TLS for a database in MC. Disabling TLS for the database disassociates all the certificates configured for that database. For more information, see Enabling or disabling TLS for a database in MC .

Disassociating a certificate from a database in MC

Before you can remove a certificate from MC, you must be sure the certificate is not associated with (being used by) any databases. The MC administrator can disassociate a certificate from a database in MC using either of these methods:

Configuring a new certificate on the database in MC

When you configure a new certificate to serve a specific purpose on a database in MC, the new certificate replaces the old certificate. The newly configured certificate is now associated with the database, and the old certificate is no longer associated and can be removed.

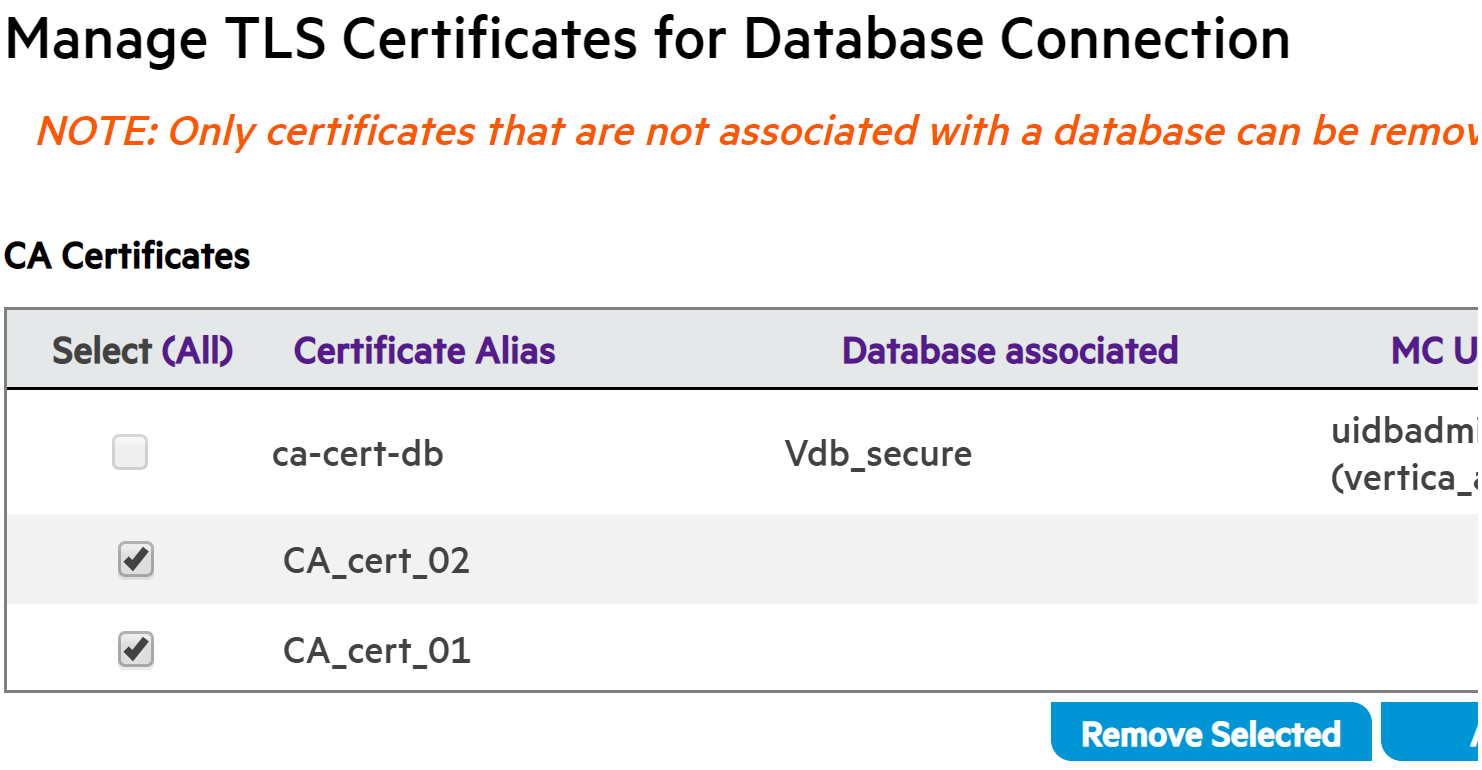
Navigate to **Databases and Clusters > DbName > Database Settings > Configure TLS.**

For details, see Configuring TLS for a monitored database in MC

Removing the TLS configuration on the database

You can remove one or more TLS certificates from the MC, provided the certificates are not associated with a database. To remove a certificate:

1. From the MC home page, navigate to **MC Settings > SSL/TLS Certificates** .
2. In the **Manage TLS Certificates for Database Connection** section, locate the row or rows for one or more CA or client certificates you want to remove. This example shows only the **CA Certificates** pane:



3. If the **Database associated** field is empty for that certificate, you can click to select the certificate for removal, and click **Remove Selected** . In the illustration above, CA_cert_02 and CA_cert_01 are selected for removal.

> **Note**
> If you remove one client certificate that is part of a certificate chain, MC removes the entire certificate chain.

MC icons display database TLS status

MC displays an icon at top left of the database in the Database and Cluster/Infrastructure view, that shows the current TLS status of the database. These same icons appear in the breadcrumbs to the left of the database name, to show the current TLS security status of the database:

Icon	Description
	No icon. The database is not configured to use TLS, and MC is not configured to connect to the database using TLS, either. When neither side has TLS configured, all connections are open and unsecured.
	Gray lock icon. TLS is configured on this database and also in MC.
	Gray lock icon with orange alert. Database is configured to use TLS but MC is not configured to use a TLS connection. An internal MC job checks the status of MC's connection.
	Red lock with red X. Both the database and MC are configured for TLS, but MC is not able to connect using TLS.

Bulk-configure a group of MC users for TLS

You as the MC administrator can create multiple MC users and map them all to the same database user id on the Vertica database server side. You map the users in MC when you create them. For details, see Creating an MC user .

You can then configure all the MC users that are mapped to a single Vertica database user id, to use the same client certificate or certificate chain and private key in MC, in a single bulk configuration process:

1. Navigate to **MC Home > Databases and Clusters > DbName> Settings > Security.**
2. Click **Configure TLS Connection** to launch the MC certificates wizard.
3. Complete steps 1 through 3 in the wizard to configure a CA certificate and the client certificate or certificate chain and key that you want to use for multiple MC users. For details, see MC certificates wizard .
4. After you complete these steps, the wizard displays the **Apply TLS configuration to MC users mapped to database** page as step 4 in the left wizard pane.
5. To apply the same CA certificate, client certificate and key you just configured to one or more additional users, click the check boxes for those users.

   > **Note**
   > All the users you select must be mapped to the same Vertica database user id.

6. To complete the configuration, click Review. MC displays a confirmation screen:
7. To complete the configuration of this CA certificate for the database and this client certificate/key pair for the selected MC users, click **Configure TLS for DB.**
8. MC confirms that the action was a success. Click **Close** to close the Certificate wizard.

Upgrading Management Console manually

If you installed MC manually, follow the procedure below to upgrade MC.

If you installed MC automatically on AWS resources, see Upgrading MC automatically on AWS .

Backing up MC before you upgrade

1. Log in as root or a user with sudo privileges on the server where MC is already installed.
2. Open a terminal window and shut down the MC process:

   ```
   # /etc/init.d/vertica-consoled stop
   ```

   For versions of Red Hat 7/CentOS 7 and above, use:

   ```
   # systemctl stop vertica-consoled
   ```

3. Back up MC to preserve configuration metadata.

Important
A full backup is required in order to restore MC to its previous state. Restoring MC is essential if the upgrade fails, or you decide to revert to the previous version of Vertica. For details, see Backing up MC .

4. Stop the database if MC was installed on an Ubuntu or Debian platform.

Extended monitoring upgrade recommendations

If you use Extended monitoring to monitor a database with MC, Vertica recommends the following upgrade procedure to avoid data loss.

1. Log in to MC as an administrator.
2. To stop the monitored database, navigate to the Existing Infrastructure > Databases and Clusters page, select the monitored database and click **Stop** .
3. On MC Settings > MC Storage DB Setup, click **Disable Streaming** to stop the storage database's collection of monitoring data.
4. To stop the storage database, navigate to the Existing Infrastructure > Databases and Clusters page, select the monitored database and click **Stop** .
5. Upgrade MC and Vertica according to Upgrade MC and Upgrading Vertica instructions.
6. To start the storage database, navigate to the Existing Infrastructure > Databases and Clusters page, select the monitored database and click **Start** .
7. Start the monitored database.
8. On MC Settings > MC Storage DB Setup, click **Enable Streaming** to enable collection of monitoring data.
   Important
   To avoid data loss, enable streaming soon after starting your monitored database. While your storage database is down and streaming is disabled, the Kafka server can retain data from your running monitored database for a limited amount of time. Data loss occurs when the data exceeds the Kafka retention policy's log size or retention time limits.

Upgrading MC

> **Note**
> After you upgrade, you are prompted to set up security questions and select a new password.

1. Download the MC package from the Vertica website :
   ```
   vertica-console-current-version.Linux-distro)
   ```
   Save the package to a location on the target server, such as /tmp .
2. On the target server, log in as root or a user with sudo privileges.
3. Change to the directory where you saved the MC package.
4. Install MC using your local Linux distribution package management system—rpm, yum, zypper, apt, dpkg. For example:
   Red Hat 8
   ```
   # rpm -Uvh vertica-console-current-version.x86_64.rpm
   ```
   Debian and Ubuntu
   ```
   # dpkg -i vertica-console-current-version.deb
   ```
5. If you stopped the database before upgrading MC, restart the database.
   As the root user, use the following command:
   ```
   /etc/init.d/verticad start
   ```
   For versions of Red Hat 8/CentOS 8 and above, run:
   ```
   # systemctl start verticad
   ```
6. Open a browser and enter the URL of the MC installation, one of the following:
   - IP address:
     ```
     https://ip-address:mc-port/
     ```
   - Server host name:
     ```
     https://hostname:mc-port/
     ```

   By default, *mc-port* is 5450.

If MC was not previously configured, the Configuration Wizard dialog box appears. Configuration steps are described in Configuring MC .

If MC was previously configured, Vertica prompts you to accept the end-user license agreement (EULA) when you first log in to MC after the upgrade.

Additionally, you can choose to provide Vertica with analytic information about your MC usage. For details, see Management Console settings .

Upgrading MC automatically on AWS

If you automatically installed Management Console (MC) version 9.1.1 or later on AWS resources, you can automatically upgrade it from the MC interface using the Upgrade wizard.

This process provisions a new Management Console instance and copies any current MC configuration data to the new MC. All MC settings, users, and monitored clusters will be transferred.

After upgrading, you can terminate the previous Management Console instance.

In addition, when you revive an Eon Mode database through the upgraded Management Console, that database will also be automatically upgraded to the same Vertica version as MC.

Upgrade MC automatically

Automatic upgrade is only available if the existing MC has been installed automatically through the AWS Marketplace.

1. From the MC home page, select **MC Settings** .
2. From the menu on the left side of the page, select **Upgrade MC** . The Upgrade MC page displays current Management Console information and indicates whether you are using the latest version of MC, or if a newer version is available.
3. Click **Start MC Upgrade** at the bottom of the page (this button is only displayed if a newer version of MC is available). The Upgrade wizard appears.
4. Go through the wizard and enter the following information when prompted:
   - AWS access key ID and AWS secret key (only required if existing MC was not installed using an IAM role)
   - AWS key pair
   - MC version to upgrade to
   - EC2 instance type for new MC host
   - EC2 instance tags (optional)
5. When upgrade is successful, the wizard displays the URL for the upgraded Management Console. Save this URL; this how to access your new MC. It is important to save this URL for future use; after you terminate your previous MC, the new MC URL will *not* be available elsewhere. (The MC URL referenced from the original stack when you created MC will continue to reference the previous MC, not the new MC.)
6. Follow the URL and log into your new MC.
7. To terminate the previous version of MC:
   1. If necessary, disable termination protection for the previous MC instance. You can do so from the AWS console. See the AWS guide for enabling and disabling instance termination protection .
   2. From the AWS console, terminate the instance on which the previous MC resides. See the AWS guide for how to terminate instances .

> **Note**
> Do not delete other associated resources for the previous MC. Some of these resources may still be in use by the new MC, or by any clusters that were created using the previous MC.

Next steps

If you plan to upgrade an Eon Mode database from Vertica version 9.1.0 or above to a later version, you can do so automatically by reviving it through a newer version of Management Console. As MC revives the database, it will also upgrade the Eon Mode database to the same Vertica version as the upgraded MC. See Reviving an Eon Mode database on AWS in MC .

Localizing user interface text

You can translate Management Console (MC) user interface (UI) text with language files in the Vertica server. After you translate the UI text, users can select the language from the language selector, a dropdown located to the right of the username in the MC toolbar.

The required language files include the locales.json file and the resource bundle, a directory of JSON-formatted files that contain the text strings translated into the target language.

Language files are located in the /opt/vconsole/temp/webapp/resources/i18n/lang directory.

locales.json

The locales.json file contains an array of JSON objects with a name key and properties, where each object represents a language that the MC supports. For example, the following object represents Mexican Spanish:

```
"es_MX": {
   "code": "es_MX",
   "name": "Spanish - Mexico",
   "country_code": "MX"
}
```

The preceding object provides the following information that you use to translate UI text:

- The object name key is a two- or four-letter country code. In the preceding example, the name key is es_MX . The resource bundle name must match this country code, or the MC cannot detect the translation files.
- The MC lists the name value in the language selector. If you translated the UI text into Mexican Spanish, the language selector would list Spanish - Mexico .

Resource bundle

The resource bundle is a directory in /opt/vconsole/temp/webapp/resources/i18n/lang that stores a collection of JSON-formatted files that contain the UI text strings that you can translate. By default, Vertica provides the following resource bundles:

- en_US (American English)
- zh_CN (Simplified Chinese)

Creating a custom resource bundle

Important

Custom resource bundles are not preserved when you upgrade Management Console versions. Before you upgrade, you must back up your resource files when you back up your Management Console configuration.

For details, see Upgrading Management Console manually .

To create a resource bundle, you must manually create a new directory and copy files from one of the default resource bundles. For example, to create a resource bundle for Mexican Spanish:

1. Navigate to the directory that contains the language files:
   ```
   $ cd /opt/vconsole/temp/webapp/resources/i18n/lang
   ```
2. Create a new directory named es_MX :
   ```
   $ mkdir es_MX
   ```

   > **Note**
   > If the name of a resource bundle does not exactly match a name key in locales.json , the MC cannot detect the translation, and the translated language is not available in the language selector.

3. Copy all files from the default en_US resource bundle into the new es_MX resource bundle:
   ```
   $ cp en_US/* es_MX
   ```

Text string file structure

Each JSON file in the resource bundle contains text strings for a specific section of the MC interface. For example, the homepage.json file stores text strings for the Management Console home page . Each JSON file represents MC pages and any child UI components that contain text—including subsections, tabs, and buttons—as individual objects. The file nests these pages and child components hierarchically to convey the page structure. For example, the homepage.json file uses the following structure:

```
{
   "homepage": {
      ...
   },
   "recentDatabase": {
      ...
   },
   "copyright": {
      ...
   },
   ...
```

The object properties represent the UI text as key/value pairs, where the key is the component with UI text, and the value is the text string that the MC displays in the UI.

Translating text

To translate a text string, edit the text string value. For example, to translate the title of the **Recent Databases** section into Spanish, open the homepage.json file in a text editor and update homepage.recentDatabase.title value:

```
{
   "homepage": {
      ...
      },
      "recentDatabase": {
        "title": "Bases de Datos Recientes",
        ...
}
```

Getting started with MC

Use Management Console to monitor the performance of your Vertica clusters. This tool provides a graphical view of your Vertica database cluster, nodes, network status, and detailed monitoring charts and graphs.

MC allows you to:

- Create, import, and connect to Vertica databases.
- Manage your Vertica database and clusters.
- Receive and view messages regarding the health and performance of your Vertica database and clusters.
- View diagnostics and support information for Management Console.
- Manage application and user settings for Management Console.

MC installation process

To install MC, complete these tasks:

1. Follow the steps listed in Installing Management Console .
2. After you have installed MC, configure it according to the instructions in Configuring Management Console .

In this section

- Connecting to MC
- Management Console home page
- Creating a cluster using MC
- Monitoring existing infrastructure using MC

Connecting to MC

To connect to Management Console:

1. Open a supported web browser .
2. Enter the IP address or host name of the host on which you installed MC (or any cluster node if you installed Vertica first), followed by the MC port you assigned when you configured MC . For example:
   Enter the IP address and port:
   `https://10.20.30.40:5450/`

   Enter the host name and port:
   `https://hostname:5450/`
3. When the MC logon dialog appears, enter your MC username and password and click **Log in** .

> **Note**
> When MC users log in to the MC interface, MC checks their privileges on Vertica Data collector (DC) tables on MC-monitored databases. Based on DC table privileges, along with the role assigned the MC user, each user's access to the MC's Overview, Activity and Node details pages could be limited. See Users, roles, and privileges for more information.

If you do not have an MC username/password, contact your MC administrator.

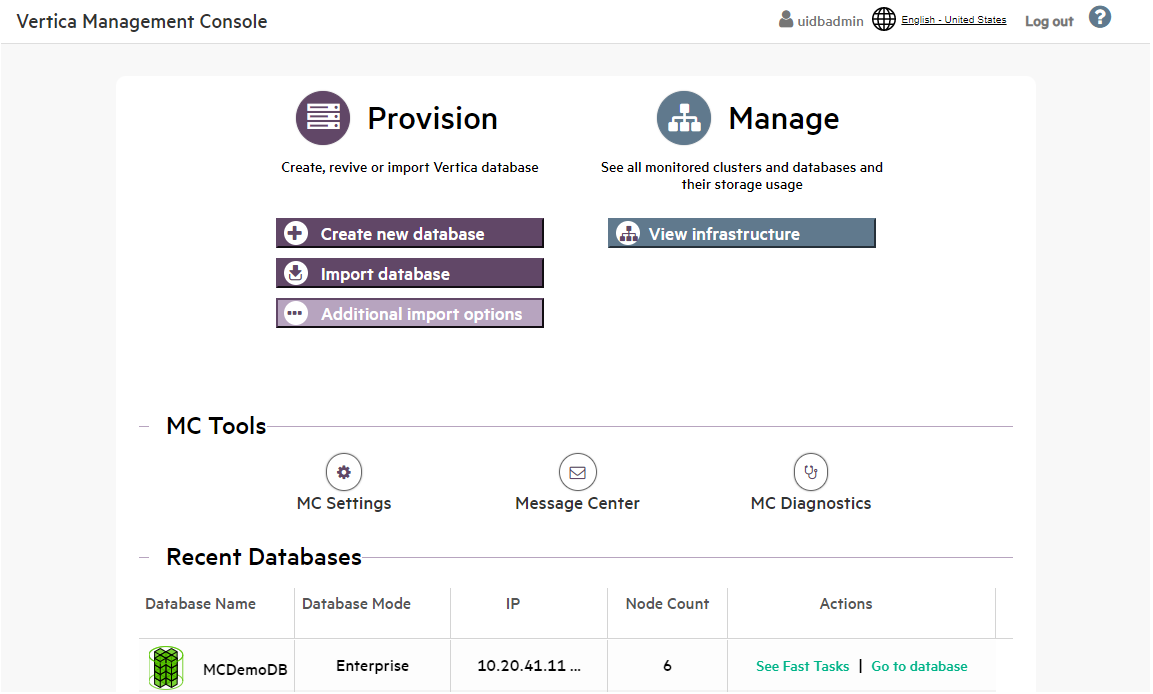
Managing client connections

Each client session to MC uses a connection from MaxClientSessions , a database configuration parameter. This parameter determines the maximum number of sessions that can run on a single database cluster node. Sometimes multiple MC users, mapped to the same database account, are concurrently monitoring the Overview and Activity pages.

> **Tip**
> You can increase the value for MaxClientSessions on an MC-monitored database to account for extra sessions. See Managing sessions for details.

Management Console home page

After you connect to MC and sign in, the Home page displays. This page is the entry point to all Vertica database clusters and users managed by MC. Information on this page, as well as throughout the MC interface, will appear or be hidden, based on the permissions (access levels) of the user who is logged in. The following image is what an MC super administrator sees.



Operations you can perform in Management Console are grouped into the following areas:

- **Provision** . Create new Vertica databases, or import existing ones to manage and monitor with MC. You can also import a Vertica cluster that resides in a Hadoop environment. See Managing database clusters .
- **Manage** . View all the clusters and databases monitored by MC, stop and remove databases, and view details about databases and clusters. See Viewing cluster infrastructure .

MC tools

- **MC Settings** . Configure MC and user settings, as well as use the MC interface to install Vertica on a cluster of hosts. See Management Console settings .
- **Message Center** . View, sort, and search database messages and optionally export messages to a file. See Monitoring Database Messages in MC.
- **MC Diagnostics** . View and resolve MC-related issues, as well as browse Vertica agent and audit logs. See Troubleshooting with MC diagnostics .

Recent databases

The **Recent Databases** section displays all databases that you created on or imported into MC. An empty space under Recent Databases means that you have not yet created or imported a database into the MC interface, or do not have permission to view any databases managed by MC.

Recent Databases lists each database name, its number of nodes, and two actions: **Fast Tasks** and **Go to Database.**

Fast tasks

Fast Tasks are key tasks you can perform on that database using MC. The Fast Tasks page provides the following options:

- **Manage and View Your Vertica Database** . See the monitoring dashboard for your database. Explore the tabs below your dashboard, which provide more ways to manage the database.
- **Connect to your Vertica Database Using SQL** . Use Management Console to run SQL queries on your database from within your browser.

- **Load Your Data into Vertica Database** . The Data Load Activity page allows you to monitor and perform data loading jobs. To load data from an S3 bucket into pre-existing tables in your database, select the Instance tab and click **New S3 Data Load.**

If you installed Management Console with Provisioning through the AWS Marketplace, you had the option to install a database pre-loaded with example clickstream analytic data. If you did so, the Fast Tasks page lists two additional links:

- **Play with Example Data in Your Vertica Database (PDF)** . This PDF guide provides instructions and example SQL queries you can use in Vertica to analyze the example data.
- **Vertica Workbook for Tableau (PDF)** . This PDF guide provides instructions on how to analyze the example data pre-loaded into Vertica, using a Tableau dashboard.

Go to database

View the Overview page , which displays a dynamic dashboard of your database's health and activity. Explore the tabs below your dashboard, which provide more ways to manage the database.

You can install and manage multiple databases with MC, but you can have only one database running on a single cluster at a time. UP databases appear in green and DOWN databases are red.

Creating a cluster using MC

Enterprise Mode only

After you install and configure MC, you can use Management Console to install a Vertica cluster on hosts where Vertica software has not been installed. The Cluster Installation assistant lets you specify the hosts you want to include in your Vertica cluster, loads the Vertica software onto the hosts, validates the hosts, and assembles the nodes into a cluster.

Complete the following tasks:

1. Prepare the hosts - Prepare each host that will become a node in the cluster.
2. Create a private key file - MC needs password-less SSH to connect to hosts and install Vertica software. Create a private key to enable MC access to the hosts.
3. Use the MC cluster installation wizard - Use the wizard to install a Vertica cluster on hosts that do not have Vertica software already installed on them.
4. Validate hosts and create the cluster - Host validation is the process where the MC runs tests against each host in a proposed cluster . You must validate hosts before the MC can install Vertica on each host.

After you successfully create a cluster using MC, see Create a database on a cluster .

In this section

- Prepare the hosts
- Create a private key file
- Use the MC cluster installation wizard
- Validate hosts and create the cluster
- Create a database on a cluster

Prepare the hosts

This topic applies only to on-premises installations.

Before you can install a Vertica cluster using the MC, you must prepare each host that will become a node in the cluster. The cluster creation process runs validation tests against each host before it attempts to install the Vertica software. These tests ensure that the host is correctly configured to run Vertica.

Validate the hosts

The validation tests provide:

- Warnings and error messages when they detect a configuration setting that conflicts with the Vertica requirements or any performance issue
- Suggestions for configuration changes when they detect an issue

> **Note**
> The validation tests do not automatically fix all problems they encounter.

All hosts must pass validation before the cluster can be created.

If you accepted the default configuration options when installing the OS on your host, then the validation tests will likely return errors, since some of the default options used on Linux systems conflict with Vertica requirements. See Operating system configuration overview for details on OS settings. To speed up the validation process you can perform the following steps on the prospective hosts before you attempt to validate the hosts. These steps are based on Red Hat Enterprise Linux and CentOS systems, but other supported platforms have similar settings.

Create a private key file

Before you can install a cluster, Management Console must be able to access the hosts on which you plan to install Vertica. MC uses password-less SSH to connect to the hosts and install Vertica software using a private key file.

If you already have a private key file that allows access to all hosts in the potential cluster, you can use it in the cluster creation wizard.

> **Note**
> The private key file is required to complete the MC cluster installation wizard.

Create a private key file

1. Log into the server as root or as a user with sudo privileges.
2. Change to your home directory.
   ```
   $ cd ~
   ```
3. Create an .ssh directory if one does not already exist.
   ```
   $ mkdir .ssh
   ```
4. Generate a passwordless private key/public key pair.
   ```
   $ ssh-keygen -q -t rsa -f ~/.ssh/vid_rsa -N ''
   ```
   This command creates two files: *vid_rsa* and *vid_rsa.pub* . The vid_rsa file is the private key file that you upload to the MC so that it can access nodes on the cluster and install Vertica. The vid_rsa.pub file is copied to all other hosts so that they can be accessed by clients using the vid_rsa file.
5. Make your .ssh directory readable and writable only by yourself.
   ```
   $ chmod 700 /root/.ssh
   ```
6. Change to the .ssh directory.
   ```
   $ cd ~/.ssh
   ```
7. Edit sshd.config as follows to disable password authentication for root:
   ```
   PermitRootLogin without-password
   ```
8. Concatenate the public key into to the file vauthorized_keys2.
   ```
   $ cat vid_rsa.pub >> vauthorized_keys2
   ```
9. If the host from which you are creating the public key will also be in the cluster, copy the public key into the local-hosts authorized key file:
   ```
   cat vid_rsa.pub >> authorized_keys
   ```
10. Make the files in your .ssh directory readable and writable only by yourself.
    ```
    $ chmod 600 ~/.ssh/*
    ```
11. Create the .ssh directory on the other nodes.
    ```
    $ ssh <host> "mkdir /root/.ssh"
    ```
12. Copy the vauthorized key file to the other nodes.
    ```
    $ scp -r /root/.ssh/vauthorized_keys2 <host>:/root/.ssh/.
    ```
13. On each node, concatenate the vauthorized_keys2 public key to the authorized_keys file and make the file readable and writable only by the owner.
    ```
    $ ssh <host> "cd /root/.ssh/;cat vauthorized_keys2 >> authorized_keys; chmod 600 /root/.ssh/authorized_keys"
    ```
14. On each node, remove the vauthorized_keys2 file.
    ```
    $ ssh -i /root/.ssh/vid_rsa <host> "rm /root/.ssh/vauthorized_keys2"
    ```
15. Copy the *vid_rsa* file to the workstation from which you will access the MC cluster installation wizard. This file is required to install a cluster from the MC.

A complete example of the commands for creating the public key and allowing access to three hosts from the key is below. The commands are being initiated from the docg01 host, and all hosts will be included in the cluster (docg01 - docg03):

```
ssh docg01
cd ~/.ssh
ssh-keygen -q -t rsa -f ~/.ssh/vid_rsa -N ''
cat vid_rsa.pub > vauthorized_keys2
cat vid_rsa.pub >> authorized_keys
chmod 600 ~/.ssh/*
scp -r /root/.ssh/vauthorized_keys2 docg02:/root/.ssh/.
scp -r /root/.ssh/vauthorized_keys2 docg03:/root/.ssh/.
ssh docg02 "cd /root/.ssh/;cat vauthorized_keys2 >> authorized_keys; chmod 600 /root/.ssh/authorized_keys"
ssh docg03 "cd /root/.ssh/;cat vauthorized_keys2 >> authorized_keys; chmod 600 /root/.ssh/authorized_keys"
ssh -i /root/.ssh/vid_rsa docg02 "rm /root/.ssh/vauthorized_keys2"
ssh -i /root/.ssh/vid_rsa docg03 "rm /root/.ssh/vauthorized_keys2"
rm ~/.ssh/vauthorized_keys2
```

Use the MC cluster installation wizard

The Cluster Installation Wizard guides you through the steps required to install a Vertica cluster on hosts that do not already have Vertica software installed.

> **Note**
> If you are using MC with the Vertica AMI on Amazon Web Services, note that the Create Cluster and Import Cluster options are not supported.

Prerequisites

Before you proceed, make sure you:

- Installed and configured Management Console .
- Prepared the hosts that you will include in the Vertica database cluster.
- Created the private key (pem) file and copied it to your local machine.
- Obtained a copy of your Vertica license if you are installing the Premium Edition. If you are using the Community Edition, a license key is not required.
- Downloaded the Vertica server RPM (or DEB file).
- Have read/copy permissions on files stored on the local browser host that you will transfer to the host on which MC is installed.

Permissions on files to transfer to MC

On your local workstation, you must have at least read/write privileges on files you'll upload to MC through the Cluster Installation Wizard. These files include the Vertica server package, the license key (if needed), the private key file, and an optional CSV file of IP addresses.

Create a Vertica cluster using MC

1. Connect to Management Console and log in as an MC administrator.
2. On MC's Home page, click the **Provisioning** task. The Provisioning dialog appears.
3. Click **Create a cluster** .
4. The Create Cluster wizard opens. Provide the following information:
    1. Cluster name—A label for the cluster. Choose a name that is unique within MC. IF you do not enter a name here, MC assigns a random unique cluster name. You can edit the name later when you view the cluster on the Infrastructure page. Note that this name is an alias that exists only in MC. If you reimport the cluster, you would need to edit the cluster name again to reestablish this name.
    2. Vertica Admin User—The user that is created on each of the nodes when they are installed, typically 'dbadmin'. This user has access to Vertica and is also an OS user on the host.
    3. Password for the Vertica Admin User—The password you enter (required) is set for each node when MC installs Vertica.

        > **Note**
        > MC does not support an empty password for the administrative user.

    4. Vertica Admin Path—Storage location for catalog files, which defaults to /home/dbadmin unless you specified a different path during MC configuration (or later on MC's Settings page).
    Important

The Vertica Admin Path must be the same as the Linux database administrator's home directory. If you specify a path that is not the Linux dbadmin's home directory, MC returns an error.

5. Click **Next** and specify the private key file and host information:
   1. Click **Browse** and navigate to the private key file (vid_rsa) that you created earlier.

      > **Note**
      > You can change the private key file at the beginning of the validation stage by clicking the name of the private key file in the bottom-left corner of the page. However, you cannot change the private key file after validation has begun unless the first host fails validation due to an SSH login error.

   2. Include the host IP addresses. You have three options:
      Specify later (but include number of nodes). This option allows you to specify the number of nodes, but not the specific IPs. You can specify the specific IPs before you validate hosts.
      Import IP addresses from local file . You can specify the hosts in a CSV file using either IP addresses or host names.
      Enter a range of IP addresses . You can specify a range of IPs to use for new nodes. For example 192.168.1.10 to 192.168.1.30. The range of IPs must be on the same or contiguous subnets.
6. Click **Next** and select the software and license:
   1. Vertica Software. If one or more Vertica packages have been uploaded, you can select one from the list. Otherwise, select **Upload a new local vertica binary file** and browse to a Vertica server file on your local system.
   2. Vertica License. Click **Browse** and navigate to a local copy of your Vertica license if you are installing the Premium Edition. Community Edition versions require no license key.
7. Click **Next** . The Create cluster page opens. If you did not specify the IP addresses, select each host icon and provide an IP address by entering the IP in the box and clicking **Apply** for each host you add.

You are now ready to Validate hosts and create the cluster .

Validate hosts and create the cluster

Host validation is the process where the MC runs tests against each host in a proposed cluster .

You can validate hosts only after you have completed the cluster installation wizard. You must validate hosts before the MC can install Vertica on each host.

At any time during the validation process, but before you create the cluster, you can add and remove hosts by clicking the appropriate button in the upper left corner of the page on MC. A Create Cluster button appears when all hosts that appear in the node list are validated.

How to validate hosts

To validate one or more hosts:

1. Connect to Management Console and log in as an MC administrator.
2. On the MC Home page, click the **Databases and Clusters** task.
3. In the list of databases and clusters, select the cluster on which you have recently run the cluster installation wizard (**Creating...** appears under the cluster) and click **View** .
4. Validate one or several hosts:
   - To validate a single host, click the host icon, then click **Validate Host** .
   - To validate all hosts at the same time, click **All** in the Node List, then click **Validate Host** .
   - To validate more than one host, but not all of them, Ctrl+click the host numbers in the node list, then click **Validate Host** .
5. Wait while validation proceeds.
   The validation step takes several minutes to complete. The tests run in parallel for each host, so the number of hosts does not necessarily increase the amount of time it takes to validate all the hosts if you validate them at the same time. Hosts validation results in one of three possible states:
   - Green check mark—The host is valid and can be included in the cluster.
   - Orange triangle—The host can be added to the cluster, but warnings were generated. Click the tests in the host validation window to see details about the warnings.
   - Red X—The host is not valid. Click the tests in the host validation window that have red X's to see details about the errors. You must correct the errors re-validate or remove the host before MC can create the cluster.
     To remove an invalid host: Highlight the host icon or the IP address in the Node List and click **Remove Host** .

All hosts must be valid before you can create the cluster. Once all hosts are valid, a **Create Cluster** button appears near the top right corner of the page.

How to create the cluster

1. Click **Create Cluster** to install Vertica on each host and assemble the nodes into a cluster.
   The process, done in parallel, takes a few minutes as the software is copied to each host and installed.
2. Wait for the process to complete. When the **Success** dialog opens, you can do one of the following:
   - Optionally create a database on the new cluster at this time by clicking **Create Database**
   - Click **Done** to create the database at a later time

See Creating a Database on a Cluster for details on creating a database on the new cluster.

Create a database on a cluster

After you use the MC Cluster Installation Wizard to create a Vertica cluster, you can create a database on that cluster through the MC interface. You can create the database on all cluster nodes or on a subset of nodes.

If a database had been created using the Administration Tools on any of the nodes, MC detects (autodiscovers) that database and displays it on the Manage (Cluster Administration) page so you can import it into the MC interface and begin monitoring it.

MC allows only one database running on a cluster at a time, so you might need to stop a running database before you can create a new one.

The following procedure describes how to create a database on a cluster that you created using the MC Cluster Installation Wizard . To create a database on a cluster that you created by running the install_vertica script, see Creating an Empty Database .

Create a database on a cluster

To create a new empty database on a new cluster:

1. If you are already on the **Databases and Clusters** page, skip to the next step. Otherwise:
   1. Connect to MC and sign in as an MC administrator.
   2. On the Home page, click **Existing Infrastructure** .
2. If no databases exist on the cluster, continue to the next step. Otherwise:
   1. If a database is running on the cluster on which you want to add a new database, select the database and click **Stop** .
   2. Wait for the running database to have a status of *Stopped* .
3. Click the cluster on which you want to create the new database and click **Create Database** .
4. The Create Database wizard opens. Provide the following information:
   - Database name and password. See Creating a database name and password for rules.
   - Optionally click **Advanced** to open the advanced settings and change the port, and catalog path, and data path. By default the MC application/web server port is 5450 and paths are /home/dbadmin , or whatever you defined for the paths when you ran the cluster creation wizard. Do not use the default agent port 5444 as a new setting for the MC application/web server port. See **MC Settings > Configuration** for port values.
5. Click **Continue** .
6. Select nodes to include in the database.
   The Database Configuration window opens with the options you provided and a graphical representation of the nodes appears on the page. By default, all nodes are selected to be part of this database (denoted by a green check mark). You can optionally click each node and clear **Include host in new database** to exclude that node from the database. Excluded nodes are gray. If you change your mind, click the node and select the **Include** check box.
7. Click **Create** in the Database Configuration window to create the database on the nodes.
   The creation process takes a few moments and then the database is started and a **Success** message appears.
8. Click **OK** to close the success message.
   The Database Manager page opens and displays the database nodes. Nodes not included in the database are gray.

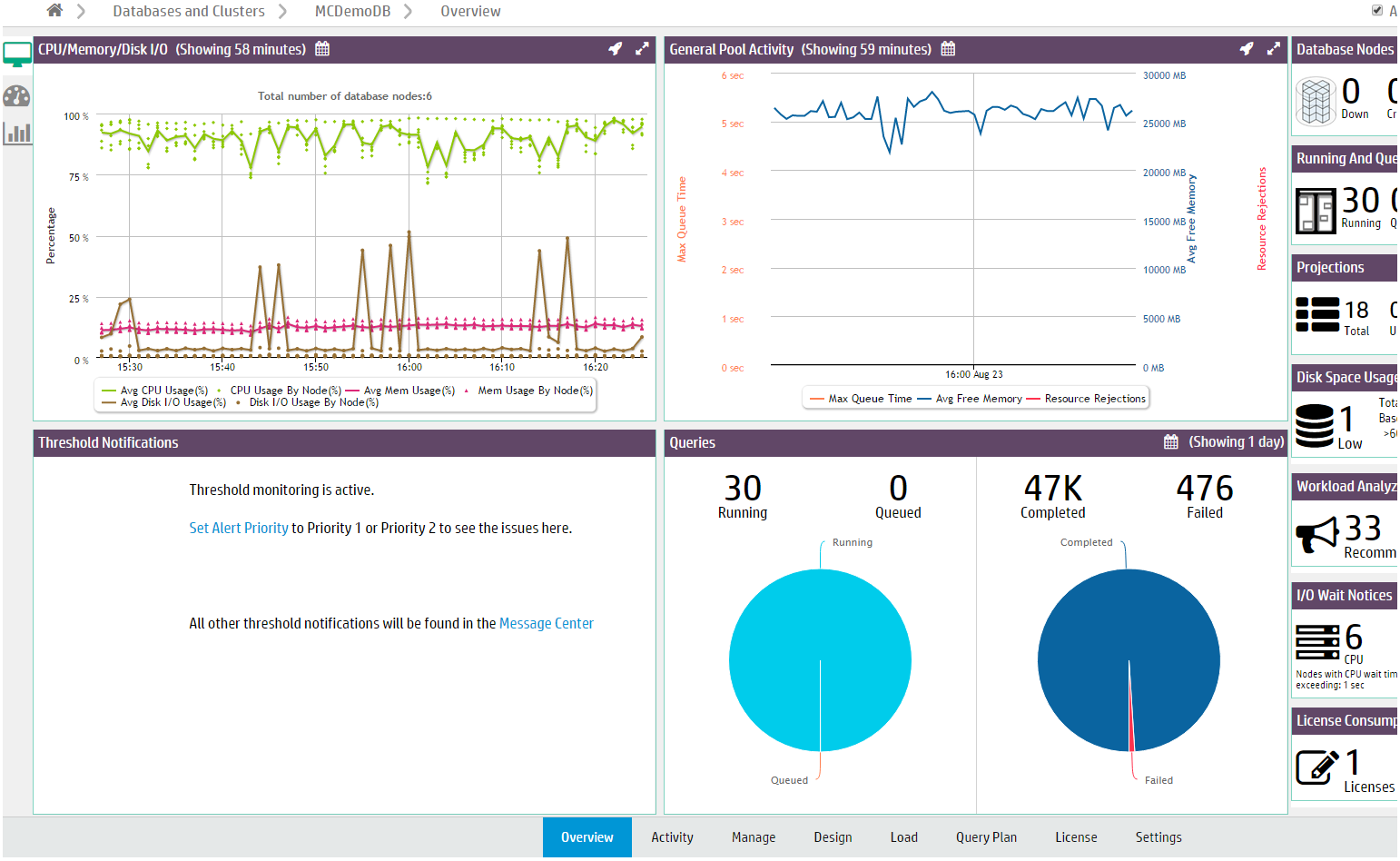
Monitoring existing infrastructure using MC

Use Management Console to monitor the health of your Vertica databases and clusters. Click the **Infrastructure** button on the Home page to see the Databases and Clusters page. Then click the cluster of interest to view the health of the nodes in that cluster and the key information associated with the cluster such as:

- Vertica version
- Number of hosts
- CPU type
- Last updated date
- Node list.

You can also zoom in and out for better view of this page.

On the Databases and Clusters page or the Home page, click the database which you want to monitor, to go to its **Overview** page:



You can perform the following tasks from the Overview page:

- View **Quick Stats** to get instant alerts and information about your cluster's status.
- View **Status Summary** that provides a general overview of the status of your cluster (as shown in preceding figure).
- Analyze **System Health** using a comprehensive summary of your system resource usage and node information, with configurable statistics that allow you to specify acceptable ranges of resource usage.
- Use **Query Synopsis** to monitor system query activity and resource pool usage.

For Eon mode databases, the **Status Summary** and **Query Synopsis** pages allow you to display information for the entire database. If subclusters are defined, you can also display information for a specific subcluster or node, or the nodes not assigned to a subcluster.

Additionally, you can perform the following tasks from the Overview page:

- Monitoring cluster nodes with MC
- Monitoring cluster CPU and Memory with MC
- Monitoring cluster performance with MC

Monitoring system resources

On the main window, you can click the database, and navigate to the **MC Activity** tab to monitor system resources such as:

- Queries
- Internal Sessions
- User Sessions
- Memory Usage
- System Bottlenecks
- User Query Phases
- Table Treemap
- Query Monitoring
- Resource Pool Monitoring

Monitoring node and MC user activity

You can use the **MC Manage** page to monitor node activity. When you click the node you want to investigate, the Node Detail page opens and provides:

- Summary information for the node
- Resources consumed by the node for last three hours

You can also browse and export log-level data from AgentTools and Vertica log files. MC retains a maximum of 2000 log records. See Monitoring node activity with MC for further details.

Use **MC Diagnostics** tab and navigate to **Audit Log** page to manage MC User activity. See Monitoring MC user activity using audit log .

Monitoring messages in databases managed by MC

You can view critical database related messages from **MC Message Center** . The **MC Message Center** reports on several critical database-related conditions using a color code to indicate the message severity. For further details, see Monitoring database messages and alerts with MC .

You can also search and sort database messages, mark messages read or unread and delete them. You can filter messages by message type, and export messages. For additional information, refer to Message center and Exporting MC-managed database messages and logs .

Monitoring and configuring resource pools

Use the **MC Activity** page to monitor resource pools. Select the resource pool you want to monitor. MC displays the following charts for the selected pool:

- Resource Usages in Pool
- Memory Usage in Node and Subclusters
- Average Query Execution and Query Time in Pool
- Resource Rejections in Pool

If you are a database administrator, you can click the database you want on the main window. You can then use the **MC Settings** tab to view and edit the resource pool parameters. Only the database administrator can monitor and configure the resource pools in Management Console.

See Monitoring resource pools with MC for further information.

Users, roles, and privileges

If you are a Management Console (MC) administrator, you can use **MC Settings** to grant MC users privileges to one or more Vertica users.

Users

MC users are not the same as system (Linux) users. MC users are external to the database, and their information is stored on an internal database on the MC application or web server. See Users in Management Console for further details.

The MC super account is the default user. The super user needs to create all other MC users.

You can create MC users using either of two authentication techniques, **LDAP** or **MC (internal)** . See Creating an MC user . After you create the MC users, you can manage them from **MC Settings** page. Refer to Managing MC users .

To control the level of access for the MC Users, you can grant them privileges (through roles) from the **MC Settings** page.

Roles and privileges

You control user access to MC through groups of privileges (also referred to as access levels) that fall into two types, those that apply to MC configuration, and those that apply to MC-managed Vertica databases.

MC permission groups

- MC configuration privileges are made up of roles that control what users can configure on the Management Console, such as modify MC settings, create and import Vertica databases, restart MC, create a Vertica cluster through the MC interface, and create and manage MC users.
- MC database privileges are made up of roles that control what users can see or do on a Vertica database monitored by MC, such as view the database cluster state, query and session activity, monitor database messages and read log files, replace cluster nodes, and stop databases.

> **Note**
> When you grant an MC user a database role, that user inherits the privileges assigned to the database user account to which the MC user is mapped. For maximum access, use the dbadmin username and password.

MC database privileges cannot alter or override the Vertica database user's privileges and roles.

> **Note**
> If TLS/SSL is configured in mutual mode on the Vertica database, each MC user must be configured with an individual client certificate and

private key, to log into the database from MC. See Configuring mutual TLS for MC users . If the individual certificate has not been configured, you see an error message. See your Management Console administrator.

See also

- Users in Management Console
- Creating an MC user
- Database privileges
- Creating an MC user
- Granting database access to MC users

In this section

- Users in Management Console
- Configuration privileges
- Database privileges

Users in Management Console

Unlike database users, which you create on the Vertica database and then grant privileges and roles through SQL statements, you create MC users on the Management Console interface. MC users are external to the database. Their information is stored on an internal database on the MC application/web server. Their access to both MC and to databases managed by MC is controlled by groups of privileges (also referred to as access levels). MC users are not system (Linux) users; they are entries in the MC internal database.

Permission group types

There are two types of permission groups on MC, those that apply to MC configuration and those that apply to database access:

- MC configuration privileges are made up of roles that control what users can configure on the Management Console, such as modify MC settings, create and import Vertica databases, restart MC, create a Vertica cluster through the MC interface, and create and manage MC users.
- MC database privileges are made up of roles that control what users can see or do on a Vertica database monitored by MC, such as view the database cluster state, query and session activity, monitor database messages and read log files, replace cluster nodes, and stop databases.

If you are using MC, you might want to allow one or more users in your organization to configure and manage MC, and you might want other users to have database access only. You can meet these requirements by creating MC users and granting them a role from each privileges group. See Creating an MC user for details.

MC user types

The following table describes the five types of role-based users on MC:

User Type	Description
SUPER Role (MC)	The default superuser administrator (Linux account) who gets created when you install and configure MC and oversees all of MC.
ADMIN Role (MC)	Users who can configure all aspects of MC and control all databases managed by MC.
MANAGER Role (MC)	Users who can configure MC user settings and monitor all databases managed by MC.
IT Role (MC)	Users who can configure some aspects of MC user settings and monitor all databases managed by MC.
NONE Role (MC)	Users who cannot configure MC and have access to one or more databases managed by MC.

For details about each role, see Configuration privileges .

You create users and grant them privileges (through roles) on the **MC Settings** page in the **User management** tab.

Creating users and choosing an authentication method

You create users and grant them privileges (through roles) on the **MC Settings** page. You can also choose how to authenticate their access to MC.

- To add users who are authenticated against the MC, click **User Management**
- To add users who are authenticated through your organization's LDAP repository, click **Authentication**

MC supports only one method for authentication, so if you choose MC, all MC users will be authenticated using their MC login credentials.

Default MC users

The MC super account is the only default user. The super or another MC administrator must create all other MC users.

See also

- Users, roles, and privileges
- Granting database access to MC users

In this section

- Creating an MC user
- Managing MC users
- Granting database access to MC users
- Managing MC user passwords

Creating an MC user

MC provides two authentication schemes for MC users: LDAP or MC (internal). The method you choose when you configure MC is the method MC uses to authenticate *all* MC users. It is not possible to authenticate some MC users against LDAP and other MC users against credentials in the database through MC.

- **MC (internal) authentication.** Internal user authorization is specific to MC itself. You create a user with a username and password combination. This method stores MC user information in an internal database on the MC application/web server, and encrypts passwords. Note that these MC users are not system (Linux) users; they are entries in the MC's internal database.
- **LDAP authentication.** All MC users—except for the MC super administrator, which is a Linux account—are authenticated based on search criteria against your organization's LDAP repository. MC uses information from LDAP for authentication purposes only and does not modify LDAP information. Also, MC does not store LDAP passwords but passes them to the LDAP server for authentication.

Instructions for creating new MC users are in this topic.

- If you chose MC authentication, follow the instructions under **Create a New User Authenticated by MC** .
- If you chose LDAP authentication, follow the instructions under **Create a New User from LDAP** .

See Configuring Management Console , Users in Management Console and LDAP authentication for more information.

Prerequisites

Before you create an MC user, ensure that:

- You have created a database directly on the server or through the MC interface, or you imported an existing database cluster into the MC interface. See Managing database clusters .
- You have created a database user account (source user) on the server, which has the privileges and/or roles you want to map to the new (target) MC user. See Creating a database user .
- You know which MC privileges you want to grant to the new MC user. See Users, roles, and privileges .
- You will be mapping the MC user to a Vertica DB user who has sysmonitor privileges assigned, or to the Vertica database super user. Without sysmonitor (or super user) privileges, the mapped MC user will not be able to view information in MC monitoring tables, and will not be able to load Kafka streaming data.

If you have not yet met the first two above prerequisites, you can still create new MC users; you just won't be able to map them to a database until after the database and target database user exist. To grant MC users database access later, see Granting database access to MC users .

Create a new user authenticated by MC

1. Sign in to MC as an administrator and navigate to **MC Settings > User Management** .
2. Click **Add** .
3. Enter the MC username.

> **Note**
> It is not necessary to give the MC user the exact same name as the database user account you'll map the MC user to in Step 7. What matters is that the source database user has privileges and/or roles similar to the database role you want to grant the MC user. The most likely scenario is that you map multiple MC users to a single database user account.

4. Let MC generate a password or create one by clicking **Edit password** . If LDAP has been configured, the MC password field will not appear.
5. Optionally enter the user's e-mail address.
6. Select an **MC configuration permissions** level. See Configuration privileges . Your choice in this field also fills in the appropriate **User API Key** value.

7. Next to the **DB access levels** section, click **Add** to grant this user database permissions.
    1. **Choose a database.** Select a database from the list of MC-discovered (databases that were created on or imported into the MC interface).
    2. **Database username.** Enter an existing database user name or, if the database is running, click the ellipsis [...] to browse for a list of database users, and select a name from the list.
    3. **Database password.** Enter the password to the database user account (not this username's password).
    4. **Restricted access.** Choose a database level (ADMIN, IT, or USER) for this user.
    5. Click **OK** to close the **Add permissions** dialog box.
    6. If the Vertica database is configured to require TLS, select **Yes** in the **Use TLS Connection** drop-down. MC launches the Certificates wizard to let you configure TLS. See MC certificates wizard .
8. Leave the user's **Status** as enabled (the default). If you need to prevent this user from accessing MC, select disabled.
9. Click **Add User** to finish.

Create a new LDAP-authenticated user

When you add a user from LDAP on the MC interface, options on the **Add a new user** dialog box are slightly different from when you create users without LDAP authentication. Because passwords are store externally (LDAP server) the password field does not appear. An MC administrator can override the default LDAP search string if the user is found in another branch of the tree. The **Add user** field is pre-populated with the default search path entered when LDAP was configured.

1. Sign in to MC and navigate to **MC Settings > User management** .
2. Click **Add** and provide the following information:
    1. LDAP user name.
    2. LDAP search string.
    3. User attribute, and click **Verify user** .
    4. User's email address.
    5. MC configuration role. NONE is the default. See Configuration privileges for details.
    6. Database access level. See Database privileges for details.
    7. Accept or change the default user's **Status** (enabled).
3. Click **Add user** .

If you encounter issues when creating new users from LDAP, you'll need to contact your organization's IT department.

How MC validates new users

After you click **OK** to close the Add permissions dialog box, MC tries to validate the database username and password entered against the selected MC-managed database or against your organization's LDAP directory. If the credentials are found to be invalid, you are asked to re-enter them.

If the database is not available at the time you create the new user, MC saves the username/password and prompts for validation when the user accesses the Database and Clusters page later.

See also

- Configuring Management Console
- Users, roles, and privileges
- Users in Management Console
- Granting database access to MC users
- Creating a database user

Managing MC users

You manage MC users through the following pages on the Management Console interface:

- **MC Settings > User management**
- **MC Settings > Resource access**

Who manages users

The MC superuser administrator (SUPER Role (MC)) and users granted ADMIN role (MC) manage all aspects of users, including their access to MC and to MC-managed databases.

Users granted IT Role (MC) can enable and disable user accounts. For more information, see Users, roles, and privileges and Users in Management Console .

Editing an MC user's information follows almost the same steps as creating a new user , except that you select an existing user and click Edit. The user's information will be pre-populated, so that you can edit and save it.

The only user account you cannot alter or remove from the MC interface is the MC super account.

For details about each user type, see Configuration privileges .

What kind of user information you can manage

You can change the following user properties:

- MC password
- Email address. This field is optional. If the user is authenticated against LDAP, the email field is pre-populated with that user's email address if one exists.
- Configuration privileges role
- Database privileges role

You can also change a user's status (enable/disable access to MC) and delete users.

About user names

After you create and save a user, you cannot change that user's MC user name, but you can delete the user account and create a new user account under a new name. The only thing you lose by deleting a user account is its audit activity, but MC immediately resumes logging activity under the user's new account.

Granting database access to MC users

If you did not grant an MC user a database-level role when you created the user account, you can do so in the User Management tab in MC Settings.

Granting the user an MC database-level role associates the MC user with a database user's privileges and ensures that the MC user cannot do or see anything not allowed by the privileges set up for the user account on the server database. When that MC user logs in to MC, his or her MC privileges for database-related activities are compared to that user's privileges on the database itself. Only when the user has both MC privileges and corresponding database privileges will the operations be exposed in the MC interface.

Prerequisites

Before you grant database access to an MC user, see the prerequisites in Creating an MC user .

Grant a database-level role to an MC user

1. Log in to Management Console as an administrator and navigate to **MC Settings > User management** .
2. Select an MC user and click **Edit** .
3. Verify the Configuration privileges are what you want them to be. NONE is the default.
4. Next to the **DB access levels section** , click **Add** and provide the following database access credentials:
    1. **Choose a database.** Select a database from the list of MC-discovered (databases that were created on or imported into the MC interface).
    2. **Database username.** Enter an existing database user name or, if the database is running, click the ellipsis [...] to browse for a list of database users, and select a name from the list.
    3. **Database password.** Enter the password to the database user account (not this username's password).
    4. **Restricted access.** Choose a database level (ADMIN, IT, or USER) for this user.
    5. Click **OK** to close the **Add permissions** dialog box.
    6. If the Vertica database is configured to require TLS, select **Yes** in the **Use TLS Connection** drop-down. MC launches the Certificates wizard to let you configure TLS. See MC certificates wizard .
5. Optionally change the user's **Status** (enabled is the default).
6. Click **Save** .

How MC validates new users

After you click **OK** to close the Add permissions dialog box, MC tries to validate the database username and password entered against the selected MC-managed database or against your organization's LDAP directory. If the credentials are found to be invalid, you are asked to re-enter them.

If the database is not available at the time you create the new user, MC saves the username/password and prompts for validation when the user accesses the Database and Clusters page later.

Managing MC user passwords

MC user passwords must be different from the username and must contain at least one of each of the following character types:

- Upper-case
- Lower-case
- Numeric
- Special: ~ ! @ # $ % ^ & * ( ) _ + = - [ ] / ? > < , .

Configurable password requirements

You can configure additional password requirements by navigating to **Home** > **MC Settings** > **Configuration** > **MC Password configuration settings** .

Parameter	Settings
Minimum password length	The minimum number of characters required in a password. • Minimum: 8 (default) • Maximum: 30

Security questions

When a user creates an MC account, they must set security questions. If a user forgets their password, they can reset their password by answering these questions. These answers:

- must only contain letters, numbers, and spaces
- must be between 2 and 30 characters in length, inclusive
- are case insensitive

Configuration privileges

When you create a Management Console (MC) user, you assign them an MC configuration access level (role). MC roles control a user's ability to create users and manage MC settings on the MC interface.

You can assign a user one of the following MC access levels:

- ADMIN Role (MC) : Full access to all MC functionality.
- Manager Role (MC) : Access to MC user management functionality. Access to non-database MC alerts.
- IT Role (MC) : Limited access to MC user management functionality. Access to MC log and to non-database MC alerts.
- NONE Role (MC) : Database access only, to the databases an administrator assigns to this user.

You grant MC configuration privileges at the same time you create the user's account, on the User Management tab of the MC Settings page. You can change MC access levels using this page. See Creating an MC user for details.

You can also use the User Management tab to grant users access to one or more databases managed by MC. See Database privileges for details.

SUPER role (MC)

The default superuser administrator, called **Super** on the MC UI, is a Linux user account that gets created when you install and configure MC . During the configuration process, you can assign the Super any name you like; it need not be dbadmin.

The MC SUPER role, a superset of the ADMIN Role (MC) , has the following privileges:

- Oversees the entire Management Console, including all MC-managed database clusters

  > **Note**
  > This user inherits the privileges/roles of the user name supplied when importing a Vertica database into MC. Vertica recommends that you use the database administrator's credentials.

- Creates the first MC user accounts and assigns them an MC configuration role
- Grants MC users access to one or more MC-managed Vertica databases by assigning Database privileges to each user

The MC super administrator account is unique. Unlike other MC users you create, including other MC administrators, the MC super account cannot be altered or dropped, and you cannot grant the SUPER role to other MC users. The only property you can change for the MC super is the password. Otherwise the SUPER role has the same privileges on MC as the ADMIN Role (MC) .

On MC-managed Vertica databases, SUPER has the same privileges as ADMIN Role (DB) .

The MC super account does not exist within the LDAP server. This account is also different from the special dbadmin account that gets created during a Vertica installation, whose privileges are governed by the DBADMIN . The Vertica-created dbadmin is a Linux account that owns the database catalog and storage locations and can bypass database authorization rules, such as creating or dropping schemas, roles, and users. The MC super does not have the same privileges as dbadmin.

ADMIN role (MC)

This user account is the user who can perform all administrative operations on Management Console, including configure and restart the MC process and add, change, and remove all user accounts. By default, MC administrators inherit the database privileges of the main database user account used to set up the database on the MC interface. Therefore, MC administrators have access to all MC-managed databases. Grant the ADMIN role to users you want to be MC administrators.

The difference between this ADMIN user and the default Linux account, the MC SUPER Role , is you cannot alter or delete the MC SUPER account, and you can't grant the SUPER role to any other MC users. You can, however, change the access level for other MC administrators, and you can delete this user's accounts from the MC interface.

There is also the ADMIN Role (DB) that controls a user's access to MC-managed databases. The two ADMIN roles are similar, but they are not the same, and you do not need to grant users with the ADMIN (mc) role an ADMIN (db) role because MC ADMIN users automatically inherit all database privileges of the main database user account that was created on or imported into MC.

MANAGER role (MC)

Users assigned the Manager role can configure user settings in MC. The Manager role allows full access to the User Management tab in MC Settings. Managers can also view a full list of databases monitored by MC on the Home page, view the MC log, and see non-database MC alerts.

The Manager role has similar configuration privileges to the IT configuration role. Unlike IT users, Managers can also create, edit, and delete users in User Settings.

IT role (MC)

MC IT users can monitor all MC-managed databases, view MC-level (non database) messages, logs, and alerts, disable or enable user access to MC, and reset non-LDAP user passwords. You can also assign MC IT users specific database privileges, which you do by mapping IT users to a user on a database. In this way, the MC IT user inherits the privileges assigned to the database user that they are mapped to.

There is also an IT Role (DB) that controls a user's access to MC-managed databases. If you grant an MC user both IT roles, it means the user can perform some configuration on MC and also has access to one or more MC-managed databases. The database mapping is not required, but it gives the IT user wider privileges.

NONE role (MC)

The default role for all newly-created users on MC is NONE, which prevents users granted this role from configuring the MC. When you create MC users with the NONE role, you grant them an MC database-level role . This assignment maps the MC user to a user account on a specific database and specifies that the NONE user inherits the database user's privileges to which he or she is mapped.

Which database-level role you grant this user with NONE privileges—whether ADMIN (db) or IT (db) or USER (db)—depends on the level of access you want the user to have on the MC-managed database. Database roles have no impact on the ADMIN and IT roles at the MC configuration level.

MC configuration privileges by user role

You grant the following configuration privileges by MC role.

MC access privileges	ADMIN	MANAGER	IT	NONE
Configure MC settings: • Configure storage locations and ports • Upload new SSL certificates • Manage LDAP authentication • Update Vertica installation • Change MC theme • Map to an external data source	Yes			
Configure user settings: • Add, edit, delete users • Add, change, delete user permissions • Map users to one or more databases	Yes	Yes		
Configure user settings: • Enable or disable user access to MC • Reset user passwords	Yes	Yes	Yes	

Monitor user activity on MC using audit log	Yes			
Create and manage databases and clusters: • Create a new database or import an existing one • Create a new cluster or import an existing one • Remove databases and clusters from MC	Yes			
Reset MC to its original, preconfigured state	Yes			
Restart Management Console	Yes			
View full list of databases monitored by MC	Yes	Yes	Yes	
View MC log	Yes		Yes	
View non-database MC alerts	Yes	Yes	Yes	Yes

See also

- Users, roles, and privileges
- Users in Management Console
- Users, roles, and privileges
- Database privileges
- Creating an MC user
- Granting database access to MC users

Database privileges

When you create Management Console (MC) users , you first assign them MC configuration privileges, which controls what they can do on the MC itself. In the same user-creation operation, you grant access to one or more MC-managed databases. MC database access does not give the MC user privileges directly on Vertica; it provides MC users varying levels of access to assigned database functionality through the MC interface.

Assign users an MC database level through one of the following roles:

- ADMIN Role (DB) : Full access to all databases managed by MC. Actual privileges ADMINs inherit depend on the database user account used to create or import the Vertica database into the MC interface.
- Associate Role (DB) : Full access to all databases managed by MC. Cannot start, stop, or drop a database. Actual privileges that Associates receive depend on those defined for the database user account to which the Associate user is mapped.
- IT Role (DB) : Can start and stop a database but cannot remove it from the MC interface or drop it.
- USER Role (DB) : Can view database information through the database Overview and Activities pages but is restricted from viewing more detailed data.

ADMIN role (DB)

ADMIN is a superuser with full privileges to monitor MC-managed database activity and messages. Other database privileges (such as stop or drop the database) are governed by the user account on the Vertica database that this ADMIN (db) user is mapped to. ADMIN is the most permissive role and is a superset of privileges granted to the Associate, IT, and USER roles .

> **Note**
> Database access granted through Management Console never overrides roles granted on a specific Vertica database.

There is also an MC configuration administrator role that defines what the user can change on the MC itself. The two ADMIN roles are not the same. Unlike the MC configuration role of ADMIN, which can manage all MC users and all databases imported into the UI, the MC database ADMIN role has privileges only on the databases you map this user to. See ADMIN Role (MC) for additional details.

Associate role (DB)

The Associate role is an MC database access role. It is similar to the Admin role. It has privileges to monitor activity and messages on databases managed by MC. Unlike Admin users, Associate users cannot start, stop, or drop the database. The Associate user role is mapped to a user account on the database. This mapped user role determines what other database privileges the Associate role has (such as modifying settings, installing licenses, and viewing the database designer).

The following database operations depend on the database user's role that you mapped this Associate user to:

- Install or audit a license
- Manage database settings
- View Database Designer
- View the database Activity page

> **Note**
> Database access granted through Management Console never overrides roles granted on a specific Vertica database.

IT role (DB)

IT can view most details about an MC-managed database, such as messages (and mark them read/unread), the database overall health and activity/resources, cluster and node state, and MC settings. You grant and manage user role assignments through the **MC Settings > User management** page on the MC.

There is also an IT role at the MC configuration access level. The two IT roles are similar, but they are not the same. If you grant an MC user both IT roles, it means the user can perform some configuration on MC and also has access to one or more MC-managed databases. For additional details, see IT Role (MC) .

User role (DB)

USER has limited database privileges, such as viewing database cluster health, activity/resources, and messages. MC users granted the USER database role might have higher levels of permission on the MC itself, such as the IT Role (MC) . Alternatively, USER users might have no (NONE) privileges to configure MC. How you combine the two levels is up to you.

Mapping MC users to a database to avoid conflicts

When you assign an MC database level to an MC user, map the MC user account to a database user account to ensure that:

- The MC user inherits the privileges assigned to that database user
- You prevent the MC user from doing or seeing anything not allowed by the privileges for the user account on the server database

Privileges assigned to the database user supersede privileges of the MC user if there is a conflict, such as stopping a database. When the MC user logs into MC using an MC user name and password, Vertica compares privileges for database-related activities to the privileges on the database account to which you mapped the MC user. Vertica allows the user to perform operations in MC only when that user has both MC privileges and corresponding database privileges.

> **Tip**
> As a best practice, you should identify, in advance, the appropriate Vertica database user account that has privileges or roles similar to one of the MC database roles.

See Creating an MC user for more information.

MC database privileges by role

The following table summarizes MC database-level privileges by user role. The table shows the default privileges each role has. Operations marked "database user privilege" are dependent on the privileges of the Vertica database user account to which the MC user is mapped.

Default database-level privileges	ADMIN	ASSOCIATE	IT	USER
View database Overview page	Yes	Yes	Yes	Yes
View database messages	Yes	Yes	Yes	Yes
Delete messages and mark read/unread	Yes	Yes	Yes	
Audit and install Vertica licenses	Database user privilege	Database user privilege		

View database Activity page: • Queries chart • Internal Sessions chart • User Sessions chart • System Bottlenecks chart • User Query Phases chart	Yes	Database user privilege	Database user privilege	Database user privilege
View database Activity page: • Queries chart > Detail page • Table Treemap chart • Query Monitoring chart • Resource Pools Monitoring chart	Database user privilege	Database user privilege		
Start a database	Yes			
Rebalance, stop, or drop databases	Database user privilege			
View Manage page	Yes	Yes	Yes	Yes
View node details	Yes	Yes	Yes	
Replace, add, or remove nodes	Database user privilege			
Start/stop a node	Yes			
View database Settings page	Yes	Yes	Yes	
Modify database Settings page	Database user privilege	Database user privilege		
View Database Designer	Database user privilege	Database user privilege		

Database management

The Management Console provides tools to manage Eon Mode and Enterprise Mode databases, including creating a database, provisioning resources, subcluster management, and query optimization.

In this section

- Creating a database using MC
- Provisioning databases using MC
- Managing database clusters
- Subclusters in MC
- Eon Mode on-premises
- Managing queries using MC
- Working with query plans in MC
- Creating a database design in Management Console
- Running queries in Management Console
- Working with workload analyzer recommendations in MC
- Running Database Designer using MC
- Using the Management Console to replace nodes
- Import and monitor a database in a Hadoop environment

Creating a database using MC

If you installed the Management Console using an RPM, there is a wizard to help you create a new database on an existing Vertica cluster.

> **Note**
> If you did not install MC with an RPM, see one of the following:

- For Management Console on Amazon Web Services (AWS) using a CloudFormation Template , see Creating an Eon Mode database in AWS with MC or Creating an Enterprise Mode database in AWS with MC .
- For Google Cloud Platform (GCP), see Provision an Eon Mode cluster and database on GCP in MC .
- To provision a new database and cluster on-premises, see Creating a cluster using MC .

1. Connect to Management Console, and log in.
2. On the home page, click **View Infrastructure** to go to the **Database and Cluster View** . This tab provides a summary of your environment, clusters, and databases.
3. If the **Database** row displays a database running on the cluster that you want to add a new database to, select the database and click **Stop** . Wait until the database status is **Stopped** .
4. In the **Clusters** row, click the existing cluster that you want to create a database on. If a database is already running on it, you must stop the database.
5. Click **Create Database** in the window to start the database creation wizard.
6. Follow the steps in the wizard to successfully create a database.

You can close the web browser during the process and sign back in to MC later. The creation process continues unless an unexpected error occurs.

See also

- Creating a cluster using MC
- Troubleshooting with MC diagnostics
- Restarting MC

Provisioning databases using MC

Management Console allows all users to create, import, and connect to Vertica databases using the **MC Provision Databases** tab.

- Import cluster or database using IP discovery
- Create a new cluster
- Import and monitor a database in a Hadoop environment

Managing database clusters

Management Console allows you to monitor multiple databases on one or more clusters at the same time. MC administrators can see and manage all databases and clusters monitored by MC, while non-administrative MC users see only databases on which they have been assigned the appropriate access levels .

Depending on your access level, you can use the MC to perform the following database and cluster-related management operations:

- Create an Eon Mode and Enterprise Mode database.
- Install an Eon Mode and Enterprise Mode database in a cloud or on-premises environment.
- Create an empty database in an existing cluster .
- Import an existing database or cluster into the MC interface.
- Start the database, unless it is already running.
- Stop the database, if no users are connected.
- Remove the database from the MC interface.

> **Note**
> Remove does not drop the database. A Remove operation leaves it in the cluster, hidden from the UI. To add the database back to the MC interface, import it using the IP address of any cluster node. A Remove operation also stops metrics gathering on that database, but statistics gathering automatically resumes after you re-import.

- Drop the database when you are certain that no users are connected. Drop is a permanent action that drops the database from the cluster.

Database clusters in the cloud

When you use the Management Console to create a database or cluster on a supported cloud provider , you can perform the following operations on individual machines or the entire cluster:

- Start
- Stop

- Revive
- Reboot
- Terminate

For details, see Viewing and managing your cluster .

In this section

- Viewing cluster infrastructure
- Viewing and managing your cluster
- Importing an existing database into MC

Viewing cluster infrastructure

For a summary of all databases and clusters currently monitored by MC, click **View Infrastructure** on the MC Home page.

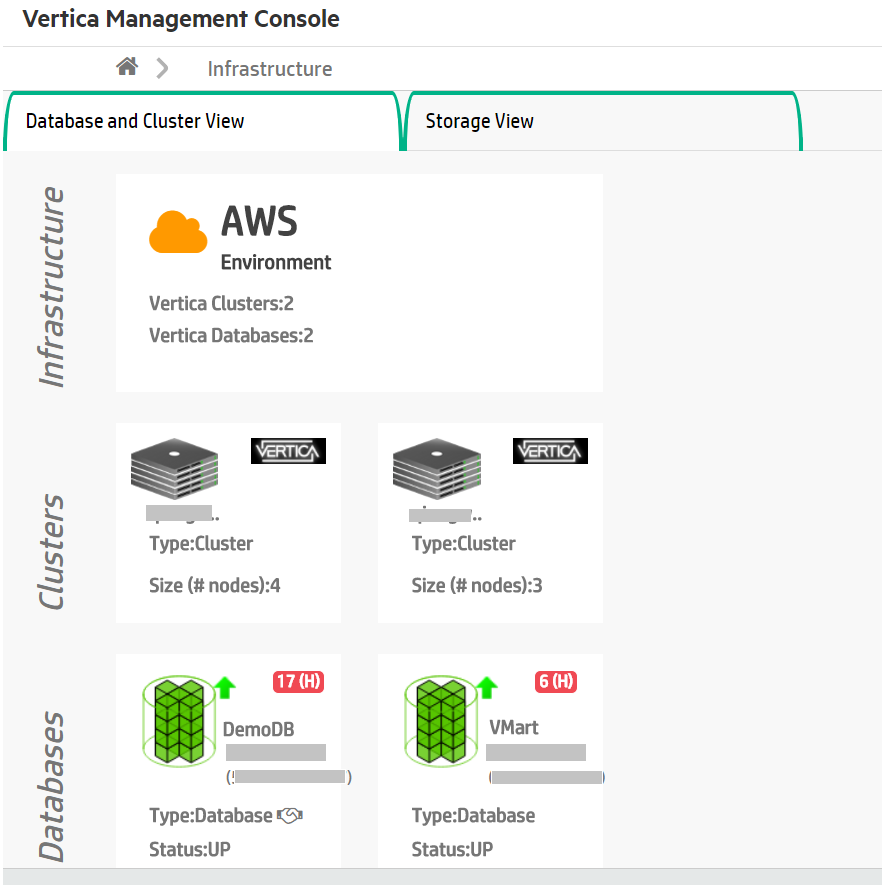
> **Note**
> Some of the features on this page are currently available in MC only on AWS and GCP.

The first tab on the Infrastructure page, **Database and Cluster View** , is overview of the infrastructure of all the clusters and databases currently monitored by MC.

Three rows are displayed: Infrastructure, Clusters, and Databases.

- **Infrastructure.** Specifies the type of environment on which your clusters reside:
  - Cloud: Displays the name of the cloud platform, such as AWS or GCP
  - On premises: Displays "Data Center"
  - Apache Hadoop: Displays "Hadoop Environment"
- **Clusters.** You can click a cluster to see its full details. From the dialog that opens, you can:
  - Add the cluster's master API key
  - View or Manage the cluster page
  - Remove the cluster from MC monitoring
  - Create a new database on the cluster (if all other databases on the cluster are stopped)
- **Databases.** A numbered badge on the top right displays the number of highest-priority messages from that database that are in your inbox. If a handshake icon ( ) displays next to "Type," that indicates the database is running in Eon Mode. If the handshake is absent, the database is running in Enterprise Mode. Click any database for more details. From the dialog that opens, you can:
  - View the database's Overview page
  - Stop or start the database
  - Drop the database (if the database is stopped)
  - Remove the database from MC monitoring

In the illustration below, MC is monitoring two different clusters that both reside on an AWS environment. One database is running on each cluster. The DemoDB database, displayed on the left, has a handshake icon next to its "Type" label that indicates it is running in Eon Mode. The VMart database on the 3-node cluster, displayed on the right, is running in Enterprise Mode.



Viewing and managing your cluster

The **Cluster** page in Management Console shows a node-based visualization of your cluster. This page shows the cluster's host addresses, the installed version of Vertica running, and a list of the databases on the cluster that MC is currently monitoring.

> **Note**
> Some of the features on this page are currently available in MC only on AWS and GCP.

From the **Cluster** page, you can also create a new empty database on the cluster, or import any existing databases MC discovers on the cluster. (These features are currently available only on AWS and GCP.)

MC displays different options depending on whether you imported the cluster to MC, or created the cluster using MC:

- Imported cluster: MC displays monitoring information about the cluster.
- Cluster created using MC: MC displays both monitoring information and management options for third-party cloud platforms such as AWS. For clusters you created using the current MC, the **Cluster** page provides cluster and instance management options.

Cluster and instance management option availability

For Eon Mode databases, MC supports actions for subcluster and node management for the following public and private cloud providers:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Pure Storage FlashBlade

> **Note**
> Enterprise Mode does not support subclusters.

For Enterprise Mode databases, MC supports these actions:

- In the cloud on AWS: Add Node action, Add Instance action.

- On-premises: Add Node action.

> **Note**
> In the cloud on GCP, Enterprise Mode databases are not supported.

Go to the cluster page

To view the **Cluster** page:

1. On the MC Home page, click **View Infrastructure** to go to the Infrastructure page . This page lists all the clusters the MC is monitoring.
2. Click any cluster shown on the Infrastructure page.
3. Select **View** or **Manage** from the dialog that displays, to view its Cluster page. (In a cloud environment, if MC was deployed from a cloud template the button says "Manage". Otherwise, the button says "View".)

> **Note**
> You can click the pencil icon beside the cluster name to rename the cluster. Enter a name that is unique within MC.

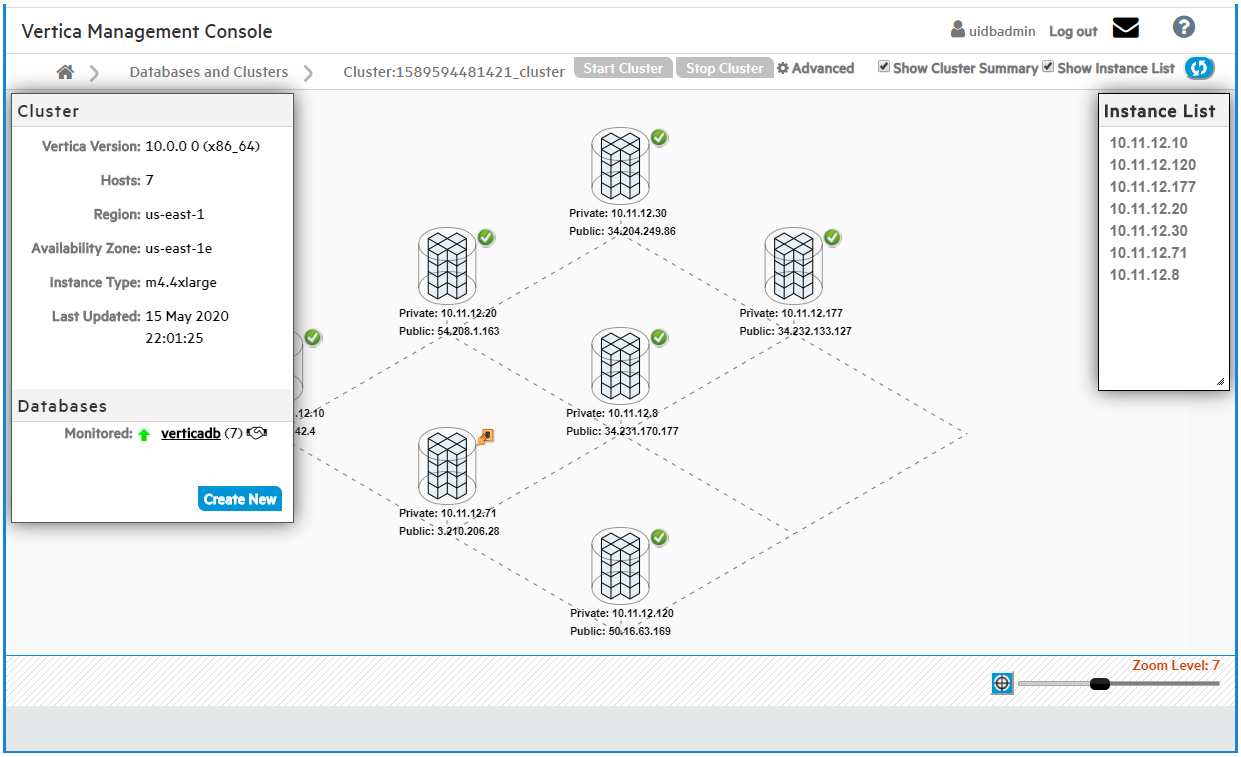
Monitor imported clusters

Whether you have imported or created a cluster using MC, you can view information about it through the Cluster page.

This page includes the following information:

- **Node visualization** : A visualization of all nodes within the cluster. Icons at the top right of each node indicate if the nodes are up. Click any node to see details about its host name, CPU information and total memory. If there are many nodes in the cluster, use the **Zoom Level** slider at the bottom right of the page to zoom the visualization in or out.
- **Instance List** : A list of all the instance IPs within the cluster. Click any instance in the list to see details about it.
- **Cluster Summary** : A summary of details about the cluster, including the version of Vertica running on the cluster and the number of hosts. If the cluster is running on cloud-platform resources such as AWS, you can also see region and instance type information.
- **Databases** : Lists all databases monitored by MC and their current state.
  - At the bottom of the **Databases** section, click **Create New** to create a new empty database .
  - This section also lists any Vertica databases on this cluster, if existing, that are not yet monitored by Management Console. See Importing an existing database into MC for the process of importing a discovered database.

The following image shows an overview of a 7-node cluster, which was created in MC using a Cloud Formation Template. This cluster has a running Eon Mode database on it.

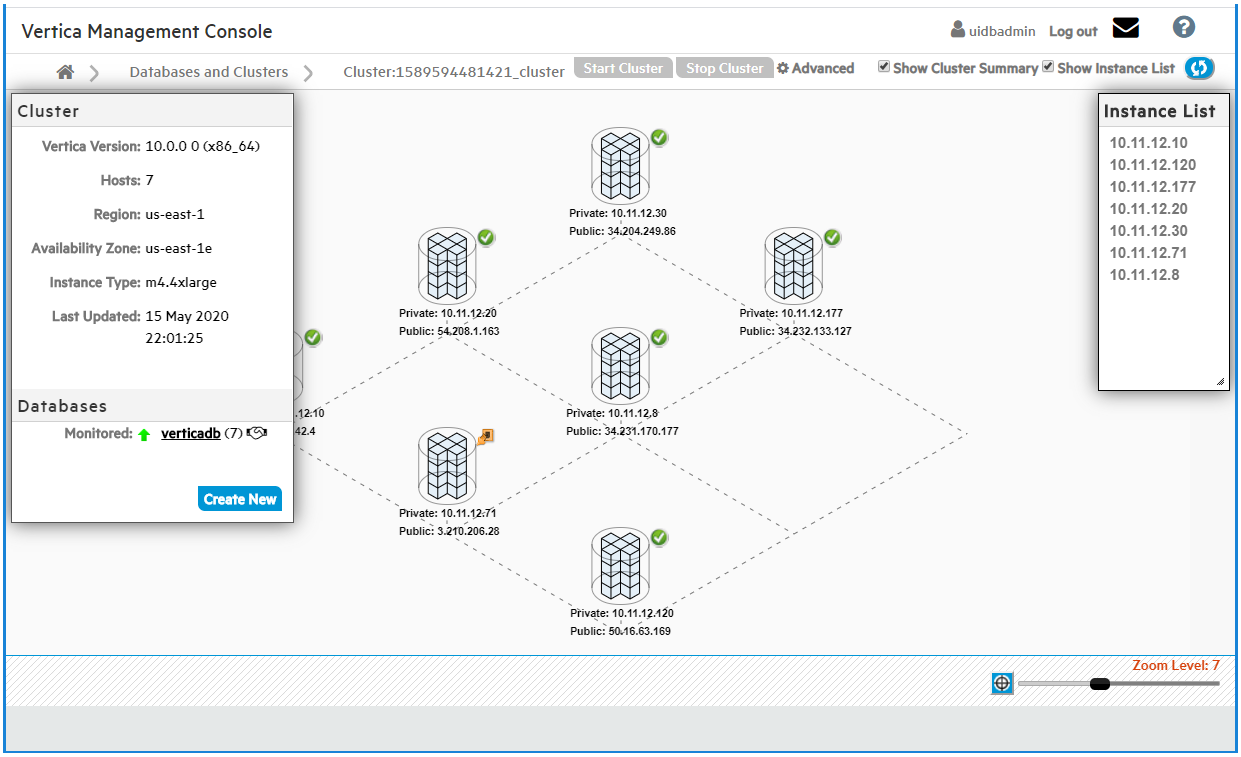


Manage a cluster created on AWS with the cluster creation wizard

If you installed Vertica from the AWS Marketplace using AWS resources , MC offers cluster management operations that are specific to the cloud. Using MC, you can manage a cluster running on AWS without going to the AWS console.

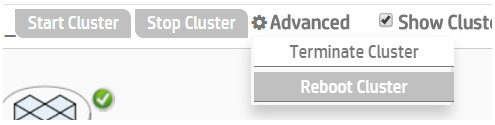
> **Note**
> **Note:** The AWS management operations to add or terminate instances are only available for clusters on AWS resources that were created using the *current* MC. Add and terminate capabilities are disabled for any cluster imported to MC, even if the imported cluster is on an AWS environment.
>
> If you upgrade the cluster's Vertica version manually through the command line, AWS management operations in MC become disabled for the cluster, even if you created that cluster using MC. Make sure to upgrade the cluster's Vertica version through MC in order to preserve AWS management capabilities for that cluster in MC.

In the screen capture below, the Cluster page shows a 7-node cluster, which was provisioned using the Cluster Creation wizard. Use the wizard to create an Eon Mode or Enterprise Mode .



Cluster management actions (Eon Mode and Enterprise Mode)

You can perform the following operations on your cluster through the **Cluster** page. These options are available at the top of the **Cluster** page or in the **Advanced** menu at the top of the page:



You can perform the following operations on your cluster through the cluster page:

- **Start Cluster** : Start all the instances in the cluster. Available at the top of the Cluster page.
- **Stop Cluster** : Stop all the instances in the cluster. You must first stop any running database on the cluster. Available at the top of the cluster page.
- **Reboot Cluster** : Restart all instances in the cluster. Available under the **Advanced** menu at the top of the page. **Note: Reboot Cluster** is currently available only on AWS.
- **Terminate Cluster** : Terminate all instances in the cluster, the databases on it, and all AWS resources from the cluster. The **Terminate Cluster** operation is available under the **Advanced** menu at the top of the page.
  - For Enterprise Mode databases, this operation *permanently deletes* any data you had on the cluster or its databases.
  - For Eon Mode databases, the data is preserved in communal storage and you can later revive the database in a new cluster . When you choose to terminate the cluster, MC gives you the option to also stop the database before termination, which is recommended in order to safely revive later.

View cluster instance details

You can view the details for any instance in your cluster. Select the IP address of an instance in the **Instance List** . MC displays a popup beside that instance showing information about its private and public IP addresses, host name, total memory, and other details.

Manage individual instances/nodes in Eon Mode

If your database is Eon Mode, you use actions available on the **Database > Manage > Subclusters** tab in MC to manage individual nodes.

To change the state of individual nodes in your Eon Mode database, you can:

- Start, stop, or terminate a node in a subcluster.

See Node action rules in MC .

To change the number of nodes in your Eon Mode database, you can:

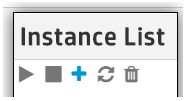
- Add or subtract individual nodes from a subcluster by scaling the subcluster up or down .
- Add or terminate an entire subcluster.

See Subcluster action rules in MC .

Manage individual instances in Enterprise Mode

If your database is Enterprise Mode, the **Cluster** page **Instance List** includes action icons you use to manage individual instances in your cluster.

In the **Instance List** panel of the **Cluster** page, select the IP address of any instance in your cluster that you want to perform the action on. Then click an icon from the icon menu at the top of the panel. Hover over an icon to read the action it performs.



- **Start Instance** : Start an individual instance in the cluster.
- **Stop Instance** : Stop an individual instance in the cluster.
- **Add Instance** : Add another instance to your cluster. When you select this action, Management Console opens the Add AWS Instance wizard, where you specify volume and storage information for the instance. You must supply your AWS key pair (and a Vertica Premium Edition license if you are adding more nodes to the cluster than a Community Edition license allows). You can add up to 10 instances at a time using the Add Instance action.
- **Restart Instance** : Restart an individual instance in the cluster.
- **Terminate Instance** : Permanently remove the instance from your cluster.

Importing an existing database into MC

If you have already created a Vertica database, you can import it into MC to monitor its health and activity.

When you install MC on the same cluster as the existing database you intend to monitor, MC automatically discovers the cluster and any databases installed on it, whether those databases are currently running or are down.

> **Note**
> If you haven't created a database and want to create one through the MC, see Creating a database using MC .

Import a database existing on a monitored cluster

The following procedure describes how to import an existing database that is on a cluster MC is already monitoring.

1. Connect to Management Console and sign in as an MC administrator.
2. On the MC Home page, click **View Your Infrastructure** .
3. On the Databases and Clusters page, click the cluster and click **View** in the dialog box that opens.
4. On the left side of the page under the Databases heading, click **Import Discovered** .

> **Tip**
> A running database appears as Monitored; any non-running databases appear as Discovered. MC supports only one running database on a single cluster at a time. You must shut down a running database on a cluster in order to monitor another database on that cluster.

5. In the **Import Database** dialog box:
   - Select the database you want to import.
   - Optionally clear auto-discovered databases you don't want to import.
   - Supply the database administrator username and password and click **Import** . (Supplying a non-administrator username prevents MC from displaying some charts after import.)
   - If the Vertica database is configured for TLS security, you need to configure TLS for all Management Console connections to this database over JDBC. Click **Use TLS.** Management Console launches the Certificates wizard. See MC certificates wizard .

After Management Console connects to the database it opens the **Manage** page, which provides a view of the cluster nodes. See Monitoring Cluster Status for more information.

You perform the import process once per existing database. Next time you connect to Management Console, your database appears under the Recent Databases section on the Home page, as well as on the Databases and Clusters page.

> **Note**
> The system clocks in your cluster must be synchronized with the system that is running Management Console to allow automatic discovery of local clusters.

Import a database existing on a new cluster

If the database you intend to monitor is on a cluster MC is not currently monitoring, MC cannot automatically discover it. You can import it with the following procedure.

1. Connect to Management Console and sign in as an MC administrator.
2. On the MC Home page, click **Import a Vertica Database Cluster** .
3. Enter the IP address of one of the database's cluster nodes.
4. Enter the master API key for the cluster. Find the key here: /opt/vertica/config/apikeys.dat
5. In the **Import Database** dialog box:
   - Select the database you want to import.
   - Optionally clear auto-discovered databases you don't want to import.
   - Supply the database administrator username and password and click **Import** . (Supplying a non-administrator username prevents MC from displaying some charts after import.)
   - To configure TLS security for all Management Console connections to this database over JDBC, click **Use TLS.** Management Console launches the Certificates wizard. For instructions on completing the wizard, see Configuring TLS while importing a database on MC .

Subclusters in MC

In Eon Mode databases, you can use subclusters (groups of nodes) to separate different workloads, to control how those workloads use resources, and to facilitate scaling your database up and down as workloads fluctuate. This allows you to better manage your cloud resource expenses or data center resources. For an overview of subcluster concepts and how subclusters work, see Subclusters .

MC makes it easy to view and manage your subclusters. You can track how queries are performing and how well your subcluster resources are balanced. Using MC, you can use the information to adjust the number and size of your subclusters to improve your query throughput and system performance.

Visualizing your subclusters

The charts on the **Database Overview** page allow you to view and drill down into the resource usage of your database at any level. You can look at the resource usage of all nodes, or all subclusters, or individual subclusters. For details, see Charting subcluster resource usage in MC .

You can view statistics for the individual nodes in each subcluster in table format on the **Database Manage** page, in the **Subclusters** tab.

For a tour of the monitoring features of the **Manage > Subclusters** tab, see Viewing subcluster layout in MC .

Managing your subclusters

For Eon Mode databases, MC supports actions for subcluster and node management for the following public and private cloud providers:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Pure Storage FlashBlade

> **Note**

Enterprise Mode does not support subclusters.

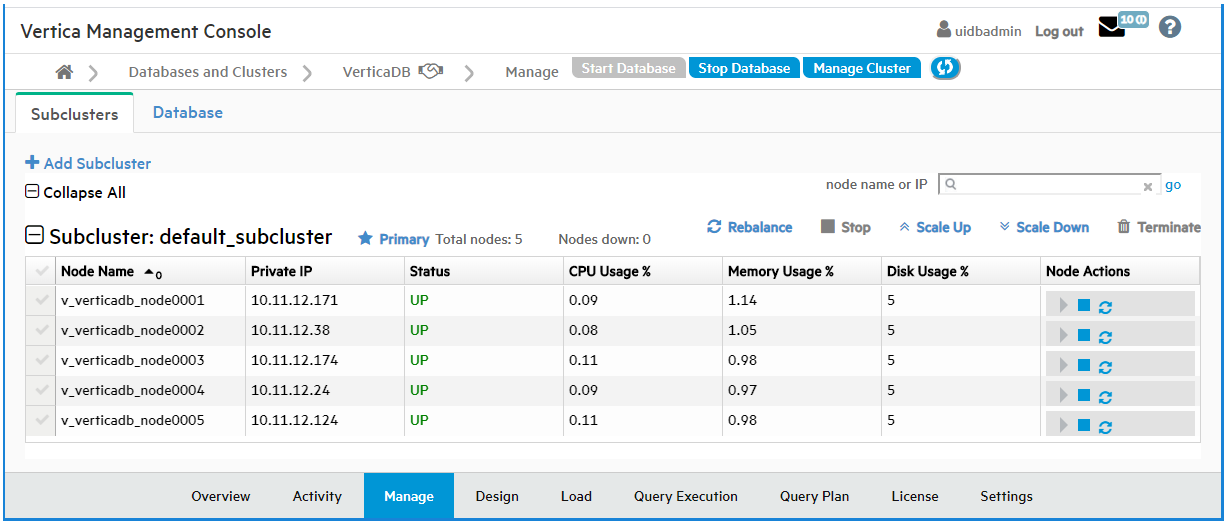
For Enterprise Mode databases, MC supports these actions:

- In the cloud on AWS: Add Node action, Add Instance action.
- On-premises: Add Node action.

Note
In the cloud on GCP, Enterprise Mode databases are not supported.

To view and manage your subclusters, select your database from the **MC Home** page or the **Databases and Clusters** page. MC displays your database's **Overview** page. Select **Manage** at the bottom of the **Overview** page.

To view the Subclusters page, click the **Manage > Subclusters** tab:



Eon Mode in the cloud

In Eon Mode on cloud platforms, you can use the **Manage > Subclusters** tab to add subclusters, rebalance your subclusters, stop and start subclusters, scale subclusters up or down, or terminate a subcluster. You can also stop or start a node, or restart a database node and its underlying instance.

Eon Mode on-premises

In Eon Mode on-premises, available subcluster and node actions behave a little differently than in the cloud, because your Vertica nodes reside on actual machines in your data center rather than on cloud instances.

Subcluster Actions in Eon Mode on Premises

- In Eon Mode on-premises, you can use the **Manage > Subclusters** tab to add subclusters, rebalance your subclusters, or delete a subcluster.
- You can add (create) a subcluster only if additional Vertica host machines are available.
- When you delete a subcluster, MC deletes the subcluster from the database but does not delete the actual machines. MC stops the nodes in the subcluster, removes them from the subcluster, and deletes the subcluster. The Vertica host machines are then available to be added to other subclusters.
- When you start or stop a subcluster in an Eon Mode database on-premises, MC starts or stops the subcluster nodes on the Vertica host machines, but not the machines themselves.
- When you scale up a subcluster on-premises, the MC wizard displays a list of the available Vertica host machines that are not currently part of a database. You select the ones you want to add to the subcluster as nodes, then confirm that you want to scale up the subcluster.
  When you scale down a subcluster on-premises, MC removes the nodes from the subcluster in the database, but does not terminate the Vertica host machines. The hosts are now available for scaling up other subclusters.
- **Scale Up** displays the IP addresses of all available Vertica hosts that are not part of the database. The **Scale Up** button is grayed out if there are no Vertica hosts in the cluster that are not part of the database.

Node actions in Eon Mode on premises

When you start or stop a node on-premises, MC starts or stops the node in the database, but does not start or stop the Vertica host machine. The Restart Node action is not available for on-premises Eon Mode databases.

In this section

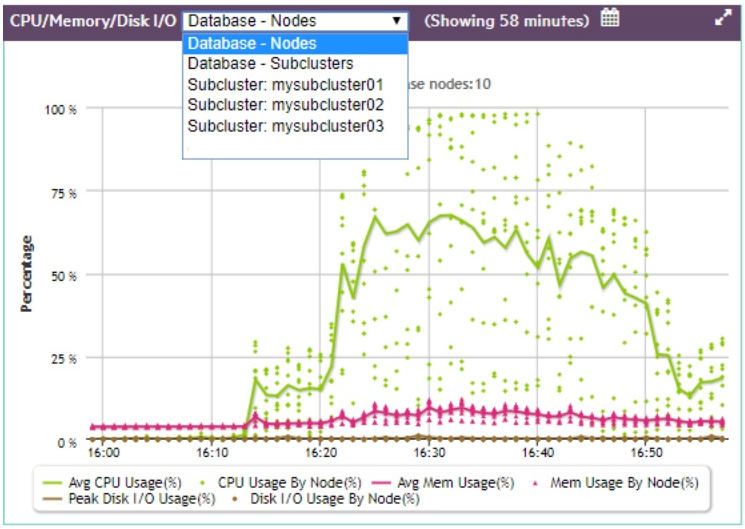
- Charting subcluster resource usage in MC
- Viewing subcluster layout in MC
- Adding subclusters in MC
- Rebalancing data using Management Console
- Subcluster action rules in MC
- Starting and stopping subclusters in MC
- Scaling subclusters in MC
- Terminating subclusters in MC
- Node action rules in MC
- Starting, stopping, and restarting nodes in MC

Charting subcluster resource usage in MC

On the **Database Overview** page, in the **CPU/Memory/Disk I/O** chart and the **Database General Pool Usage** chart, you can use the dropdown in the title bar to focus the chart on:

- All nodes in the database
- All subclusters in the database
- An individual subcluster, by name

For example, the CPU/Memory/Disk I/O chart dropdown allows you to choose the database nodes, one named subcluster, or all subclusters.



If you choose **Database - Subclusters,** the line in a given color represents the trend of all subclusters averaged together for that statistic, and each dot of the same color represents an individual subcluster at a certain time, for that same statistic.

For more in-depth information on how expand detail areas of charts and drill into the details, see Viewing the overview page .

Viewing subcluster layout in MC

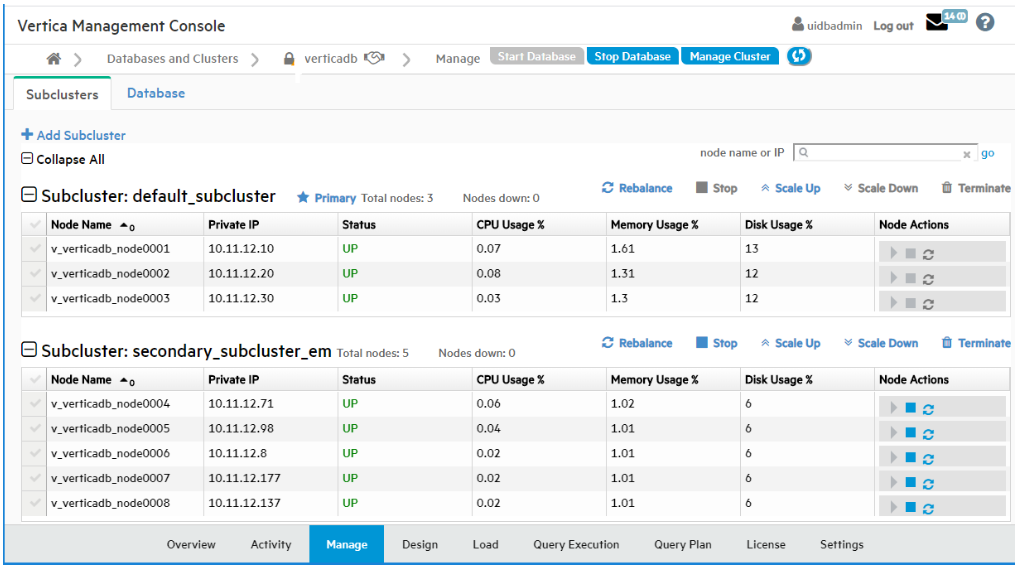
The MC **Database Manage** page displays two tabs, the **Subclusters** tab and the **Database** tab.

This topic describes the monitoring functions of the **Subclusters** tab. To monitor your subclusters, on the **Subclusters** tab you can view, sort, and search for subclusters and view their layout and statistics.

For information about using the **Subclusters** tab to make changes — adding, rebalancing, stopping and starting, scaling up or down, or terminating subclusters — see Subclusters in MC and its subtopics.

The **Subclusters** tab includes the following statistics in table form for the nodes in each subcluster:

- Node name
- Private IP address
- Status (UP or DOWN)
- CPU Usage %
- Memory Usage %
- Disk Usage %



Searching for nodes

To find a particular node, enter its node name or IP address in the "node name or IP" search field at the top right of the **Subclusters** tab. Searching for nodes is especially helpful if your cluster is very large. To find a specific node, enter its complete node name or IP address. You can enter a partial node name or IP address to find all nodes whose name or IP address contains that string. For example, if you enter "240" in the search field, MC would find both of the following nodes:

- Node name: MyNode24018
- Node IP address: 1.160.10.240

> **Note**
> Wildcard characters are not supported in the search field.

Starting, stopping, or removing nodes

The right column provides icons for executing node actions. You can start, stop, or remove a node in the subcluster. Removing a node also removes it from the database. Only the Start, Stop, and Remove actions are available on this page. For details, see Starting, stopping, and restarting nodes in MC

> **Note**
> If you change the layout of your subcluster, for example by removing nodes, you must rebalance shards. See REBALANCE_SHARDS .

Sorting nodes within a subcluster

You can sort the nodes within each subcluster by the values in any column, by clicking on the column heading.

Collapsing or expanding a subcluster, or the entire table

To collapse a subcluster section to one summary row, click the minus icon or the subcluster heading. To collapse the entire table to summary rows, click the minus icon or "Collapse All".

To expand a collapsed subcluster section, click the plus icon or the subcluster heading. To expand the entire table, click the plus icon or **Expand All.**

Adding subclusters in MC

You can add a subcluster to an Eon Mode database on-premises or in the cloud, to provide your database with more compute power, and to separate workloads.

For more information about the rules governing subclusters, see Subclusters .

Adding a subcluster on a cloud platform

When you use MC to add an Eon Mode subcluster on a cloud platform, MC provisions the requested instances, configures them as nodes in your database cluster, and forms those nodes into a new subcluster.

You can add a second primary subcluster in order to stop the original primary. Be sure to make the replacement primary subcluster at least one node larger than the original. (If you make them the same size, they both count equally toward the quorum, and stopping either would violate the quorum, so you cannot stop either one. For more information, see Data integrity and high availability in an Eon Mode database .)

Amazon Web Services (AWS)

The following steps add a subcluster on AWS:

1. On the **Manage** > **Subclusters** tab, click **Add Subcluster** to open the Add Subcluster window.
2. On **Enter AWS Credentials and preferences** , some values might populate based on your current configuration. Accept the defaults or enter the following values:
   - **AWS Region** : Enter the same region as your cluster.
   - **AWS Subnet** : The subnet for your cluster. By default, Vertica creates your cluster in the same subnet as your MC instance.
     Important
     Use security groups and network access control lists (ACLs) to secure your subnet. For details, see the Amazon documentation .
   - **AWS Key Pair** : Your Amazon key pair for SSH access to EC2 instances.
3. Select **Next** . On the **Specify Subcluster Information** screen, supply the following information:
   - **Subcluster Name** : Enter a name for your subcluster.
   - Subcluster type from the dropdown list. Select **Primary** or **Secondary** .
   - **Number of instances in this subcluster** : Select the number of nodes that you want in the subcluster. If you are using the free Community Edition license, you are limited to 3 nodes. If you enter a value greater than 3, you are prompted to enter an upgraded license to continue.
   - **Vertica License** : Click **Browse** to locate and upload your Vertica license key file.
     If you do not supply a license key file here, the MC uses the Vertica Community Edition license. This license has a three node limit, so the value in **Number of instances in this subcluster** cannot be larger than 3 if you do not supply a license. See Managing licenses for more information.
   - **Node IP setting** : Choose a node IP setting. If you choose **Public IP** , the address is not persistent across instance stop and start.
4. Select **Next** . On the **Specify cloud instance and data storage info** screen, **Database Depot Path** is populated to match the original cluster configuration. Choose or enter a value in the following:
   - **EC2 Instance Type** : For recommended instance types, see Choosing AWS Eon Mode Instance Types . For a comprehensive list of Vertica supported instance types, see Supported AWS instance types .
   - **Database Depot Path** : This field is populated with the existing subcluster's depot path.
   - **EBS Volume Type** : This field is populated with the volume configuration defaults for the associated instance type. Select a new value to change the default.
   - **EBS Volume Size (GB) per Volume per Available Node** : This field is populated with the volume configuration defaults for the associated instance type. Enter a new value to change the default.
5. Select **Next** . On the **Specify additional storage and tag info** screen, the following fields are populated:
   - **Database Catalog Path** : The path to a persistent storage location.
   - **Database Temp Path** : The path to an ephemeral storage location if the node instance type includes the ephemeral storage option.
     Under each path, there are **EBS Volume Type** and **EBS Volume Size (GB) per Volume per Available Node** fields that are populated with volume configuration defaults for the associated instance type. Select or enter a new value to change the default path.
   - Optionally, select **Tag EC2 Instances** to assign distinctive, searchable metadata tags to the instances in your new subcluster. Existing tags are displayed.
6. Select **Next** . On **Review 'Add Subcluster' information** , confirm the configuration details for your new subcluster.
7. Select **Add Subcluster** to create the subcluster.

After the subcluster is created, The Subcluster page displays. The MC automatically subscribes the nodes in your new subcluster to shards so that the nodes are ready to use.

Google Cloud Platform (GCP)

The following steps add a subcluster on GCP:

1. On the **Manage** > **Subclusters** tab, click **Add Subcluster** to open the Add Subcluster window.
2. On **Specify Subcluster Information** , supply the following information:
   - **Subcluster Name** : Enter a name for your subcluster.
   - Subcluster type from the dropdown list. Select **Primary** or **Secondary** .
   - **Number of instances in this subcluster** : Select the number of nodes that you want in the subcluster. If you are using the free Community Edition license, you are limited to 3 nodes. If you enter a value greater than 3, you are prompted to enter an upgraded license to continue.
   - **Vertica License** : Click **Browse** to locate and upload your Vertica license key file.
     If you do not supply a license key file here, the MC uses the Vertica Community Edition license. This license has a three node limit, so the value in **Number of instances in this subcluster** cannot be larger than 3 if you do not supply a license. See Managing licenses for more information.
   - **Node IP setting** : Choose a node IP setting. If you choose **Public IP** , the address is not persistent across instance stop and start.

3. Select **Next** . Cluster configuration might take a few minutes.
4. On **Specify cloud instance and depot storage info** , supply the following information:
   - **Instance Type** : Select the instance type. For recommended instance types, see GCP Eon Mode instance recommendations . For a comprehensive list of Vertica supported instance types, see Supported GCP machine types .
   - **Database Depot Path** : This value is populated to match the original cluster configuration.
   - **Disk Type** : Volume type for each node. This value is populated with the volume configuration defaults for the associated instance type.
   - **Volume Size (GB) per Volume per Available Node** : Volume size for each node. This value is populated with the volume configuration defaults for the associated instance type.
5. Select **Next** . On Specify additional storage and label info, supply the following:
   - **Database Catalog Path** : The path to a persistent storage location.
   - **Disk Type** : Volume type for the database catalog.
   - **Size (GB) per Available Node** : Volume size for the database catalog.
   - **Database Temp Path** : The path to an ephemeral storage location if the node instance type includes the ephemeral storage option.
   - **Disk Type** : Volume type for the database temp storage location.
   - **Size (GB) per Available Node** : Volume size for the database temp storage location.
   - **Label Instances** : Optional. Assign distinctive, searchable metadata tags to the instances in your new subcluster. Existing tags are displayed.
6. Select **Next** . On **Review Information** , confirm the configuration details for your new subcluster, and accept the terms and conditions.
7. Select **Add Subcluster** to create the subcluster.

After the subcluster is created, The **Subcluster** page displays. The MC automatically subscribes the nodes in your new subcluster to shards so that the nodes are ready to use.

Microsoft Azure

The following steps add a subcluster on Azure:

1. On the **Manage** > **Subclusters** tab, click **Add Subcluster** to open the Add Subcluster window.
2. On **Specify Subcluster Information** , supply the following information:
   - **Subcluster Name** : Enter a name for your subcluster.
   - Subcluster type from the dropdown list. Select **Primary** or **Secondary** .
   - **Number of instances in this subcluster** : Select the number of nodes that you want in the subcluster. If you are using the free Community Edition license, you are limited to 3 nodes. If you enter a value greater than 3, you are prompted to enter an upgraded license to continue.
   - **Vertica License** : Click **Browse** to locate and upload your Vertica license key file.
     If you do not supply a license key file here, the MC uses the Vertica Community Edition license. This license has a three node limit, so the value in **Number of instances in this subcluster** cannot be larger than 3 if you do not supply a license. See Managing licenses for more information.
   - **Node IP setting** : Choose a node IP setting. If you choose **Public IP** , the address is not persistent across instance stop and start.
3. Select **Next** . Cluster configuration might take a few minutes.
4. On **Specify cloud instance and depot storage info** , supply the following information:
   - **Virtual Machine (VM) Size** : Select the instance type. For a comprehensive list of Vertica supported instance types, see Recommended Azure VM types and operating systems .
   - **Database Depot Path** : This value is populated to match the original cluster configuration.
   - **Managed Disk Volume Type** : Volume type for each node. This value is populated with the volume configuration defaults for the associated instance type.
   - **Managed Disk Volume Size (GB) per Volume per Available Node** : Volume size for each node. This value is populated with the volume configuration defaults for the associated instance type.
5. Select **Next** . On Specify additional storage and label info, supply the following:
   - **Database Catalog Path** : The path to a persistent storage location.
   - **Managed Disk Volume Type** : Volume type for the database catalog.
   - **Managed Disk Volume Size (GB) per Available Node** : Volume size for the database catalog.
   - **Database Temp Path** : The path to an ephemeral storage location if the node instance type includes the ephemeral storage option.
   - (E16_v4 VMs only) **Managed Disk Volume Type** : Volume type for the database temp storage location.
   - (E16_v4 VMs only) **Managed Disk Volume Size (GB) per Available Node** : Volume size for the database temp storage location.
   - **Label Instances** : Optional. Assign distinctive, searchable metadata tags to the instances in your new subcluster. Existing tags are displayed.
6. Select **Next** . On **Review Information** , confirm the configuration details for your new subcluster.
7. Select **Add Subcluster** to create the subcluster.

After the subcluster is created, The **Subcluster** page displays. The MC automatically subscribes the nodes in your new subcluster to shards so that the nodes are ready to use.

On-premises

For an Eon Mode database on-premises, you can use MC to create additional subclusters. MC displays all available Vertica hosts that are not part of the database. It configures the ones you select to become the nodes in the new subcluster in your database.

1. On the **Manage > Subclusters** tab, click **Create Subcluster** at top left. MC opens the **Create Subcluster** wizard.
2. In the first screen, respond in the following fields:
   - **Subcluster Name:**Enter a name for the new subcluster.
   - **Subcluster type dropdown (unlabeled):**Select **Primary** or **Secondary** .
   - **Select nodes that will be added to a subcluster:**Select from the list of IP addresses. MC displays all available Vertica hosts within the cluster that are not currently members of a database.
   - **Confirmation:**Click the **Confirmation** check box to indicate that you want to create the named subcluster, to include the Vertica hosts you selected as nodes.
3. After you click the **Confirmation** check box, the **Proceed** button becomes active. Click **Proceed** .

MC displays a progress screen while it creates the requested subcluster. Wait for all steps to complete.

See also

Subcluster action rules in MC

Rebalancing data using Management Console

Vertica can rebalance your subcluster when you add or remove nodes. If you notice data skew where one node shows more activity than another (for example, most queries processing data on a single node), you can manually rebalance the sublcluster using MC if the database is imported into the MC interface.

On the Management Console **Manage** page in the **Subclusters** tab, click **Rebalance** above the subcluster to initiate the rebalance operation.

During a rebalance operation, you cannot perform any other activities on the database, such as start, stop, add, or remove nodes.

Subcluster action rules in MC

The table below summarizes when each subcluster action is available for the primary subcluster or a secondary subcluster.

For Eon Mode databases, MC supports actions for subcluster and node management for the following public and private cloud providers:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Pure Storage FlashBlade

> **Note**
> Enterprise Mode does not support subclusters.

In the cloud

The explanations in this table apply to subcluster actions in the cloud. For the differences on-premises, see On-Premises further below.

Subcluster Action	Primary Subcluster	Secondary Subcluster
Add Subcluster	Allowed. Must include at least one node. The subcluster cannot be empty. Vertica recommends having only one primary subcluster. You can add a second primary subcluster in order to stop the original primary. Be sure to make the replacement primary subcluster at least one node larger than the original. (If you make them the same size, they both count equally toward the quorum, and stopping either would violate the quorum, so you cannot stop either one. For more information, see Data integrity and high availability in an Eon Mode database .)	Allowed. Creates a new subcluster. Provisions cloud instance(s). Adds nodes to that subcluster. Defaults to minimum of one node.

Rebalance • This button applies to shard subscriptions in Eon Mode.	Always allowed. **Note** When you scale down a subcluster, Vertica rebalances the shards automatically, whereas when you scale up a subcluster, you must rebalance the shards yourself.	Always allowed. See note for primary subcluster.
Start Subcluster • Includes starting cloud instances if not already running.	Available if primary subcluster is stopped.	Available if: • Secondary subcluster is stopped. • Primary subcluster is running.
Stop Subcluster • Includes stopping cloud instances.	**Stop Subcluster** is available for the primary subcluster only under certain conditions: • Theprimary subcluster must be running. • One or more additional primary subclusters must exist in the database. • It must be true that stopping this primary subcluster will shut down less than 50% of the primary nodes in the database.	Available if running.
Scale Up • Adds cloud instances which become added nodes.	Always allowed. • Vertica adds nodes that are the same instance type as the existing nodes that are already in the target subcluster.	Always allowed.
Scale Down • Removes nodes. • Terminates instances.	Allowed only under the following conditions: • If K-safety is >= 1, the remaining number of nodes after scaling down the primary subcluster must be >= 3. • The number of nodes you remove must be fewer than half the nodes.	Same as for primary.
Terminate Subcluster • Drops subcluster from database. • Terminates instances.	Available if another primary subcluster exists in the database.	Always allowed. • Removes subcluster entry from SUBCLUSTERS table.

On-premises

When you start or stop a subcluster in an Eon Mode database on-premises, MC starts or stops the subcluster nodes on the Vertica host machines, but not the machines themselves.

When you scale up a subcluster on-premises, the MC wizard displays a list of the available Vertica host machines that are not currently part of a database. You select the ones you want to add to the subcluster as nodes, then confirm that you want to scale up the subcluster.

When you scale down a subcluster on-premises, MC removes the nodes from the subcluster in the database, but does not terminate the Vertica host machines. The hosts are now available for scaling up other subclusters.

Starting and stopping subclusters in MC

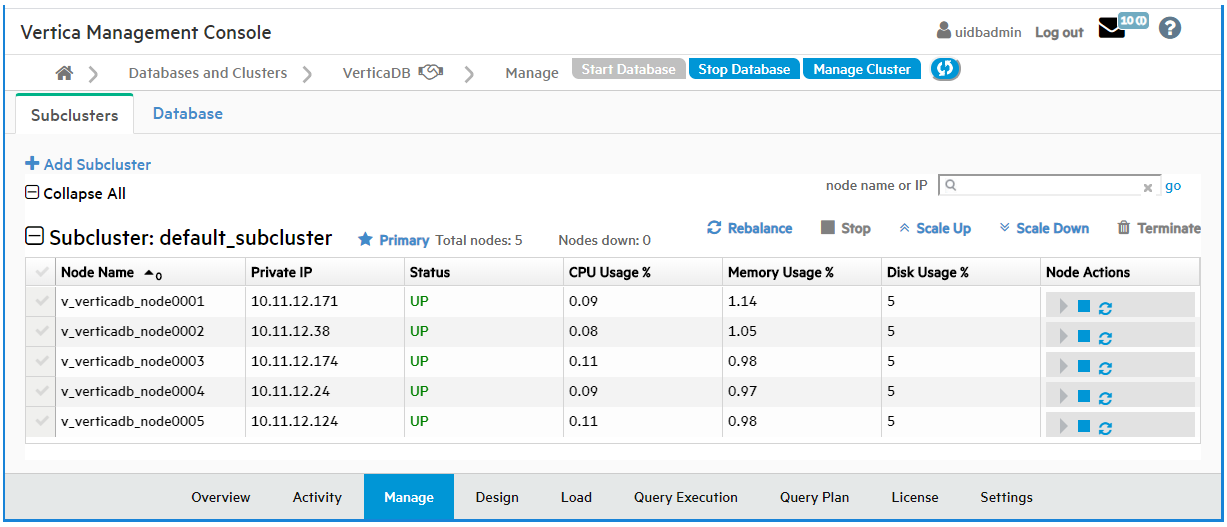
For Eon Mode databases, MC supports actions for subcluster and node management for the following public and private cloud providers:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Pure Storage FlashBlade

> **Note**
> Enterprise Mode does not support subclusters.

On the **Manage > Subclusters** tab, the tool bar displays the available subcluster actions above each subcluster. For example, the screen capture below shows the available actions for the default subcluster, which is also the primary subcluster: **Rebalance, Stop** (grayed out), **Scale Up, Scale Down,** and **Terminate** (grayed out).

> **Note**
> Terminate Subcluster is available for Eon Mode databases only in the cloud, not on premises.



Why are some actions grayed out?

Stop and **Terminate** are grayed out in this example because if you stopped or terminated the only primary subcluster, the database would shut down. For each subcluster, **Stop** displays if the subcluster is currently running, or **Start** displays if the subcluster is currently stopped. Both **Stop** and **Terminate** are grayed out if their execution would be unsafe for the database.

Starting a subcluster in the cloud

You can start any subcluster that is currently stopped.

1. In the **Manage > Subclusters** tab, locate the subcluster you want to start.
2. Just above it on the right, click **Start** .
3. In the **Start Subcluster** screen, click the check box to confirm you want to start the subcluster.
   MC displays a progress screen while the startup tasks are executing.
4. When all the tasks are checked, click **Close** .
   The **Manage > Subclusters** tab shows that your subcluster is started and its nodes are up.

Stopping a subcluster in the cloud

Important

Stopping a subcluster does not warn you if there are active user sessions connected to the subcluster. This behavior is the same as stopping an individual node. Before stopping a subcluster, verify that no users are connected to it.

You can stop a primary subcluster only if there is another primary subcluster in the database with node count greater at least by 1 node, to maintain K-safety.

You can add a second primary subcluster in order to stop the original primary. Be sure to make the replacement primary subcluster at least one node larger than the original. (If you make them the same size, they both count equally toward the quorum, and stopping either would violate the quorum, so you cannot stop either one. For more information, see Data integrity and high availability in an Eon Mode database .)

You can stop a secondary subcluster anytime, to save money on cloud resources.

1. In the **Manage > Subclusters** tab, locate the subcluster you want to stop.
   If the **Stop** button is displayed but grayed out, you cannot stop this subcluster because doing so would shut down the database.
2. Just above the subcluster on the right, click **Stop** .
3. In the **Stop Subcluster** window, click the check box to confirm you want to stop the subcluster.
   MC displays a progress screen while the subcluster stopping tasks are executing.
4. When all the tasks are checked, click **Close** .
   The **Manage > Subclusters** tab shows that your subcluster is stopped and its nodes are down.

Starting or stopping a subcluster on premises

When you start or stop a subcluster in an Eon Mode database on-premises, MC starts or stops the subcluster nodes on the Vertica host machines, but not the machines themselves.

See also

Subcluster action rules in MC

Scaling subclusters in MC

You can scale an Eon Mode subcluster up or down , to increase or decrease the number of nodes in the subcluster. This lets you add compute capacity when you need it, and reduce it to save money or redirect resources when you don't.

For Eon Mode databases, MC supports actions for subcluster and node management for the following public and private cloud providers:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Pure Storage FlashBlade

> **Note**
> Enterprise Mode does not support subclusters.

Scaling up in the cloud

When you scale up a subcluster, MC adds one or more cloud instances to your database cluster as hosts, and adds them to your subcluster as nodes.

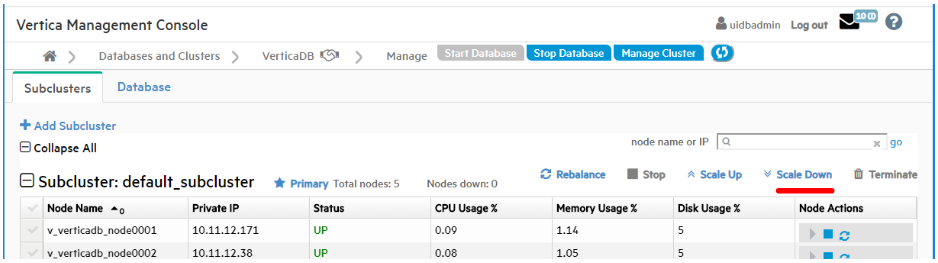
1. On the **Manage > Subclusters** tab, click **Scale Up** immediately above the subcluster you want to enlarge. MC launches the Scale Up wizard.
2. Wait a moment or two while MC pre-populates the fields on the **Enter AWS Credentials and preferences** screen with your credentials, then click **Next** .
3. On the **Specify Subcluster information** screen, enter the number of instances you want to add to the subcluster in **Number of Instances to Add** . MC pre-populates the number of existing hosts in the cluster.
4. Click **Browse** and select your Vertica license to insert in the license field, then click **Next** .
5. The **Specify cloud instance and data storage info** screen has fields that specify the cloud instance and data storage information. MC sets all the fields to the same values as the existing instances in your subcluster. Select **Next** to accept the existing values.
6. The **Specify additional storage and tag info** screen displays any tags you specified for your instances when you created the database cluster. You can use the same tags for the instances you are adding, or use the fields on the screen to add new tags for these instances. You can keep and apply or delete the existing tags. If you delete tags, they are not used for the instances you are adding now. When you are done modifying the tags, click **Next** .
7. MC displays the **Review 'Scale Up' information** screen for your approval. If the information is correct, click **Scale Up** to add the instances to the subcluster.
8. Wait until all operations on the progress screen show check marks, then click **Close** .

Scaling down in the cloud

When you scale down a subcluster, MC removes the requested number of nodes from the subcluster, and removes their hosts from the database cluster. It then terminates the underlying cloud instances. Scaling down a subcluster is allowed only when doing so will not cause the database to shut down.

For details on when Scale Down is or is not available for a subcluster, see Subcluster action rules in MC .

1. On the **Manage > Subclusters** tab, click **Scale Down** above the subcluster to launch the Scale Down page:



2. In the **Number of hosts to remove** field, enter the number of hosts you would like to remove from the subcluster.
3. Under **Confirmation** , click the checkbox to affirm that you want to scale down the named subcluster, and terminate the instances the nodes were using.
4. Click **Scaledown Subcluster** .
   The scale down operation may take a little time. When it finishes, the progress screen displays the details about the removed nodes and their hosts and the terminated instances.
5. When all steps show check marks, click **Close** .

Scaling on-premises

When you scale up a subcluster on-premises, the MC wizard displays a list of the available Vertica host machines that are not currently part of a database. You select the ones you want to add to the subcluster as nodes, then confirm that you want to scale up the subcluster.

When you scale down a subcluster on-premises, MC removes the nodes from the subcluster in the database, but does not terminate the Vertica host machines. The hosts are now available for scaling up other subclusters.

Terminating subclusters in MC

For Eon Mode databases, MC supports actions for subcluster and node management for the following public and private cloud providers:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Pure Storage FlashBlade

> **Note**
> Enterprise Mode does not support subclusters.

In the cloud

The **Terminate** action for subclusters is available for Eon Mode databases only in the cloud.

You can terminate any subcluster that will not cause the database to go down. You can terminate:

- Any secondary subcluster.
- A primary subcluster, provided that:
  - There is at least one other primary subcluster in the database.
  - The other primary subcluster is at least one node larger than the one you want to terminate.

Terminate a subcluster

1. On the **Manage > Subclusters** tab, click **Terminate** immediately above the target subcluster.
2. In the **Terminate Subcluster** window, click the check box to confirm you want to delete the chosen subcluster and terminate its instances.
3. Click **Terminate Subcluster** .
   MC displays a progress window that shows the steps it is executing to terminate the subcluster.
4. When all the steps are checked, click **Close** .

On-premises

The **Terminate** action for subclusters is not available for Eon Mode on-premises. You can stop a subcluster on-premises, provided doing so will not bring down the database. You cannot terminate a subcluster on-premises, because terminating a subcluster stops the nodes and then terminates the cloud instances those nodes reside on. MC cannot terminate on-premises Vertica host machines.

To free up some of the Vertica host machines in an on-premises subcluster, scale down the subcluster.

See also

Subcluster action rules in MC

Node action rules in MC

The table below summarizes when each node action is available for nodes in the primary subcluster or in a secondary subcluster.

For Eon Mode databases, MC supports actions for subcluster and node management for the following public and private cloud providers:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Pure Storage FlashBlade

> **Note**
> Enterprise Mode does not support subclusters.

For Enterprise Mode databases, MC supports these actions:

- In the cloud on AWS: Add Node action, Add Instance action.
- On-premises: Add Node action.

> **Note**
> In the cloud on GCP, Enterprise Mode databases are not supported.

Node Action	Primary Subcluster	Secondary Subcluster
Start Node Starts instance if needed.	Always allowed.	Always allowed.
Stop Node Stops the instance on a cloud platform.	• Allowed when K-safety is >=1. After the node is stopped, there must be 3 or more remaining UP nodes. • Not allowed when K-safety is 0.	• Allowed when K-safety is >=1. After the node is stopped, there must be 3 or more remaining UP nodes. • Allowed when K-safety is 0.
Restart Node	• Always allowed in the cloud. • Not available on premises.	• Always allowed in the cloud. • Not available on premises.
The **Remove Node** action has been deleted from existing actions for all nodes. Use Scale Down subcluster instead.		

When you start or stop a node on-premises, MC starts or stops the node in the database, but does not start or stop the Vertica host machine. The Restart Node action is not available for on-premises Eon Mode databases.

See also

Starting, stopping, and restarting nodes in MC

Starting, stopping, and restarting nodes in MC

In MC you can start, stop, and restart nodes in a subcluster in your database. This allows you to tailor the amount of compute power you are using to the current demands for the workload assigned to that subcluster.

For Eon Mode databases, MC supports actions for subcluster and node management for the following public and private cloud providers:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Pure Storage FlashBlade

> **Note**
> Enterprise Mode does not support subclusters.

For Enterprise Mode databases, MC supports these actions:

- In the cloud on AWS: Add Node action, Add Instance action.
- On-premises: Add Node action.

> **Note**
> In the cloud on GCP, Enterprise Mode databases are not supported.

On the **Manage > Subcluster** page in the right column, the **Node Actions** column displays icons that let you start, stop, or restart a node and the underlying cloud machine. Hover over an icon to read the action it performs. If an icon is grayed out, that action is not available for that node.

- **Start Node** . Start an individual node in the subcluster.
- **Stop Node** . Stop an individual node in the subcluster.
- **Restart Node (GCP only)** . Restart an individual node in the subcluster.

See also

Node action rules in MC

Eon Mode on-premises

In this section

- Reviving an Eon Mode database on premises with FlashBlade using MC
- Creating an Eon Mode database on premises with FlashBlade in MC

Reviving an Eon Mode database on premises with FlashBlade using MC

An Eon Mode database keeps an up-to-date version of its data and metadata in its communal storage location. After a cluster hosting an Eon Mode database is terminated, this data and metadata continue to reside in communal storage. When you revive the database later, Vertica uses the data in this location to restore the database in the same state on a newly provisioned cluster.
(For details on how to stop or terminate a cluster using Management Console, see Viewing and managing your cluster .)

Prerequisites

You can revive a *terminated* Eon Mode database on premises that uses a Pure Storage FlashBlade appliance as its communal storage location if you have the following facts available:

- The endpoint IP address of the FlashBlade.
- The endpoint port for the FlashBlade.
- The access key and secret key for FlashBlade.
- The S3 URL for the FlashBlade.
- The name of the stopped database stored on the FlashBlade, that you wish to revive.

Revive an Eon Mode database from communal storage on FlashBlade

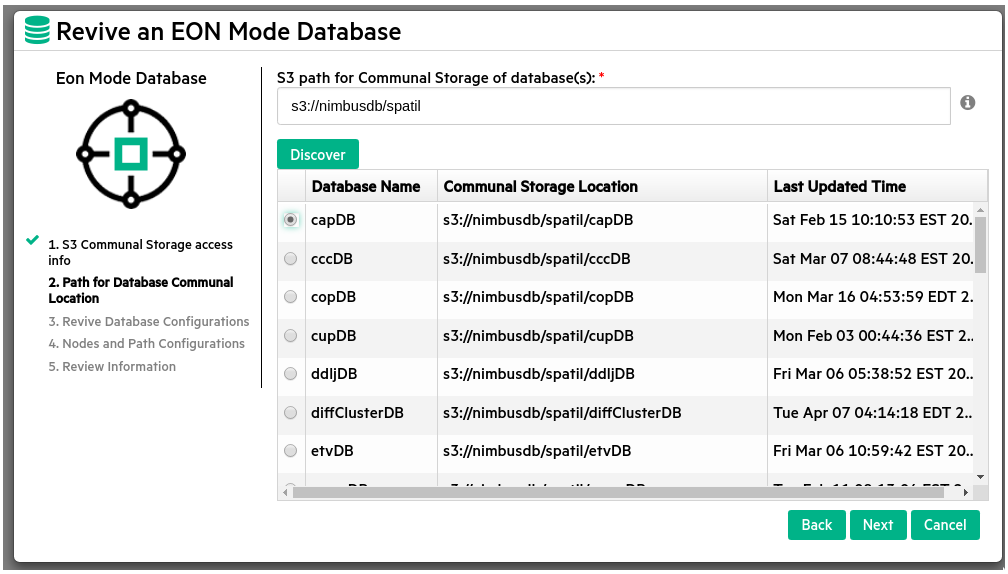
1. On the MC Infrastructure page, click the box for the cluster you wish to revive. MC displays a pop-up with cluster details and action buttons.
2. Click **Revive Database** . MC launches the **Revive an Eon Mode Database** wizard.
3. In the **S3 Communal Storage Information** screen, enter the following fields, then click **Next:**

S3-compatible End Point IP	The IP address of the Pure Storage FlashBlade appliance.
End Point IP	The port for the FlashBlade.
Access ID	The access key for the FlashBlade.
Secret Key	The secret key for the FlashBlade.

4. When you click **Next** , MC validates your credentials. If validation is successful, MC reads the list of S3 buckets on the end point (the FlashBlade).
5. In the **Path for Database Communal Location** screen, enter the following field , then click **Discover** .

S3 path for Communal Storage of database(s)	Enter the complete S3 bucket URL for the database you wish to revive, in the following format: s3:// *bucket-name* / *subfolder-name*

6. MC populates the table under the Discover button with the database names and complete S3 URLs of all the databases it finds on the S3 location.



1. Click the radio button for the database you want to revive, then click **Next** .
2. In the **Details of Selected Database to Revive** screen, MC pre-fills most of the fields. Enter and confirm your password:

Vertica **Database Name**	Database name is pre-entered.
Original Database Version	Database Vertica version is pre-filled.
Cluster Vertica Version	Cluster Vertica version is pre-filled.
Database size	Number of nodes is pre-filled.
Original Vertica Database Super User Name	Database super user name is pre-filled.
Password	Enter the database password.
Confirm Password	Re-enter the password to confirm.

3. MC validates the information. If validation is successful, the **Next** button changes from grayed out to active. Click **Next** .
4. In the **Eon Mode Database** screen, select the IP addresses of the nodes in the cluster on which you wish to revive the database. MC fills in the **Number of Nodes** field and displays the catalog path and depot path that were configured for the database when it was created.
5. In the **Temp Path** field, enter the complete path for the Temp directory. Then click **Next** .
6. MC displays the **S3 Provider Details** screen with a summary of all the information you have entered to revive the database. Review the information to verify it is correct. Then click **Revive Database** .
7. MC displays a progress screen indicating the database revival tasks that have been completed and the overall percentage of the revival that is complete. Wait until the revive is 100% complete, then click **Close** .

When the work is complete, Vertica displays the MC landing page.

Creating an Eon Mode database on premises with FlashBlade in MC

This topic describes how to create an Eon Mode database using only on-premises machines, with Pure Storage FlashBlade as the communal storage reservoir, using Management Console.

Step 1: create a bucket and credentials on the Pure Storage FlashBlade

To use a Pure Storage FlashBlade appliance as a communal storage location for an Eon Mode database you must have:

- The IP address of the FlashBlade appliance. You must also have the connection port number if your FlashBlade is not using the standard port 80

or 443 to access the bucket. All of the nodes in your Vertica cluster must be able to access this IP address. Make sure any firewalls between the FlashBlade appliance and the nodes are configured to allow access.
- The name of the bucket on the FlashBlade to use for communal storage.
- An access key and secret key for a user account that has read and write access to the bucket.

See the Pure Storage support site for instructions on how to create the bucket and the access keys needed for a communal storage location.

Step 2: install and configure MC

Install and configure Management Console on one of the on-premises machines.

Step 3: create or import a Vertica cluster

In MC, create a Vertica cluster on a group of on-premises machines, or import a previously created cluster to MC.

Step 4: create an Eon Mode database with FlashBlade as communal storage

Create an Eon Mode database on the on-premises cluster, using Pure Storage FlashBlade as your S3-compatible communal storage, as explained below.

1. In MC on the **Infrastructure** page, click the square for the specific cluster where you want to create the database. MC displays a pop-up with cluster details and action buttons.
2. Click **Create Database.** MC launches the **Create a New Database** wizard.
3. In the **Vertica Database Mode** screen, click the icon for **Eon Mode Database** , then click **Next** .
4. In the **S3 Communal Storage Information** screen, enter the following information, then click **Next** :

S3-compatible End Point IP	Enter the IP address of the Pure Storage FlashBlade appliance.
End Point Port	Enter the port of the FlashBlade appliance for a valid connection with the Management Console (80 for an unencrypted connection or 443 for an encrypted connection).
Access ID	Enter the access key for the FlashBlade.
Secret Key	Enter secret key for the FlashBlade.
Communal Location URL	Enter a communal location URL beginning with s3:// that points to the third-party storage appliance you will be using, for example Pure Storage FlashBlade. For example, s3:// *bucket* / *subfolder* or s3:// *bucket* / *folder* / *subfolder* . The *bucket* must already exist, and the *subfolder* must not exist.

5. In the **Database Parameters** screen, enter these fields, then click **Next** :

Database Name	Enter 1-30 letters, numbers, and underscores. The first character must be alphabetic.
Password	Enter a new administrator password for the new Eon Mode database. Allowed characters: Alphanumeric (letters and digits) and ASCII special characters. Maximum is 100 characters.
Confirm Password	Enter the same password again to confirm it.
Port	Currently, Vertica supports only port 5433.
Catalog Path	Enter the directory path for the catalog of the Eon Mode database. The catalog should always reside on persistent storage. This path must exist on the host machines. If the path does not exist on the host machines you must create it manually before specifying it in this wizard.
Depot Path	Enter the directory path for the Eon Mode depot. MC populates the depot with recently used data. Queries that find their data in the depot get better performance. The directory must exist on the filesystem that was mounted during cluster creation. If the path does not exist on the host machines you must create it manually before specifying it in this wizard.
Depot Size	Enter the percentage of total disk space to use for the Eon Mode depot. The default is 60% of available disk. The maximum is 99,999T, or the equivalent in G or M units.

Temp Path	Directory path to use for temp data. This path must exist on the host machines. If the path does not exist on the host machines you must create it manually before specifying it in this wizard. For Temp, you may want to use a scratch disk.

6. In the **Specify Node Preferences** screen, select the IP addresses of the hosts for the database nodes. Then enter these fields, and click **Next** :

Number of nodes selected for the database	This value is set automatically. Always equal to the number of IP addresses you select.
Data Segmentation Shards	The number of shards is the number of segments that the data will be divided into in communal storage. For best performance, Vertica recommends you choose a number of shards that is a multiple of the number of nodes in the database. Vertica requires a minimum of 1 and allows a maximum of 960 shards. A good strategy is to choose an even number divisible by multiple factors. Consider the future growth of your database, as Vertica requires that the number of nodes be no greater than the number of shards, and you add nodes later but you cannot change the number of shards later.

7. MC displays a confirmation screen labeled **S3 Provider Details** , that summarizes all of your choices for configuring the Eon database. Verify the details are correct, then click **Create Database** .
8. MC creates the database, displaying a progress indicator screen. Wait for all steps to complete, which may take several minutes.

When the work is complete, navigate to the MC landing page to use your new database.

Managing queries using MC

Management Console allows you to view the query plan of an active query or a manually entered query specified by the user.

1. On the MC Home Page, click the database you want to view the **Overview** page.
2. Select the **Activity** tab to view the query activity.
3. Click the **Explain** tab to access the query plan.

See Working with query plans in MC and Accessing query plans in Management Console for further information.

Management Console provides two options for viewing the query plan: **Path Information** and **Tree Path** . For details on each, refer Query plan view options .

Additionally, you can also Viewing projection and column metadata using the **MC Explain** tab.

See also

- Expanding and collapsing query paths
- Clearing query data

In this section

- About profile data in Management Console
- Profiling queries using MC
- Viewing profile data in MC

About profile data in Management Console

After you profile a specific query, the Management Console Explain page displays profile data like query duration, projection metadata, execution events, optimizer events, and metrics in a pie chart.

See the following links for more information on the kinds of profile data you can review on the Management Console Explain page:

- Projection metadata
- Query phase duration
- Profile metrics
- Execution events
- Optimizer events

In this section

- Projection metadata
- Query phase duration
- Profile metrics
- Execution events

- Optimizer events

Projection metadata

To view projection metadata for a specific projection, click the projection name in the EXPLAIN output. Metadata for that projection opens in a pop-up window.

To view projection data for all projections accessed by that query, click the **View Projection Metadata** button at the top of the **Explain** page. The metadata for all projections opens in a new browser window.

> **Note**
> If the **View Projection Metadata** button is not enabled, click **Profile** to retrieve the profile data, including the projection metadata.

The projection metadata includes the following information:

- Projection ID
- Schema name
- Whether or not it is a superprojection
- Sort columns
- IDs of the nodes the projection is stored on
- Whether or not it is segmented
- Whether or not it is up to date
- Whether or not it has statistics
- Owner name
- Anchor table name

To display a SQL script that can recreate the projection on a different cluster, click **Click to get export data** . This script is identical to the output of the EXPORT_OBJECTS function. The SQL script opens in a pop-up window.

Copy and paste the command from this window, and click **Close** .

Query phase duration

This pie chart appears in the upper-right corner of the Query Plan window. It shows what percentage of total query processing was spent in each phase of processing the query.

The phases included in the pie chart (when applicable) are:

- Plan
- InitPlan
- SerializePlan
- PopulateVirtualProjection
- PreparePlan
- CompilePlan
- ExecutePlan
- AbandonPlan

Hover over the slices on the pie chart or over the names of the phases in the box to get additional information. You can see the approximate number of milliseconds (ms) and percentage used during each phase.

> **Note**
> The time data in the profile metrics might not match the times in the query phase duration. These times can differ because the query phase duration graph uses the longest execution time for a given phase from all the nodes. Network latency can add more data, which is not taken into account in these calculations.

Profile metrics

In the **Path Information** view, the area to the right of each query path contains profile metrics for that path.

- **Disk** —Bytes of data accessed from disk by each query path. If none of the query paths accessed the disk data, all the values are 0.

- **Memory** —Bytes of data accessed from memory by each query path.
- **Sent** —Bytes of data sent across the cluster by each query path.
- **Received** —Bytes of data received across the cluster by each query path.
- **Time** —Number of milliseconds (ms) that the query path took to process on a given node, shown on progress bars. The sum of this data does not match the total time required to execute the query. This mismatch occurs because many tasks are executed in parallel on different nodes. Hover over the progress bars to get more information, such as total bytes and percentages.

> **Note**
> The time data in the profile metrics might not match the times in the Query phase duration chart. These times can differ because the query phase duration graph uses the longest execution time for a given phase from all the nodes. Network latency can add more data, which is not taken into account in these calculations.

Execution events

To help you monitor your database system, Vertica logs significant events that affect database performance and functionality. Click **View Execution Events** to see information about the events that took place while the query was executing.

If the **View Execution Events** button is not enabled, click **Profile** to retrieve the profile data, including the execution events.

The arrows on the header of each column allow you to sort the table in ascending or descending order of that column.

The execution events are described in the following table.

Event Characteristic	Details
Time	Clock time when the event took place.
Node Name	Name of the node for which information is listed.
Session ID	Identifier of the session for which profile information is captured.
User ID	Identifier of the user who initiated the query.
Request ID	Unique identifier of the query request in the user session.
Event Type	Type of event processed by the execution engine. For a list of events and their descriptions, see Initial process for improving query performance .
Event Description	Generic description of the event.
Operator Name	Name of the Execution Engine component that generated the event. Examples include but are not limited to: • DataSource • DataTarget • NetworkSend • NetworkRecv • StorageUnion Values from the Operator name and Path ID columns let you tie a query event back to a particular operator in the query plan. If the event did not come from a specific operator, then this column is NULL.
Path ID	Unique identifier that Vertica assigns to a query operation or a path in a query plan. If the event did not come from a specific operator, this column is NULL.
Event OID	A unique ID that identifies the specific event.
Event Details	A brief description of the event and details pertinent to the specific situation.

Suggested Action	Recommended actions (if any) to improve query processing.

Optimizer events

To help you monitor your database system, Vertica logs significant events that affect database performance and functionality.Click **View Optimizer Events** to see a table of the events that took place while the optimizer was planning the query.

If the **View Optimizer Events** button is not enabled, click **Profile** to retrieve the profile data, including the optimizer events.

The arrows on the header of each column allow you to sort the table in ascending or descending order of that column.

The following types of optimizer events may appear in the table:

Event characteristic	**Details**
Time	Clock time when the event took place.
Node Name	Name of the node for which information is listed.
Session ID	Identifier of the session for which profile information is captured.
User ID	Identifier of the user who initiated the query.
Request ID	Unique identifier of the query request in the user session.
Event Type	Type of event processed by the optimizer.
Event Description	Generic description of the event.
Event OID	A unique ID that identifies the specific event.
Event Details	A brief description of the event and details pertinent to the specific situation.
Suggested Action	Recommended actions (if any) to improve query processing.

Profiling queries using MC

Management Console allows you to view profile data for a query.

- On the MC Home Page, click the database to view the **Overview** page.
- Click the **Explain** tab to perform tasks related to profiling a query.

See Viewing profile data in MC for further details.

On the **Explain** tab, you can view the following profile data using MC:

- Query phase duration
- Projection metadata
- Execution events
- Optimizer events
- Profile metrics

You can use any of the four different formats to view the profile data:

- Path Information view
- Query Drilldown view
- Tree Path view
- Profile Analysis view

See Viewing different profile outputs for detailed explanation of each view.

Additionally, Management Console supports different color codes for viewing the progress of profiling a query. For an explanation of these color codes, see Monitoring profiling progress .

See also

- Viewing profile data in MC

Viewing profile data in MC

Management Console allows you to view profile data about a single query. You can:

- Review the profile data in multiple views
- View details about projection metadata, execution events, and optimizer events
- Identify how much time was spent in each phase of query execution and which phases took the most amount of time

After you select the database you want to use, you can view the profile data using Management Consolein either of two ways:

- Focus on specific areas of database activity, such as spikes in CPU usage
- Review the profile data for a specific query

To focus on specific areas of database activity:

1. At the bottom of the Management Console window, click the **Activity** tab.
2. From the list at the top of the page, select **Queries** .
3. On the activity graph, click the data point that corresponds to the query you want to view.
4. In the **View Plan** column, click **Profile** next to the command for which you want to view the query plan. Only certain queries, like SELECT, INSERT, UPDATE, and DELETE, have profile data.
5. In The **Explain Plan** window, Vertica profiles the query.
6. You can view the output in Path Information view, Query Plan Drilldown view, Tree Path view, or Profile Analysis view. To do so, click the respective buttons on the left of the output box.

To review the profile data for a specific query:

1. In the **Explain** window, type or paste the query text into the text box. Additionally, you can monitor queries that are currently running. To do so, perform one of the following. In the **Find a Query By ID** input window:
   - Enter the query statement and transaction ID
   - Click the **Browse Running Queries** link

   > **Caution**
   > If you enter more than one query, Management Console profiles only the first query.

2. To receive periodic updates about the query's progress and resources used, select the **Enable Monitoring** check box. As a best practice, avoid specifying an interval time of less than 60 seconds because doing so may slow your query's progress.
3. Click the **Profile** button.
   While Vertica is profiling the query, a **Cancel Query** button is enabled briefly, allowing you to cancel the query and profiling task. If the **Cancel Query** button is disabled, that means Management Console does not have the proper information to cancel the query or the query is no longer running in the database.
   When processing completes, the profile data and metrics display below the text box. You can view the output in Path Information view, Query Plan Drilldown view, Tree Path view , or Profile Analysis view. To do so, click the respective view buttons on the left of the output box.

In this section

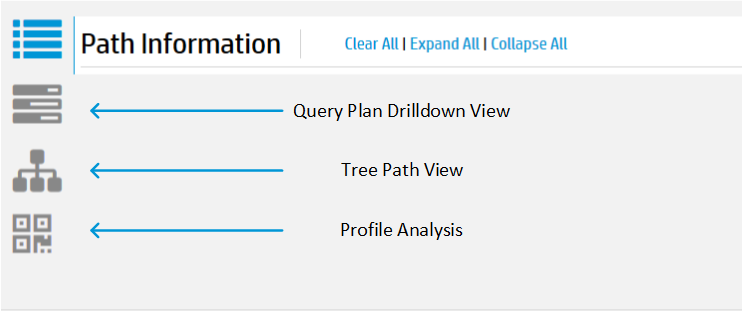
- Viewing different profile outputs
- Monitoring profiling progress
- Expanding and collapsing query path profile data

Viewing different profile outputs

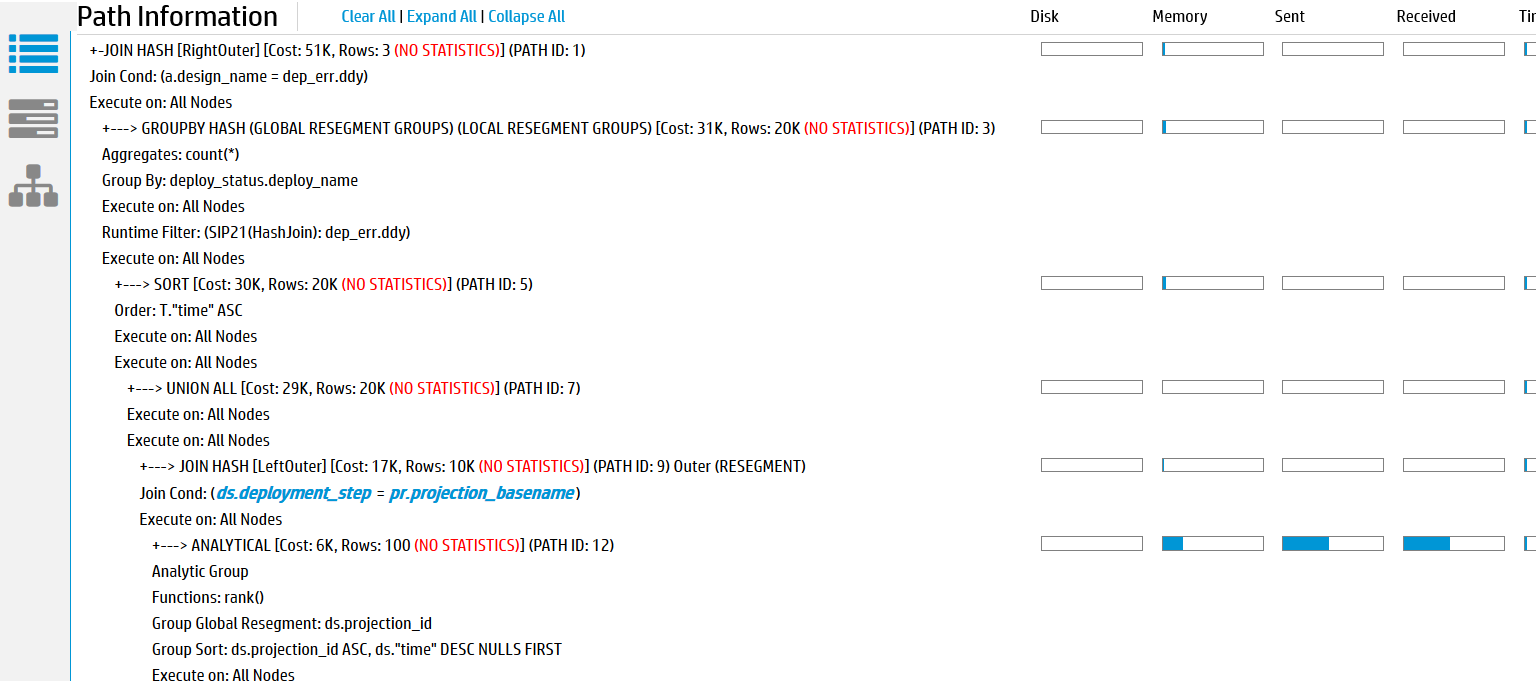
Vertica Management Console allows you to examine the results of your query profile in multiple views. You can view your profile in the following formats:

- Path Information view
- Query Drilldown view
- Tree Path view
- Profile Analysis view

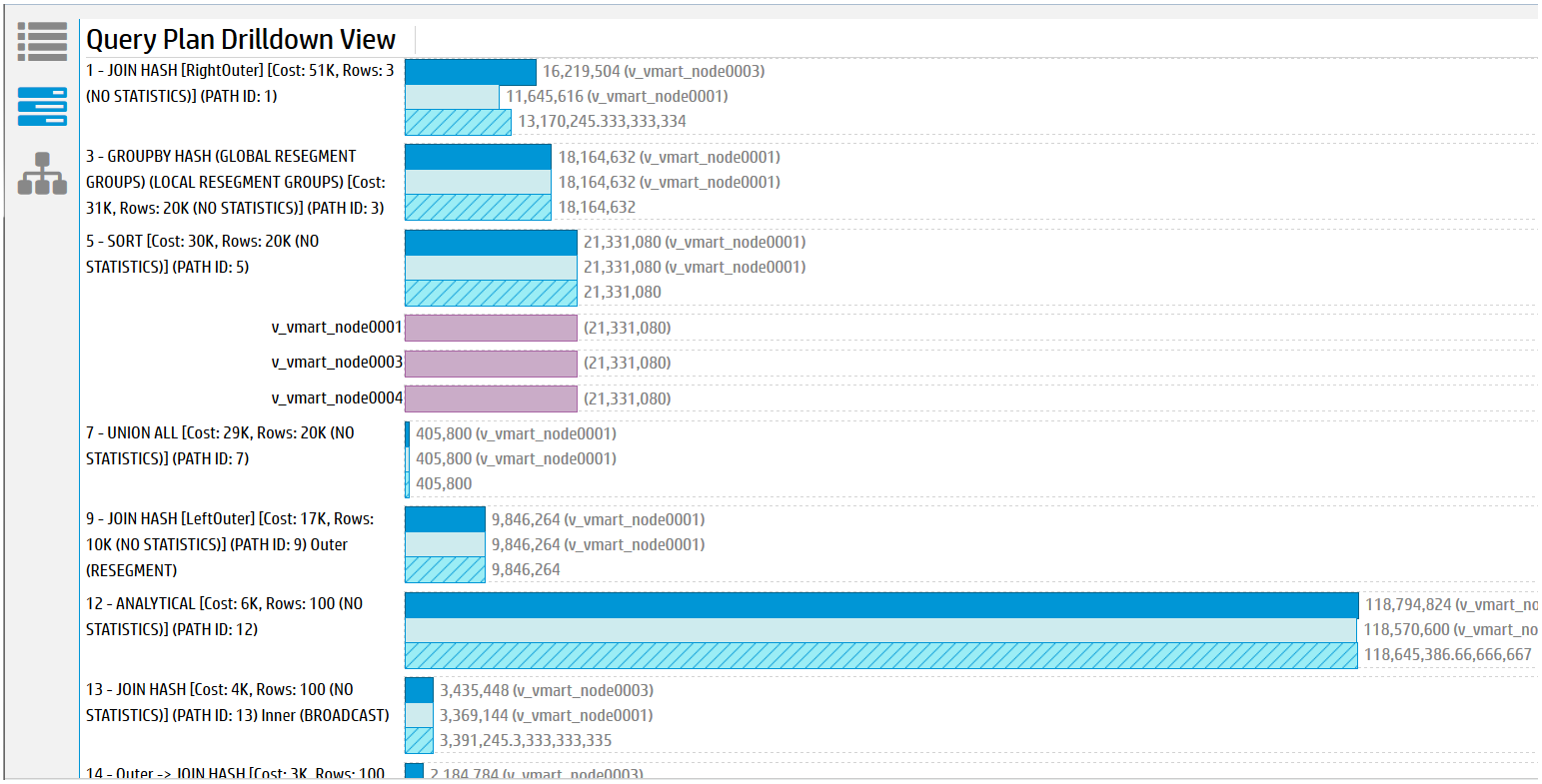
You can change the query profile output using the icons on the bottom portion of the **Explain** page.



The **Path Information** view displays the query plan path along with metric data. If you enable profile monitoring, the data will update at the specified interval. To view metadata for a projection or a column, click the object name in the path output. A pop-up window displays the metadata if it is available.

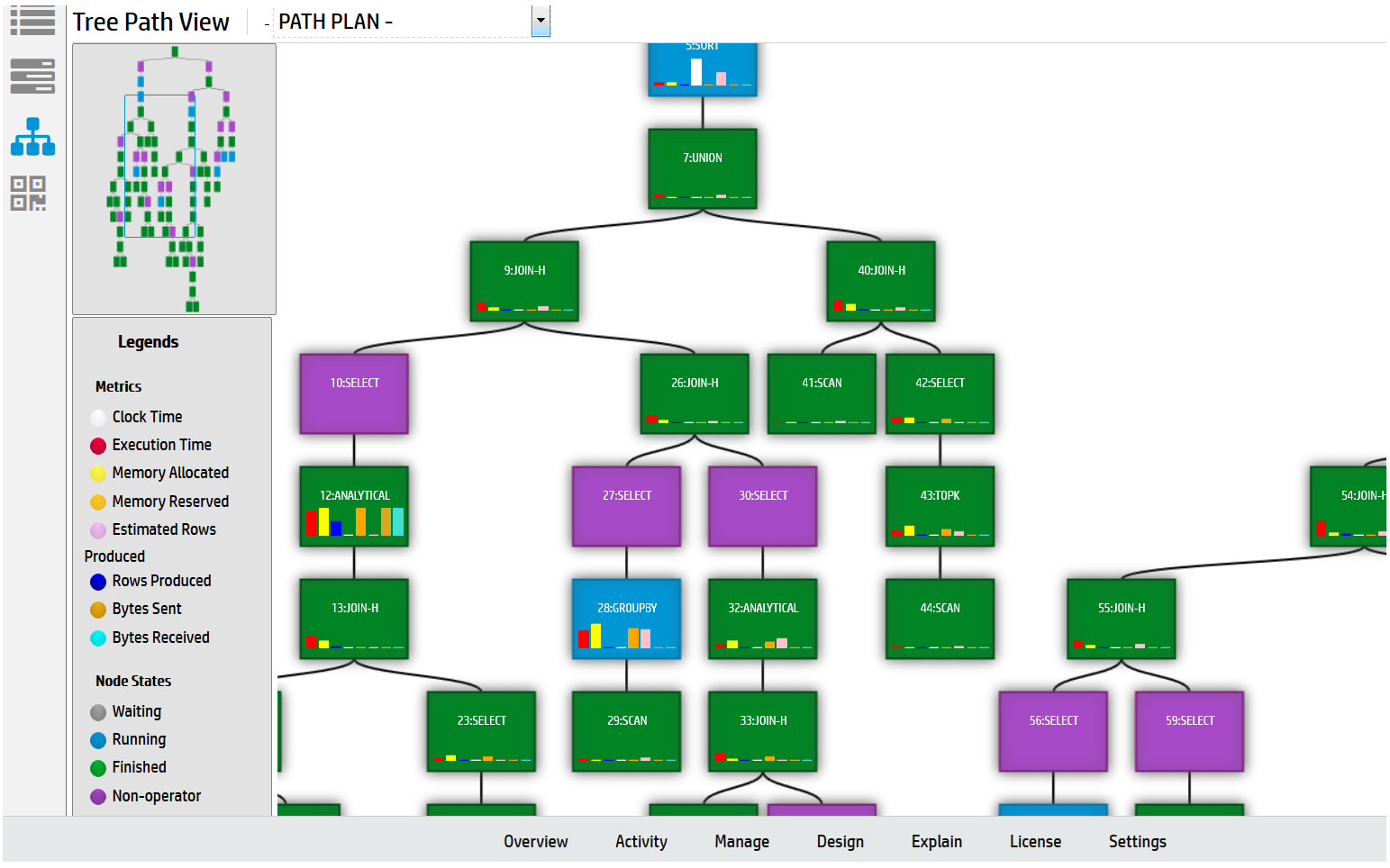


The **Query Plan Drilldown** view shows detailed counter information at the node and operator level.

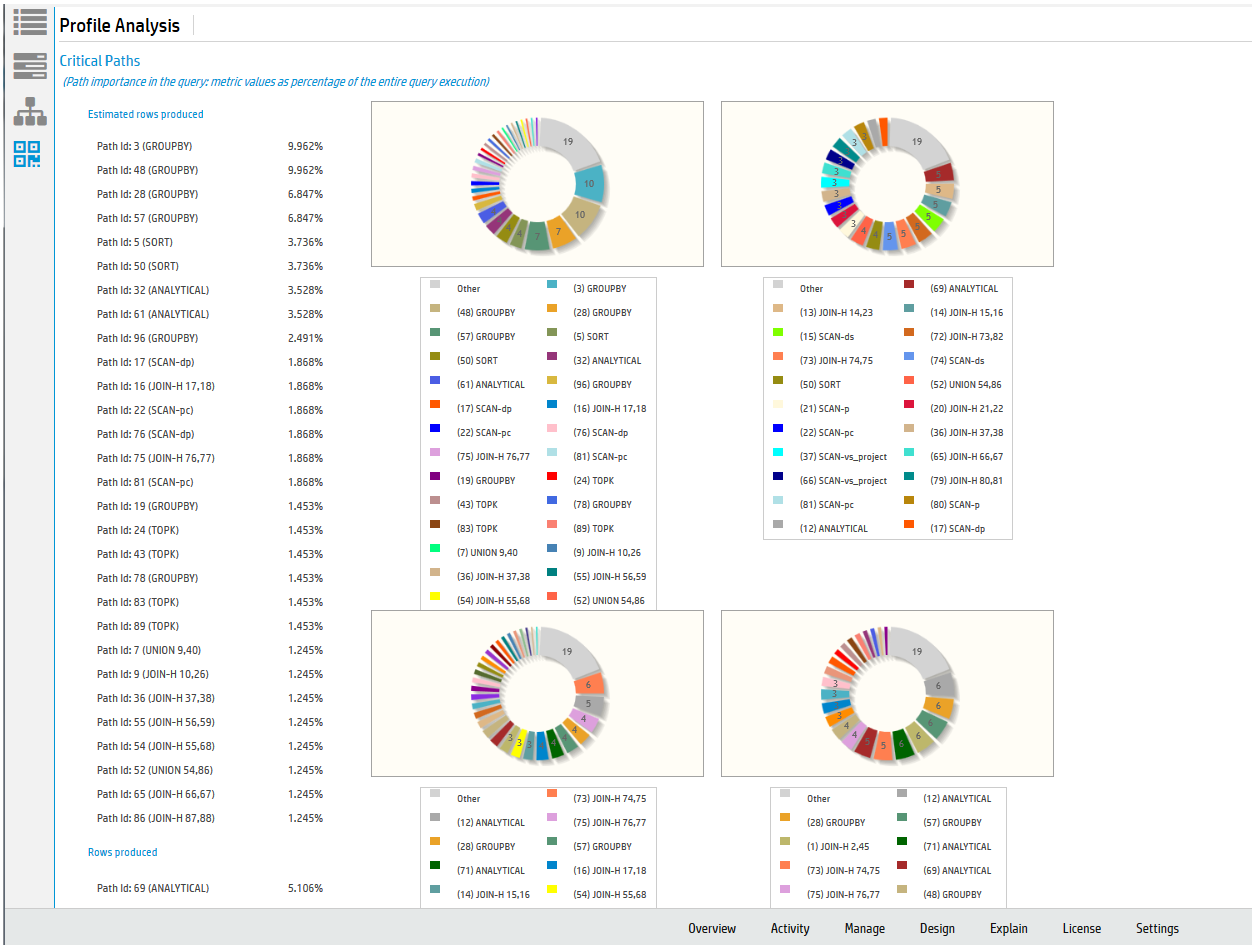


For each path, the path number is listed along with statistical information on the node and operator level. This view allows you to see which nodes are acting as outliers. Click on any of the bars to expand details for that node.

The **Tree Path** details the query plan in the form of a tree. If you enable monitoring, the state of the path blocks will change depending on whether the path is running, done, or has not started. Metric information is displayed in each path block for the counters you specified in the Profile Settings.



The **Profile Analysis** view allows you to identify any resource outliers. You can compare the estimated rows produced count with the actual rows produced count, view execution time per path, and identify memory usage per path.



When you profile a query, you will also see a pie chart detailing the query phase duration. You can also view projection metadata, execution events, and optimizer events by clicking on the respective buttons next to the pie chart.

Monitoring profiling progress

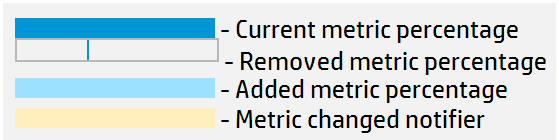
While loading profile data for a query, Management Console can provide updates about the query's progress and resources used.

To enable profiling progress updates, select the Enable Monitoring check box when profiling a query. See Viewing Profile Data in Management Console .

The default interval time is 60 seconds. At the specified interval, Management Console displays an updated view of the query's progress. Note that interval times of less than 60 seconds may slow down your query.

Viewing updated profile metrics

At every interval, Management Console displays a new set of profile metrics. You can view these metrics in Path Information view, Query Plan Drilldown view, or Tree view by clicking the respective view buttons on the left of the output box.



- A dark blue bar indicates the current metric percentage.
- When a metric bar has decreased, a dark blue line indicates the previous metric percentage.
- When a metric bar has increased, a light blue bar indicates the added percentage. The previous percentage appears as a dark blue bar.
- A metric bar highlighted in yellow indicates it has changed since the last interval.
- A metric bar highlighted in red indicates the absolute value of the metric has decreased. This typically means Vertica reported the previous value incorrectly, and has readjusted. (For example, if Vertica previously reported path's Time value as 75 seconds, then reports it as 50 seconds at the next interval, the metric bar turns red to indicate the decrease in absolute Time value.)

Expanding and collapsing query path profile data

When you have a query on the EXPLAIN window, the profile data displays in the right-hand side of the lower half of the window. The query path information can be lengthy, so you can collapse path information that is uninteresting, or expand paths that you want to focus on.

- To collapse all the query paths, click **Collapse All** .
- To expand all the query paths, click **Expand All** .
- To expand an individual query path so you can see details about that step in processing the query, click the first line of the path information. Click the first line again to collapse the path data.

For information about what the profile data means, see About profile data in Management Console .

Working with query plans in MC

Management Console can show you a query plan in easy-to-read format, where you can review the optimizer's strategy for executing a specific query. You can view a query plan in either of two ways:

- View the plan of an active query.
- View the plan for any query that you manually specify.

Access the plan of an active query

1. At the bottom of the Management Console window, click the **Activity** tab.
2. From the list at the top of the page, select **Queries** .
3. On the activity graph, click the data point that corresponds to the query you want to view.
4. In the View Plan column, click **Explain** next to the command for which you want to view the query plan. Only certain queries use query plans—for example, SELECT, INSERT, DELETE, and UPDATE.
5. In the Explain Plan window, click **Explain** . Vertica generates the query plan.
6. (Optional) View the output in Path Information view or Tree Path view. To do so, click the respective view buttons on the left of the output box.

Access the plan for a specific query

1. Locate the query for which you want to see the query plan in either of the following ways:
   - **Queries Not Running** — In the Explain window, type or paste the query text into the text box.
   - **Queries Currently Running** — In the Find a Query By ID input window, perform one of the following actions:
     - Enter the query statement and transaction ID.
     - Click the **Browse Running Queries** link.

   > **Caution**
   > Entering the word EXPLAIN before the query results in a syntax error.

2. Click **Explain** . Vertica generates the plan.

   If the query is invalid, Management Console highlights in red the parts of your query that might have caused a syntax error.
3. (Optional) View the output in Path Information view or Tree Path view. To do so, click the respective view buttons on the left of the output box.

In this section

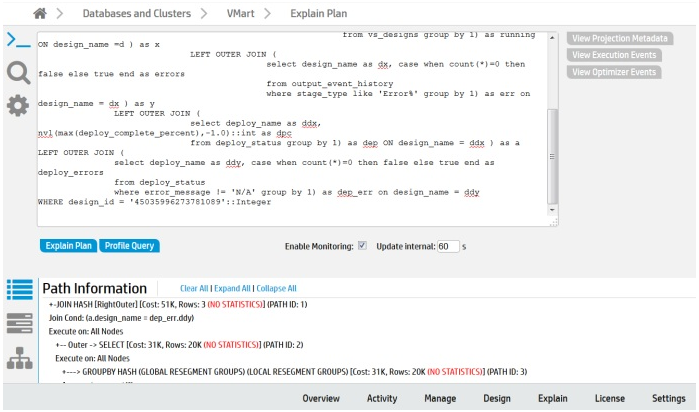
- Accessing query plans in Management Console
- Query plan view options
- Expanding and collapsing query paths
- Clearing query data
- Viewing projection and column metadata

Accessing query plans in Management Console

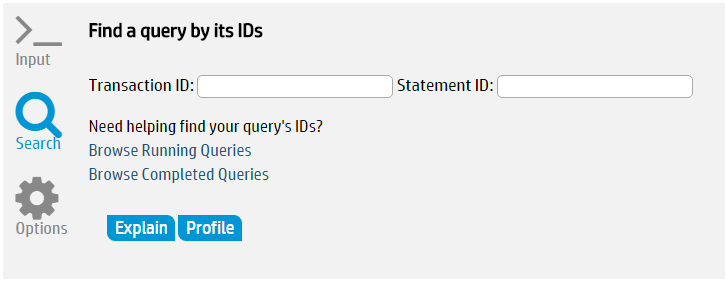
You can access query plans in Management Console in two ways:

- In the Detail page for query-related charts on the database's Activity page, click **Explain** next to a query to view a plan for that query.
- Enter a query manually on the Explain page and click **Explain Plan** .

In both cases, the following window opens:



You can also enter the transaction ID and statement ID or browse running or completed queries in the Find a Query input window:



In the output window, you can perform the following tasks related to the query you entered:

- Expand and collapse query paths.
- Clearing query data.
- View projection and column metadata.
- Use different query plan views.

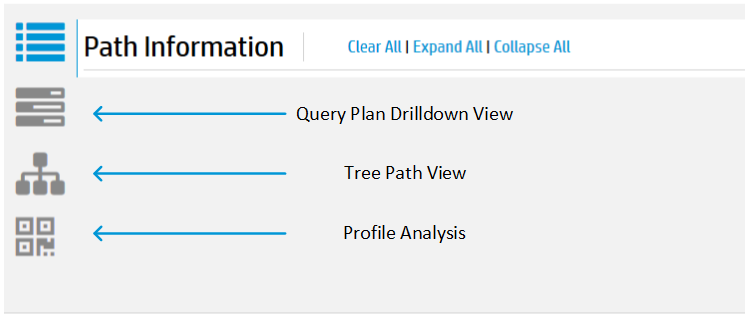
Query plan view options

Vertica Management Console provides two views for displaying query plans:

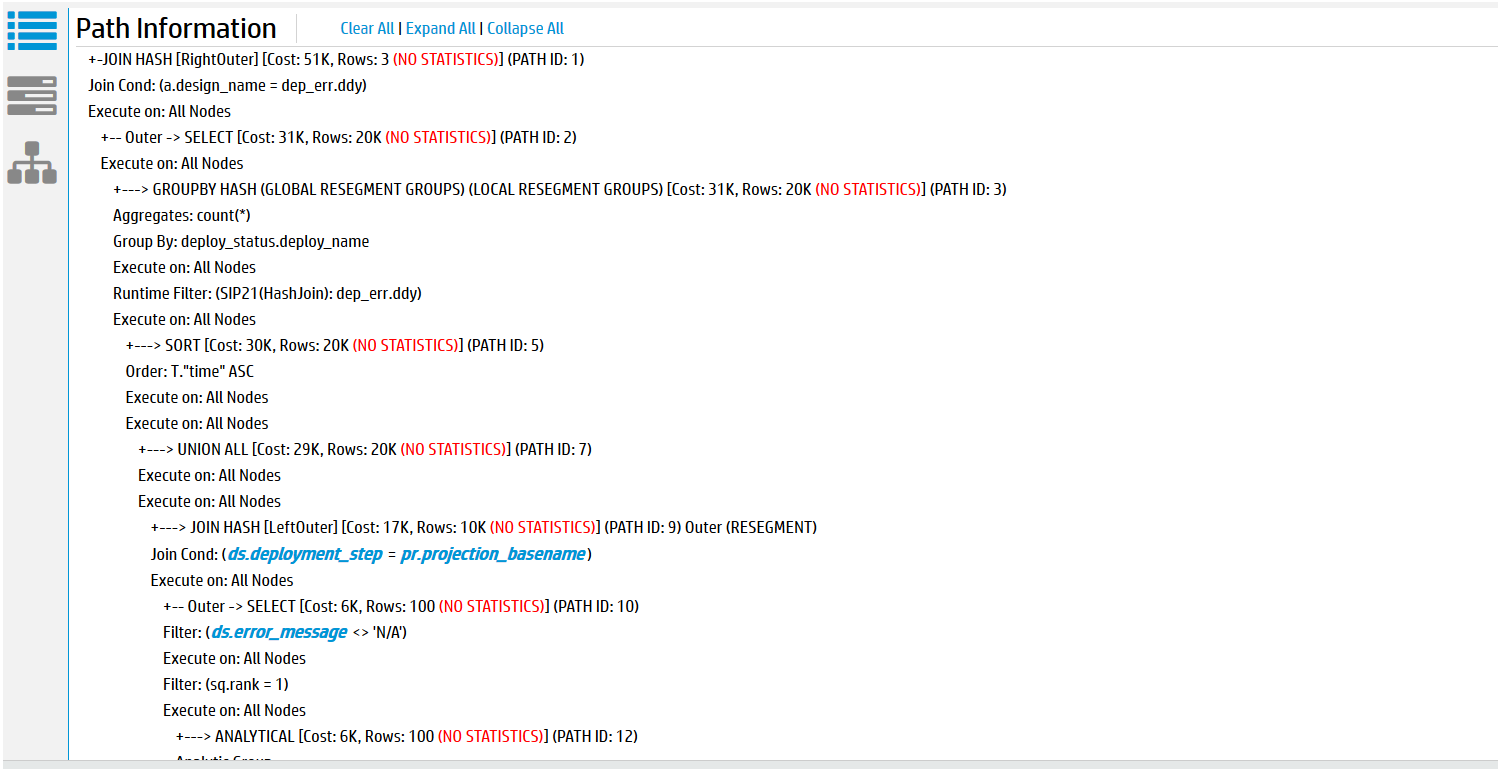
- Path Information
- Tree Path

> **Note**
> Query Plan Drilldown and Profile Analysis output views are only available when you run PROFILE .

You can change the query plan view using the icons on the bottom portion of the **Explain** page.

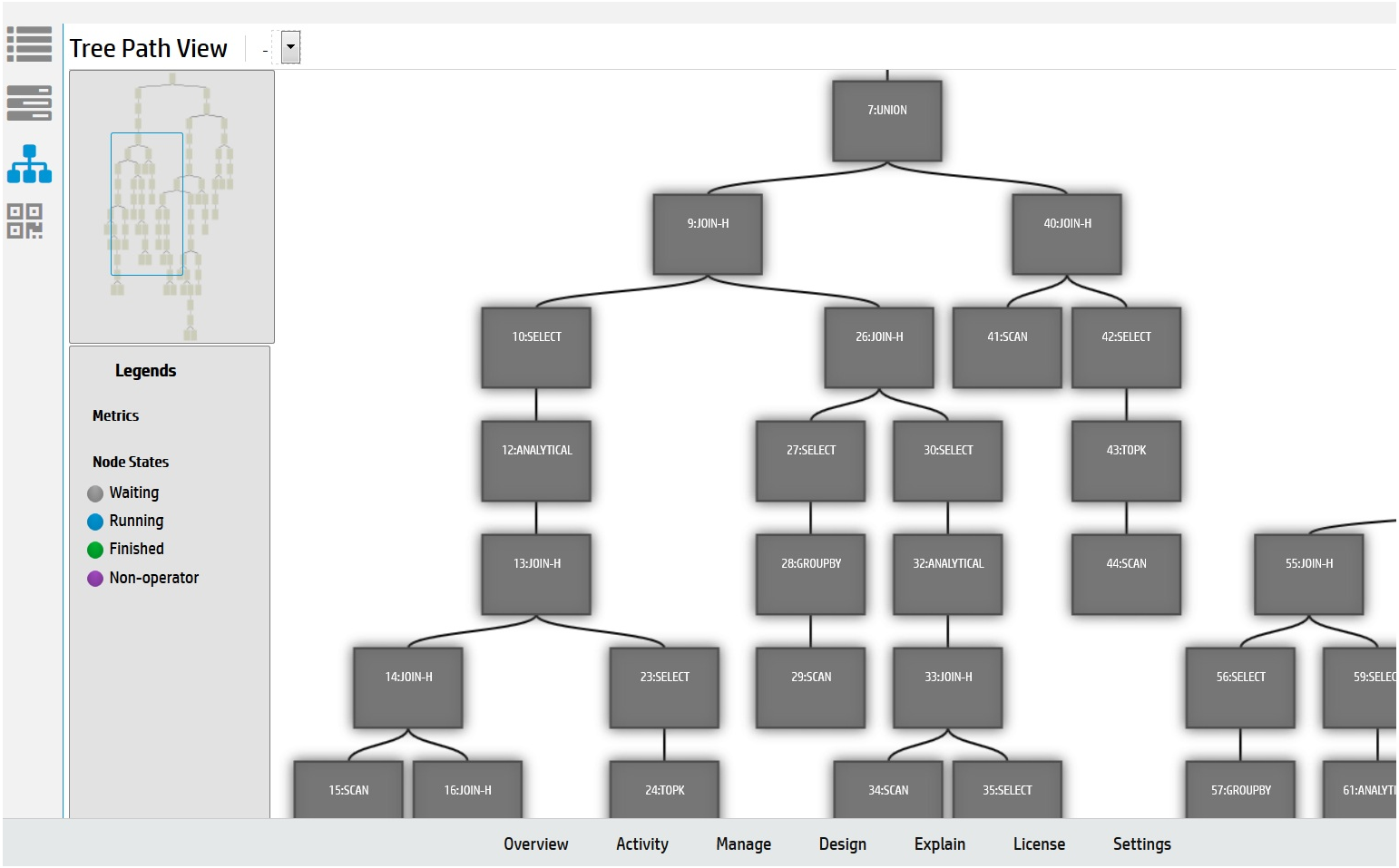


The **Path Information** view displays the query plan path. You can expand or collapse the view to see different levels of detail. To view metadata for a projection or a column, click the object name in the path output. A pop-up window displays the metadata, if it is available.



The

The **Tree Path** view details the query plan in the form of a tree. When you run EXPLAIN, the tree view does not contain any metrics because the query has not yet executed.



Expanding and collapsing query paths

The EXPLAIN window initially displays the full query plan as generated by the EXPLAIN command. Query plans can be lengthy, so you might want to modify the display so you can focus only on areas of interest:

- Collapse All collapses all query paths, and displays only a summary of each path.
- Expand All expands all query paths.
- Click the first line of a path to display details for that path. To collapse that path, click its first line again.

For details about EXPLAIN command output, see EXPLAIN-Generated query plans .

Clearing query data

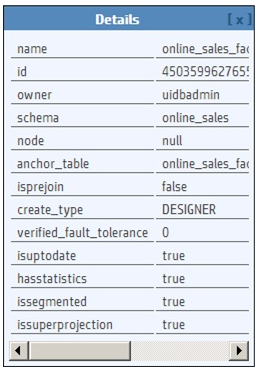
After you finish reviewing the current query data, click Clear All to clear the query text and data. Alternatively, to display information about another query, enter the query text and click Explain or Profile.

Viewing projection and column metadata

In the Management Console EXPLAIN window, when query paths are expanded in the Path Information view, Projection lines contain a projection name and Materialize lines contain one or more column names.

To view metadata for a projection or a column, click the object name. A pop-up window displays the metadata. The following image on the left shows example projection metadata and the image on the right shows example column metadata.

> **Note**
> Most system tables do not have metadata.





When you are done viewing the metadata, close the pop-up window.

Creating a database design in Management Console

Database Designer creates an design that provides excellent performance for ad-hoc queries and specific queries while using disk space efficiently. Database Designer analyzes the logical schema definition, sample data, and sample queries, and creates a physical schema that you can deploy.

For more about how Database Designer works, see the Creating a database design section of the documentation, and About Database Designer .

To use Management Console to create an optimized design for your database, you must be a DBADMIN user or have been assigned the DBDUSER role.

Management Console provides two ways to create a design:

- **Wizard** —This option walks you through the process of configuring a new design. Click **Back** and **Next** to navigate through the Wizard steps, or **Cancel** to cancel creating a new design.
  To learn how to use the Wizard to create a design, see Using the wizard to create a design .
- **Manual** —This option creates and saves a design with the default parameters.
  To learn how to create a design manually, see Creating a design manually .

> **Tip**
> If you have many design tables that you want Database Designer to consider, it might be easier to use the Wizard to create your design. In the Wizard, you can submit all the tables in a schema at once; creating a design manually requires that you submit the design tables one at a time.

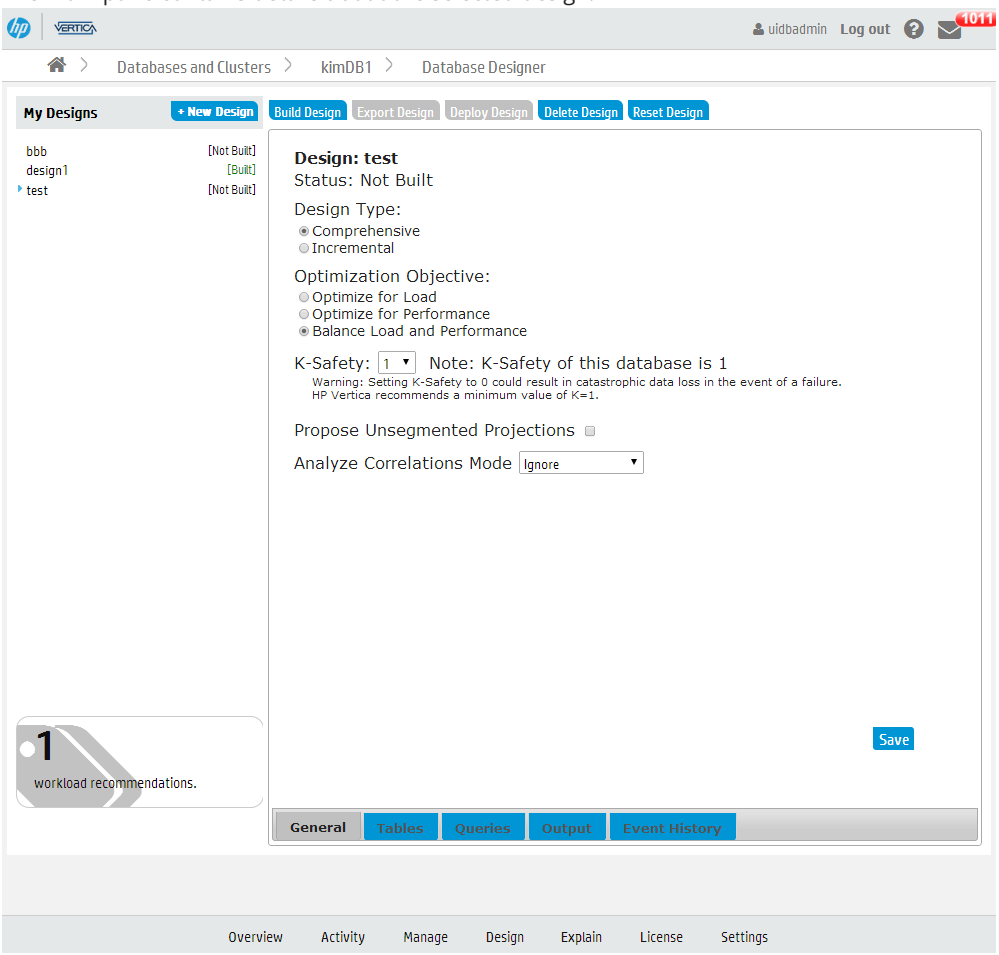
In this section

- Using the wizard to create a design
- Creating a design manually

Using the wizard to create a design

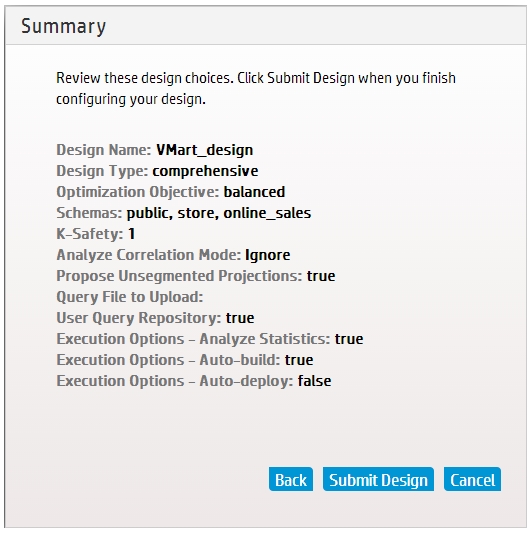
Take these steps to create a design using the Management Console's Wizard:

1. On your database's dashboard, click the **Design** tab at the bottom of the page to navigate to the Database Designer page.
   The left side of the Database Designer page lists the database designs you own, with the most recent design you worked on highlighted. That pane also lists the current status of the design. Details about the most recent design appear in the main pane.
   The main pane contains details about the selected design.



2. To create a new design, click **New Design** .
3. Enter a name for your design, and click **Wizard** .
4. Navigate through the Wizard using the **Back** and **Next** buttons.
5. To build the design immediately after exiting the Wizard, on the **Execution Options** window, select **Auto-build** .
   Important
   Vertica does not recommend that you auto-deploy the design from the Wizard. There might be a delay in adding the queries to the design, so if the design is deployed but the queries have not yet loaded, deployment might fail. If this happens, reset the design, check the **Queries** tab to make sure the queries are loaded, and deploy the design.

6. After you enter all the information, the Wizard displays a summary of your choices. Click **Submit Design** to build your design.



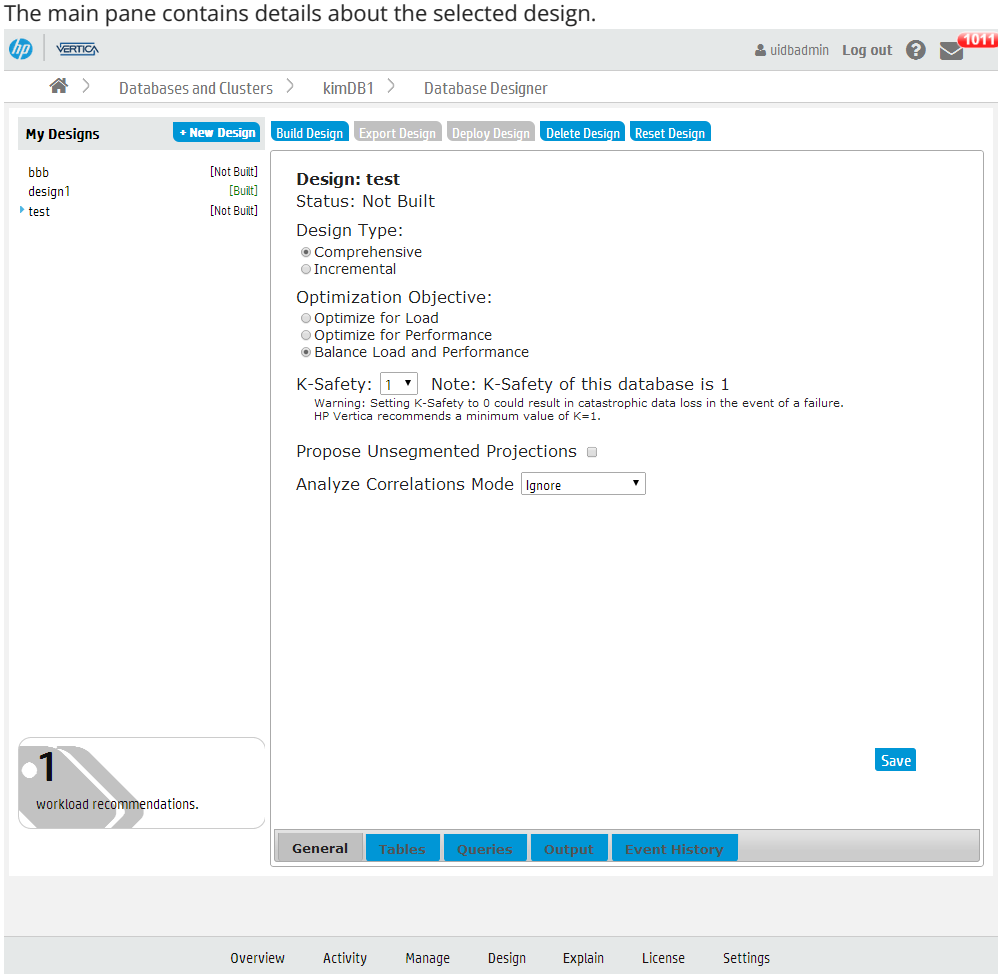
See also

- About Database Designer
- Creating a design manually

Creating a design manually

To create a design using Management Console and specify the configuration, take these steps.

1. On your database's dashboard, click the **Design** tab at the bottom of the page to navigate to the Database Designer page.
   The left side of the Database Designer page lists the database designs you own, with the most recent design you worked on highlighted. That pane also lists the current status of the design. Details about the most recent design appear in the main pane.
   The main pane contains details about the selected design.



2. To create a new design, click **New Design** .
3. Enter a name for your design and select **Manual.**

The main **Database Design** window opens, displaying the default design parameters. Vertica has created and saved a design with the name you specified, and assigned it the default parameters.

4. On the **General** window, modify the design type, optimization objectives, K-safety, Analyze Correlations Mode, and the setting that allows Database Designer to create unsegmented projections.
   If you choose **Incremental** , the design automatically optimizes for the desired queries, and the K-safety defaults to the value of the cluster K-safety; you cannot change these values for an incremental design.
   Analyze Correlations Mode determines if Database Designer analyzes and considers column correlations when creating the design.
5. Click the **Tables** tab. You must submit tables to your design.
6. To add tables of sample data to your design, click **Add Tables** . A list of available tables appears; select the tables you want and click **Save** . If you want to remove tables from your design, click the tables you want to remove, and click **Remove Selected** .
   If a design table has been dropped from the database, a red circle with a white exclamation point appears next to the table name. Before you can build or deploy the design, you must remove any dropped tables from the design. To do this, select the dropped tables and and click **Remove Selected** . You cannot build or deploy a design if any of the design tables have been dropped.
7. Click the **Queries** tab. To add queries to your design, do one of the following:
   - To add queries from the QUERY_REQUESTS system table, click **Query Repository** . In the Queries Repository dialog that appears, you can sort queries by most recent, most frequently executed, and longest running. Select the desired queries and click **Save** . All valid queries that you selected appear in the **Queries** window.

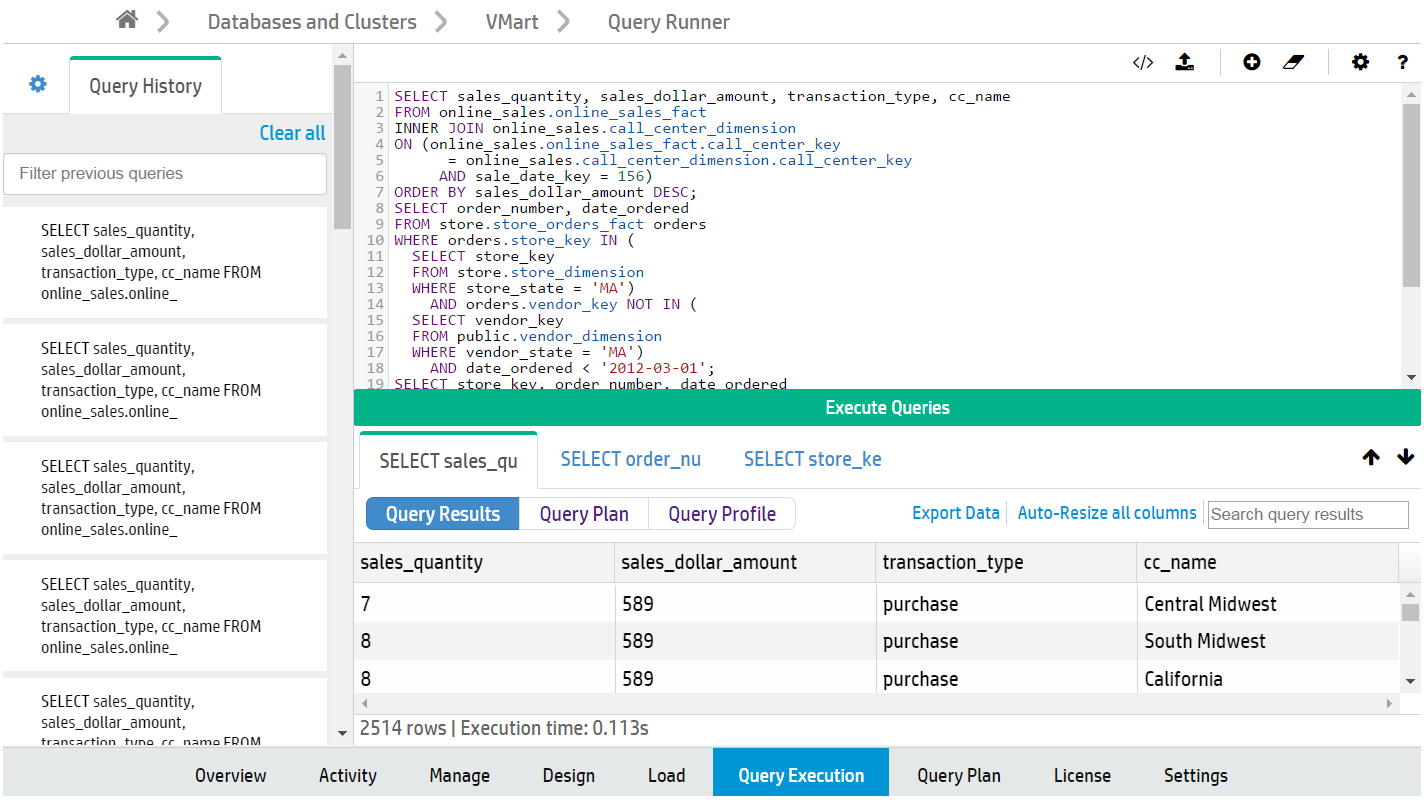
   Database Designer checks the validity of the queries when you add the queries to the design and again when you build the design. If it finds invalid queries, it ignores them.
   If you have a large number of queries, it may take time to load them. Make sure that all the queries you want Database Designer to consider when creating the design are listed in the **Queries** window.
8. Once you have specified all the parameters for your design, you should build the design. To do this, select your design and click **Build Design** .
9. Select **Analyze Statistics** if you want Database Designer to analyze the statistics before building the design.
   For more information see Statistics Analysis .
10. If you do not need to review the design before deploying it, select **Deploy Immediately** . Otherwise, leave that option unselected.
11. Click **Start** . On the left-hand pane, the status of your design displays as **Building** until it is complete.
12. To follow the progress of a build, click **Event History** . Status messages appear in this window and you can see the current phase of the build operation. The information in the Event History tab contains data from the OUTPUT_EVENT_HISTORY system table.
13. When the build completes, the left-hand pane displays **Built** . To view the deployment script, select your design and click **Output** .
14. After you deploy the design using Management Console, the deployment script is deleted. To keep a permanent copy of the deployment script, copy and paste the SQL commands from the **Output** window to a file.
15. Once you have reviewed your design and are ready to deploy it, select the design and click **Deploy Design** .
16. To follow the progress of the deployment, click **Event History** . Status messages appear in this window and you can see the current phase of the deployment operation.
    In the Event History window, while the design is running, you can do one of the following:
    - Click the blue button next to the design name to refresh the event history listing.
    - Click **Cancel Design Run** to cancel the design in progress.
    - Click **Force Delete Design** to cancel and delete the design in progress.
17. When the deployment completes, the left-hand pane displays **Deployment Completed** . To view the deployment script, select your design and click **Output** .

Your database is now optimized according to the parameters you set.

Running queries in Management Console

You can use the Query Runner to run SQL queries on your database through Management Console (MC). After executing a query, you can also get the query plan and profile information for the query on this page.

To reach the Query Runner, select your database from the Home page or the Databases and Clusters page to view your database's Overview page. Select Query Execution at the bottom of the Overview page.



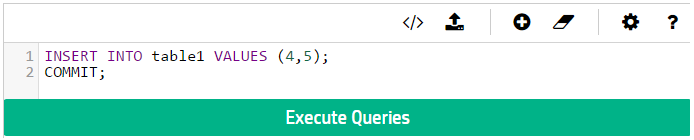
To familiarize yourself with how queries work in Vertica, you can refer to the Queries section of the documentation, as well as the SQL reference.

Limitations

You cannot execute COPY LOCAL statements using the Query Runner. To do so, use the vsql client installed on the server. See Using vsql. (To use MC to import data from Amazon S3 storage to your Vertica database, see Loading data from Amazon S3 using MC.)

Manually commit any transactions (INSERT and COPY statements) you perform by adding the COMMIT statement in the text box after the transaction statements. If you do not do so, the transaction rolls back.

In the following example, to insert values into table1, include a COMMIT statement in the text box and execute the two statements together: INSERT INTO table1 VALUES (1,2); COMMIT;



Format

To input a series of queries, delimit them with a semicolon (;).

To automatically format the SQL text you have input, click the Format icon (</>).

Privileges

It is important when running queries in MC that the database administrator has correctly set up MC user privileges. The administrator must map all MC user profiles to their corresponding database user.

The Query Runner only permits MC users to perform actions that their corresponding Vertica database roles allow.

To set up user mappings, go to Home > MC Settings > User Management.

For more about how mapping MC user profiles to database users works, see Granting database access to MC users. For information about database-level users and privileges, see the Database users and privileges section of the documentation.

Execute a query

The Query Runner provides several ways to input a query to run:

- **Input text.** Enter the text for a query or series of queries into the text box.
- **Import a SQL script.** Click the Upload icon ( ) to the top right of the text box to upload a SQL script (plain text file, typically with an extension of .sql). The queries from that file appears in the text box.

- **Enter a previous query from the Query History tab.** The Query History tab, on the left side of the page, displays the last 100 queries you have executed using the Query Runner on your current device and browser. Click any previous query in this tab to enter that query into the text box.

  Hover over a query in the Query History tab to view all the query text. To clear queries from your history, hover over an individual query and click **x** , or click **Clear all** at the top of the tab. Click the star to the left of any query to favorite it, so it won't be cleared when you click **Clear all** .

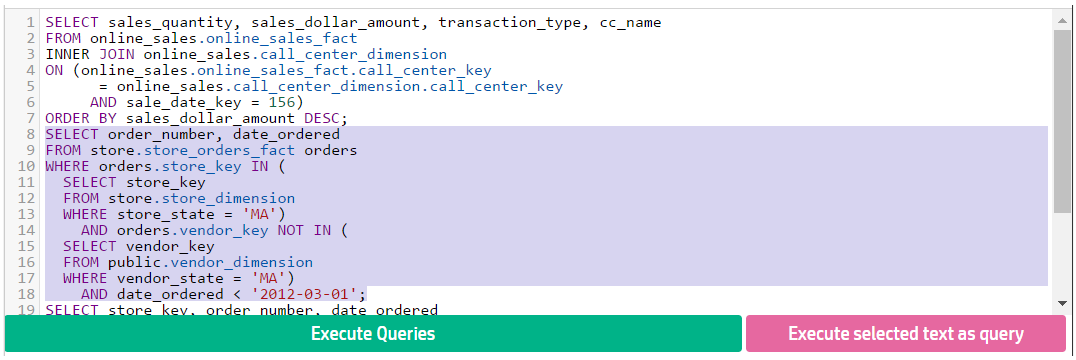
Cick **Execute Query** to run the queries you have input.

You can also execute only a portion of the text entered into the text box, as long as the selected text is a valid query. To do so, select that portion of the text. The **Execute selected text as query** button then appears below the text box.

For example, you might execute only a part of the entered text if you have uploaded a SQL script that containing multiple queries, but you decide to run only one of those queries.

To customize your execution settings, click the Settings icon (⚙) at the top right of the text box:

- **Row Limit:** Set the maximum number of rows to return. By default, the limit is 10000 rows.
- **Search Path:** Specify the schema to query.

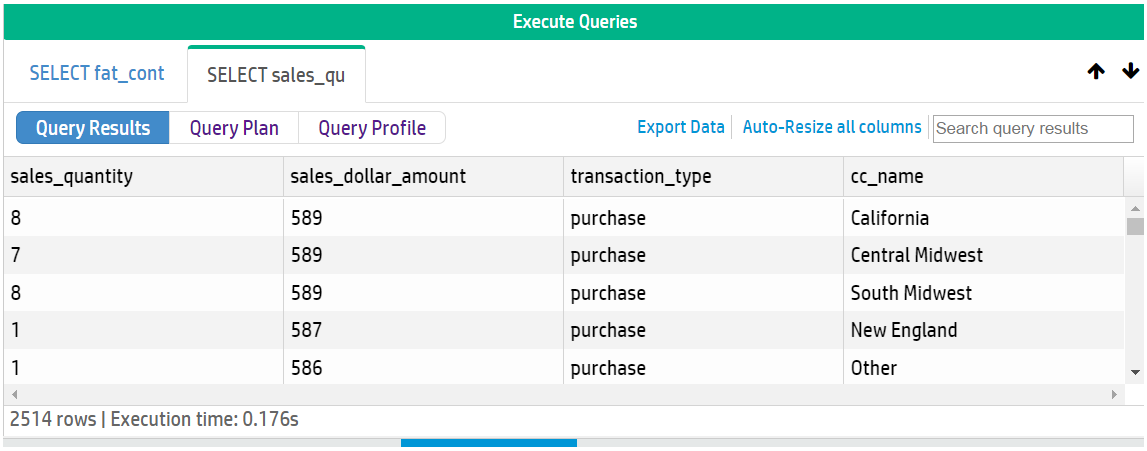


Get query results

The Query Runner returns results in a table format. If you ran multiple queries simultaneously, the results window displays a tab for each set of results. View the number of rows returned and the query execution time at the bottom of the results window.

If your result returns many columns, you can click **Auto-resize all columns** in the top right of the results window for a better fit, or click and drag column borders to manually resize individual columns.

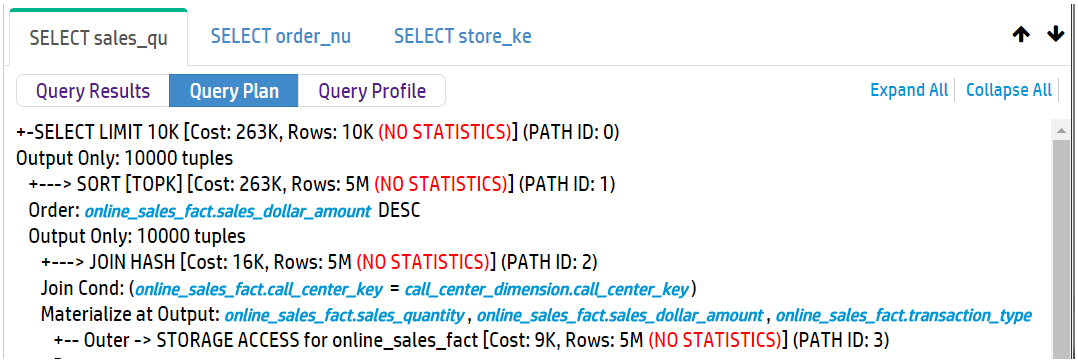
Sort results by clicking on a column name, or use the search bar to narrow down results.



Query plans and profiles

Each query result also displays an option to retrieve the plan or profile for that query.

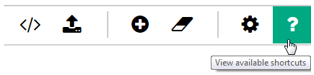
After retrieving a plan or profile, you can expand or collapse the results view to see different levels of detail. To view metadata for a projection or a column, click the object name in the path output. A pop-up window displays the metadata, if it is available.



Note that the Query Runner does not automatically provide query profiles for queries that run for less than 1 second. To do so, prepend the word PROFILE to the query and run it.

You can also profile your query on the **Query Plan** page. The Query Plan page provides more details about both plan and profile results, including a query plan drilldown by node, a tree path view, and a profile analysis.

Keyboard shortcuts



The Query Runner provides the following keyboard shortcuts:

- **?** : Press the question mark to display or dismiss a list of the available keyboard shortcuts. (You can also click the question mark icon at the top right of the text box to view this list.)
- **alt + ↑** : Press alt + up arrow to decrease the height of the text box.
- **alt + ↓** : Press alt + down arrow to increase the height of the text box.
- **ctrl + enter** : Press ctrl + enter to run the query.
- **ctrl + shift + enter** : Press ctrl + shift + enter to run selected text.

See also

- Granting database access to MC users
- Database users and privileges
- Queries
- SQL reference
- Using vsql

Working with workload analyzer recommendations in MC

If queries perform sub-optimally, use Workload Analyzer to get tuning recommendations and hints about optimizing database objects.

Workload Analyzer is a Vertica utility that analyzes system information in Vertica system tables. It then returns a set of tuning recommendations based on statistics, system and data collector events, and database/table/projection design. You can use these recommendations to tune query performance.

Configuring the workload analyzer execution time

By default, Workload Analyzer runs each day at 2 AM. To optimize when Workload Analyzer uses resources, you can set Workload Analyzer to run at a different time for any or all databases that Management Console monitors. Alternately, you can set Management Console to never run Workload Analyzer automatically.

> **Note**
> Workload Analyzer automatically begins monitoring data one minute after the Management Console process starts. Workload Analyzer then runs once per day, or immediately after you import a database to Management Console. It continually gathers data in the background as long as the database is running. If you have not yet created a database, or if the database is down, Workload Analyzer does nothing until the database is back up.

1. On the Home page, click **MC Settings** .
2. Click the **Monitoring** tab.
3. Under the **Workload Analyzer Assistant** section of the Monitoring page, select your time zone.
4. Select the radio button for one of the options:

- **All Databases:** Select a time from the list. Workload Analyzer will run at that time on all databases that MC monitors.
- **Specific Database at Specific Time:** Select a database and a time from the list. At the time you specify, Workload Analyzer will run at that time on the database you selected.
- **Do Not Run Workload Analyzer On Any Database:** MC will never run Workload Analyzer automatically on any database it monitors.

5. Click **Apply** at the top right of the page.

For additional information about tuning recommendations and their triggering event, see Workload analyzer recommendations .

View workload analyzer recommendations

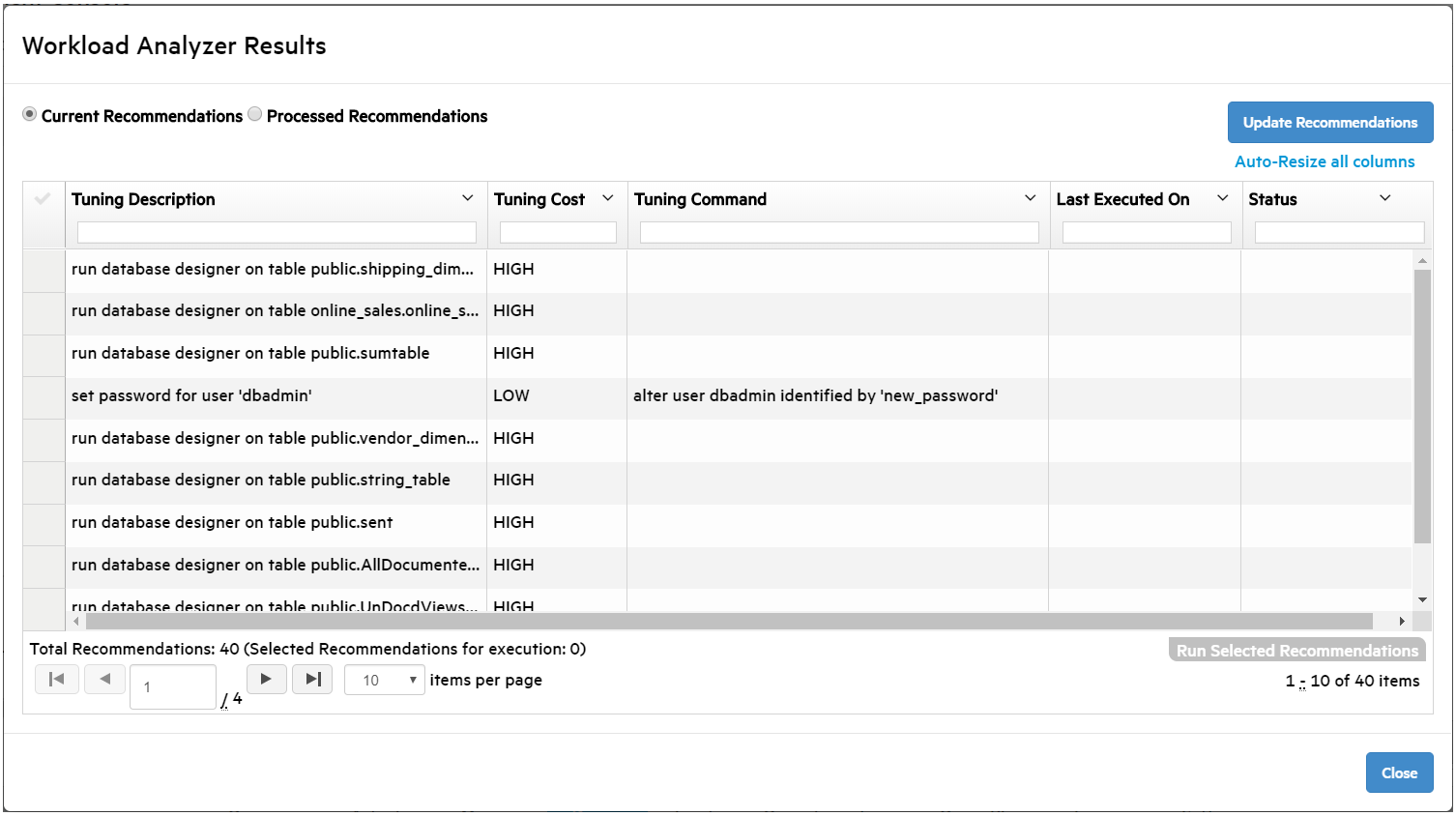
Workload Analyzer recommendations are available from the Quick Stats sidebar on the right of the database's **Overview** page. The Workload Analyzer module displays the number of tuning recommendations that the Workload Analyzer has generated.

To view the Workload Analyzer Results on the Database Designer page, click the number in the Workload Analyzer module.

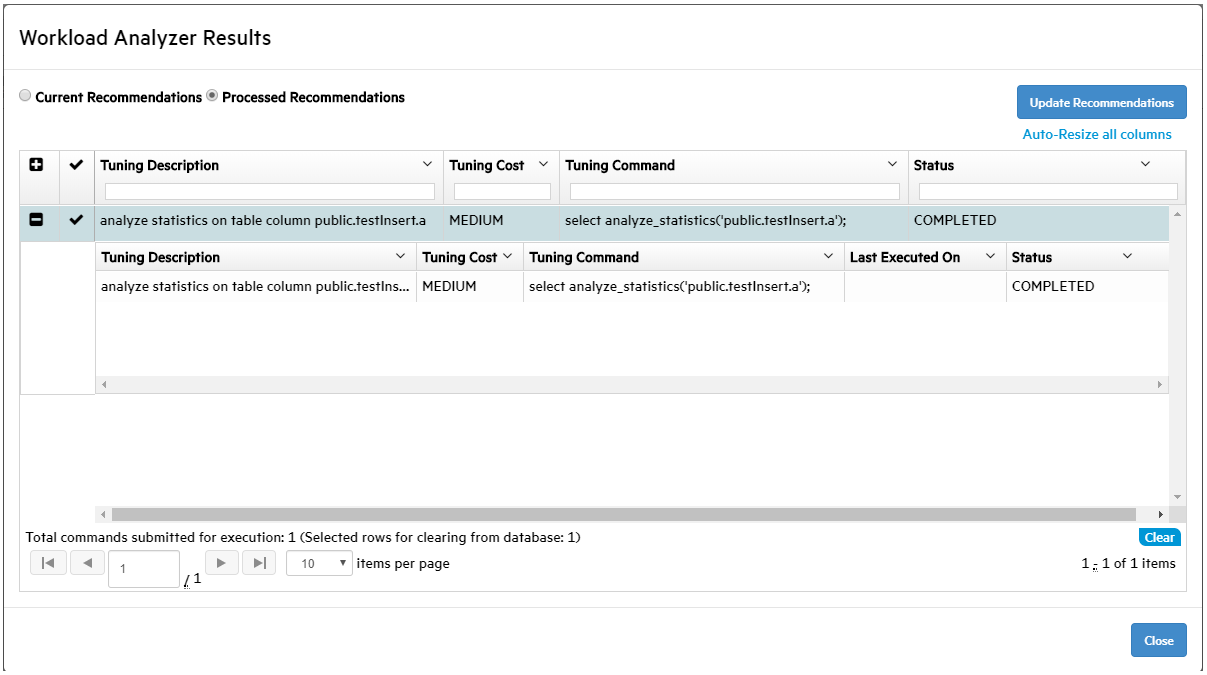


The Workload Analyzer Results window allows you to view details about and perform actions using current and processed recommendations.

Click the **Current Recommendations** radio button to display available Workload Analyzer recommendations. When ANALYZE_STATISTICS is returned as a tuning recommendation, select the check mark to the left of the row and click **Run Selected Recommendations** to execute the recommendation automatically.



Click the **Processed Recommendations** radio button to display the Workload Analyzer recommendations that you previously executed. To remove a recommendation from the list, click the check mark to the left of the row and click **Clear** , located above the **Close** button in the bottom-right of the window. To expand or hide the processed recommendation's execution history, click the plus or minus sign to the left of the row.



You can force the Workload Analyzer task to run immediately by clicking **Update Recommendations** , located above the **Status** column.

The total recommendations and the number of recommendations currently selected to run are displayed under the recommendations grid. Use the settings under the grid to view more recommendations per page or to cycle through the recommendations that do not fit on the page.

The following columns are used to describe recommendations:

- **Tuning Description** — Describes the Workload Analyzer recommendation.
- **Tuning Cost** — Resource cost of running each command (LOW, MEDIUM, or HIGH).

> **Tip**
> When the tuning cost is HIGH, consider running the recommended tuning during off-peak load times.

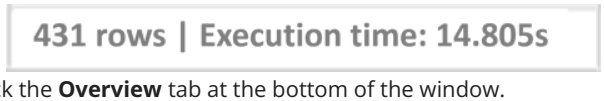
- **Tuning Command** — SQL command used to execute the recommendation.
- **Last Executed On** — Date that the recommendation was last run. In MM/DD/YYYY format.
- **Status** — Describes the execution stage of a tuning recommendation ran from Workload Analyzer Results.

For more information about tuning recommendations, see Getting tuning recommendations and ANALYZE_WORKLOAD .

Running workload analyzer recommendations to optimize a query

When the Workload Analyzer recommends that you run ANALYZE_STATISTICS to optimize a query, you can run the recommendation automatically from Workload Analyzer Results.

1. Record the data source and execution time for a query that is running sub-optimally.
   1. Click the **Query Execution** tab at the bottom.
   2. Use the Query Runner to execute a query that you want to optimize.
   3. Record the database table or tables in the query's FROM clause, and record the **Execution time** , located under the **Query Results** table.

2. Click the **Overview** tab at the bottom of the window.
3. On the **Overview** page, click the number in the **Workload Analyzer** box on the right.

   Workload Analyzer Results opens.
4. To filter the recommendations, enter the sub-optimal query's database table or tables in the field at the top of the **Tuning Description** column.

5. Select one or more ANALYZE_STATISTICS recommendations by clicking the check mark to the left of the row. To select all of the filtered ANALYZE_STATISTICS recommendations, click the check mark to the left of the **Tuning Description** column header.

Tuning Description	Tuning Cost	Tuning Command	Last Executed On	Status
public.testInsert				
run database designer on table public.testInsert	HIGH			
✔ analyze statistics on table column public.testInsert.a	MEDIUM	select analyze_statistics('public.testInsert.a');		

6. Click **Run Selected Recommendations** , located in the bottom-right of the window.
   This process might take several minutes.
7. After the tuning recommendations are completed, click the **Processed Recommendations** radio button at the top of the window.
   The previously executed recommendations are displayed.
8. Locate any recommendations that you just executed, and verify that the **Status** column says **COMPLETED** .
9. Verify that the query was optimized.
   1. Click the **Query Execution** tab at the bottom of the Management Console.
   2. Execute the query that was performing sub-optimally. Note the **Execution time** under the query results to verify the performance increase.



See also

Analyzing workloads Getting tuning recommendations

Running Database Designer using MC

You can use Database Designer to create a comprehensive design, which allows you to create new projections for all tables in your database.

Additionally, you can use Database Designer to create an incremental design. An incremental design creates projections for all tables referenced in the queries you supply.

To run Database Designer using MC, follow the steps listed at Running Database Designer with Management Console .

Using the Management Console to replace nodes

On the MC **Manage** page, you can quickly replace a DOWN node in the database by selecting one of the STANDBY nodes in the cluster.

A DOWN node shows up as a red node in the cluster. Click the DOWN node and the Replace node button in the Node List becomes activated, as long as there is at least one node in the cluster that is not participating in the database. The STANDBY node will be your replacement node for the node you want to retire; it will appear gray (empty) until it has been added to the database, when it turns green.

> **Tip**
> You can resize the Node List by clicking its margins and dragging to the size you want.

When you highlight a node and click **Replace** , MC provides a list of possible STANDBY nodes to use as a replacement. After you select the replacement node, the process begins. A node replacement could be a long-running task.

MC transitions the DOWN node to a STANDBY state, while the node you selected as the replacement will assume the identity of the original node, using the same node name, and will be started.

Assuming a successful startup, the new node will appear orange with a status of RECOVERING until the recovery procedure is complete. When the recovery process completes, the replacement node will turn green and show a state of UP.

Import and monitor a database in a Hadoop environment

You can use Management Console to connect to and monitor a Vertica database that resides in an Apache Hadoop environment. To monitor the database in the Hadoop environment, you must connect to an Apache Ambari server.

Prerequisites

Before you begin, you must:

- Install Vertica on a Hadoop cluster
- Install Apache Ambari version 1.6.1 or 2.1.0
- Enable Ganglia on your Hadoop cluster, to get the most information from your Hadoop environment

Importing Vertica within a Hadoop environment

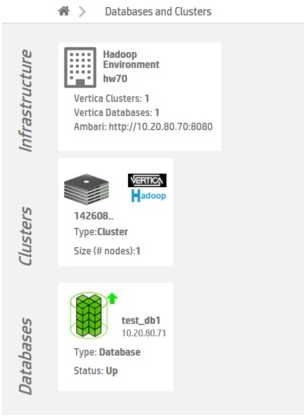
To import your Vertica database that resides in a Hadoop environment, connect to that Hadoop environment in Management Console through an Apache Ambari server.

1. From the Management Console home page, select **Additional import options** .
2. In **Provisioning** , select **Connect using an Ambari server to impot** Vertica **within a Hadoop environment** .
3. The **Provision Access Within Hadoop Environment** window provides the following options:
   - **Connect to a new Ambari server** : Choose this option to create a enter your username and password for a new Ambari server connection.
   - **Known Ambari URLs** : If you have a pre-existing Ambari connection that you want to use, select it from the drop-down list.
4. In the next window, select the Hadoop cluster with the Vertica database that you want to monitor.
   Management Console automatically discovers Hadoop clusters that are currently monitored by the Ambari server that you specify. If Management Console does not monitor Vertica clusters in the specified Hadoop environment, you can import clusters at this time.
   After you select the Hadoop cluster, you receive confirmation that your Hadoop cluster is saved.
5. Enter the IP address for the Vertica database you want to import and monitor. If Vertica is running on multiple hosts, enter the IP address of one of them.
6. Enter the API key for the Vertica cluster. The API key is generated during Vertica installation, and you can find it in the /opt/vertica/config/apikeys.dat file.
7. The next window displays the discovered databases. Select one or more databases you want to import, and enter the corresponding username and password.
8. If the import is successful, you receive a success message. Click **Done** to go to the **Existing Infrastructure** page.

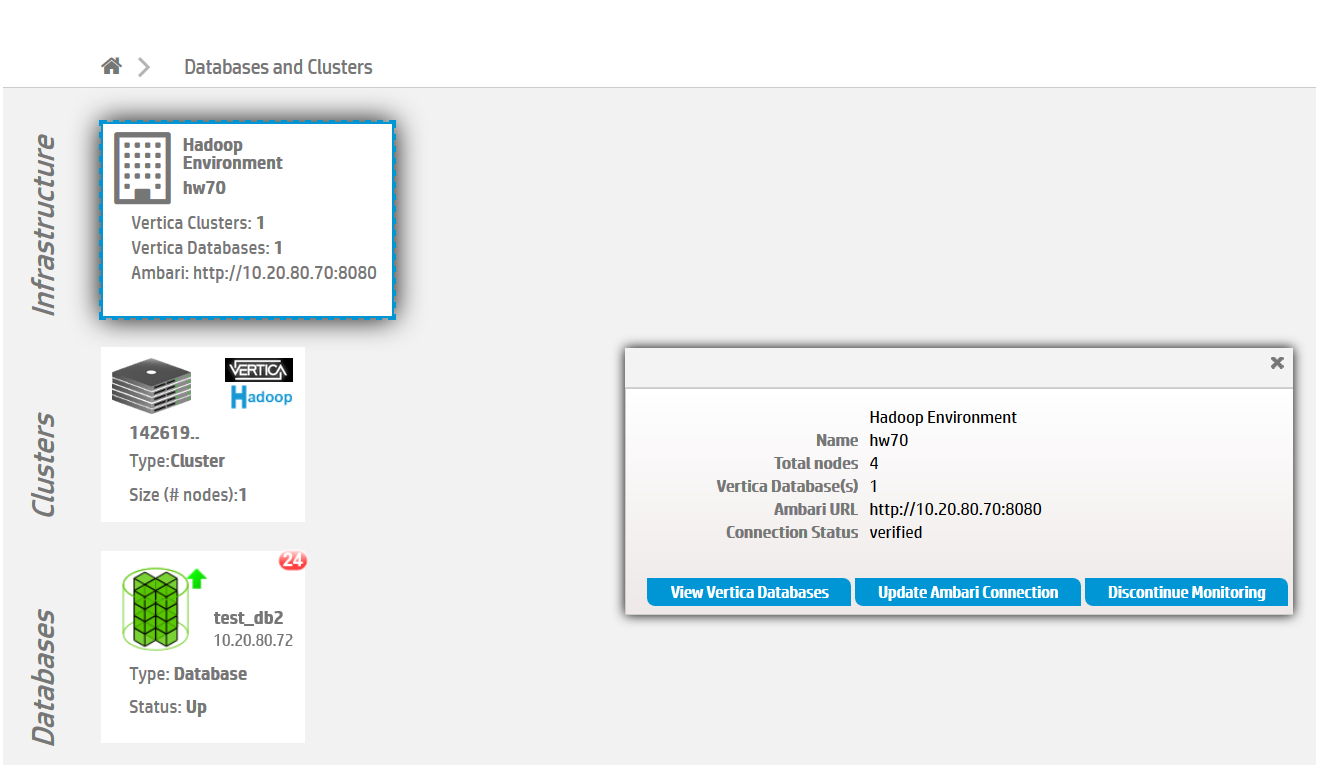
To import an additional Vertica cluster within a Hadoop environment, click **Import Cluster or database using IP address discovery** under **Provisioning** . Management Console will automatically associate the cluster with the existing Hadoop environment.

Monitoring Vertica within a Hadoop environment

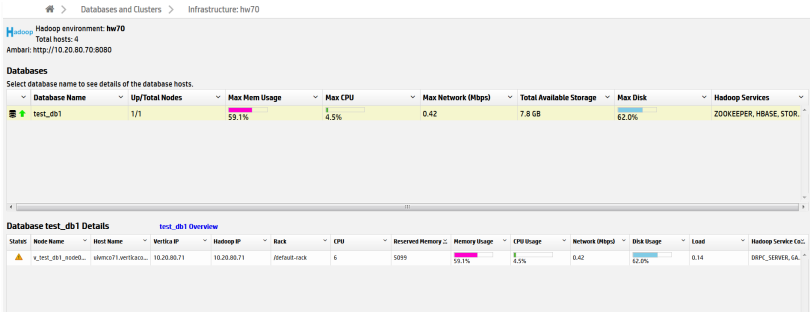
To monitor the Vertica clusters in a Hadoop environment, navigate to the **Existing Infrastructure** page:



Click to select the Hadoop environment, and then click **View Vertica Databases** .



The Management Console displays information about the Vertica databases that reside in a Hadoop environment:



You can monitor information like resource usage, Hadoop services, and database and connection status.

Update or remove an ambari connection

To update or remove an existing Ambari connection, go to the MC **Existing Infrastructure** page, and click on the relevant Hadoop environment.

To update a connection, click **Update Ambari Connection** . Step through the wizard to update the connection.

To remove a connection, select **Update Ambari Connection** and choose **Remove Connection** , or click **Discontinue Monitoring** and then confirm that you want to remove the connection. Removing the connection also removes all Vertica databases associated with this connection from MC monitoring. You can re-import the databases later if needed.

See also:

Apache Hadoop integration

Cloud platforms

Vertica supports Eon Mode database, subcluster, and node actions for the following cloud providers:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Private on-premises cloud storage

Management Console provides workflows that simplify resource provisioning and database management. Additionally, Vertica provides configuration defaults and recommendations for each cloud provider.

In this section

- Managing an Eon Mode database in the cloud
- Amazon Web Services in MC
- Microsoft Azure in MC

- Google Cloud Platform in MC

Managing an Eon Mode database in the cloud

After you provision your cluster and database in the cloud, the screens and techniques you use to monitor the database are the same regardless of the database mode or cluster platform. (Exceptions are noted in the documentation for particular features.)

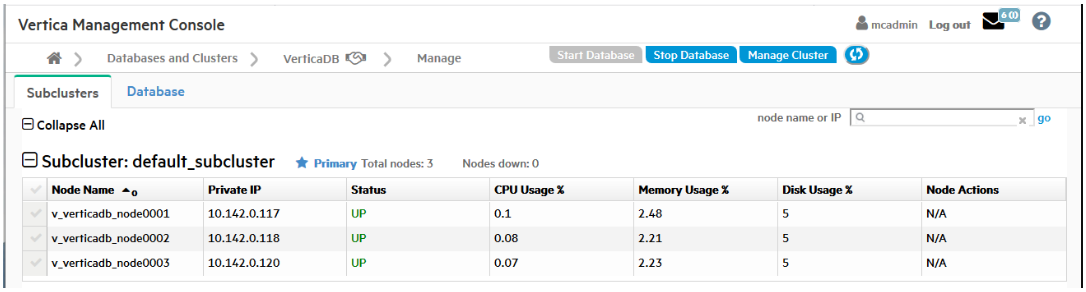
Using MC to monitor your cluster, you can monitor the nodes in your subcluster, load data, run queries, and perform all other monitoring tasks for subclusters and nodes.

For details about Vertica and the supported cloud providers, see Set up Vertica on the cloud .

Monitoring primary and secondary subclusters

The **Cluster > Manage** page **Database** tab lets you monitor your database nodes in visual format. MC supports only the monitoring features on the Database tab, as described in Viewing and managing your cluster .

You can use the **Manage > Subclusters** tab to monitor your subcluster and nodes:



The **Subclusters** tab displays the following subcluster information:

- Subcluster name
- Whether it is primary or secondary
- Total number of nodes
- Number of down nodes in that subcluster

In addition, the tab displays the following node information:

- Node name
- Private IP address
- UP or DOWN status
- CPU
- Memory
- Disk usage percentages
- Available node actions

Searching for nodes

On the **Manage > Subclusters** tab in MC, you can search for a specific node or group of nodes.

In the node name or IP field above the subcluster:

- To find a single node, enter a complete node name or IP address.
- To find a related group of nodes, enter a partial node name or IP address that those nodes share.

Amazon Web Services in MC

Management Console provides specific resources for managing database clusters on AWS.

You can provision an Eon Mode or Enterprise Mode database cluster on AWS.

You can revive an Eon Mode database cluster on AWS. For more information, see Reviving an Eon Mode database on AWS in MC .

The MC provision and revive wizards for AWS configure separate volumes for the data, depot, catalog, and temp database directories. The specific volumes it uses for each directory depend upon the mode and the specific AWS instance type you select when you provision or revive the cluster. For details on the volumes configured for clusters on AWS, see:

- Eon Mode volume configuration defaults for AWS

- Enterprise Mode volume configuration defaults for AWS

In this section

- Creating an Eon Mode database in AWS with MC
- Creating an Enterprise Mode database in AWS with MC
- Reviving an Eon Mode database on AWS in MC
- Eon Mode volume configuration defaults for AWS
- Enterprise Mode volume configuration defaults for AWS
- Add nodes to a cluster in AWS using Management Console
- Loading data from Amazon S3 using MC

Creating an Eon Mode database in AWS with MC

After you deploy Management Console on Amazon Web Services (AWS) with a CloudFormation Template , you can provision a cluster and database. Vertica clusters are provisioned in the same Virtual Private Cloud (VPC) as the Vertica Management Console. You can create an initial cluster of up to 60 hosts.

> **Note**
> See Creating a cluster using MC if you installed Management Console with an RPM, or to provision an on-premises database and cluster.

Prerequisites

Before you begin, complete or obtain the following:

- Deploy MC and AWS resources with a CloudFormation template
- AWS credentials and environment details
- S3 communal storage URL
- Vertica credentials
- Vertica Management Console Credentials

Choose one of the following setup options:

- Quick Setup : Select your cluster size based on your estimated compressed working data size . The Management Console calculates the volume size per node, and reserves part of the disk for catalog and temp storage.
- Advanced Setup : This option provides more granular control over configuration settings related to subnet, Node IP, and depot, temp, and catalog volume sizes.

Quick setup

1. Log in to the Vertica Management Console.
2. On the Management Console home page under **Provision** , click **Create new database** .
3. On **Database Storage Mode** , click Eon Mode.
4. Click **Next** . On Vertica **License** , select one of the following license mode options:
   - **Community Edition** : A free Vertica license to preview Vertica functionality. This license provides limited features. If you use a Community Edition license for your deployment, you can upgrade the license later to expand your cluster load. See Managing licenses form more information.
   - **Premium Edition** : Use your Vertica license. After you select this option, click **Browse** to locate and upload your Vertica license key file, or manually enter it in the field.

1. Click **Next** . On **Setup Path** , select **Quick Setup** .
2. Click **Next** . In the Vertica **Settings** section, select the desired Vertica database version. You can select from the latest hotfix of recent Vertica releases. For each database version, you can also select the operating system. See Choose a Vertica AMI Operating Systems for available OS and major version options.
3. In the **AWS EC2 Instance Type** section, select from one of the following instance types:
   - Ephemeral Depot
   - EBS Depot
4. In the **Cluster Size** section, select the number of instances to deploy with your cluster based on your working data size. For details about working data size, see Configuring your Vertica cluster for Eon Mode .

> **Note**
> If you are using a Community Edition license, your cluster size selections are limited to **Small** , **Medium** , and **Large** .

5. In the **Database Parameters** section, supply the following information:
   - **Database Name** : The name for your new database. See Creating a database name and password for database name requirements.
   - **Administrator Username** : The name of the database superuser .
   - **Administrator Password** : The password for the database administrator user account. For details, see Password guidelines .
   - **Confirm Password** : Reenter the **Administrator Password** .
   - **Load Sample Data** : Optional. Click the slider to the right to preload your database with example clickstream data. This option is useful if you are testing features and want some preloaded data in the database to query.
6. In the **AWS Environment** section, supply the following information:
   - **AWS Key Pair** : Your Amazon key pair for SSH access to EC2 instances.
   - **IP Access** : The cluster IP address range for SSH and client access to cluster hosts.
   - **S3 Communal Storage URL** : The path to a new subfolder in your existing AWS S3 bucket for Communal Storage of your Eon Mode database. Vertica creates the subfolder in the existing S3 bucket.

     > **Note**
     > Use an existing S3 bucket in the same region as your Management Console instance.

   - **Tag EC2 Instances** : Optional. Assign distinct, searchable metadata tags to the instances in this cluster. Many organizations use labels to organize, track responsibility, and assign costs for instances.
     To add a tag, click the slider to the right to display the **Tag Name** and **Tag Value** fields. Click **Add** to create the tag. Added tags are displayed below the fields.
7. Click **Create Database Cluster** to create a Eon Mode cluster on AWS.

Advanced setup

1. Log in to the Vertica Management Console.
2. On the Management Console home page under **Provision** , click **Create new database** .
3. On **Database Storage Mode** , click Eon Mode.
4. Click **Next** . On Vertica **License** , select one of the following license mode options:
   - **Community Edition** : A free Vertica license to preview Vertica functionality. This license provides limited features. If you use a Community Edition license for your deployment, you can upgrade the license later to expand your cluster load. See Managing licenses form more information.
   - **Premium Edition** : Use your Vertica license. After you select this option, click **Browse** to locate and upload your Vertica license key file, or manually enter it in the field.

1. Click **Next** . On **Setup Path** , select **Advanced Setup** .
2. Click **Next** . On **AWS Environment** , supply the following information:
   - **AWS Key Pair** : Your Amazon key pair for SSH access to EC2 instances.
   - **AWS Subnet** : The subnet for your cluster. By default, Vertica creates your cluster in the same subnet as your MC instance.
     Important
     Use security groups and network access control lists (ACLs) to secure your subnet. For details, see the Amazon documentation .
   - **IP Access** : The cluster IP address range for SSH and client access to cluster hosts.
   - **Node IP Setting** : Select **Private** , **Public** , or **Elastic** . For details about each option, see the Amazon documentation .
   - **S3 Communal Storage URL** : The path to a new subfolder in your existing AWS S3 bucket for Communal Storage of your Eon Mode database. Vertica creates the subfolder in the existing S3 bucket.

     > **Note**
     > Use an existing S3 bucket in the same region as your Management Console instance.

   - **Tag EC2 Instances** : Optional. Assign distinct, searchable metadata tags to the instances in this cluster. Many organizations use labels to organize, track responsibility, and assign costs for instances.
     To add a tag, click the slider to the right to display the **Tag Name** and **Tag Value** fields. Click **Add** to create the tag. Added tags are displayed below the fields.
3. Click **Next** . **Database Parameters** accepts information about your Vertica license. Supply the following information:
   - **Database Name** : The name for your new database. See Creating a database name and password for database name requirements.
   - **Administrator Username** : The name of the database superuser .
   - **Administrator Password** : The password for the database administrator user account. For details, see Password guidelines .
   - **Confirm Password** : Reenter the **Administrator Password** .

- **Vertica Version** : Select the desired Vertica database version. You can select from the latest hotfix of recent Vertica releases. For each database version, you can also select the operating system. See Choose a Vertica AMI Operating Systems for available OS and major version options.
- **Load Sample Data** : Optional. Click the slider to the right to preload your database with example clickstream data. This option is useful if you are testing features and want some preloaded data in the database to query.

4. Click **Next** . On **AWS Configuration** , supply the following information:
   - **Number of Nodes** : The initial number of nodes for your database.
   - **Number of** Vertica **Database Shards** : Sets the number of shards in your database. Vertica suggests a number of shards automatically, based on your node count. After you set this value, you cannot change it later. The shard count must be greater than or equal to the maximum subcluster count. Be sure to allow for node growth. See Configuring your Vertica cluster for Eon Mode for recommendations.
   - **EC2 Instance Type** : The instance types used for the nodes. See Choosing AWS Eon Mode Instance Types for a list of recommended AWS instances. For details about each instance type, see the Amazon EC2 Instance Types documentation.
   - **Local Storage** : Customize your cluster according to your storage needs. For guidance, see Eon Mode volume configuration defaults for AWS for the Vertica default settings for each supported instance.
5. Click **Next** . On **Review** , confirm your selections. Click **Edit** to return to a previous section and make changes.
6. When you are satisfied with your selections, click the **I accept the terms and conditions** checkbox.
7. Click **Create Cluster** to create a Eon Mode cluster on AWS.

After creating the database

After you create the database, click **Get Started** to view the Fast Tasks page. To view your database, select **Manage and View Your Vertica Database** to go to the database **Overview** .

You can also view your database from the **Recent Databases** section of the MC home page.

For additionalinformation about managing your cluster, instances, and database using Management Console, see Managing database clusters .

Creating an Enterprise Mode database in AWS with MC

After you deploy Management Console on Amazon Web Services (AWS) with a CloudFormation Template , you can provision a cluster and database. Vertica clusters are provisioned in the same Virtual Private Cloud (VPC) as the Vertica Management Console. You can create an initial cluster of up to 60 hosts.

> **Note**
> See Creating a cluster using MC if you installed Management Console with an RPM, or to provision an on-premises database and cluster.

Prerequisites

Before you begin, complete or obtain the following:

- Deploy MC and AWS resources with a CloudFormation template
- AWS credentials and environment details
- S3 communal storage URL
- Vertica credentials
- Vertica Management Console Credentials

Provisioning a cluster and database

1. Log in to the Vertica Management Console.
2. On the Management Console home page under **Provision** , click **Create New Database** .
3. On **Database Storage Mode** , click Enterprise Mode.
4. Click **Next** . **Create a New** Vertica **Cluster | mode: Enterprise** provides you with two workflow options for creating your database. Select one of the following:
   - **Quick Create** : Vertica configures your EC2 instances with default settings.
   - **Custom Create** : You can specify EC2 instance types and other database settings.
5. Click **Next** . On **Enter AWS Credentials and preferences** , **AWS Region** is filled with the region of the Management Console host. Supply the following information:
   - **AWS Subnet** : Under **Show Advanced Options** . Select the subnet used to create your cluster.
   - **AWS Access Key ID** : Displayed if MC was configured to use the AWS Access Keys authentication method. Enter your access key.
   - **AWS Secret Access Key** : Displayed if MC was configured to use the AWS Access Keys authentication method. Enter the password associated with the **AWS Access Key ID** .
   - **AWS Key Pair** : Your Amazon key pair for SSH access to EC2 instances.
   - **CIDR Range** : The cluster IP address range for SSH and client access to cluster hosts.

6. Click **Next** . **Enter** Vertica **database name and login credentials** accepts information about your Vertica license. Supply the following information:
   - Vertica **Database Name** : The name for your new database. See Creating a database name and password for database name requirements.
   - Vertica **Version** : **Custom Create** mode only. Select the desired Vertica database version. You can select from the latest hotfix of recent Vertica releases. For each database version, you can also select the operating system. See Choose a Vertica AMI Operating Systems for available OS and major version options.
   - Vertica **Database User Name** : The name of the database superuser .
   - **Password** : The password associated with the database username. For details, see Password guidelines .
   - **Confirm Password** : Reenter the **Password** .
   - **Database Node Count** : The number of nodes that you want to deploy in this cluster. **Quick Create** mode provides options for 1 or 3 database node counts.
   - **Vertica License** : **Custom Create** mode only. Click **Browse** to locate and upload your Vertica license key file. If you do not supply a license key file here, the wizard deploys your database with a Vertica Community Edition license. This license has a three node limit, so the value in the Database Size filed cannot be larger than 3 if you do not supply a license. If you use a Community Edition license for your deployment, you can upgrade the license later to expand your cluster load more than 1TB of data. See Managing licenses form more information.
   - **Load example test data** : Optional. Click the checkbox to preload your database with example clickstream data. This option is useful if you are testing features and want some preloaded data in the database to query.
7. Click **Next** . **Specify cloud instance and main data storage info** provides options to let you customize your instance configuration. In **Quick Create** mode, the options on this screen are pre-selected and read-only.
   Database Data Path is filled with the path to your persistent database storage.
   EBS Volume Type and **EBS Volume Size (GB) per Volume per Available Node** fields are filled with default values for the selected **EC2 Instance Type** . See Eon Mode volume configuration defaults for AWS for more information.
   In **Custom Create** mode, supply information for the following:
   - **EC2 Instance Type** : The instance type your cluster deploys. See Supported AWS instance types for more information.
   - **EBS Volume Type** : The block-level storage type for each node in your cluster. See Enterprise Mode volume configuration defaults for AWS for recommendations about supported volume types.
   - **EBS Volume Size (GB) per Volume per Available Node** : The amount of disk space available on each disk attached to each node in your cluster. This field shows you the total disk space available per node in your cluster.
   - **Enable EBS Volume Encryption** : Optional. Select the checkbox if you want server-level encryption on your EC2 instances. With AWS, only 4th and 5th generation instance types (c4/5, r4/5, and m4/5) support encryption.
   - **Node IP setting** : Select **Private** , **Public** , or **Elastic** . For details about each option, see the Amazon documentation .
8. Click **Next** . **Specify additional storage and tag info** lets you allocate additional storage for your cluster. In **Quick Create** mode, the options on this screen are pre-selected and read-only.
   - **Database Catalog Path** is the location of the local copy of the database catalog. **Database Temp Path** is the temporary storage space for each node, if the node instance type includes the temporary storage option.
   - In **Custom Create** mode, select or enter a value for **EBS Volume Type** , **EBS Volume Size (GB) per Volume per Available Node** , and **Enable EBS Volume Encryption** under each path. Each field has the same definition as described in the previous step.
   - **Tag EC2 instances** : Optional. Assign distinct, searchable metadata tags to the instances in this cluster. Many organizations use labels to organize, track responsibility, and assign costs for instances.
     After you click the checkbox, the **Tag Name** and **Tag Value** fields are displayed. Click **Add** to create the tag. Added tags are displayed below the fields.
9. Click **Next** . On the **Review** screen, confirm your selections. To edit your selections, click **Back** until you reach the screen containing information that you want to edit.
10. When you are satisfied with your selections, click the **Accept terms and conditions of the "Software Only Terms" for your territory** checkbox.
11. Click **Create** to create an Enterprise Mode cluster on AWS.

After the cluster and database is successfully created, click **Get Started** to view the Fast Tasks page. To view your database, select **Manage and View Your Vertica Database** to go to the database **Overview** .

You can also view your database from the **Recent Databases** section of the MC home page.

See Managing database clusters for further managing your cluster, instances, and database using Management Console.

Reviving an Eon Mode database on AWS in MC

Important
You can also revive a database using admintools . You must use admintools in order to revive a database on an existing cluster. For example, use admintools if you stopped a cluster whose hosts use instance storage where data is not persistently stored, and plan to bring back the database on the same cluster.

An Eon Mode database keeps an up-to-date version of its data and metadata in its communal storage location. After a cluster hosting an Eon Mode database is terminated, this data and metadata continue to reside in communal storage. When you revive the database later, Vertica uses the data in

this location to restore the database in the same state on a newly provisioned cluster.
If Management Console has been installed using a CloudFormation template from the AWS Marketplace, you can use the Provision and Revive wizard in Management Console.

During a revive of your database, when you select a Vertica version that is higher than the version of the original database in the communal storage, Vertica upgrades your database to match the Vertica version you selected. This upgrade may cause the database revive to take longer. To bypass this upgrade, select the Vertica version of your original database.

> **Note**
> After upgrading a Vertica database, you can not revert to an earlier version.

Prerequisites

- Communal storage location (an AWS S3 bucket) of the stopped Eon Mode database you plan to revive. For guidance, see Viewing and managing your cluster .
- Username and password of the Eon Mode database you plan to revive.
- AWS account with permissions to create a VPC, subnet, security group, instances, and roles.
- Amazon key pair for SSH access to an instance.

Revive the database on the cloud

You use a wizard in Management Console to provision a new cluster on AWS and revive a database onto it. For the new cluster, Management Console automatically provisions the same number of AWS instances used by the database when it was last shut down.

> **Caution**
> You should not use the same communal storage location for running multiple databases, as it causes data corruption. To avoid corruption, you should also never use the revive functionality to simultaneously run the same Eon Mode database on different clusters.

1. From the Management Console home page, click **Provision and Revive an Eon Mode Database** . The Provision and Revive an Eon Mode Database wizard opens.
2. Enter your cloud credentials and cluster preferences. Your cluster must be in the same region as your communal storage location's S3 bucket. To revive the cluster in a new region, you must:
   - Create an S3 bucket in the new region.
   - Copy the previous S3 bucket's contents into it.
   - Provide the new S3 bucket URL in Step 3.
3. By default, Vertica creates your cluster in the same subnet as your Management Console instance. If you want to manage all Vertica clusters in the same VPC, you can provision your Vertica database in a different subnet than the Management Console instance. To do so, on the **AWS Credentials** page, select **Show Advanced Options** and enter a value in the **Subnet** field.
   Important
   If you specify a different subnet for your database, make sure to secure the subnet.
4. Enter the S3 URL of the database you are reviving. When you enter an S3 bucket location, Management Console discovers all known Eon Mode databases.
5. Select the correct database to revive.
6. Provide the database administrator credentials for the database you are reviving. These credentials are the same as the ones used by the database in the previous cluster.
7. In the **Database Version** field, choose the desired Vertica database version. Select from the latest hotfix of recent Vertica releases. For each Vertica version, you can select from a list of associated Linux operating systems.
   If you select a Vertica version that is higher than the version of the original database in the communal storage, Vertica upgrades your database to match the Vertica version you selected. This upgrade may cause the database revive to take longer. To bypass this upgrade, select the Vertica version of your original database.

   > **Note**
   > After your Vertica database has been upgraded, you can not downgrade your database later.

8. Choose instance types for the cluster. Management Console provisions the same number of instances used by the database when it was last shut down.
   The MC populates the existing paths for the depot, catalog, and temp directories.

Note
You cannot persist the catalog with ephemeral instances.

The last step displays a confirmation page showing the configured volumes. For details on the volume configurations that MC provides, see Eon Mode volume configuration defaults for AWS and Enterprise Mode volume configuration defaults for AWS .

Caution
To ensure against data loss, choose instances that use EBS storage. EBS storage is durable, while instance store is temporary. For Eon mode, Management Console displays an alert to inform the user of the potential data loss when terminating instances that support instance store.

9. Choose whether to encrypt your EBS volumes. With AWS, only 4th and 5th generation instance types (c4, r4, and m4; c5, r5, and m5) support encrypting EBS volumes.
10. Optionally, you can tag the instances. In the **Tag EC2 instances** field, if another cluster is already running, Management Console fills those fields with the tag values for the first instance in the cluster. You can accept the defaults, or enter new tag values.
11. Review your choices, accept the license agreement, and click **Create** to revive the database on a new cluster. If the version of Management Console you use to revive is higher than that of the database, Management Console first notifies you that it is about to automatically upgrade the database. After starting the revive process, the wizard displays its progress. After a successful revive, the database starts automatically.

Important
If the database fails to start automatically, check the controlmode setting in /opt/vertica/config/admintools.conf . This setting must be compatible with the network messaging requirements of your Eon implementation. AWS relies on unicast messaging, which is compatible with a controlmode setting of point-to-point (pt2pt). If the source database controlmode setting was broacast and you migrate to S3/AWS communal storage, you must change controlmode with admintools:

```
$ admintools -t re_ip -d dbname -T
```

When the revive process is complete, click **Get Started** to navigate to the Fast Tasks page.

Eon Mode volume configuration defaults for AWS

When you provision or revive an Eon Mode database cluster, Management Console configures separate volumes for the depot, catalog, and temp directories. The specific volumes and sizes that Management Console configures vary depending on the AWS instance type you select when provisioning or reviving the cluster.

MC follows these rules when allocating resources for these directories for an Eon Mode database cluster:

- Depot: Allocate instance store if available with the selected instance type. Otherwise, allocate EBS volumes. (In Eon Mode on AWS, S3 is the backup.)
- Catalog: Always allocate an EBS volume, to ensure the catalog is durable.
- Temp: Allocate instance store if available with the selected instance type. Otherwise, allocate EBS volumes.

Note
Table below includes all instances types that include instance store.

Instance Type API Name	Instance Storage	Depot	Catalog	Temp
Supported EBS Volumes	N/A	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Configurable 1 EBS volume Default to 100G

c3.4xlarge	320 G (2 * 160 GiB SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance store 2x160G
c3.8xlarge	640 G (2 * 320 GiB SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance store 2x320G
c5d.4xlarge	4000 G (1 * 400 GiB SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance store 1x400G
i3.4xlarge	3800 G (2 * 1900 GiB NVMe SSD)	Instance store 1x1900G	Configurable 1 EBS volume Default to 50G	Instance store 1x1900G
i3.8xlarge	7600 G (4 * 1900 GiB NVMe SSD)	Instance store 3x1900G	Configurable 1 EBS volume Default to 50G	Instance store 1x1900G
i3.16xlarge	15200 G (8 * 1900 GiB NVMe SSD)	Instance store 6x1900G	Configurable 1 EBS volume Default to 50G	Instance store 2x1900G
i3en.3xlarge	7500 G (1 * 7500 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 1x7500G
i3en.6xlarge	15000 G (2 * 7500 GiB NVMe SSD)	Instance store 1x7500G	Configurable 1 EBS volume Default to 50G	Instance Store 1x7500G
i3en.12xlarge	30000 G (4 * 7500 GiB NVMe SSD)	Instance store 3x7500G	Configurable 1 EBS volume Default to 50G	Instance Store 1x7500G
i4i.4xlarge	3750 G (1 * 3750 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance store 1x3750G

i4i.8xlarge	7500 G (2 * 3750 GiB NVMe SSD)	Instance Store 1x3750G	Configurable 1 EBS volume Default to 50G	Instance store 1x3750G
i4i.16xlarge	15000 G (4 * 3750 GiB NVMe SSD)	Instance store 3x3750G	Configurable 1 EBS volume Default to 50G	Instance store 1x3750G
m5d.4xlarge	600 G (2 * 300 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance store 2x300G
m5d.8xlarge	1200 G (2 * 600 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance store 2x600G
m5d.12xlarge	1800 G (2 * 900 GiB NVMe SSD)	Instance store 1x900G	Configurable 1 EBS volume Default to 50G	Instance store 1x900G
m5d.16xlarge	2400 G (4 * 600 GiB NVMe SSD)	Instance store 3x600G	Configurable 1 EBS volume Default to 50G	Instance store 1x600G
r3.4xlarge	32G (1 * 320G SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance store 320G
r3.8xlarge	640 G (2 * 320 GiB SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance store 2x320G
r5d.4xlarge	600 G (2 * 300 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance store 2x300G
r5d.8xlarge	1200 G (2 * 600 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance store 2x600G

r5d.12xlarge	1800 G (2 * 900 GiB NVMe SSD)	Instance store 1x900G	Configurable 1 EBS volume Default to 50G	Instance store 1x900G
r5d.16xlarge	2400 G (4 * 600 GiB NVMe SSD)	Instance store 3x600G	Configurable 1 EBS volume Default to 50G	Instance store 1x600G

Supported EBS volumes

- c4.4xlarge
- c4.8xlarge
- c5.4xlarge
- c5.9xlarge
- c6i.4xlarge
- c6i.8xlarge
- c6i.12xlarge
- c6i.16xlarge
- c6i.24xlarge
- c6i.32xlarge
- m4.4xlarge
- m4.10xlarge
- m5.4xlarge
- m5.8xlarge
- m5.12xlarge
- r4.4xlarge
- r4.8xlarge
- r4.16xlarge
- r5.4xlarge
- r5.8xlarge
- r5.12xlarge
- r6i.4xlarge
- r6i.8xlarge
- r6i.12xlarge
- r6i.16xlarge
- r6i.24xlarge
- r6i.32xlarge

Enterprise Mode volume configuration defaults for AWS

When you provision an Enterprise Mode database cluster, Management Console configures separate volumes for the data, catalog, and temp directories.

The specific volumes and sizes that MC uses vary depending on the AWS instance type you select when provisioning the cluster.

MC follows these rules when selecting resources for these directories for an Enterprise Mode database cluster:

- Data: Always use EBS volumes, to ensure the data is durable.
- Catalog: Always use an EBS volume, to ensure the catalog is durable.
- Temp: If the chosen instance type offers instance store, then use volumes from instance store.

> **Note**
> Table below includes all instance types that include instance store.

Instance Type API Name	Instance Storage	Data	Catalog	Temp

Supported EBS Volumes	N/A	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Configurable 1 EBS volume Default to 100G
c3.4xlarge	320 G (2 * 160 GiB SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x160G
c3.8xlarge	640 G (2 * 320 GiB SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x320G
c5d.4xlarge	400 G (1 * 400 GiB SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 1x400G
i3.4xlarge	3800 G (2 * 1900 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x1900G
i3.8xlarge	7600 G (4 * 1900 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 4x1900G
i3.16xlarge	15200 G (8 * 1900 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 8x1900G
i3en.3xlarge	7500 G (1 * 7500 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 1x7500G
i3en.6xlarge	15000 G (2 * 7500 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x7500G

i3en.12xlarge	30000 G (4 * 7500 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 4x7500G
i4i.4xlarge	3750 G (1 * 3750 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 1x3750G
i4i.8xlarge	7500 G (2 * 3750 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x3750G
i4i.16xlarge	15000 G (4 * 3750 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 4x3750G
r3.4xlarge	320 G (1 * 320 GiB SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 320G
m5d.4xlarge	600 G (2 * 300 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x300G
m5d.8xlarge	1200 G (2 * 600 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x600G
m5d.12xlarge	1800 G (2 * 900 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x900G
m5d.16xlarge	2400 G (4 * 600 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 4x600G

r3.8xlarge	640 G (2 * 320 GiB SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x320G
r5d.4xlarge	600 G (2 * 300 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x300G
r5d.8xlarge	1200 G (2 * 600 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x600G
r5d.12xlarge	1800 G (2 * 900 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 2x900G
r5d.16xlarge	2400 G (4 * 600 GiB NVMe SSD)	Configurable 8 EBS volumes Default to 75G per volume	Configurable 1 EBS volume Default to 50G	Instance Store 4x600G

Supported EBS volumes

- c4.4xlarge
- c4.8xlarge
- c5.4xlarge
- c5.9xlarge
- c6i.4xlarge
- c6i.8xlarge
- c6i.12xlarge
- c6i.16xlarge
- c6i.24xlarge
- c6i.32xlarge
- m4.4xlarge
- m4.10xlarge
- m5.4xlarge
- m5.8xlarge
- m5.12xlarge
- r4.4xlarge
- r4.8xlarge
- r4.16xlarge
- r5.4xlarge
- r5.8xlarge
- r5.12xlarge
- r6i.4xlarge
- r6i.8xlarge
- r6i.12xlarge
- r6i.16xlarge
- r6i.24xlarge
- r6i.32xlarge

Add nodes to a cluster in AWS using Management Console

When you use MC to add nodes to a cluster in the cloud, MC provisions the instances, adds the new instances to the existing Vertica cluster, and then adds those hosts to the database.

In the Vertica Management Console, you can add nodes in several ways, depending on your database mode.

For Eon Mode databases, MC supports actions for subcluster and node management for the following public and private cloud providers:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Pure Storage FlashBlade

> **Note**
> Enterprise Mode does not support subclusters.

For Enterprise Mode databases, MC supports these actions:

- In the cloud on AWS: Add Node action, Add Instance action.
- On-premises: Add Node action.

> **Note**
> In the cloud on GCP, Enterprise Mode databases are not supported.

Adding nodes in an Eon Mode database

In an Eon Mode database, every node must belong to a subcluster. To add nodes, you always add them to one of the subclusters in the database:

- By scaling up an existing subcluster by one or more nodes.
- By adding a new subcluster of one or more nodes.

Adding nodes in an Enterprise Mode database on AWS

In an Enterprise Mode database on AWS, to add an instance to your cluster:

1. On the MC Home page, click **View Infrastructure** to go to the Infrastructure page. This page lists all the clusters the MC is monitoring.
2. Click any cluster shown on the Infrastructure page.
3. Select **View** or **Manage** from the dialog that displays, to view its Cluster page. (In a cloud environment, if MC was deployed from a cloud template the button says "Manage". Otherwise, the button says "View".)

   > **Note**
   > You can click the pencil icon beside the cluster name to rename the cluster. Enter a name that is unique within MC.



1. Click the Add (**+**) icon on the **Instance List** on the **Cluster Management** page.

MC adds a node to the selected cluster.

Loading data from Amazon S3 using MC

You can use the Data Load Activity page in Management Console to import data from Amazon S3 storage to an existing Vertica table. When you run a load job, Vertica appends rows to the target table you provide. If the job fails, or you cancel the job, Vertica commits no rows to the target table.

When you view your load history on the Instance tab, loading jobs initiated in MC using Amazon S3 have the name MC_S3_Load in the Stream Name column.

Prerequisites

To use the MC Load feature, you must have:

- Access to an Amazon S3 storage account.
- An existing table in your Vertica database to which you can copy your data. You must be the owner of the table.
- (For non-CloudFormation Template installs) An S3 gateway endpoint.

If you aren't using a CloudFormation Template (CFT) to install Vertica, you must create an S3 gateway endpoint in your VPC. For more information, see the AWS documentation .

For example, the Vertica CFT has the following VPC endpoint:

```
"S3Enpoint" : {
   "Type" : "AWS::EC2::VPCEndpoint",
   "Properties" : {
   "PolicyDocument" : {
      "Version":"2012-10-17",
      "Statement":[{
      "Effect":"Allow",
      "Principal": "*",
      "Action":["*"],
      "Resource":["*"]
      }]
   },
   "RouteTableIds" : [ {"Ref" : "RouteTable"} ],
   "ServiceName" : { "Fn::Join": [ "", [ "com.amazonaws.", { "Ref": "AWS::Region" }, ".s3" ] ] },
   "VpcId" : {"Ref" : "VPC"}
}
```

Create a loading job

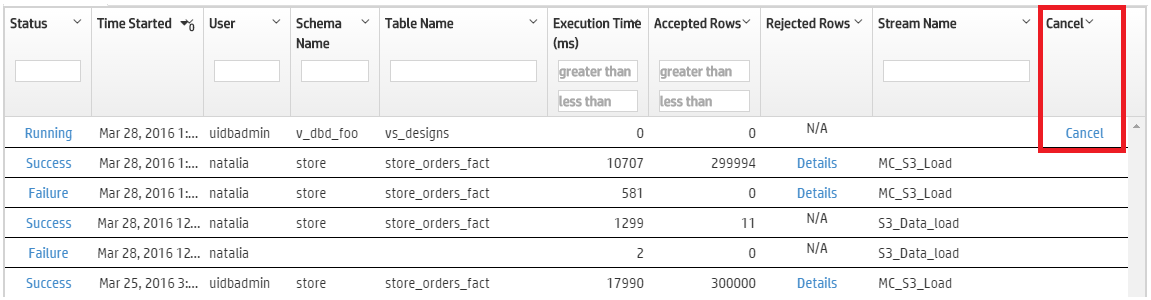
To load data from an Amazon S3 bucket to an existing table in your target database:

1. On the target database MC dashboard, click on the Load tab at the bottom of the page to view the Data Load Activity page.
2. Click on the Instance tab.
3. Click New S3 Data Load at the top-right of the tab. The Create New Amazon S3 Loading Job dialog box opens.
4. Enter your AWS account credentials and your target location information in the required fields, which are indicated by asterisks (*). Use the format S3:// for the bucket name.
5. (Optional) Specify additional options by completing the following fields:
   - COPY Parameters
   - Capture rejected data in a table
   - Reject max

   For more about using these fields, see About configuring a data load from S3 .

Cancel an initiated loading job

If a loading job is in progress, you can cancel it using the Cancel option in the Load History tab's Cancel column. Click Cancel to cancel the loading job. When you cancel a job, Vertica rolls back all rows and does not commit any data to the target table.



Status	Time Started	User	Schema Name	Table Name	Execution Time (ms)	Accepted Rows	Rejected Rows	Stream Name	Cancel
					greater than / less than	greater than / less than			
Running	Mar 28, 2016 1:...	uidbadmin	v_dbd_foo	vs_designs	0	0	N/A		Cancel
Success	Mar 28, 2016 1:...	natalia	store	store_orders_fact	10707	299994	Details	MC_S3_Load	
Failure	Mar 28, 2016 1:...	natalia	store	store_orders_fact	581	0	Details	MC_S3_Load	
Success	Mar 28, 2016 12...	natalia	store	store_orders_fact	1299	11	N/A	S3_Data_load	
Failure	Mar 28, 2016 12...	natalia			2	0	N/A	S3_Data_load	
Success	Mar 25, 2016 3:...	uidbadmin	store	store_orders_fact	17990	300000	Details	MC_S3_Load	

See also

- Viewing load history
- About configuring a data load from S3 Apache Kafka integration
- COPY

In this section

- About configuring a data load from S3
- Viewing load history

About configuring a data load from S3

When you create an S3 Data Load using MC, you have the option of further configuring the load operation. You can optionally specify the following:

- Add COPY Parameters
- Capture Rejected Data in a Table
- Set a Rejected Records Maximum

Add COPY parameters

MC performs the load operation with COPY . You can use the COPY Parameters field to further configure the COPY operation. This field accepts parameters that are specified after the COPY statement's FROM clause. For details on these parameters and special requirements, see Parameters .

> **Note**
> The FILTER and PARSER parameters must appear in that order and precede all other parameters.

For example, you can specify the DELIMITER and SKIP parameters to separate columns with a comma, and skip one record of input data, respectively:

```
DELIMITER ',' SKIP 1
```

You can also add comments in this field with standard C comment notation .

> **Note**
> This field does not support SQL comment notation (double hyphen --).

Capture rejected data in a table

Set **Capture rejected data in a table** to Yes to create a table that contains rejected row data. You can view this data in the Load History tab.

This table uses the following naming convention:

```
schema.s3_load_rejections_target-table-name
```

You must have CREATE privilege on the schema if the table doesn't already exist. When you invoke multiple load processes for the same target table, MC appends all rejections data to the same table. For details, see Saving rejected data to a table .

Set a rejected records maximum

Set **Reject Max** to the maximum number of rows that can be rejected before the load operation fails. If COPY rejects the specified maximum rows, Vertica rolls back the entire load operation.

See also

- Loading data from Amazon S3 using MC
- Viewing load history

Viewing load history

You can view a history of all your continuous and instance loading jobs in Vertica on the Data Load Activity page.

- **Continuous jobs:** Loading jobs that continuously monitor a source and stream data from the source.
- **Instance jobs:** Loading jobs that batch load from a source. Instance jobs are of a fixed length and shorter-term than continuous loads.

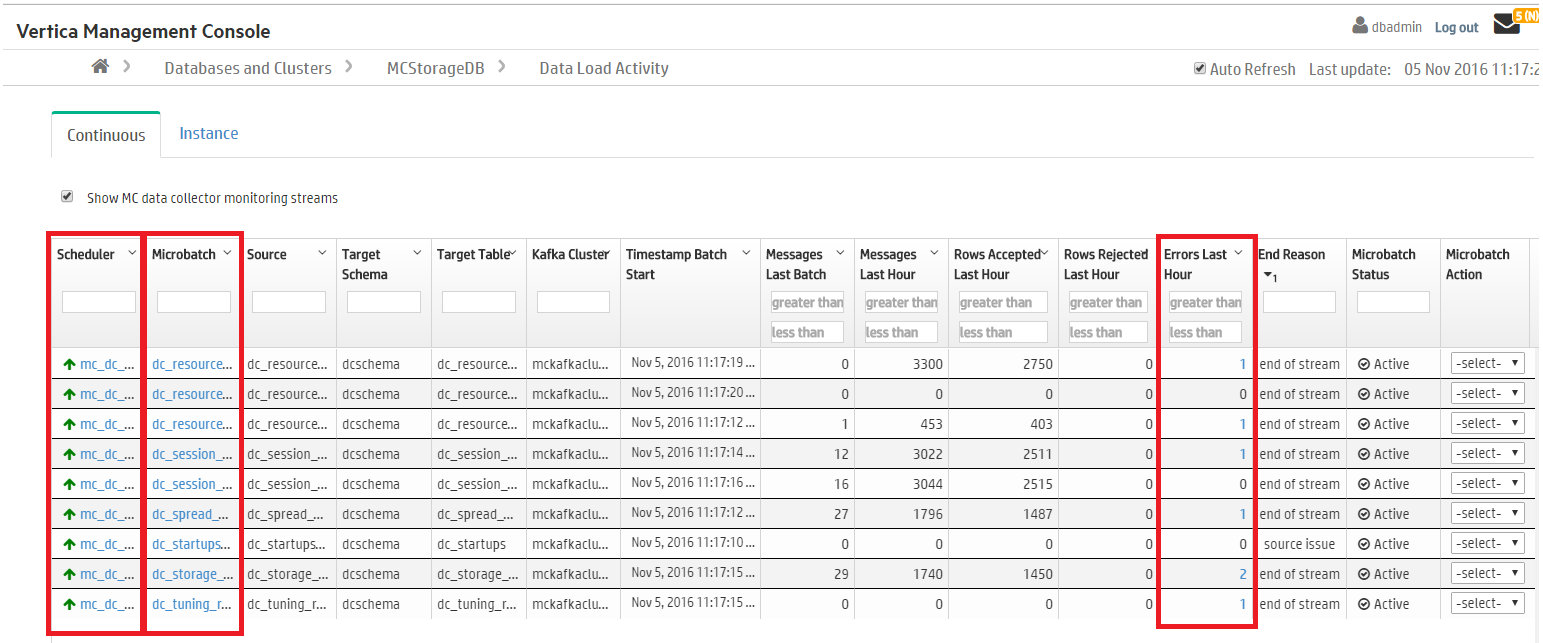
View continuous loads

The Continuous tab on the Data Load Activity page displays history of your database's continuous loading jobs. For example, you can see loading jobs you create using the Vertica integration with Kafka (see Apache Kafka integration). Additionally, if you enable the MC extended monitoring feature, the Continuous tab displays the continuous jobs that stream data from your monitored database to a storage database. (See Extended monitoring for more on how MC can use Kafka to monitor databases externally.)

Use the Continuous tab to view details about continuous jobs, such as their source, target tables, and other microbatch configuration details.

If extended monitoring is enabled, jobs streaming to the MC storage database show mc_dc_kafka_config as the scheduler name. Deselect **Show MC data collector monitoring streams** at the top of the tab to remove these jobs from the display.

In the Continuous tab, click the labels in the **Scheduler** , **Microbatch** , and **Errors Last Hour** to view additional details about those loading jobs.



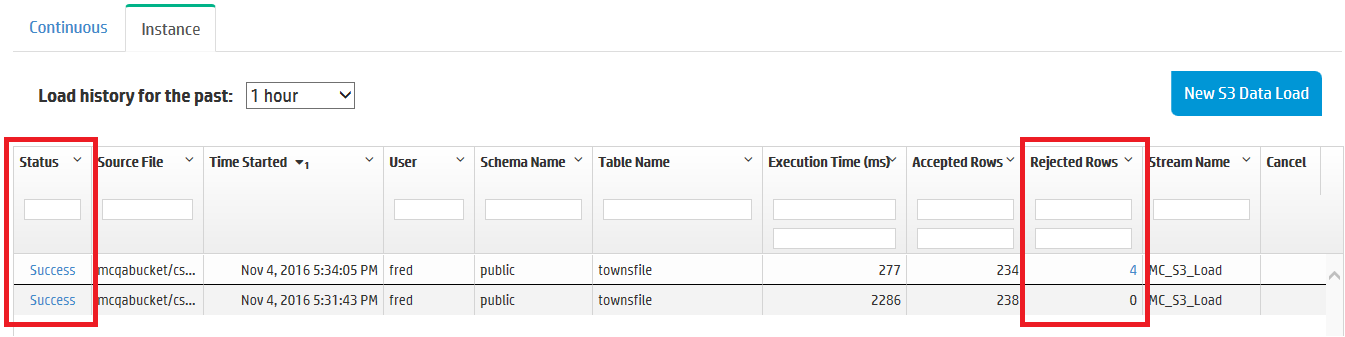
Scheduler	Microbatch	Source	Target Schema	Target Table	Kafka Cluster	Timestamp Batch Start	Messages Last Batch	Messages Last Hour	Rows Accepted Last Hour	Rows Rejected Last Hour	Errors Last Hour	End Reason	Microbatch Status	Microbatch Action
mc_dc_...	dc_resource...	dc_resource...	dcschema	dc_resource...	mckafkaclu...	Nov 5, 2016 11:17:19 ...	0	3300	2750	0	1	end of stream	Active	-select-
mc_dc_...	dc_resource...	dc_resource...	dcschema	dc_resource...	mckafkaclu...	Nov 5, 2016 11:17:20 ...	0	0	0	0	0	end of stream	Active	-select-
mc_dc_...	dc_resource...	dc_resource...	dcschema	dc_resource...	mckafkaclu...	Nov 5, 2016 11:17:12 ...	1	453	403	0	1	end of stream	Active	-select-
mc_dc_...	dc_session_...	dc_session_...	dcschema	dc_session_...	mckafkaclu...	Nov 5, 2016 11:17:14 ...	12	3022	2511	0	1	end of stream	Active	-select-
mc_dc_...	dc_session_...	dc_session_...	dcschema	dc_session_...	mckafkaclu...	Nov 5, 2016 11:17:16 ...	16	3044	2515	0	0	end of stream	Active	-select-
mc_dc_...	dc_spread_...	dc_spread_...	dcschema	dc_spread_...	mckafkaclu...	Nov 5, 2016 11:17:12 ...	27	1796	1487	0	1	end of stream	Active	-select-
mc_dc_...	dc_startups...	dc_startups...	dcschema	dc_startups	mckafkaclu...	Nov 5, 2016 11:17:10 ...	0	0	0	0	0	source issue	Active	-select-
mc_dc_...	dc_storage_...	dc_storage_...	dcschema	dc_storage_...	mckafkaclu...	Nov 5, 2016 11:17:15 ...	29	1740	1450	0	2	end of stream	Active	-select-
mc_dc_...	dc_tuning_r...	dc_tuning_r...	dcschema	dc_tuning_r...	mckafkaclu...	Nov 5, 2016 11:17:15 ...	0	0	0	0	1	end of stream	Active	-select-

For more on continuous data streaming terminology, see Data streaming integration terms .

View load instances

In the Instance tab, you can see a history of your database's one-time loading jobs. For example, you can view instance jobs you created using the COPY command in vsql (see COPY), or instance jobs you created in MC to copy data from an Amazon S3 bucket. (For more about initiating loading jobs in MC, see Loading data from Amazon S3 using MC .)

In the Instance tab, click the labels in the Status column and Rejected Rows column to view more details about completed jobs. For more about rejected rows, see Handling messy data .



Status	Source File	Time Started	User	Schema Name	Table Name	Execution Time (ms)	Accepted Rows	Rejected Rows	Stream Name	Cancel
Success	mcqabucket/cs...	Nov 4, 2016 5:34:05 PM	fred	public	townsfile	277	234	4	MC_S3_Load	
Success	mcqabucket/cs...	Nov 4, 2016 5:31:43 PM	fred	public	townsfile	2286	238	0	MC_S3_Load	

The number of load history results on the Instance tab depends on the Data collector retention policy for Requests Issued and Requests Completed. To change the retention policy, see Configuring data retention policies .

See also

- Loading data from Amazon S3 using MC
- Apache Kafka integration
- Data streaming integration terms
- COPY

Microsoft Azure in MC

Management Console (MC) supports both Eon Mode and Enterprise Mode clusters on Microsoft Azure as described in the following table:

Architecture	Description
Eon Mode	Deploy an MC instance, and then provision and create an Eon Mode database from the MC. For more details, see the following: • Creating an Eon Mode cluster and database in Azure in MC • Adding subclusters in MC
Enterprise Mode	Deploy a four-node database comprised of one MC instance and three database nodes. In Enterprise Mode, the database uses the MC primarily as a monitoring tool. For example, you cannot provision and create a database with an Enterprise Mode MC. For information about creating and managing an Enterprise Mode database, see Create a database using administration tools .

For additional details, see Vertica on Microsoft Azure .

Provision and monitor clusters

You can use MC to provision an Eon Mode database cluster on Azure. For details, see Creating an Eon Mode cluster and database in Azure in MC .

MC provides specific resources for monitoring database clusters on Azure. For details, see Managing an Eon Mode database in the cloud .

You can revive a stopped Eon Mode database on Azure using MC. For details, see Reviving an Eon Mode database on Azure in MC .

Managing your cluster in MC

1. On the MC home page, click **View Infrastructure** . MC displays the Database and Cluster View. This view shows your infrastructure platform, cluster, and database.
2. On the left side of the screen next to **Clusters** , click the square for the cluster you want to manage. MC displays a window with your cluster name, an information summary, and several buttons.
3. Click **Manage** . The **Cluster** page displays.
4. On the **Cluster** page, you can view the following information:
   - The instances in your cluster in visual format.
   - The status of each instance, whether it is running.
   - The private and public IP address for each cluster instance.
   - The Vertica version that is running, your region, and your instance type in the **Cluster** pane.

Cluster actions on Azure in MC

On the **Cluster** page, you can perform the following cluster actions:

- **Start Cluster** : Starts the instances, then starts the database. For Eon Mode databases, MC repopulates the nodes with data from the storage account container.
- **Stop Cluster** : Stops the nodes in the database, then stops their cloud instances.
- **Advanced > Terminate** : Stops the database, then terminates the cloud instances.

Subcluster management

You can add, Scale Up, Scale Down, remove, and terminate subclusters with MC. For details, see the following:

- Subclusters in MC
- Scaling subclusters in MC
- Subcluster action rules in MC

Node management

You add or delete nodes by scaling subclusters up or down. You can also start, stop, and restart nodes. For details, see the following:

- Add nodes to a cluster in AWS using Management Console

- Starting, stopping, and restarting nodes in MC
- Node action rules in MC

Restrictions

You cannot revive an Eon Mode database with the MC.

In this section

- Creating an Eon Mode cluster and database in Azure in MC
- Eon Mode volume configuration defaults for Azure
- Reviving an Eon Mode database on Azure in MC

Creating an Eon Mode cluster and database in Azure in MC

After you deploy a Management Console instance on Azure, you can provision a cluster and create an Eon Mode database.

Prerequisites

Before you begin, complete or obtain the following:

- Complete Deploy Vertica from the Azure Marketplace
- Complete the cluster and storage requirements described in Eon Mode on Azure prerequisites
- SSH public key used when deploying the MC instance
- Review Recommended Azure VM types and operating systems
- Vertica credentials
- Vertica Management Console credentials

Creating the database

Complete the following steps from the Management Console:

1. Log in to the Vertica Management Console.
2. On the Management Console home page, click Eon Mode.
3. On Vertica License, select one of the following license mode options:
   - **Community Edition** : A free Vertica license to preview Vertica functionality. This license provides limited features. If you use a Community Edition license for your deployment, you can upgrade the license later to expand your cluster load. See Managing licenses form more information.
   - **Premium Edition** : Use your Vertica license. After you select this option, click **Browse** to locate and upload your Vertica license key file, or manually enter it in the field.
4. Click **Next** . On **Azure Environment** , supply the following information:
   - **SSH Public Key** : Paste the same public key used when you deployed the MC instance in the Azure Marketplace.
   - **Azure Subnet** : The subnet for your cluster. Select the same subnet used when you deployed the MC instance in the Azure Marketplace. If your organization requires multiple subnets for security purposes, see the Azure documentation for additional information.
   - **CIDR Range** : The range of IP addresses for client and SSH access. Azure requires that the last octet is 0 and the prefix is 24. For example, 10.20.30.0/24.
   - **Node IP Setting** : Choose **Public IP - Dynamic** , **Public IP - Static** , or **Private IP** . For details, see the Azure documentation .
   - **Communal Storage URL** : The path to a new subfolder in your existing Azure Blob storage account and container. The subfolder must not already exist.
   - **Tag Azure Resources** : Optional. Assign distinct, searchable metadata tags to the instances in this cluster. Many organizations use labels to organize, track responsibility, and assign costs for instances.
     To add a tag, click the slider to the right to display the **Tag Name** and **Tag Value** fields. Click **Add** to create the tag. Added tags are displayed below the fields. Vertica recommends that you use lowercase characters in tag fields.
5. Click **Next** . **Database Parameters** accepts identifying information about your database and OS version:
   - **Database Name** : The name for your new database. See Creating a database name and password for database name requirements.
   - **Administrator Username** : The name of the database superuser .
   - **Administrator Password** : The password for the database administrator user account. For details, see Password guidelines .
   - **Confirm Password** : Reenter the **Administrator Password** .
   - **Vertica Version** : Select the desired Vertica database version. You can select from the latest hotfix of recent Vertica releases. For each database version, you can also select the operating system. For available OS and major version options, see Recommended Azure Operating Systems .
   - **Load Sample Data** : Optional. Click the slider to the right to preload your database with example clickstream data. This option is useful if you are testing features and want some preloaded data in the database to query.
6. Click **Next** . On Azure configuration, supply the following information:
   - **Number of Nodes** : The initial number of nodes for your database.

- **Number of** Vertica **Database Shards** : Sets the number of shards in your database. Vertica suggests a number of shards automatically, based on your node count. After you set this value, you cannot change it later. The shard count must be greater than or equal to the maximum subcluster count. Be sure to allow for node growth. See Configuring your Vertica cluster for Eon Mode for recommendations.
- **Virtual machine (VM) size** : The instance types used for the nodes. For a list of recommended instances, see Recommended Azure VM types and operating systems .
- **Local Storage per Node** : Customize your cluster according to your storage needs. For the Vertica default settings for each supported instance, see Eon Mode volume configuration defaults for Azure .

7. Click **Next** . On **Review** , confirm your selections. Click **Edit** to return to a previous section and make changes.
8. When you are satisfied with your selections, click the **I accept the terms and conditions** checkbox.
9. Click **Create Cluster** to create a Eon Mode cluster on Azure.

After creating the database

After you create the database, click **Get Started** to view the Fast Tasks page. To view your database, select **Manage and View Your Vertica Database** to go to the database **Overview** .

You can also view your database from the **Recent Databases** section of the MC home page.

For additionalinformation about managing your cluster, instances, and database using Management Console, see Managing database clusters .

Eon Mode volume configuration defaults for Azure

When you provision an Eon Mode database cluster, Management Console (MC) configures separate volumes for the depot, catalog, and temp directories. The specific volumes and sizes that Management Console configures vary depending on the Azure instance type that you select when provisioning the cluster.

MC follows these rules when allocating resources for these directories for an Eon Mode database cluster:

- Depot: Use Standard or Premium LRS to ensure the data is durable.
- Catalog: Use Standard or Premium LRS to ensure the data is durable.
- Temp: Allocate instance store if available with the selected instance type. Otherwise, allocate Standard or Premium LRS volumes.

If NVMe or Local SSD are displayed as volume types for an instance, there are no other choices. You must choose a different VM to change the volume type.

For details about each disk type, see the Azure documentation .

Instance Type	Instance Storage	Depot	Catalog	Temp
E16ds v4	1 x 600GB	Configurable 8 Data Disks/Managed Disk/Remote Storage 8 x 75 GB Premium/Standard LRS **Default:** Premium LRS	Data Disk 50 GB Premium/Standard LRS - Durable **Default:** Premium LRS	1 x 600GB Temporary local SSD (ephemeral) Display: "Local SSD"
E20ds v4	1 x 750 GB	Data Disk 8 x 75 GB	Data Disk 50 GB	1 x 600GB Temporary local SSD (ephemeral) Display: "Local SSD"
E32ds v4	1 x 1200 GB	Data Disk 8 x 75 GB	Data Disk 50 GB	1 x 600GB Temporary local SSD (ephemeral) Display: "Local SSD"

E48ds v4	1 x 1800 GB	Data Disk 8 x 75 GB	Data Disk 50 GB	1 x 600GB Temporary local SSD (ephemeral) Display: "Local SSD"
E64ds v4	1 x 2400 GB	Data Disk 8 x 75 GB	Data Disk 50 GB	1 x 600GB Temporary local SSD (ephemeral) Display: "Local SSD"
E80ids v4	1 x 2400 GB	Data Disk 8 x 75 GB	Data Disk 50 GB	1 x 600GB Temporary local SSD (ephemeral) Display: "Local SSD"
E16s v4	Instance Store not supported	Configurable 8 Data Disks 8 x 75 GB (user defined)	Data Disk 50 GB	Data Disks 1 x 300 GB (user-defined) No less than 300 GB Premium/Standard LRS - Durable **Default:** Premium LRS 300GB **Min** : 50 GB **Max** : 10000 GB **Note** This instance type supports Remote storage only. Temp uses Data Disks.
E20s v4	Instance Store not supported	Data Disk 8 x 75 GB	Data Disk 50 GB	Data Disks 1 x 300 GB (user-defined) No less than 300 GB Premium/Standard LRS - Durable **Default:** Premium LRS 300GB **Min** : 50 GB **Max** : 10000 GB

E32s v4	Instance Store not supported	Data Disk 8 x 75 GB	Data Disk 50 GB	Data Disks 1 x 300 GB (user-defined) No less than 300 GB Premium/Standard LRS - Durable **Default:** Premium LRS 300GB **Min** : 50 GB **Max** : 10000 GB
E48s v4	Instance Store not supported	Data Disk 8 x 75 GB	Data Disk 50 GB	Data Disks 1 x 300 GB (user-defined) No less than 300 GB Premium/Standard LRS - Durable **Default:** Premium LRS 300GB **Min** : 50 GB **Max** : 10000 GB
E64s v4	Instance Store not supported	Data Disk 8 x 75 GB	Data Disk 50 GB	Data Disks 1 x 300 GB (user-defined) No less than 300 GB Premium/Standard LRS - Durable **Default:** Premium LRS 300GB **Min** : 50 GB **Max** : 10000 GB
E80s v4 **Note** You cannot use this type to revive an Eon Mode database.	Instance Store not supported	Data Disk 8 x 75 GB	Data Disk 50 GB	Data Disks 1 x 300 GB (user-defined) No less than 300 GB Premium/Standard LRS - Durable **Default:** Premium LRS 300GB **Min** : 50 GB **Max** : 10000 GB
E16s v3	1 x 256 GB	Data Disk 8 x 75 GB	Data Disk 50 GB	1 x 256 GB Use Temporary local SSD (instance store is ephemeral)

E20s v3	1 x 320 GB	Data Disk 8 x 75 GB	Data Disk 50 GB	1 x 320 GB Use Temporary local SSD (instance store is ephemeral)
E32s v3	1 x 512 GB	Data Disk 8 x 75 GB	Data Disk 50 GB	1 x 512 GB Use Temporary local SSD (instance store is ephemeral)
E48s v3	1 x 768 GB	Data Disk 8 x 75 GB	Data Disk 50 GB	1 x 768 GB Use Temporary local SSD (instance store is ephemeral)
E64s v3	1 x 864 GB	Data Disk 8 x 75 GB	Data Disk 50 GB	1 x 864 GB Use Temporary local SSD (instance store is ephemeral)
E64is v3 **Note** You cannot use this type to revive an Eon Mode database.	1 x 864 GB	Data Disk 8 x 75 GB	Data Disk 50 GB	1 x 864 GB Use Temporary local SSD (instance store is ephemeral)

Reviving an Eon Mode database on Azure in MC

An Eon Mode database keeps an up-to-date version of its data and metadata in its communal storage location. After a cluster hosting an Eon Mode database is terminated, this data and metadata continue to reside in communal storage. When you revive the database later, Vertica uses the data in this location to restore the database in the same state on a newly provisioned cluster.
Many of the fields in the revive workflow are populated with information provided during provisioning. For details about fields with existing values, see Creating an Eon Mode cluster and database in Azure in MC .

The following steps revive an Eon Mode database on Azure:

1. On the MC home page, select **Revive Eon Mode database** . The **Provision & Revive an** Eon Mode **Database** window opens.
2. On **Specify cluster access preferences** , supply the following information:
   - **Azure Subnet** : The subnet for your cluster. Select the same subnet used when you deployed the MC instance in the Azure Marketplace. If your organization requires multiple subnets for security purposes, see the Azure documentation for additional information.
   - **SSH Public Key** : Paste the same public key used when you deployed the MC instance in the Azure Marketplace.
   - **CIDR Range** : The range of IP addresses for client and SSH access.
3. Select **Next** . On **Specify Azure AZB path for Communal Storage of database** , you can enter a parent directory to list all available Eon Mode databases within that directory:
   1. Enter the directory name. At minimum, you must provide the account and container name, and optionally subfolder names in the following format:
      azb:// *storage-account* / *container* [/ *subfolder-name* /...]
   2. Select **Discover** . The MC lists all available Eon Mode databases within the container or subfolder.
   3. Select the database that you want to revive from the list.
4. Select **Next** . On **Enter revive database configurations** , supply the following information:
   - **Revive to Vertica Version** : Lists the currently available Vertica versions. If you select a version that is later than the version that you used to provision the database, the MC upgrades the database version automatically.
   - **Password** : The password for the database superuser .

    - **Confirm Password** : Reenter **Password** .
5. Select **Next** . On **Specify cloud instance and depot data storage** , supply the following information about the depot :
    - **Virtual Machine (VM) Size** : The machine types used for the nodes. For recommended machine types, see the memory optimized machine types in Recommended Azure VM types and operating systems .
    - **Managed Disk Volume Type** : Available for Azure managed disks only. For details about each disk type, see the Azure documentation .
    - **Managed Disk Volume Size (GB) per Volume per Available Node** : Volume size for each node. This value is populated with the volume configuration defaults for the associated instance type.
    - **Node IP Setting** : Choose **Public IP - Dynamic** , **Public IP - Static** , or **Private IP** . For details, see the Azure documentation .
6. Select **Next** . On **Specify additional storage and tag info** , supply the following information:
    - **Managed Volume Type** : Available only for Azure managed disks. For details about each disk type, see the Azure documentation .
    - For the catalog and temp paths, provide the following information:
        - **Managed Disk Volume Type** : Available only for Azure managed disks. For details about each disk type, see the Azure documentation .
        - **Managed Disk Volume Size (GB) per Available Node** : Volume size for each node. This value is populated with the volume configuration defaults for the associated VM type.
    - **Tag Azure Resources** : Optional. Assign distinct, searchable metadata tags to the instances in this cluster. Many organizations use labels to organize, track responsibility, and assign costs for instances.
      To add a tag, select the checkbox to make the **Tag Name** and **Tag Value** available. Click **Add** to create the tag. Tags are displayed below the fields. Vertica recommends that you use lowercase characters in tag fields.
7. Select **Next** . On **Review revive information** , confirm your selections. Select **Back** to return to a previous section and make changes.
8. When you are satisfied with your selections, select the **Accept terms and conditions** checkbox.
9. Select **Revive Database** to revive the Eon Mode database on Azure.

For details about database clusters in Management Console, see Managing database clusters .

Google Cloud Platform in MC

Management Console (MC) supports cluster, subcluster, and node actions on Google Cloud Platform (GCP).

Provision, monitor, and revive clusters

You can use MC to provision an Eon Mode database cluster on GCP. For details, see Provision an Eon Mode cluster and database on GCP in MC .

MC provides specific resources for monitoring database clusters on GCP. For details, see Managing an Eon Mode database in the cloud .

You can revive a stopped Eon Mode database on GCP using MC. For details, see Reviving an Eon Mode database on GCP in MC .

Managing your cluster in MC

1. On the MC home page, click **View Infrastructure** . MC displays the Database and Cluster View. This view shows your infrastructure platform, cluster, and database.
2. On the left side of the screen next to **Clusters** , click the square for the cluster you want to manage. MC displays a window with your cluster name, an information summary, and several buttons.
3. Click **Manage** . The **Cluster** page displays.
4. On the **Cluster** page, you can view the following information:
    - The instances in your cluster in visual format.
    - The status of each instance, whether it is running.
    - The private and public IP address for each cluster instance.
    - The Vertica version that is running, your region, and your instance type in the **Cluster** pane.

Cluster actions on GCP in MC

On the **Cluster** page, you can perform the following cluster actions:

- **Start Cluster** : Starts the instances, then starts the database. For Eon Mode databases, MC repopulates the nodes with data from the storage account container.
- **Stop Cluster** : Stops the nodes in the database, then stops their cloud instances.
- **Advanced > Terminate** : Stops the database, then terminates the cloud instances.

> **Note**
> If a GCP instance has a local SSD attached, Google does not allow you to stop the instance. If you do shut down an instance with a local SSD through the guest operating system, you will not be able to restart the instance, and the data on the local SSD will be lost.

Subcluster management

You can add, Scale Up, Scale Down, remove, and terminate subclusters with MC. For details, see the following:

- Subclusters in MC
- Scaling subclusters in MC
- Subcluster action rules in MC

Node management

You add or delete nodes by scaling subclusters up or down. You can also start, stop, and restart nodes. For details, see the following:

- Add nodes to a cluster in AWS using Management Console
- Starting, stopping, and restarting nodes in MC
- Node action rules in MC

Restrictions

- Subclusters are supported in Eon Mode only, not in Enterprise Mode.
- Node actions are not supported in MC on GCP.

See also

Vertica on Google Cloud Platform

In this section

- GCP Eon Mode instance recommendations
- Provision an Eon Mode cluster and database on GCP in MC
- Reviving an Eon Mode database on GCP in MC
- Eon Mode volume configuration defaults for GCP

GCP Eon Mode instance recommendations

When you use the MC to deploy an Eon Mode database to the Google Cloud Platform (GCP), you choose the instance type to deploy as the database's nodes. The default instance settings in the MC are the more conservative option (currently, n1-standard-16). They are sufficient for most workloads. However, you may choose instances with more memory (such as n1-highmem-16) if your queries perform complex joins that may otherwise spill to disk. You can also choose instances with more cores (such as n1-standard-32), if you perform highly-complex compute-intensive analysis. The following links provide additional information about GCP machine type instances and Vertica:

- Machine types : Google Cloud's documentation that describes configuration details for each instance option.
- Supported GCP machine types : Available machine types when deploying Vertica to GCP.
- Eon Mode volume configuration defaults for GCP : Details about Vertica's default volume configurations.

The more powerful instance you choose, the higher the cost per hour. You need to balance whether you want to use fewer, higher-powered but more expensive instances vs. relying on more lower-powered instances that cost less. Thanks to Eon Mode's elasticity, if you choose to use the less-powerful instances, you can always add more nodes to meet peak demands. When you reduce the number of instances to a minimum during off-peak times, you'll spend less than if you had a similar number of more-powerful instances.

Storage options

The MC's deployment wizard also asks you to select the type of local storage for your instances. You can select different options for each type of local storage that Vertica uses: the catalog, the depot, and temporary space. For all of these storage locations, you choose the type of disks to use (standard vs. SSD). You will see the best performance with SSD disks. However, SSD disks cost more.

For the depot, you also choose whether to use local or persistent disks. The local option is faster, as it resides directly on the virtual machine host. However, whenever you shut down the node, this storage is wiped clean. The persistent storage is slower than the local option, as it is not stored directly on the machine hosting the instance. However, it is not wiped out whenever you shut down the instance. See the Google Cloud documentation's Storage options page for more information.

Which of these options you choose depends on how much depot warming the nodes must perform when starting. If the content of your node's depots change little over time (or you tend to frequently start and stop instances), using persistent storage makes sense. In this case, the depot's warming period will be shorter because most of the data the node needs to participate in queries may still be in its depot when it starts. It will perform fewer fetches of data from communal storage while participating in queries.

If your working data set is rapidly changing or you tend to leave nodes stopped for extended periods of time, your best choice is usually to use local storage. In this scenario, the data in the node's depot when it restarts is usually stale. To participate in queries, the node must fetch much of the data it needs from communal storage, resulting in slower performance until it has warmed its depot. Using local ephemeral storage makes sense here, because you will get the benefit of having faster depot storage. Because your nodes have to warm their depots anyhow, there is less of a downside of having the depot on ephemeral storage.

For general guidelines on scaling your cluster for Eon Mode database, see Configuring your Vertica cluster for Eon Mode .

Provision an Eon Mode cluster and database on GCP in MC

You can use Google Marketplace and MC to provision an Eon Mode database on GCP. The sections below give an overview of how to set up an Eon Mode database on GCP, with links to the detailed procedures

Prerequisites

- GCP Eon Mode instance recommendations
- Eon Mode on GCP prerequisites

Step 1: provision an MC instance using Google marketplace

These steps are an overview of the procedure. For more detailed instructions, see Deploy an MC instance in GCP for Eon Mode .

In Google Marketplace:

1. Select the Vertica Eon Mode solution.
2. Fill in the fields to configure a GCP MC instance.
3. Click the **Deploy** button to provision the MC instance.
4. Connect to and log into the MC instance.

You are now in the new MC instance running on GCP, and the MC home page appears.

Step 2: use the MC instance to provision an Eon Mode database on GCP

To use the MC to provision and deploy a new Eon Mode database on GCP:

1. From the MC home screen, click **Create new database** to launch the Create a Vertica Cluster on Google Cloud wizard.
2. On the first page of the wizard enter the following information:
   - **Google Cloud Storage HMAC Access Key** and **HMAC Secret Key** : Copy and paste the HMAC access key and secret you created earlier. You find these values on the Interoperability tab of the of the Storage Settings page. See Eon Mode on GCP prerequisites for details.
   - **Zone** : This value defaults to the zone containing your MC instance. Make this value is the same as the zone containing the Google Cloud Storage bucket that your database will use for communal storage.

     > **Caution**
     > You will see significant performance issues if you choose different zones for cluster instances, storage, or the MC.

   - **CIDR Range** : The IP address range for clients to whom you want to grant access to your database. Make this range as restrictive as possible to limit access to your database.
3. Click **Next** , and supply the following information:
   - **Vertica Database Name** : the name for your new database. See Creating a database name and password for database name requirements.
   - **Vertica Version** : select the desired Vertica database version. You can select from the latest hotfix of recent Vertica releases. For each database version, you can also select the operating system.
   - **Vertica Database User Name** : the name of the database superuser . This name defaults to dbadmin, but you can enter another user name here.
   - **Password** and **Confirm Password** : Enter a password for the database superuser account.
   - **Database Size** : The number of nodes in your initial database. If you specify more than three nodes here, you must supply a valid Vertica license file in the Vertica License field (below).
   - **Vertica License** : Click **Browse** to locate and upload your Vertica license key file. If you do not supply a license key file here, the wizard deploys your database with a Vertica Community Edition license. This license has a three node limit, so the value in the Database Size filed cannot be larger than 3 if you do not supply a license. If you use a Community Edition license for your deployment, you can upgrade the license later to expand your cluster load more than 1TB of data. See Managing licenses form more information.

     > **Note**
     > This field does not appear if you created your MC instance using a by-the-hour (BTH) launcher. The BTH license is automatically applied to all clusters you create using a BTH MC instance. For a by-the-hour license, cloud vendors charge the customer for licensed Vertica usage along with their cloud infrastructure charges.

   - **Load example data** : Check this box if you want your deployed database to load some example clickstream data. This option is useful if you are testing features and just want some preloaded data in the database to query.
4. Click **Next** and supply the following information:
   - **Instance Type** : the specifications of the virtual machine instances the MC will use to deploy your database nodes. See the Google Cloud documentation's Machine types page for details of each instance type. Also see GCP Eon Mode instance recommendations .

- **Database Depot Path** and **Disk Type** : the local mount point for the depot, and the type and number of local disks dedicated to the depot for each node. You cannot change the mount path for the depot. The disks you select in the **Disk Type** field are only used to store the depot. On the next page of the wizard, you will configure disks for the catalog and temporary disk space. You will see the best performance when using SSD disks, although at a higher cost. You can choose to use faster local storage for your depot. However, local storage is ephemeral—GCP wipes the disk clean whenever you stop the instance. This means each time you start a node, it will have to warm its depot from scratch, rather than taking advantage of any still-current data in its depot. See the Google Cloud documentation's Storage options page for more information about the local disk options.
- **Volume Size** : the amount of disk space available on each disk attached to each node in your cluster. This field shows you the total disk space available per node in your cluster. For the best practices on choosing the amount of disk space for your nodes, see Configuring your Vertica cluster for Eon Mode .
- **Data Segmentation Shards** : sets the number of shards in your database. After you set this value, you cannot change it later. See Configuring your Vertica cluster for Eon Mode for recommendations. The default value is based on the number of nodes you entered in the Database size you specified earlier. It is usually sufficient, unless you anticipate greatly expanding your cluster beyond your initial node count.
- **Communal Location** : a Google Cloud Storage URL that specifies where to store your database's communal data. See Eon Mode on GCP prerequisites for requirements.
- **Instance IP settings** : specify whether the nodes in your database will have static or ephemeral network addresses that are accessible from the internet, or addresses that are only accessible from within the internal virtual network.

5. Click **Next** . The wizard validates your communal storage location URL. If there is an problem with the URL you entered, it displays an error message and prompts you to fix the URL.
   After your communal storage URL passes validation, fill in the following information:
   - **Database Catalog Path** , **Disk Type** , and **Size (GB) per Available Node** : the mount point disk type, and disk size for the local copy of the database catalog on each node. You cannot edit the mount point. You choose the type of local disk to use for the catalog, and its size. You can only choose persistent disk storage for the catalog. SSD drives are faster, but more expensive than standard disks. The default setting for the disk size is adequate for most medium size databases. Increase the size if you anticipate maintaining a large database.
   - **Database Temp Path** , **Disk Type** , and **Size (GB) per Available Node** : the mount point disk type, and disk size for the temporary storage space on each node. You cannot edit the mount point. You choose the type of local disk to use, and its size. You can only choose persistent disk storage for the temporary disk space. SSD drives are faster, but more expensive than standard disks. The default setting is adequate for most databases. Consider increasing the temporary space if you perform many complex merges that spill to disk.
   - **Label Instances** : check this box to enable adding labels to your node's instances. Many organizations use labels to organize, track responsibility, and assign costs for instances. See the Google Cloud documentation's Labeling resources page for more information. If you choose to add labels, enter the label name and value, and click **Add** .
6. Click **Next** . Review the summary of all your database settings. If you need to make a correction, use the Back button to step back to previous pages of the wizard.
7. When you are satisfied with the database settings, check **Accept terms and conditions** and click **Create** .

The process of provisioning and creating the database takes several minutes. After it completes successfully, the MC displays a **Get Started** button. This button leads to a page of useful links for getting started with your new database.

See also

- Managing an Eon Mode database in MC
- Stopping and starting an Eon Mode cluster
- Reviving an Eon Mode database cluster

Reviving an Eon Mode database on GCP in MC

An Eon Mode database keeps an up-to-date version of its data and metadata in its communal storage location. After a cluster hosting an Eon Mode database is terminated, this data and metadata continue to reside in communal storage. When you revive the database later, Vertica uses the data in this location to restore the database in the same state on a newly provisioned cluster.
Follow these steps to revive an Eon Mode database on GCP:

1. On the MC home page, click **Revive Eon Mode Database** . MC launches the Provision and Revive an Eon Mode Database wizard.
2. On the first page of the wizard, enter the following information:
   - **Google Cloud Storage HMAC Access Key** and **HMAC Secret Key** : Copy and paste the HMAC access key and secret you created when you created the database. See Eon Mode on GCP prerequisites for details.
   - **Zone** : This value defaults to the zone containing your MC instance. Make this value the same as zone containing the Google Cloud Storage bucket your database will use for communal storage. You will see significant performance issues if you choose different zones for cluster instances, storage, or the MC.
   - **CIDR Range** : The IP address range for clients you want to grant access to your database. Make this range as restrictive as possible to limit the exposure of your database.
3. Click **Next** . On the second page of the wizard, set **Google Storage Path for Communal Storage of Database** to the URL of the communal storage bucket for the Eon Mode database to revive. For requirements, see Eon Mode on GCP prerequisites .
4. Click **Discover** . MC displays a list of all Eon Mode databases available on the specified communal storage location.

5. Select the database to revive. MC prepopulates the choices for the Data, Depot, and Temp catalogs, using the same machine types and configuration choices used when the database was created.
6. Click **Next** . Review the summary of all your database settings. If you need to make a correction, use the Back button to step back to previous pages of the wizard.
7. When you are satisfied with the database settings, check the **Accept terms and conditions** box and click **Revive Database** .

MC displays a progress screen while creating the cluster and reviving the database onto it, a process which takes several minutes. After it completes successfully, the MC displays a **Get Started** button. This button leads to a page of useful links for getting started with your new database.

Eon Mode volume configuration defaults for GCP

Vertica supports a variety of disk volume resources for provisioning instances on Google Cloud Platform (GCP).

All data is secured with Google-managed data encryption. Management Console does not support user-managed data encryption.

For performance information, see Google's Block storage performance documentation.

Persistent disk defaults

You can allocate up to 128 persistent disks (PDs). The following table describes the default persistent disk volume resources that Vertica provides:

Instance Types	Catalog	Depot	Temp
n1-standard/highmem-16 n1-standard/highmem-32 n1-standard/highmem-64 n2-standard/highmem-16 n2-standard/highmem-32 n2-standard/highmem-48 n2-standard/highmem-64	Configurable 1 PD (Standard/SSD) volume **Default:** 50 GB	Configurable 8 PD (Standard/SSD) volume **Default:** 600 GB	Configurable 1 PD (Standard/SSD) volume **Default:** 100 GB

Local SSD defaults (ephemeral storage)

Allocate up to 24 local SSDs for ephemeral storage with the following considerations:

- There is an extra cost for each local SSD.
- All local SSD are 375G fixed size, with the option of SCSI or NVMe interface. An NVMe disk has twice the input/output operations per second (IOPs) compared to a SCSI disk.

Important

When the local SSD is in use, you cannot stop or start the instance or a cluster containing local SSD instances. If you do shut down an instance with a local SSD through the guest operating system, you will not be able to restart the instance and the data on the local SSD will be lost.

For details, see Google's Adding Local SSDs page.

The following table describes the default local SSD disk volume resources that Vertica provides:

Total Local SSDs	Instance Types	Catalog (Persistent)	Depot (Ephemeral)	Temp (Ephemeral)

4	n1-standard/highmem-16 n1-standard/highmem-32 n1-standard/highmem-64 n2-standard/highmem-16 n2-standard/highmem-32	Configurable 1 PD (Standard/SSD) volume **Default:** 50 GB	3 x 375 GB	1 x 375 GB
5	n1-standard/highmem-16 n1-standard/highmem-32 n1-standard/highmem-64	Configurable 1 PD (Standard/SSD) volume **Default:** 50 GB	4 x 375 GB	1 x 375 GB
6	n1-standard/highmem-16 n1-standard/highmem-32 n1-standard/highmem-64	Configurable 1 PD (Standard/SSD) volume **Default:** 50 GB	5 x 375 GB	1 x 375 GB
8	n1-standard/highmem-16 n1-standard/highmem-32 n1-standard/highmem-64 n2-standard/highmem-16 n2-standard/highmem-32 n2-standard/highmem-48 n2-standard/highmem-64	Configurable 1 PD (Standard/SSD) volume **Default:** 50 GB	6 x 375 GB	2 x 375 GB
16	n1-standard/highmem-16 n1-standard/highmem-32 n1-standard/highmem-64 n2-standard/highmem-16 n2-standard/highmem-32 n2-standard/highmem-48 n2-standard/highmem-64	Configurable 1 PD (Standard/SSD) volume **Default:** 50 GB	12 x 375 GB	4 x 375 GB
24	n1-standard/highmem-16 n1-standard/highmem-32 n1-standard/highmem-64 n2-standard/highmem-16 n2-standard/highmem-32 n2-standard/highmem-48 n2-standard/highmem-64	Configurable 1 PD (Standard/SSD) volume **Default:** 50 GB	20 x 375 GB	4 x 375 GB

Monitoring with MC

Management Console gathers and retains history of important system activities about your MC-managed database cluster, such as performance and resource utilization. You can use MC charts to locate performance bottlenecks on a particular node, to identify potential improvements to Vertica configuration, and as a reference for what actions users have taken on the MC interface.

> **Note**
> MC directly queries Data Collector tables on the MC-monitored databases themselves. See Management Console architecture . For how to set up MC to query an alternative database for monitoring data, see Extended monitoring .

The following list describes some of the areas you can monitor and troubleshoot through the MC interface:

- Multiple database cluster states and key performance indicators that report on the cluster's overall health
- Information on individual cluster nodes specific to resources
- Database activity in relation to CPU/memory, networking, and disk I/O usage
- Layout of subclusters , and resource utilization and query workload on subclusters. (Available in Eon mode databases only, where the database includes one default subcluster, and may include additional user-defined subclusters.)
- Query concurrency and internal/user sessions that report on important events in time
- Cluster-wide messages
- Database and agent log entries
- MC user activity (what users are doing while logged in to MC)
- Issues related to the MC process
- Error handling and feedback

About chart updates

MC retrieves statistical data from the production database to keep the charts updated. The charts also update dynamically with text, color, and messages that Management Console receives from the agents on the database cluster. This information can help you quickly resolve problems.

Each client session to MC uses a connection from MaxClientSessions , a database configuration parameter. This parameter determines the maximum number of sessions that can run on a single database cluster node. Sometimes multiple MC users, mapped to the same database account, are concurrently monitoring the Overview and Activity pages.

> **Tip**
> You can increase the value for MaxClientSessions on an MC-monitored database to account for extra sessions. See Managing sessions for details.

In this section

- Viewing the overview page
- Monitoring same-name databases with MC
- Monitoring cluster nodes with MC
- Monitoring node activity with MC
- Monitoring cluster performance with MC
- Monitoring cluster CPU and Memory with MC
- Monitoring database storage with MC
- Monitoring subscription status in Eon Mode
- Monitoring system resources with MC
- Monitoring resource pools with MC
- Monitoring database messages and alerts with MC
- Monitoring MC user activity using audit log
- Monitoring external data sources with MC
- Monitoring depot activity with MC
- Monitoring depot storage with MC
- Extended monitoring

Viewing the overview page

The Overview page displays a dynamic dashboard view of your database.

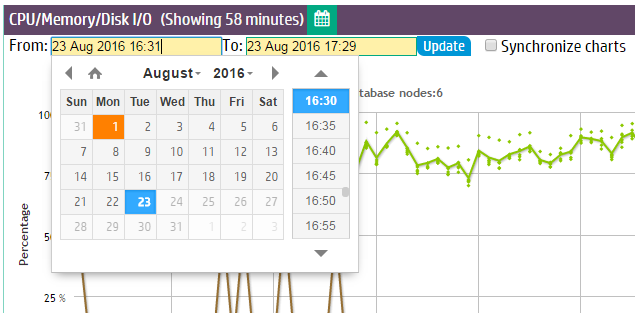
The page provides three tabs: Status Summary, System Health, and Query Synopsis. Access these tabs by clicking one of the three icons at the top left of the Overview page. Each tab contains charts and filters displaying information about your cluster. The QuickStats widgets on the right of the page display alerts and statistics about the state of your cluster.

Information on this page updates every two minutes, however you can adjust that value in the MC Settings page on the Monitoring tab. You can postpone updates by deselecting Auto Refresh in the toolbar.

Chart viewing options

You can specify time frames for some charts, which display a calendar icon in their title bars. Click the calendar icon to specify the time frame for that module.

On the Status Summary tab, you can select **Synchronize charts** to simultaneously apply the specified time frame to all charts on that tab.



If you have enabled extended monitoring on your database, MC can display longer ranges of data in certain charts. See Extended monitoring . If a chart is using extended monitoring data, the rocket ship icon appears in the title bar:



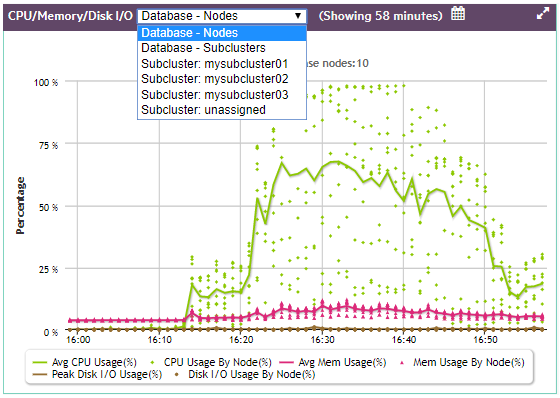
You can expand some charts to view them in larger windows. Click the expand icon in the title bar to do so:



Changing what the chart displays

The charts on the Overview page can display information about the nodes in your database, or the activity in all your database subclusters, in a single subcluster, or on the nodes that are not assigned to a subcluster. Use the dropdown in the title bar to select the type of information you want to display in the chart.

> **Note**
> The dropdown list in the CPU/Memory/Disk I/O chart below, and all other MC charts, appears only for Eon Mode databases, and only if subclusters are defined.

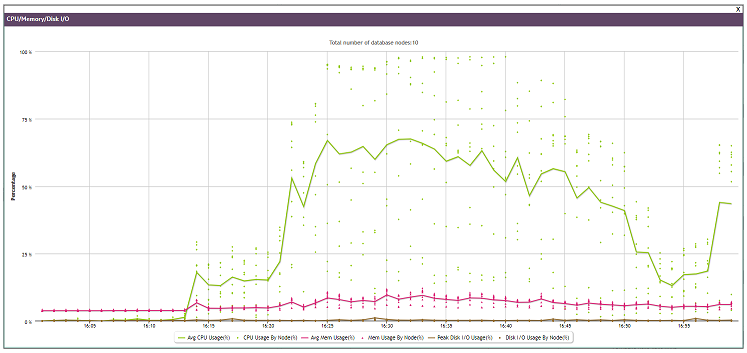


Zooming to show chart details

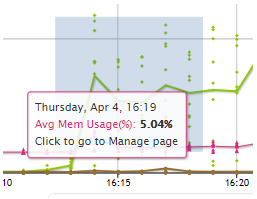
There are several steps you can take to show increasing levels of detail in a chart.

You can click the expand icon in the title bar to view the chart in a larger window:

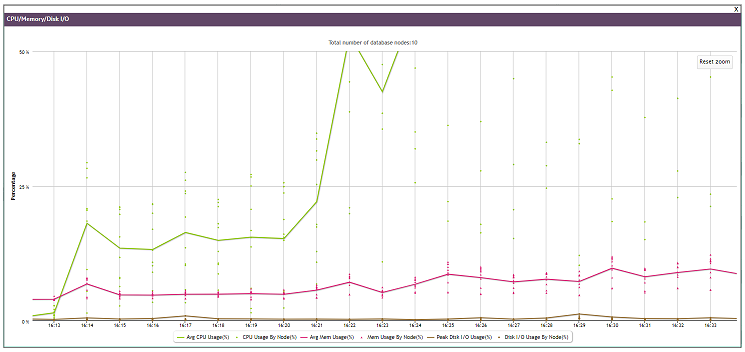




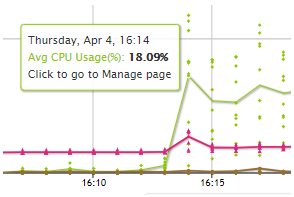
You can use the cursor to outline a small area you want to expand, shown as a gray rectangle below:



When you release the cursor, the detail area expands to full size:



Hover over any line or point on the chart to see details about those specific data points. This works before or after you expand the chart:



What the lines and dots on the chart represent

The legend below the CPU/Memory/Disk I/O chart explains what the lines and dots on the chart represent.

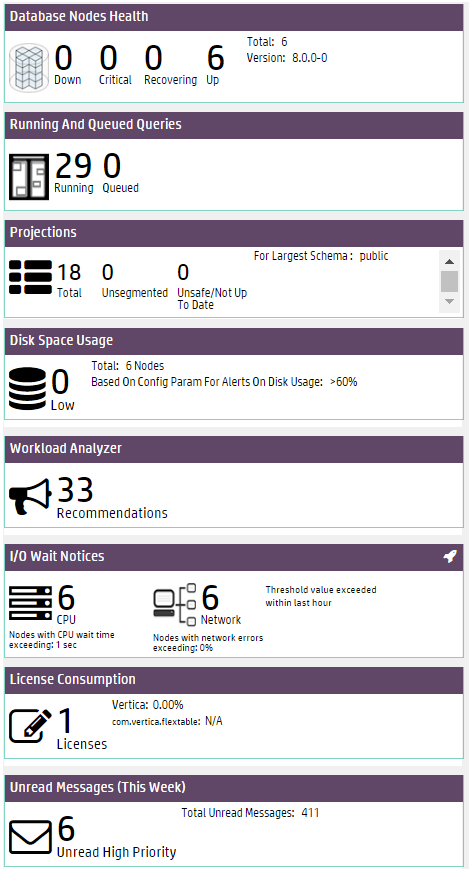
Each line represents the average of the nodes you selected in the dropdown. If you selected Database - Nodes, the line represents the average for all the nodes in the database. If you selected one subcluster, the line represents the average for the nodes in that subcluster.

Each dot represents an individual entity within your dropdown choice. If you chose Database - Nodes, each dot represents one node in the database. If you chose Database - Subclusters, each dot represents one subcluster in the database. If you chose a single subcluster or the unassigned subclusters, each dot represents an individual node within that set.

You can hover over any line or dot to see a summary about it. You can click on a dot to display the Node Details page for that dot.

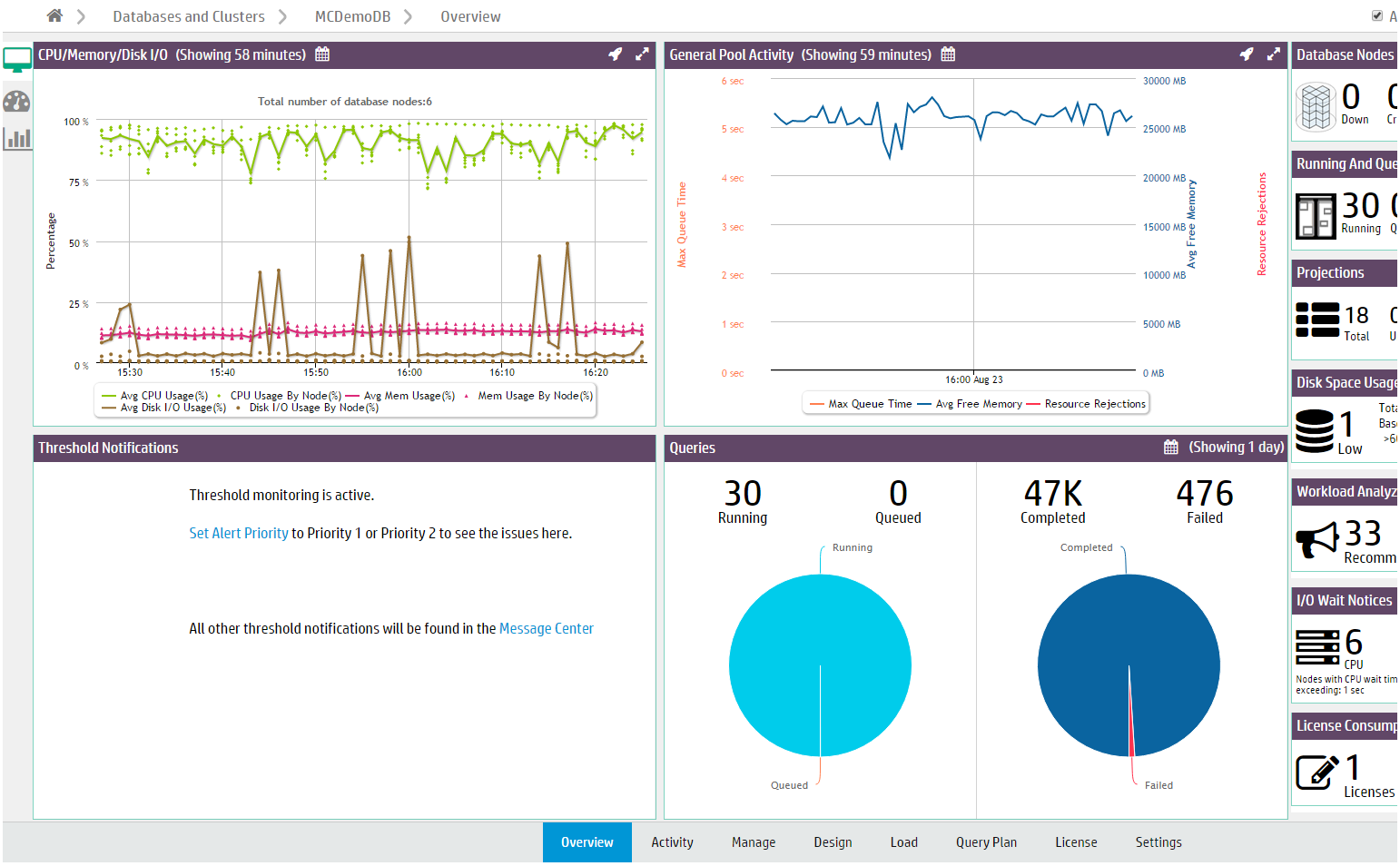
Quick stats

The Quick Stats sidebar on the right of the page provides instant alerts and information about your cluster's status.



- **Database Nodes Health** displays which nodes are down, critical, recovering, or up. Critical and recovering nodes are included in the total nodes considered "up" by the database. Click a node value to open the Manage page.
- **Running and Queued Queries** displays current queries in the database. Click the query values to open the Query Monitoring charts.
- **Projections** displays the number of total projections, unsegmented projections, and unsafe projections for the database schema with the most projections. Click a value to open the Table Treemap chart.
- **Disk Space Usage** alerts you to the number of nodes that are low on disk space. Click the value to go to the Manage page. On the Manage page, the Storage Used KPI View is displayed.
- **Workload Analyzer** analyzes system information retained in SQL system tables and provides tuning recommendations, along with the cost (low, medium, or high) of running the command. See Analyzing workloads for more information.
- **I/O Wait Notices** displays the number of nodes that, in the last hour, have recorded Disk I/O waits and Network I/O waits exceeding the wait threshold (1 second for Disk and 0 seconds for Network).
- **License Consumption** displays the number of licenses your database uses, and the percentage of your Vertica Community Edition or Premium Edition license being used.
- **Unread Messages** display the number of unread messages and alerts for the database. This count differs from the number of total messages across all your databases. Click the value to open the Message Center.

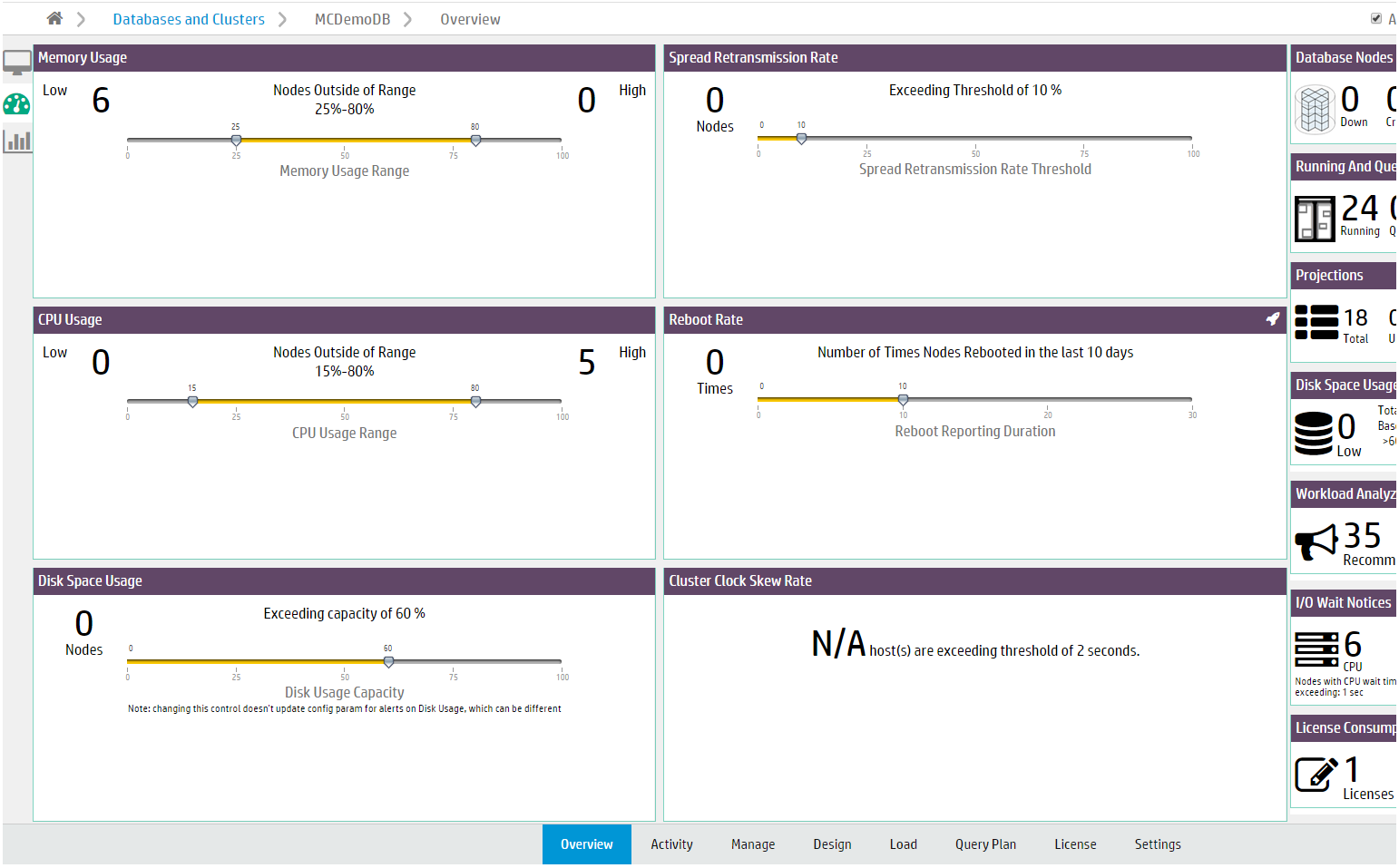
Status summary



The Status Summary tab displays four modules that provide a general overview of the status of your cluster:

- The **CPU/Memory/Disk I/O Usage** module shows cluster resource usage. The chart displays the number of nodes in the database cluster and plots average and per-node percentages for CPU, memory, and disk I/O usage.
  - Select a resource type from the legend to remove or add it from the chart display.
  - Click a data point (which represents a node) to open the Manage page. See Monitoring cluster CPU and Memory with MC .
- The **General Pool Activity** module displays GENERAL pool activity. The chart displays average query queue times, average GENERAL pool free memory, and resource rejections. Use this chart to see how much free memory there is in GENERAL pool, or if there have been high queue times.
  - Click the dropdown in the title bar to view the GENERAL pool usage for the entire database (the default), for a specific subcluster, or for the nodes not assigned to a subcluster.
  - Click the expand icon in the title bar to open the chart in a bigger window.
  - Click a data point to open the Resource Pools Monitoring chart. See Managing workloads .
- The **Thresholds Notifications** module displays alerts generated when a threshold has been exceeded in the database. Notifications are categorized by System Health and Performance.
  - In the module, you can acknowledge an alert (which marks it as read) or click the X to stop monitoring that threshold (which stops you receiving similar alerts in the future).
  - Customize thresholds and alert priorities for these notifications in the Thresholds tab of the database Settings page. See Customizing Message Thresholds.
- The **Queries** module displays query statistics. The first pie chart displays running and queued queries in the last 24 hours. The second chart displays completed and failed queries for the time frame you specify. Click a query count number above the chart to open the Query Monitoring chart. See Monitoring running queries with MC .

System health

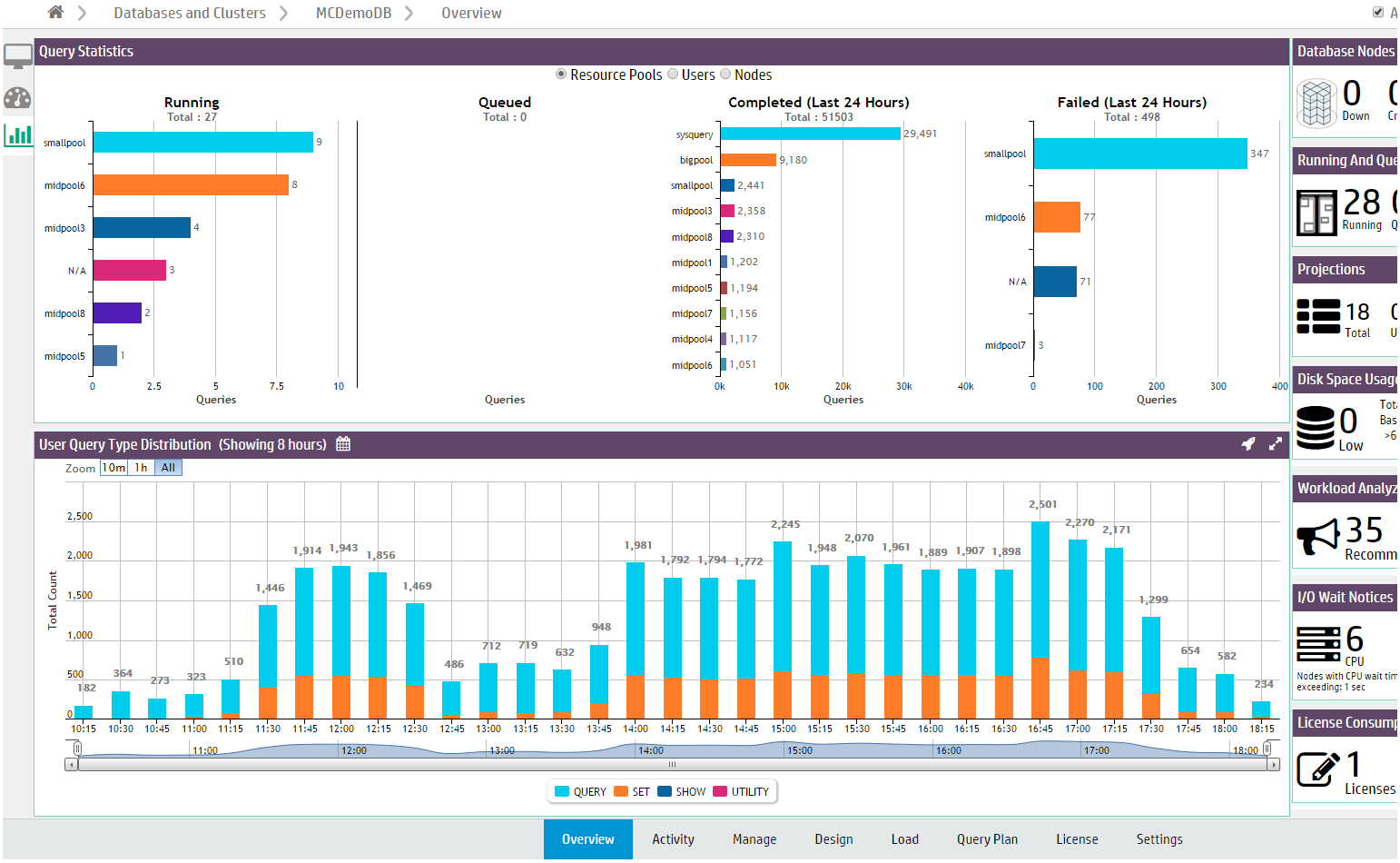


The System Health tab provides a summary of your system resource usage and node information, with filters that allow you to view resource usage within the ranges you specify.

> **Note**
> Note: Adjusting the filters on the System Health tab does not affect any database or MC settings.

- The **Memory Usage** filter displays the number of nodes with high and low memory usage. Move the sliders to adjust the memory usage range filter.
  For example, if you specify a range of 25% to 75% memory usage, the filter displays how many nodes are using less than 25% of memory (Low) and how many are using more than 75% (High). Hover your cursor over the Low and High values to see lists of what nodes fall, respectively, below or above the memory usage range you specified.
  Click a node value to go to the Manage page, which displays the Memory Utilization KPI View.
- The **Spread Retransmission Rate** filter displays the number of nodes with high spread retransmission rates. When a node's retransmission rate is too high, it is not communicating properly with other nodes. Move the slider to adjust the retransmission rate filter.
  Hover your cursor over the Nodes value to see a list of what nodes exceeded the spread retransmission rate you specified. Click the node value to view spread retransmit rate alerts in the Message Center.
- The **CPU Usage** chart displays the number of nodes with high and low CPU usage. Move the sliders to adjust the CPU usage range filter. Hover your cursor over the Low and High values to see lists of what nodes are below or above range you specified.
  Click a node value to go to the Manage page, which displays the CPU Utilization KPI View.
- The **Reboot Rate** filter displays the number of times nodes in the cluster have rebooted within the specified time frame. Use this filter to discover if nodes have gone down recently, or if there have been an unusual number of reboots. Move the slider to adjust the number of days. Hover over the Times value to see a list of the nodes that have rebooted and the times at which they did so.
- The **Disk Space Usage** filter displays the number of nodes with high disk space usage. Move the slider to adjust the disk usage filter. Hover your cursor over the Nodes value to see a list of what nodes exceed the acceptable range.
  Click the nodes value to go to the Manage page, which displays the Storage Used KPI View.
- The **Cluster Clock Skew Rate** module displays the number of nodes that exceed a clock skew threshold. Nodes in a cluster whose clocks are not in sync can interfere with time-related database functions, the accuracy of database queries, and Management Console's monitoring of cluster activity.

Query synopsis



The Query Synopsis page provides two modules that report system query activity and resource pool usage:

- The **Query Statistics** module displays four bar charts that provide an overview of running, queued queries, failed, and completed queries in the past 24 hours.
  - Select one of the options at the top of the module to group the queries by **Resource Pools** , **Users** , **Nodes** , or **Subclusters.**
  - Click a bar on the chart to view details about those queries the Query Monitoring activity chart.
- The **User Query Type Distribution** chart provides an overview of user and system query activity. The chart reports the types of operations that ran. The default is to display the types of operations that ran on all nodes in the database. Use the dropdown in the title bar to display the types of operations that ran on the nodes in a specific subcluster, or on the nodes not assigned to a subcluster.
  - Hover your cursor over chart points for more details.
  - Select a type of operation from the legend to remove or add it from the chart display.
  - To zoom to a certain time frame, you can adjust the sliders at the bottom of the chart.
  - Click a bar in the graph to open the Queries chart.

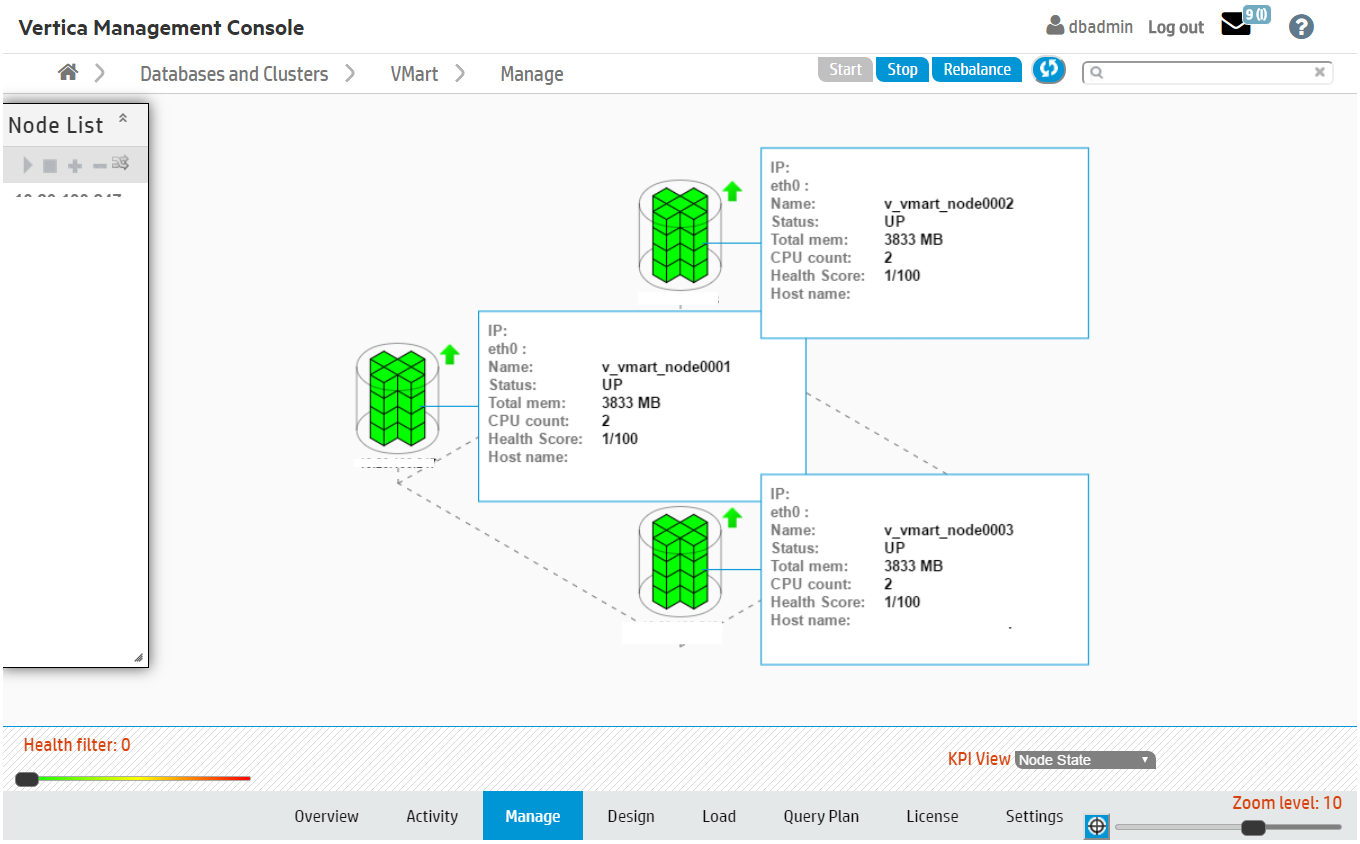
Monitoring same-name databases with MC

If you are monitoring two databases with identical names on different clusters, you can determine which database is associated with which cluster by clicking the database icon on MC's Databases and Clusters page to view its dialog box. Information in the dialog displays the cluster on which the selected database is associated.

Monitoring cluster nodes with MC

For a visual overview of all cluster nodes, click the running database on the Databases and Clusters page and click the **Manage** tab at the bottom of the page to open the cluster status page.

The cluster status page displays the nodes in your cluster.



The appearance of the nodes indicate the following states:

- **Healthy:** The nodes appear green.
- **Up:** A small arrow to the right of the node points upward.
- **Critical:** The node appears yellow and displays a warning icon to the right.
- **Down:** The node appears red. To the right of the node, a red arrow points downwards.
- **Unplugged:** An orange outlet and plug icon appears to the right. This is displayed when the MC cannot communicate with the agent running on the node.

You can get information about a particular node by clicking it, an action that opens the node details page.

Filtering what you see

If you have a large cluster, where it might be difficult to view dozens to hundreds of nodes on the MC interface, you can filter what you see. The Zoom filter shows more or less detail on the cluster overall, and the Health Filter lets you view specific node activity; for example, you can slide the bar all the way to the right to show only nodes that are down. A message next to the health filter indicates how many nodes in the cluster are hidden from view.

On this page, you can perform the following actions on your database cluster:

- Add, remove and replace nodes
- Rebalance data across all nodes
- Stop or start (or restart) the database
- Refresh the view from information MC gathers from the production database
- View key performance indicators (KPI) on node state, CPU, memory, and storage utilization (see Monitoring cluster performance with MC for details)

> **Note**
> Starting, stopping, adding, and dropping nodes and rebalancing data across nodes works with the same functionality and restrictions as those same tasks performed through the Administration tools .

If you don't see what you expect

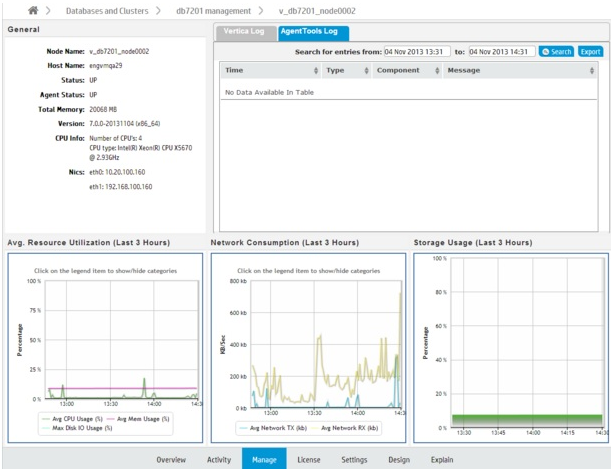
If the cluster grid does not accurately reflect the current state of the database (for example if the MC interface shows a node in INITIALIZING state, but when you use the Administration Tools to View Database Cluster State, you see that all nodes are UP), click the Refresh button in the toolbar. Doing so forces MC to immediately synchronize with the agents and update MC with new data.

Don't press the F5 key, which redisplays the page using data from MC and ignores data from the agent. It can take several seconds for MC to enable all database action buttons.

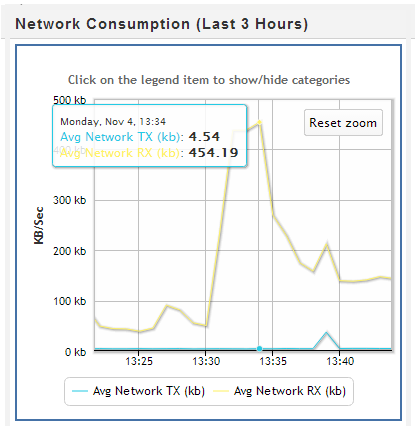
Monitoring node activity with MC

If a node fails on an MC-managed cluster or you notice one node is using higher resources than other cluster nodes—which you might observe when monitoring the Overview page —open the **Manage** page and click the node you want to investigate.

The Node Details page opens and provides summary information for the node (state, name, total memory, and so on), as well as resources the selected node has been consuming for the last three hours, such as average CPU, memory, disk I/O percent usage, network consumption in kilobytes, and the percentage of disk storage the running queries have been using. You can also browse and export log-level data from AgentTools and Vertica log files. MC retains a maximum of 2000 log records.



For a more detailed view of node activity, use the mouse to drag-select around a problem area in one of the graphs, such as the large spike in network traffic in the above image. Then hover over the high data point for a summary.



See also

- Viewing the overview page
- Monitoring cluster performance with MC

Monitoring cluster performance with MC

Key Performance Indicators (KPIs) are a type of performance measurement that let you quickly view the health of your database cluster through MC's **Manage** page. These metrics, which determine a node's color, make it easy for you to quickly identify problem nodes.

Metrics on the database are computed and averaged over the latest 30 seconds of activity and dynamically updated on the cluster grid.

How to get metrics on your cluster

To view metrics for a particular state, click the menu next to the **KPI View** label at the bottom of the Manage page, and select a state.

MC reports KPI scores for:

- **Node state** —(default view) shows node status (up, down, k-safety critical) by color; you can filter which nodes appear on the page by sliding the health filter from left to right
- **CPU Utilization** —average CPU utilization

- **Memory Utilization** —average percent RAM used
- **Storage Utilization** —average percent storage used

After you make a selection, there is a brief delay while MC transmits information back to the requesting client. You can also click **Sync** in the toolbar to force synchronization between MC and the client.

Node colors and what they mean

Nodes in the database cluster appear in color. Green is the most healthy and red is the least healthy, with varying color values in between.

Each node has an attached information dialog box that summarizes its score. It is the score's position within a range of 0 (healthiest) to 100 (least healthy) that determines the node's color *bias* . Color bias means that, depending on the value of the health score, the final color could be slightly biased; for example, a node with score 0 will be more green than than a node with a score of 32, which is still within the green range but influenced by the next base color, which is yellow. Similarly, a node with a score of 80 appears as a dull shade of red, because it is influenced by orange.

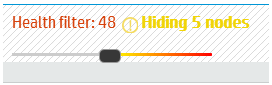
MC computes scores for each node's color bias as follows:

- 0-33: green and shades of green
- 34-66: yellow and shades of yellow
- 67-100: red and shades of red shades

If the unhealthy node were to consume additional resources, its color would change from a dull orange-red to a brighter red.

Filtering nodes from the view

The health filter is the slider in the lower left area of page. You can slide it left to right to show or hide nodes; for example, you might want to hide nodes with a score smaller that a certain value so the UI displays only the unhealthy nodes that require immediate attention. Wherever you land on the health filter, an informational message appears to the right of the filter, indicating how many nodes are hidden from view.



Filtering is useful if you have many nodes and want to see only the ones that need attention, so you can quickly resolve issues on them.

Monitoring cluster CPU and Memory with MC

On the MC Overview page, the **CPU/Memory** subsection provides a graph-based overview of cluster resources during the last hour, which lets you quickly monitor resource distribution across nodes.

This chart plots average and per-node percentages for both CPU and memory with updates every minute—unless you clear Auto Refresh Charts in the toolbar. You can also filter what the chart displays by clicking components in the legend at the bottom of the subsection to show/hide those components. Yellow data points represent individual nodes in the cluster at that point in time.

Investigating areas of concern

While viewing cluster resources, you might wonder why resources among nodes become skewed. To zoom in, use your mouse to drag around the problem area surrounding the time block of interest.

After you release the mouse, the chart refreshes to display a more detailed view of the selected area. If you hover your cursor over the node that looks like it's consuming the most resources, a dialog box summarizes that node's percent usage.

For more information, click a data point (node) on the graph to open MC's node details page. To return to the previous view, click **Reset zoom** .

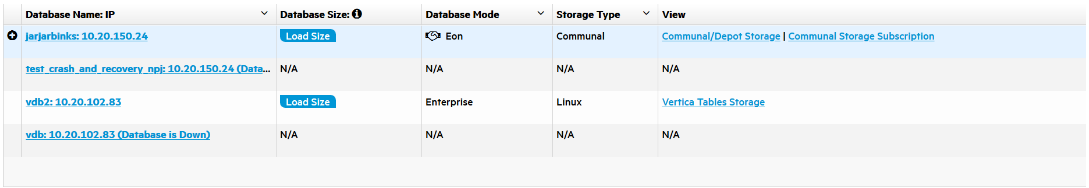
See also

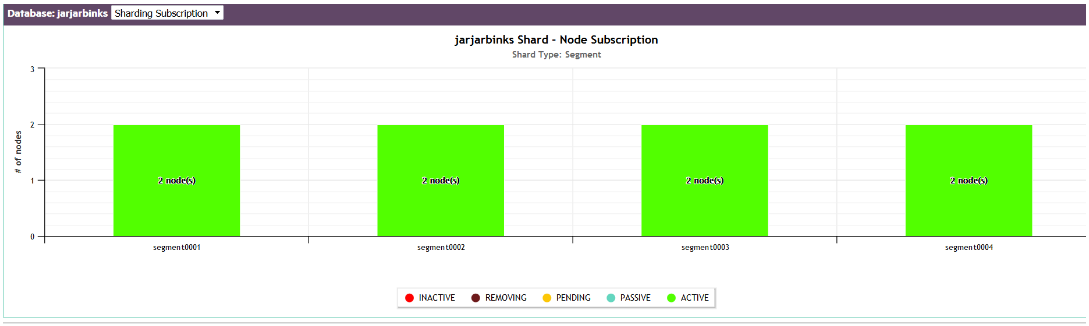
- Monitoring node activity with MC

Monitoring database storage with MC

The Infrastructure page's **Storage View** provides a summary of the amount of data stored in your database, and the persistent location of that data. Use this view to monitor how much of your storage capacity your databases are using.

For a database running in Eon Mode, MC also displays bar charts in the Storage View that illustrate shard subscription status. Use these charts to determine if your current subscription layout is optimal for querying your Eon Mode database. For information about using subscription status charts, see Monitoring subscription status in Eon Mode .





Monitor storage usage

The storage summary table lists all databases currently monitored by MC and information about their storage:

- **Database Size** . Click **Load Size** to calculate the total size of the database.
- **Database Mode** . Vertica databases run in Enterprise Mode, or Eon Mode .
- **Storage Type** . Enterprise Mode databases list the OS of the local nodes where data is stored. Eon Mode databases list the type of communal storage location where it stores its data. Eon Mode currently supports only S3-compatible storage locations.
- **View** . The options displayed in this column depend on the database mode and type of data on the database.
  - Vertica **Tables Storage** : For Enterprise Mode databases only. Click for a dialog listing the node and local directories where Vertica table data is stored.
  - **Communal/Depot Storage** : For Eon Mode databases only. Click for a dialog displaying location paths for your depot and communal storage.
  - **Communal Storage Subscription** : For Eon Mode databases only. Click to view bar charts at the bottom of the Storage View page, illustrating shard subscription status. For more about these charts, Monitoring subscription status in Eon Mode .
  - **External Tables** : Available when there are external tables in your database. Click for a dialog displaying details about all external tables. (Also see Monitoring table utilization and projections with MC .)
  - **HCatalog Details** : Available when your Vertica database has access to Hive tables. (See Using the HCatalog Connector .) Click for a dialog displaying details about HCatalog schemas. For any HCatalog schema, click View Tables for details about all tables accessible through that schema. (Also see Monitoring table utilization and projections with MC .)

In front of Eon Mode database names in the list, a plus icon displays. Click the icon to expand more details about the database's depot capacity and usage. The depot is cache-like storage where Eon Mode databases keep local copies of communal storage data for faster query access.

- Click **Percentage Used** to view the Depot Activity chart for that database.
- Click **View Depot Details by Nodes** to see a dialog displaying location paths and depot usage information.

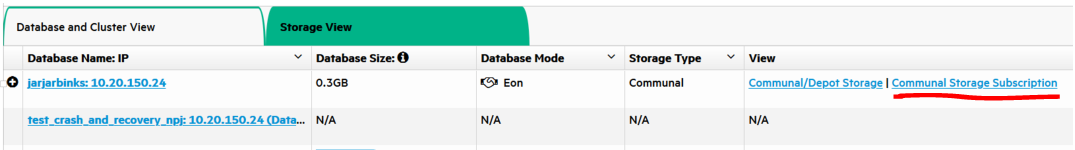
See also

- Eon Mode
- Eon Mode architecture
- Monitoring depot activity with MC
- Monitoring subscription status in Eon Mode
- Monitoring table utilization and projections with MC
- Working with external data
- Using the HCatalog Connector

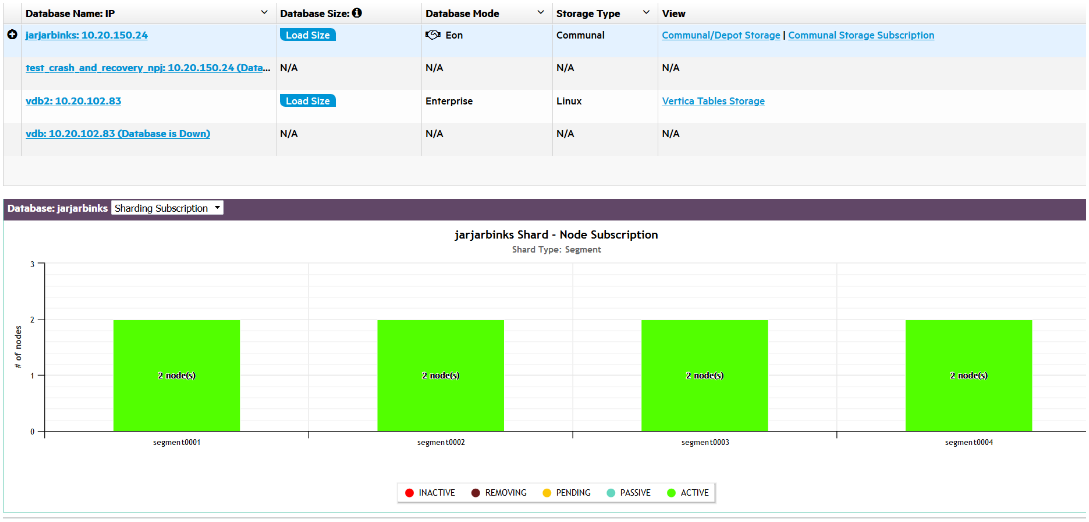
Monitoring subscription status in Eon Mode

To view subscription charts for any Eon Mode database you monitor, click **View Your Infrastructure** on the MC Home page. Then click the **Storage View** tab.

Click the **Details** action for that database in the storage summary list (highlighted in red in the image below).



When you click **Details** , two charts become available on the bottom half of the page: The Sharding Subscription chart, and the Node Subscription chart. You can switch between these two charts using the drop down menu to the right of the chart title.



Why monitor shard and node subscriptions?

Shards are segments of the data that is stored persistently in your Eon Mode database's communal storage location, for example Amazon S3 in the cloud or PureStorage if your cluster is on premises. Each node in the database subscribes to a subset of those shards. In this way, the node gets updated on when to populate its depot with new data from communal storage. (See Shards and subscriptions .)

For K-safety in an Eon Mode database, shards should have multiple node subscribers to ensure that even if a node goes down or is being used by another query, the data on that shard is still available on other nodes. If a shard has no node subscribers, that could indicate that data loss is occurring.

Subscriptions go through several transitions, which are illustrated by colors in the subscription charts:

- **Pending (Yellow).** The node is ready to subscribe to a certain shard. It cannot yet serve queries because it is not actively subscribed to the shard yet.
- **Passive (Blue/Teal).** The node could potentially serve queries for a shard it is passively subscribed to, but its depot contents for that shard may not yet be up to date, which could negatively impact query performance. The passively subscribed node is waiting for an active node subscriber of the shard to send it the most recent data.
- **Active (Green).** The node is actively subscribed to the shard, can load new data from communal storage, and can serve queries for data in that shard. The actively subscribed node sends data from that shard to other subscribed nodes.
- **Removing (Dark Red/Maroon).** The node is unsubscribing from the shard. It may have the most recent data from that shard, but that state is temporary until data from that shard is cleaned up.
- **Inactive (Red).** The subscribed node is down. It can no longer serve queries for that shard.

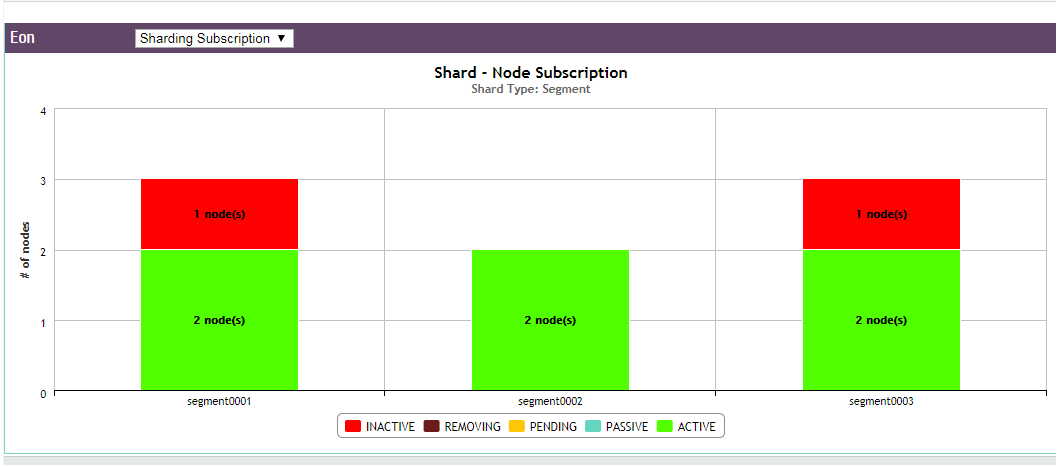
Operations such as adding or removing nodes or rebalancing shards can change which nodes subscribe to which shards. Shard subscription changes can prevent object-level restore from backups, though full restore is always possible. If shard subscriptions change, consider making a backup with the new configuration.

Monitor sharding subscription

The Sharding Subscription chart displays how many nodes are subscribed to each shard in your database, and what type of subscription it is.

You can hover over any bar in the chart to see which nodes are subscribed to the shard. Click on a subscription type in the legend to show or hide it in the chart display.

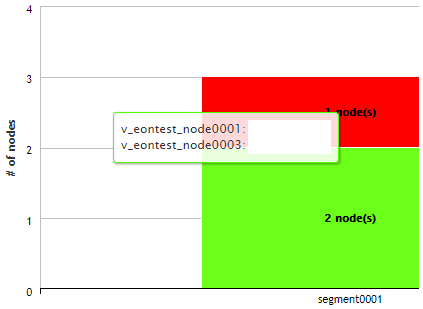
The example below shows the shard subscription status for a running Eon Mode database. The database has three nodes that are up, and one node (Node 4) that has been added to the cluster, but is down.



You can hover over any bar in the chart to see which nodes are subscribed to the shard. In this example, nodes 1 and 3 have active subscriptions to the first shard (green); nodes 1 and 2 to the second shard; and nodes 2 and 3 to the third shard.

The active subscriptions are evenly spread across the shards. This is a k-safe Eon Mode database.

Node 4 was subscribed to two shards; however, because it is down, its subscriptions to the shards are now inactive (red).



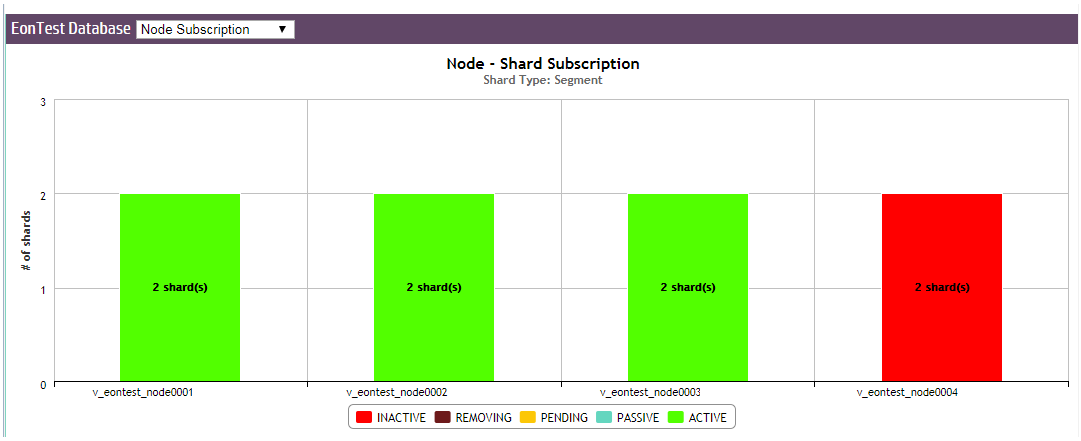
Monitor node subscriptions

Use this chart to view how many shards each node in your database is subscribed to, and the state of those subscriptions. The number of shards each node is subscribed to should be about the same to prevent overworking any given node.

Hover over any bar to see the shards it is subscribed to. The color of the bar indicates the state of each subscription. Click on a subscription type in the legend to show or hide it in the chart display.

The example below shows the same database from the Sharding Subscription example above. Nodes 1 through 3 are each actively subscribed to two shards (green). At least two nodes are subscribed to every shard in the database (which you can double check using the Sharding Subscription chart), ensuring that even if one of the nodes is down or being used in a query, another node is still actively subscribed and can access the data of that shard.

Since Node 4 is down, the chart shows that both its shard subscriptions are now inactive.



See also

- Shards and subscriptions
- Elasticity

Monitoring system resources with MC

MC's **Activity** page provides immediate visual insight into potential problem areas in your database's health by giving you graph-based views of query and user activity, hardware and memory impact, table and projection usage, system bottlenecks, and resource pool usage.

Select one of the following charts in the toolbar menu:

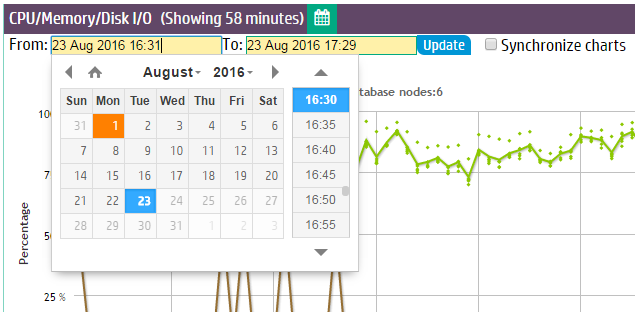
- Queries
- Internal Sessions
- User Sessions
- Memory Usage
- System Bottlenecks
- User Query Phases
- Monitoring table utilization and projections with MC
- Query Monitoring
- Resource Pool Monitoring
- Monitoring catalog memory with MC

How up to date is the information?

System-level activity charts automatically update every five minutes, unless you clear Auto Refresh in the toolbar. Depending on your system, it could take several moments for the charts to display when you first access the page or change the kind of resource you want to view.

Chart viewing options

You can specify time frames for some charts, which display a calendar icon in their title bars. Click the calendar icon to specify the time frame for that module.



If you have enabled extended monitoring on your database, MC can display longer ranges of data in certain charts. See Extended monitoring . If a chart is using extended monitoring data, the rocket ship icon appears in the title bar:

You can expand some charts to view them in larger windows. Click the expand icon in the title bar to do so:

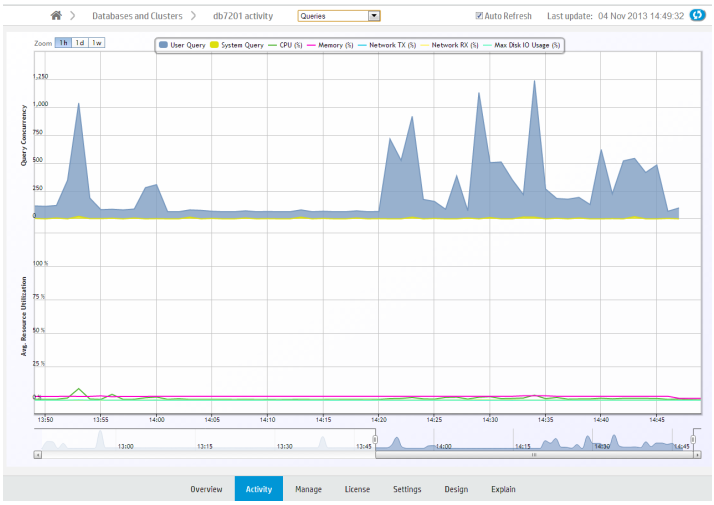


In this section

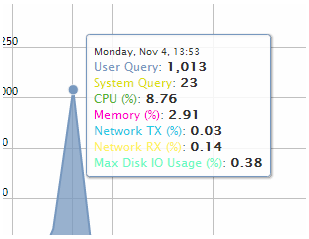
- Monitoring query activity with MC
- Monitoring internal sessions with MC
- Monitoring user sessions with MC
- Monitoring system memory usage with MC
- Monitoring system bottlenecks with MC
- Monitoring user query phases with MC
- Monitoring table utilization and projections with MC
- Monitoring running queries with MC
- Monitoring catalog memory with MC

Monitoring query activity with MC

The Queries chart displays information about query concurrency and average resource usage for CPU/memory, network activity, and disk I/O percent based on maximum rated bandwidth.



Hover over a data point for more information about percent usage for each of the resource types.



If you click a data point, MC opens a details page for that point in time, summarizing the number of user queries and system queries. This page can help you identify long-running queries, along with the query type. You can sort table columns and export the report to a file.

Monitoring key events

On the main Queries page, MC reports when a key event occurred, such as a Workload Analyzer or rebalance operation, by posting a **Workload Analyzer** (Workload Analyzer) and/or **RBL** (rebalance) label on the resource section of the chart.

Filtering chart results

The default query concurrency is over the last hour. The chart automatically refreshes every five minutes, unless you clear the Auto Refresh option in the toolbar. You can filter results for 1 hour, 1 day, or up to 1 week, along with corresponding average resource usage. You can also click different resources in the legend to show or hide those resources.

To return to the main Queries page, use the slider bar or click the 1h button.

Viewing more detail

To zoom in for detail, click-drag the mouse around a section or use the sliding selector bar at the bottom of the chart. After the detailed area displays, hover your cursor over a data point to view the resources anchored to that point in time.

For more detail about user or system queries, click a data point on one of the peaks. A **Detail** page opens to provide information about the queries in tabular format, including the query type, session ID, node name, query type, date, time, and the actual query that ran.

The bottom of the page indicates the number of queries it is showing on the current page, with Previous and Next buttons to navigate through the pages. You can sort the columns and export contents of the table to a file.

To return to the main Queries page, click **<database name> Activity** in the navigation bar.

Monitoring internal sessions with MC

The Internal Sessions chart provides information about Vertica system activities, such as Tuple Mover and rebalance cluster operations, along with their corresponding resources, such as CPU/memory, networking, and disk I/O percent used.

Hover your cursor over a bar for more information. A dialog box appears and provides details.

Filtering chart results

You can filter what the chart displays by selecting options for the following components. As you filter, the **Records Requested** number changes:

- Category: Filter which internal session types (mergeout, rebalance cluster) appear in the graph. The number in parentheses indicates how many sessions are running on that operation.

- Session duration: Lists time, in milliseconds, for all sessions that appear on the graph. The minimum/maximum values on which you can filter (0 ms to *n* ms) represent the minimum/maximum elapsed times within all sessions currently displayed on the graph. After you choose a value, the chart refreshes to show only the internal sessions that were greater than or equal to the value you select.
- Records requested: Represents the total combined sessions for the Category and Session Duration filters.

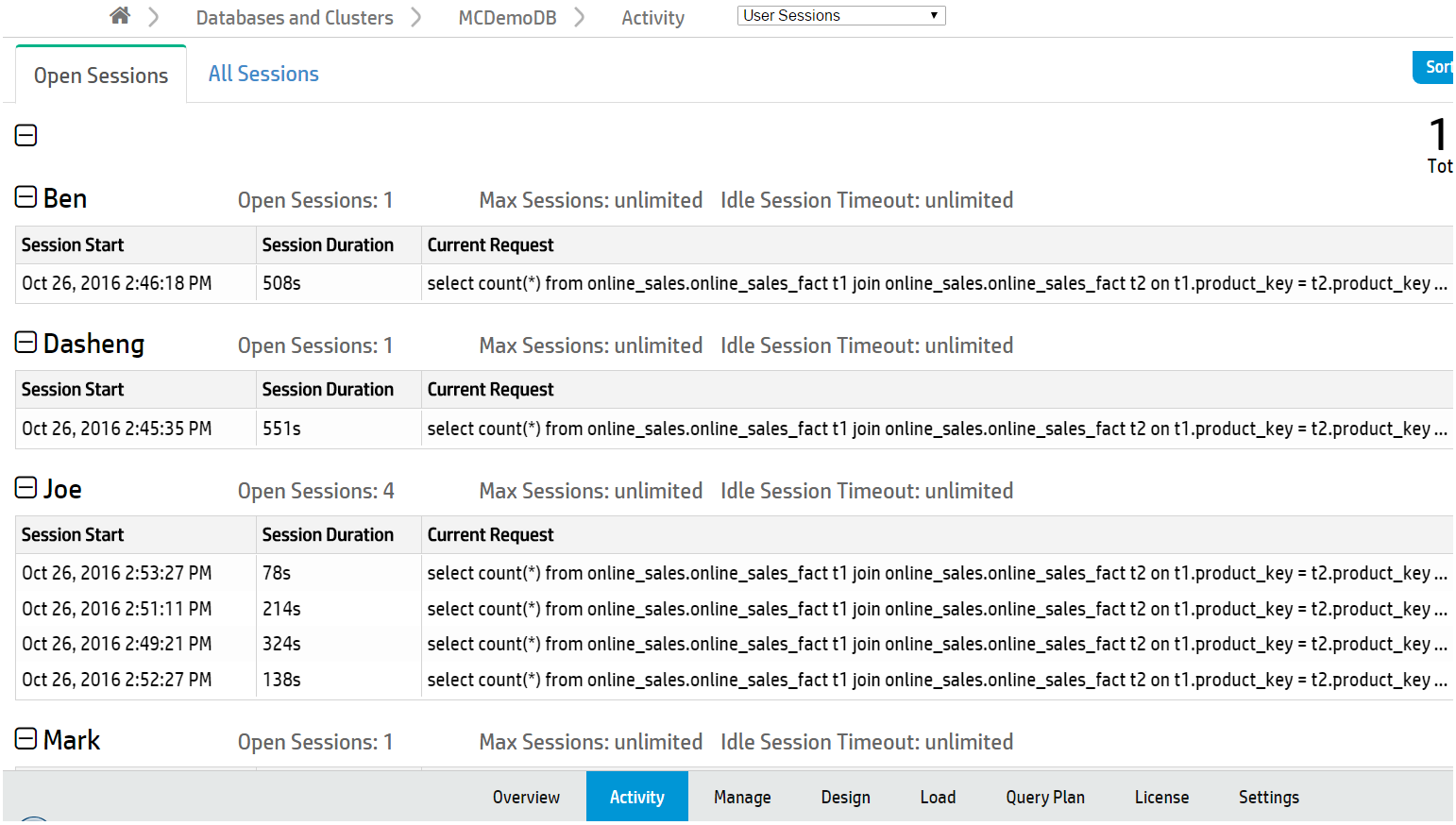
Monitoring user sessions with MC

The User Sessions charts provide information about Vertica user activities for all user connections open to MC.

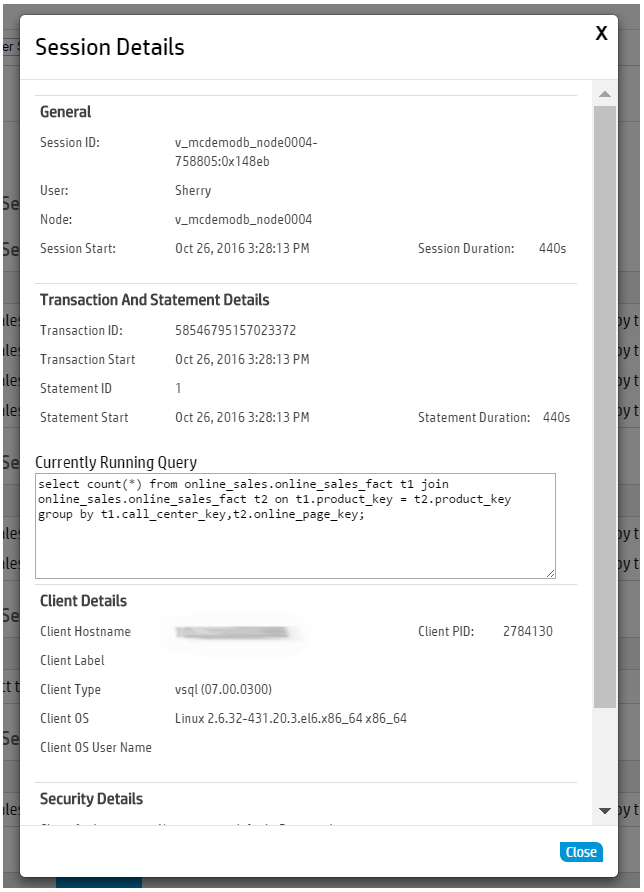
Choose **User Sessions** from the menu at the top of your database's Activity page to view these charts.

View open sessions

The Open Sessions tab displays a table of currently open sessions for each user. You can close a session or cancel a query on this tab by selecting that option from the **Actions** column.

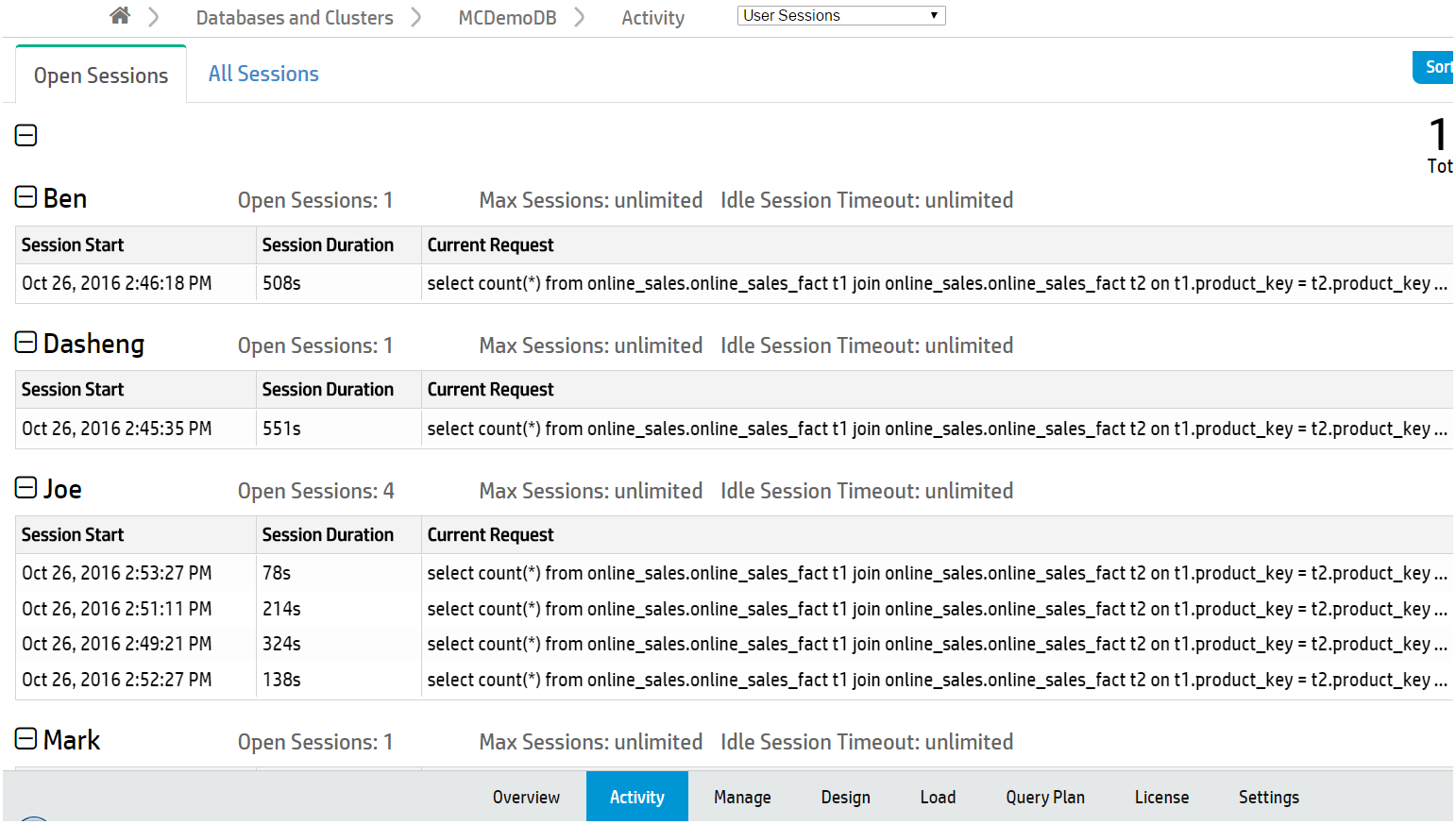


Click any row to open a **Session Details** dialog that shows more extensive information about that session.



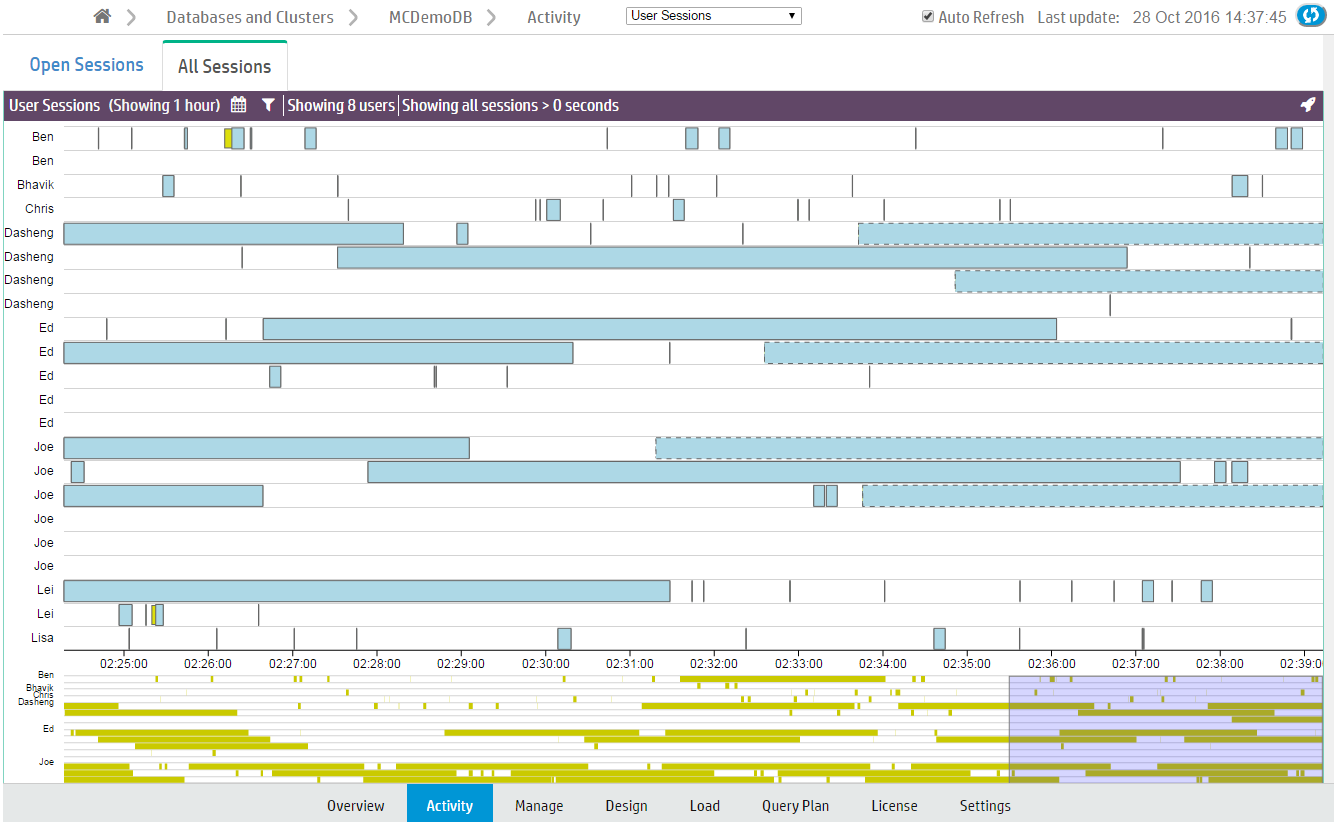
To configure the Open Sessions page display:

- Use the **Sort Users** button at the top right of the page to sort by user name or number of open sessions.
- Use the **Toggle Columns** button at the top right of the page to select which columns to display. Each table displays session information by column, such as the session start time or the



View all user sessions

The All Sessions tab displays a history of all user sessions in a swim lane chart.



What chart colors mean

Bars outlined with a dotted line are currently running sessions.

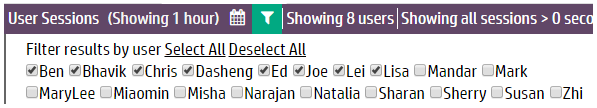
Sessions are divided into two colors, yellow and blue.

- Yellow bars represent user (system) sessions. If you click a yellow bar, MC opens a Detail page that shows all queries that ran or are still running within that session.
- Blue bars represent user requests (transactions within the session). If you click a blue bar in the graph, MC opens a Detail page that includes information for that query request only.
  When you hover your mouse over a transaction bar, a dialog box provides summary information about that request, such as which user ran the query, how long the transaction took to complete, or whether the transaction is still running.

Filter chart results

Extremely busy systems will show a lot of activity on the interface, perhaps more than you can interpret at a glance. You can filter chart results in several ways:

- **Zoom.** The context chart at the bottom of the page highlights in blue which section of the All Sessions chart you are viewing. Click and drag the blue box left or right to view earlier or later user sessions. Click and drag the edges of the blue box to zoom in or out.
- **Select fewer users.** Click the filter icon ( ) at the top of the page. A menu of a menu of all available users appears below. Deselect users to exclude from the chart.



- **Change the session duration (how long a session took to run).** Click the Filter icon ( ) at the top of the page. The **Filter sessions and queries by duration** field appears below. Enter the minimum session length (in seconds) to display on the chart and click **Update** .



- **Specify a time frame.** Click the Calendar icon ( ) at the top of the page to display the From and To fields. Using the fields, select the time frame to display in the chart and click **Update** .

Monitoring system memory usage with MC

The Memory Usage chart shows how system memory is used on individual nodes over time. Information the chart displays is stored based on Data collector retention policies, which a superuser can configure. See Configuring data retention policies .

The first time you access the Memory Usage chart, MC displays the first node in the cluster. MC remembers the node you last viewed and displays that node when you access the Activity page again. To choose a different node, select one from the Nodes drop-down list at the bottom of the chart. The chart automatically refreshes every five minutes unless you disable the Auto Refresh option.

> **Tip**
> On busy systems, the Node list might cover part of the graph you want to see. You can move the list out of the way by dragging it to another area on the page.

Types of system memory

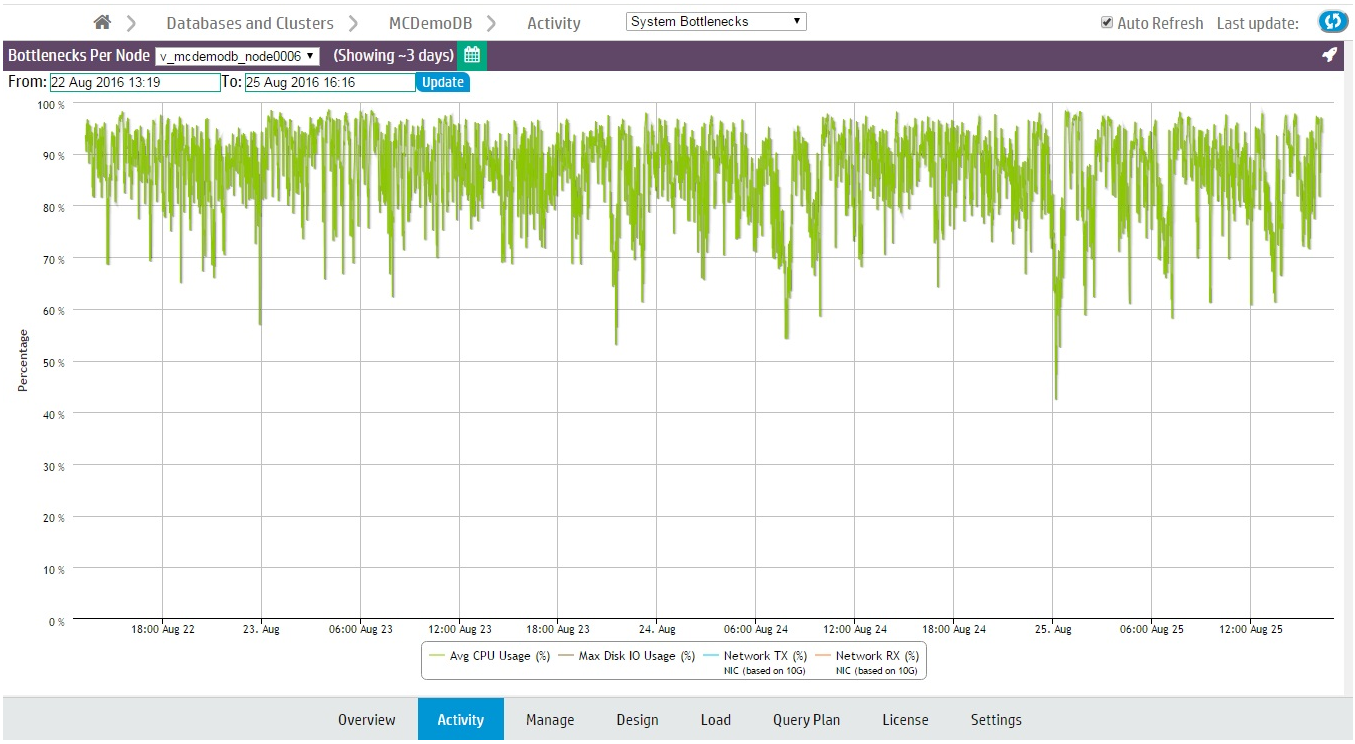
The Memory Usage chart displays a stacking area for the following memory types:

- swap
- free
- fcache (file cache)
- buffer
- other (memory in use by all other processes running on the system besides the main Vertica process, such as the MC process or agents)
- Vertica
- rcache (Vertica ROS cache)
- catalog

When you hover over a data point, a dialog box displays percentage of memory used during that time period for the selected node.

Monitoring system bottlenecks with MC

The System Bottlenecks chart helps you quickly locate performance bottlenecks on a particular node. The first time you access the Activity page, MC displays the first node in the cluster. To choose a different node, select one from the Nodes drop-down list at the bottom of the chart.

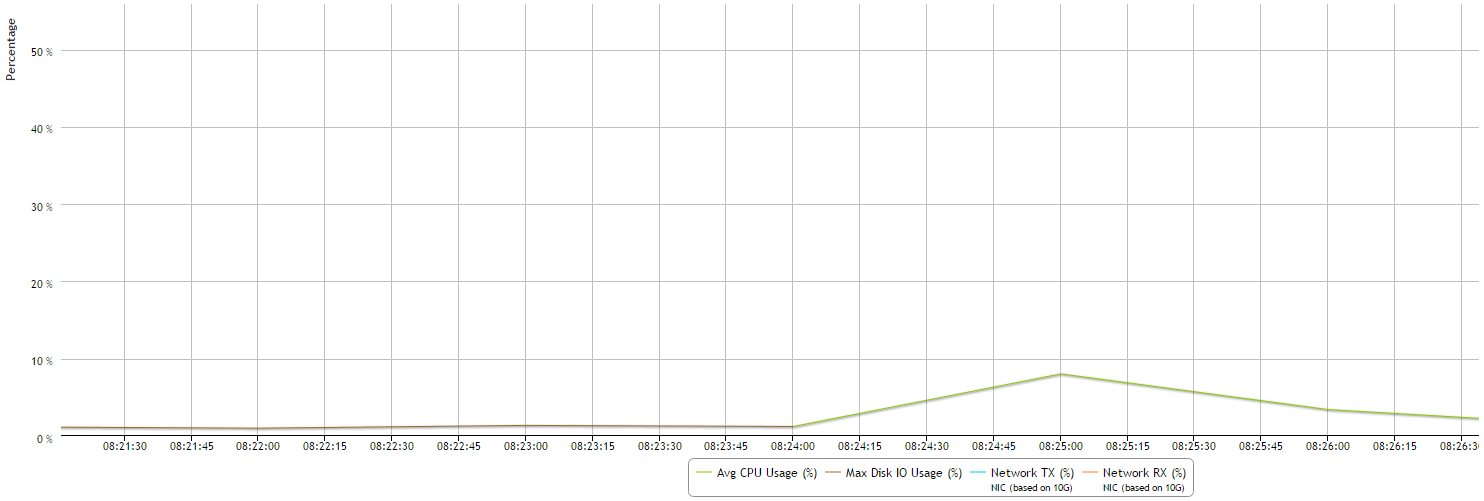


The System Bottlenecks chart reports what MC identifies as the most problematic resource during a given time interval. You can use this chart as a starting point for investigation.

How MC gathers system bottleneck data

Every 15 minutes, MC takes the maximum percent values from various system resources and plots a single line with a data point for the component that used the highest resources at that point in time. When a different component uses the highest resources, MC displays a new data point and changes the line color to make the change in resources obvious. Very busy databases can cause frequent changes in the top resources consumed, so you might notice heavy chart activity.

In the following example, at 08:24 the maximum resources used changed from Disk I/O to CPU. The System Bottlenecks charts denotes this with a change in line color from brown to green.



The components MC reports on

MC reports maximum percent values for the following system components:

- Average percent CPU usage
- Average percent memory usage
- Maximum percent disk I/O usage
- Percent data sent over the network (TX)
- Percent data received over the network (RX)

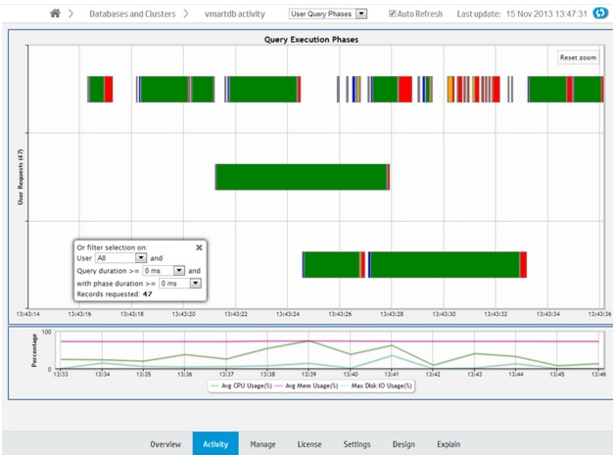
How MC handles conflicts in resources

If MC encounters two metrics with the same maximum percent value, it displays one at random. If two metrics are very close in value, MC displays the higher of the two.

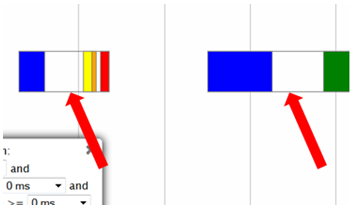
Monitoring user query phases with MC

The User Query Phases chart provides information about the query execution phases that a query goes through before completion. Viewing this chart helps you identify at a glance queries possibly delayed because of resource contention.

> **Note**
> For a list of query phases, see Query phase duration .



Each bar, bound by a gray box, represents an individual query. Within a query, a different color represents each query phase. The chart does not show phases for queries with durations of less than 4 seconds. Blank spaces within a query represent waiting times, as in the image below.

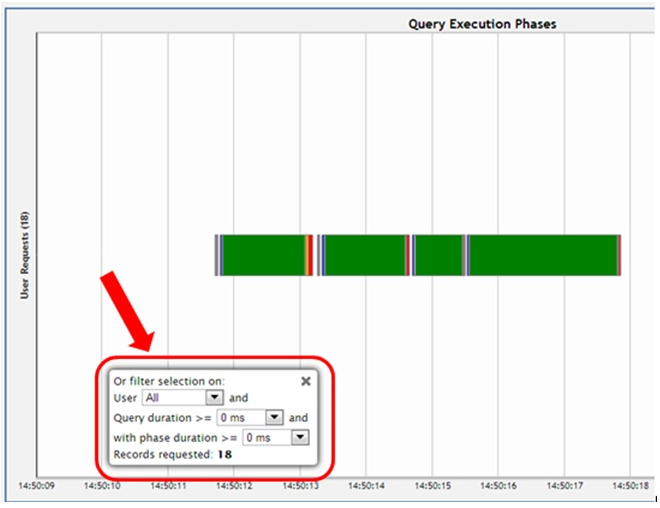


Hover over a phase in the query for information on the phase type and duration.

The chart shows queries run over the last 15 minutes. The chart automatically refreshes every five minutes, unless you clear the Auto Refresh option in the toolbar.

Filtering chart results

You can filter what the chart displays by selecting options for the user running the query, minimum query duration, and minimum phase duration.



Viewing more detail

To zoom in for detail, click-drag the mouse around a section of the chart. Click Reset Zoom, located at the top right corner of the chart, to restore the chart to its original view.

For more detail, click a query bar. The Detail page opens to provide information about the queries in tabular format, including the query type, session ID, node name, query type, date, time, the actual query that ran, and an option to run Explain Plan or profile the query. Click a table column header to sort the queries by that category.

To export the contents of the table to a file, click Export, located at the upper right of the page.

To return to the main Queries page, click Activity in the navigation bar.

Monitoring table utilization and projections with MC

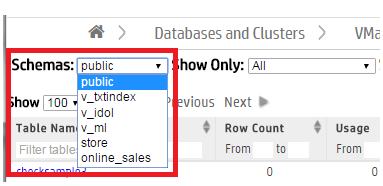
The Table Utilization activity page helps you monitor tables and projections in your database by schema.

Use the Table Utilization charts for a listing of all the tables in a schema, which you can filter and sort; or view them by size and usage in a treemap visualization. These charts allow you to identify outliers among your tables, such as those that are large or overused.

The Projections Summary , located on the right side of the page, provides an overview of the projections in the schema. You can use this summary to help identify if projections are evenly distributed across nodes.

Visualize tables

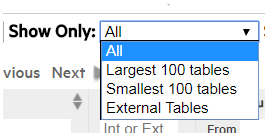
MC shows you the public schema by default. To specify which schema to view, choose one from the **Schemas** menu at the top of the activity page. The summary of tables and projections in that schema appear on the page.



MC visualizes your available tables by schema in a table chart or as a treemap chart . From the **Show As** menu, choose **Table** for a tabular chart or **Map** for a treemap chart. By default, MC displays the Table chart.



Depending on the number of tables in the schema, the chart may be crowded. To narrow it down, use the **Show Only** filter at the top of the page to display only the largest 100 tables, smallest 100 tables, or external tables.



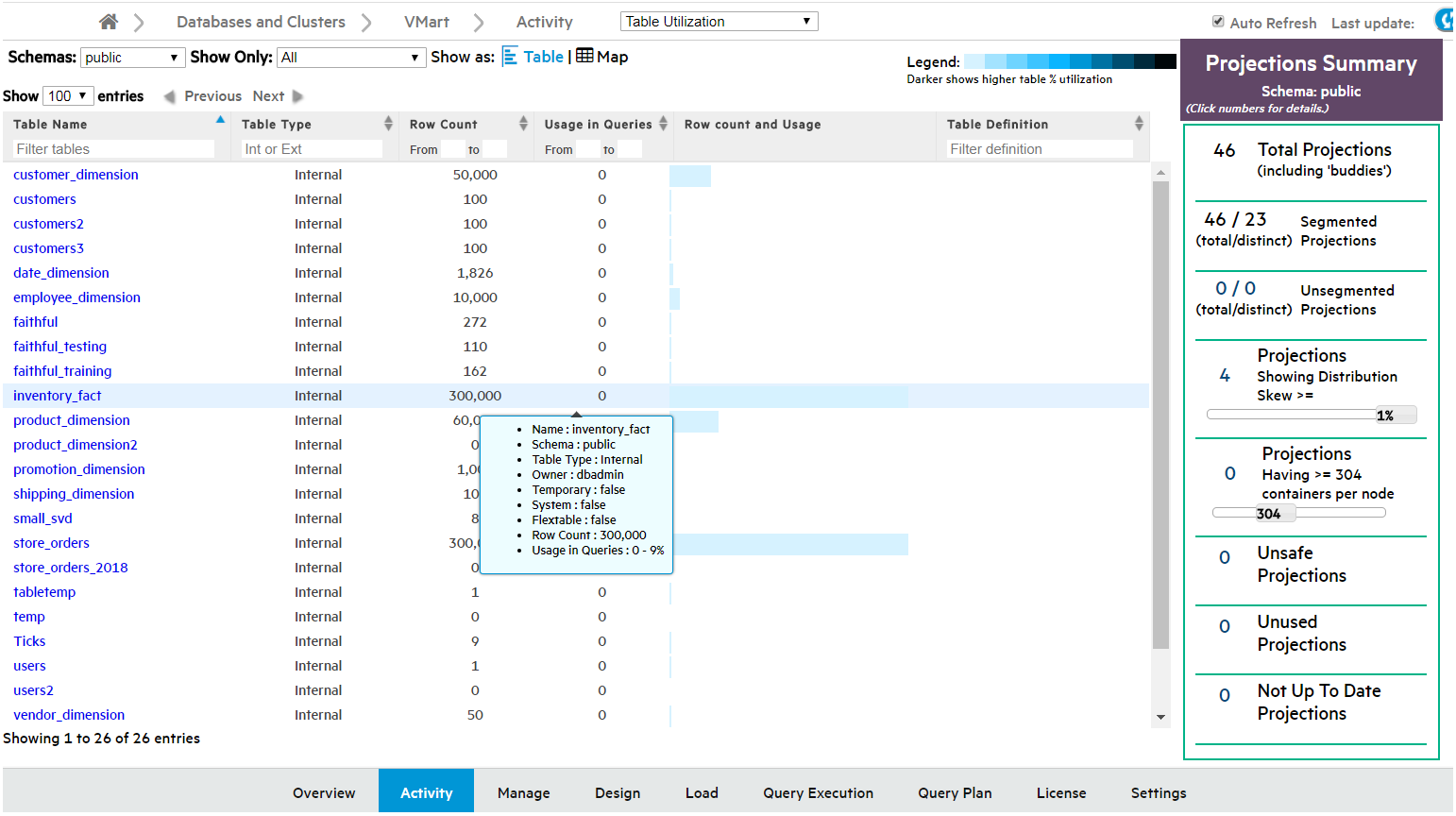
View the table chart

The Table chart is a tabular view of the schema's table data. Use the tabular view to filter or sort on any columns, and view the explicit values for row counts and utilization.

The columns display each table's:

- **Table Name** . Click this name to see the Table Details page.
- **Table Type** : Internal, Working with external data or HCatalog . (Details such as row count and usage are not available for external and HCatalog types.)
- **Row Count** .
- **Usage in Queries** , by percentage of time the table is queried.
- **Row count and Usage** , visualized as a bar. The length of the bar indicates row count; a darker color indicates higher usage.
- **Table Definition** . The COPY statement table definition, only applicable to external tables.

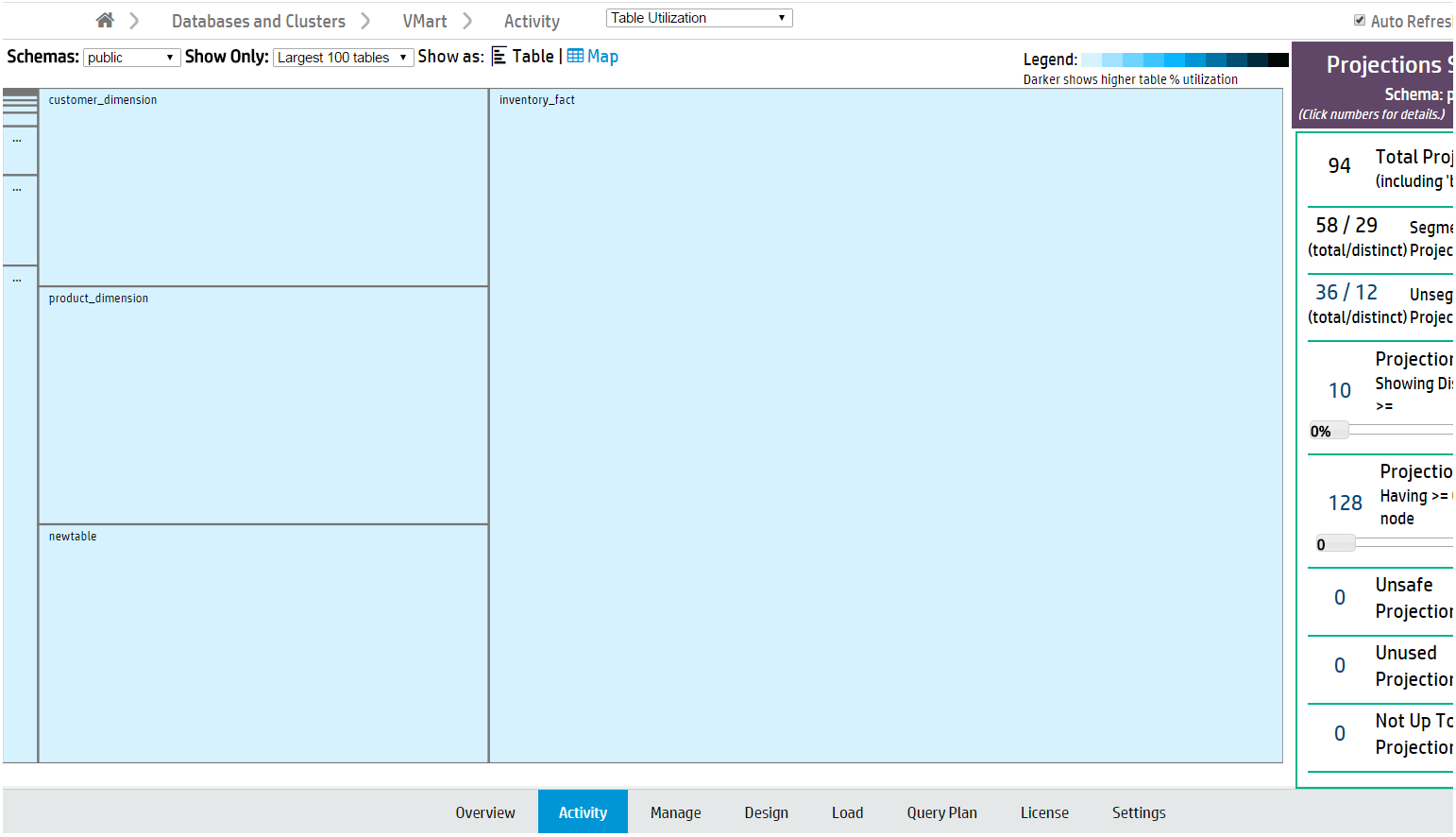
Hover over any row in the chart to view the table's properties (shown for inventory_fact in the screen capture below). Click the table name to view its more in-depth Table Details page.



View the treemap chart

In the Treemap visualization, tables are represented by boxes, nested by size and colored by usage. Darker colors indicate higher table usage.

Hover over a table to view more details, or click to view its Table Details page.

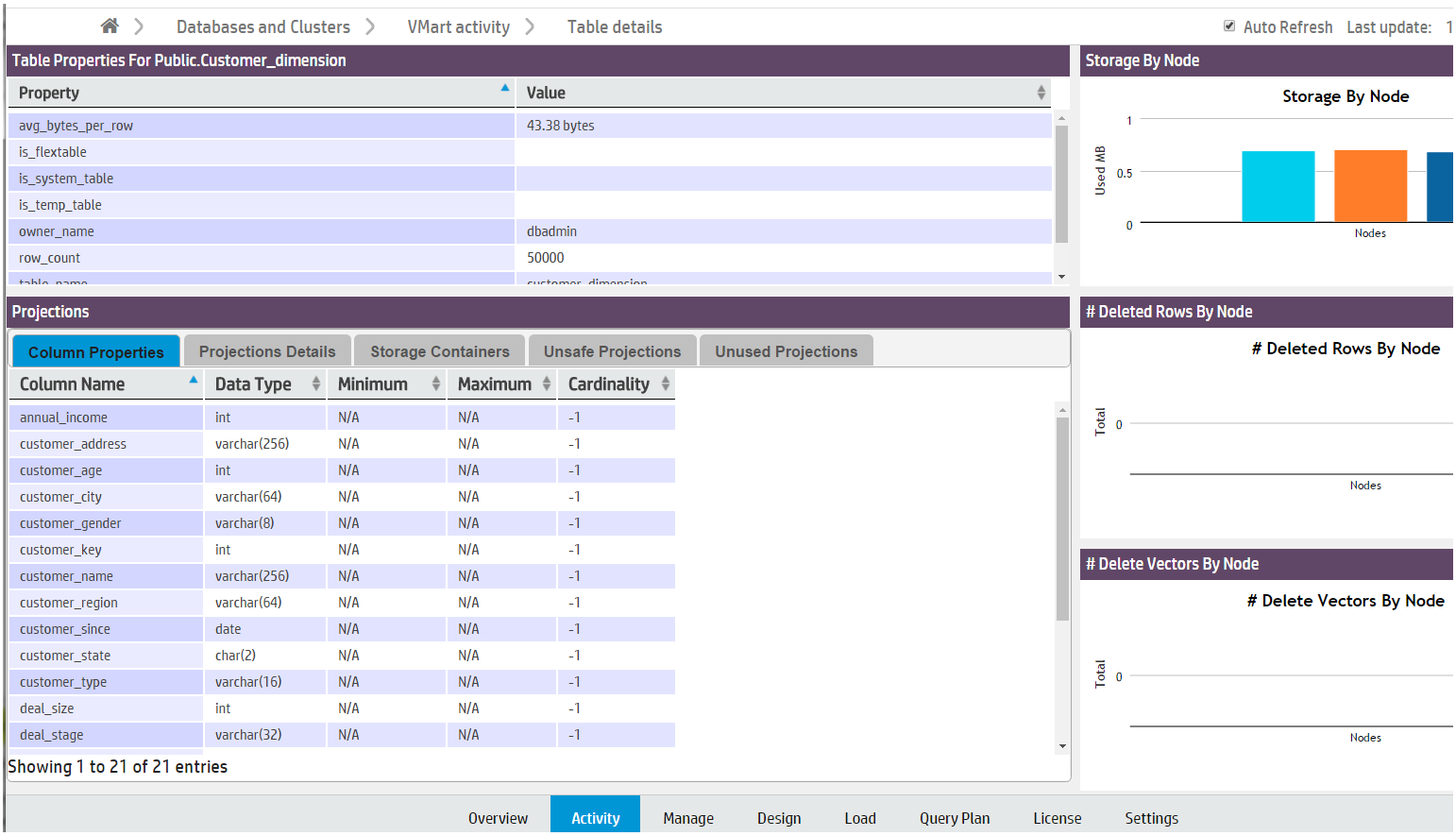


View table details

The Table Details page displays a detailed overview of internal Vertica tables. (This is not available for external and HCatalog tables.) Click a table name on the Table Utilization Activity page to open its Table Details page in a new window.

You can view the following details:

- **Table Properties** . Table properties (such as row count and owner).
- **Projections** . The properties of the table's columns and projections.
- **Storage by Node** . The table's storage utilization per node, in MB.
- **# Deleted Rows by Node** . Vertica allocates physical storage to deleted rows until they are purged by the Tuple Mover.
- **# Delete Vectors by Node** . Vertica creates small containers called delete vectors when DELETE or UPDATE statements run on a table. A large number of delete vectors can adversely impact performance. (See Deletion marker mergeout .)



Note

Note: If you have deleted table rows recently, Management Console may not display the most recent row count. MC updates the row count when mergeout occurs. See Mergeout .

Projections summary

The Projections Summary is located in a side bar on the right side of the Table Utilization page. It displays the following statistics of all projections in a schema:

- **Total projections** .
- **Segmented projections** , the number of projections segmented across multiple nodes.
- **Unsegmented projections** , the number of projections that are not segmented across multiple nodes.
- **Projections Showing Distribution Skew** , the number of projections unevenly distributed across nodes. Tables with fewer than 1000 rows are not counted. Move the slider to configure filter by distribution skew percentage.
- **Projections Having >= Containers Per Node** . Move the slider to specify the minimum number of containers.
- **Unsafe Projections** , the number of projections with a K-safety less than the database's K-safety.
- **Unused Projections** .
- **Not Up to Date Projections** .

Click a projections number to view a list of the specified projections and their properties. For more about projections, see Projections .

See also

- Working with native tables
- Projections
- Working with external data
- Using the HCatalog Connector

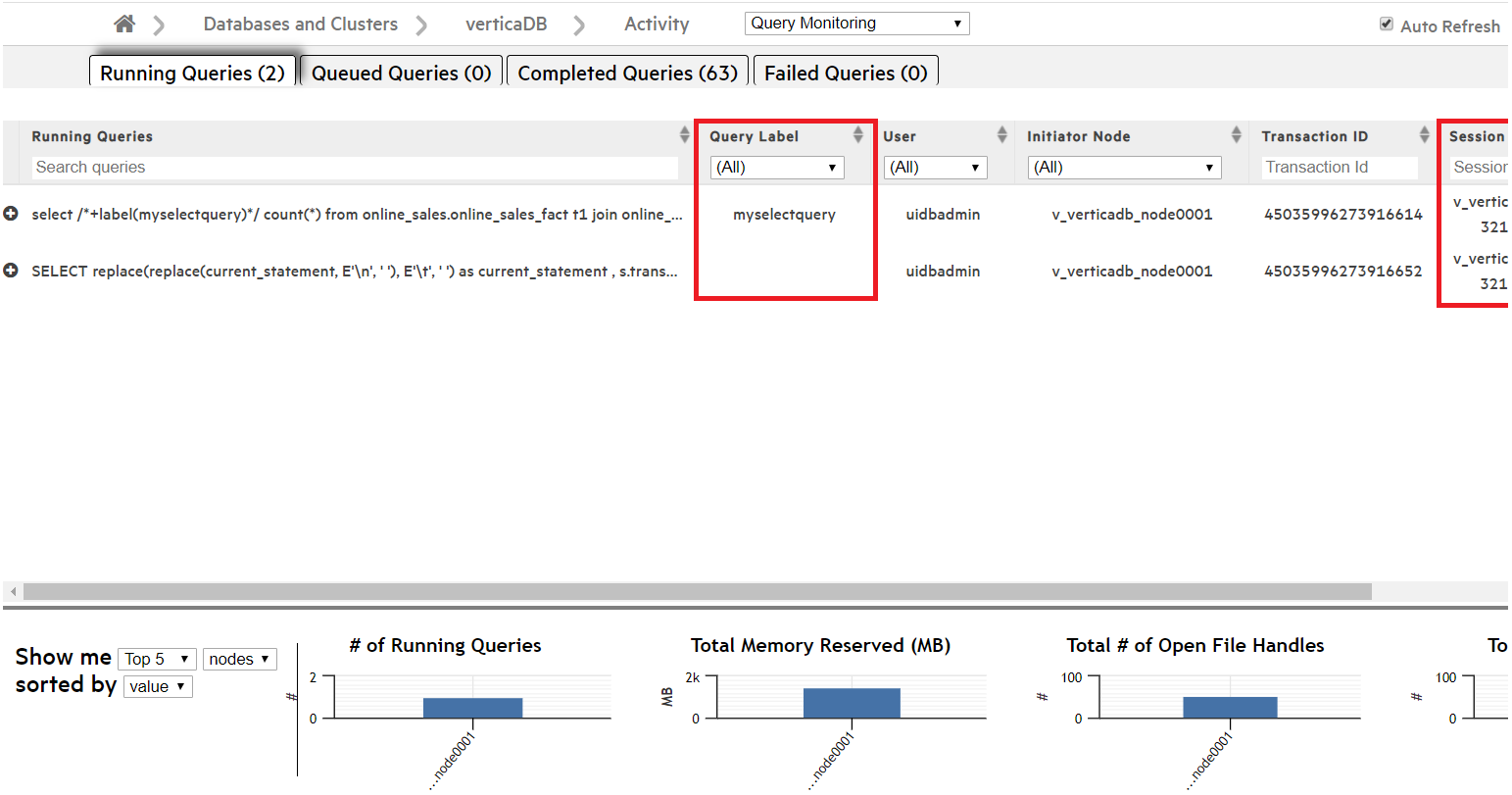
Monitoring running queries with MC

The **Query Monitoring** activity page displays the status of recent and currently running queries, and resource information by user and node. For Eon Mode databases, you can also display the status of queries by subcluster. From this page, you can profile a query or cancel a running query.

Use this page to check the status of your queries, and to quickly cancel running or queued queries to free up system resources. This page can help you identify where resources are being used, and which queries, users, nodes, or subclusters are using the most resources.

The **Query Monitoring** page includes four tables, displayed as tabs:

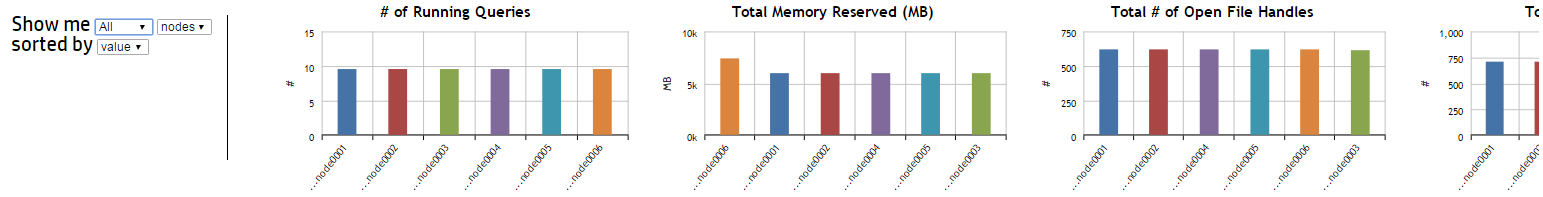
- Running queries
- Queued queries
- Completed queries
- Failed queries



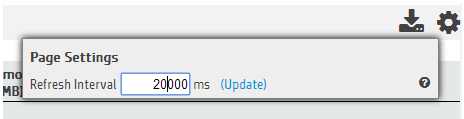
From the **Actions** column you can:

- **Cancel** . Cancel a running or queued query.
- **Close session** . Close a session for a running or queued query.
- **Explain** . Open the **Query Plan** page for any query.
- **Profile** . Profile any query on the **Query Plan** page.

The four bar charts at the bottom of the page display aggregate query usage by node or user. Hover over a bar with your cursor to see its value. When sorted by value, the left-most bar in each chart represents the node or user with the highest value.



The Query Monitoring page refreshes every 20 seconds by default. To change the refresh interval, click the **Page Settings** button in the upper-right corner of the page. A dialog appears. Type the new refresh interval, in milliseconds, in the text box.



Sorting or searching queries by session ID or client label

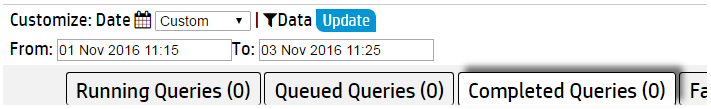
The Query Monitoring Activity > Running Queries page includes columns that display the **Session ID** and **Client Label** for each query. You can sort the queries by Session ID or Client Label, or use the search field below either column to search for queries with a specific Session ID or Client Label.

Filtering chart results

Use the search field below each column title to narrow down your chart results. (For example, if you enter the text *SELECT product_description* in the **Search Queries** field and select a specific node in the **Initiator Node** column, the chart returns only queries which both contain that text and were initiated on the node you specified.)

Click a column title to sort the order of the queries by that category.

There may be a large number of results for Completed and Failed Queries. Use the **Customize** section at the top of these two tabs to further filter your chart results. For either tab, you can select a custom date and time range for your results.

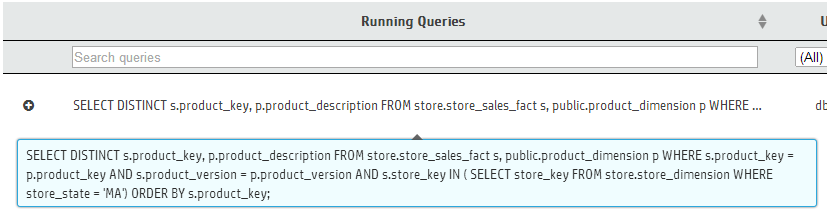


In the Completed Queries tab, click **Data** to enter additional query information to filter based on any of the following fields:

- User
- Request
- Request Duration
- Node
- Request label

Viewing more details

Click a query to view the entire query.



In the **Failed Queries** chart, click the **plus (+)** icon next to a failed query to see the failure details for each node involved in the query's execution.

To export the data in one of the **Query Monitoring** tables, click the table's tab, then click the **Export** ( ) button in the upper-right corner of the page. The browser downloads the data for that chart in a .dat file. The data exported includes columns that may not be visible in the MC, including the minimum and maximum values for Memory, Thread Count, and Open File Handles.

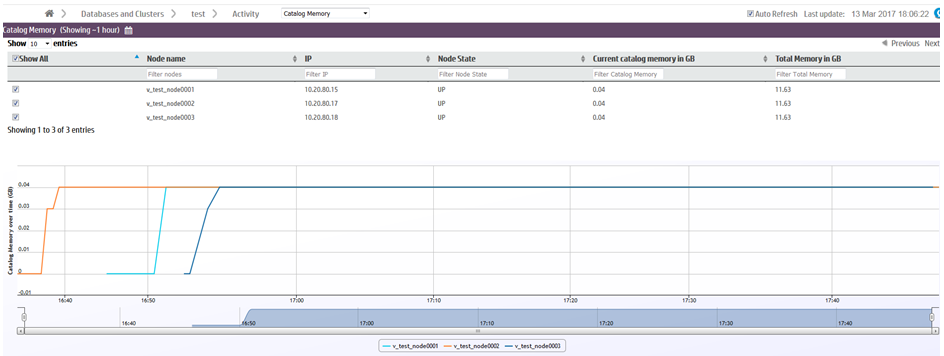
Monitoring catalog memory with MC

The Catalog Memory activity page displays the catalog memory for each node. Use this page to check for sudden changes in catalog memory, or discrepancies in memory distribution across your nodes.

The Catalog Memory page displays:

- **A node details table.** The table lists the details of each node in the database, including their current catalog memory and total memory usage.
- **A catalog memory chart.** A line graph visualization of each node's catalog memory usage over time. Each line represents a node. The color legend at the bottom of the chart indicates the color of each node's line.

In the image below, catalog memory begins at 0GB for all three nodes. Over the next twenty minutes, catalog memory increases to 0.04GB in the second node (orange), then the first node (cyan), and finally in the third node (dark blue). Starting at 16:55, note that the three overlapping node lines appear as one line when all three nodes have the same catalog memory.



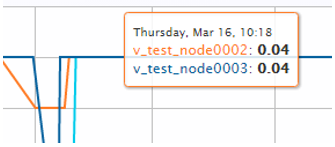
Filtering chart results

If you have many nodes in your database, you may want to display only certain nodes in the catalog memory chart. You can remove nodes from the chart in two ways:

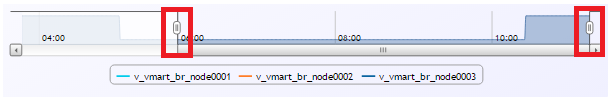
- Deselect the node's check box in the node details table.
- Deselect the node in the color legend below the chart.

Viewing more details

Hover over any line in the chart to view the time, node name, and catalog size.



At the bottom of the chart is a summary bar that shows a quick overview of the catalog memory over time. Move the sliders on either side of the chart to zoom in on a specific time frame in the chart. When zoomed in, you can use the scrollbar to move forward or backward in time.

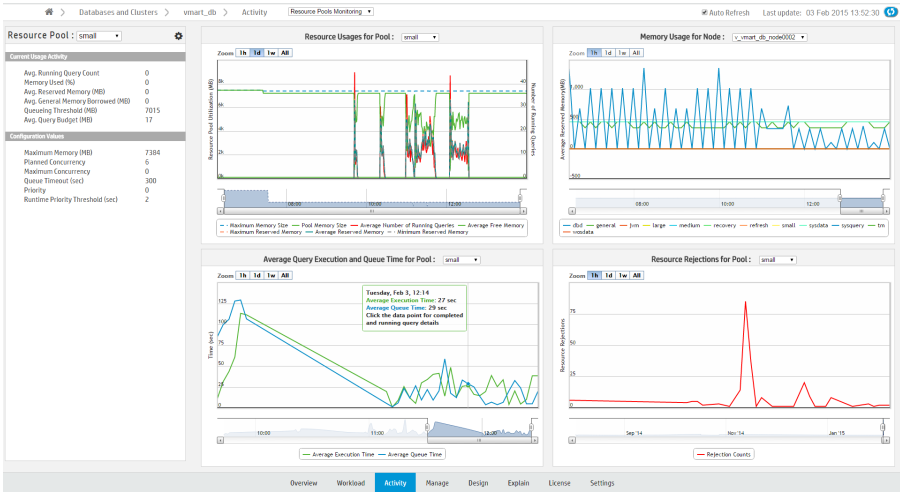


Monitoring resource pools with MC

Management Console allows database administrators to monitor and configure resource pools through the **Activity** and **Configuration** pages. These pages help you manage workloads by providing visual representations of resource usage as well as resource pool configuration options.

Monitoring resource pools charts

You can monitor your resource pools using the **Resource Pools Monitoring** charts, accessible through the Management Console **Activity** page.



Select a resource pool to view using the **Resource Pool** menu, located in the leftmost sidebar. In the sidebar, **Current Usage Activity** displays the pool's real-time statistics.

Monitor the selected resource pool using the following charts, which display the pool's historic data:

- **Resource Usages for Pool:** Shows the historically averaged acquired memory usage by each pool across all nodes. The graph uses two y-axes, one that shows memory size, and a second that shows the total number of running queries. Data is collected every hour. Hover over a data point for a summary of the memory usage at that specific point.
- **Memory Usage in Node:** Shows the historically acquired memory usages by all pools across all nodes. Data is collected every hour. Hover over a data point for a summary of the memory usage at that specific point. Use the title bar dropdown to display the memory usage for a specific node. For Eon mode databases, you can also display the memory usage for a specific subcluster , all subclusters, or nodes not assigned to a subcluster. An Eon mode database has one default subcluster, and may have additional user-defined subclusters.
- **Average Query Execution and Query Time in Pool:** Shows the averaged query queue time plotted against the query execution time by each pool across all nodes. Data is collected every minute. Hover over data to get the average query execution and queue time in the specified pool. Click a data point to show detailed individual query information.
- **Resource Rejections in Pool:** Shows the historical total number of resource requests that were rejected in each pool across all nodes. Data is collected every hour. Click a data point to show rejection details and reasons in a pop-up window.

Configuring resource pools in MC

Database administrators can view information about resource pool parameters and make changes to existing parameters through the Management Console **Configuration** page. You can also create and remove new resource pools, assign resource pool users, and assign cascading pools.

See Configuring Resource Pools in Management Console

Permissions

Only the database administrator can monitor and configure resource pools in Management Console.

See also

- Configuring Resource Pools in Management Console
- Monitoring resource pools

In this section

- Configuring resource pools with MC

Configuring resource pools with MC

Database administrators can view information about resource pool parameters and make changes to existing parameters in MC through the Resource Pools Configuration page. You can also create and remove new resource pools, assign resource pool users, and assign cascading pools.

Access the Resource Pools Configuration page from the Settings page by selecting the Resource Pools tab.

You can also access the Configuration page from the Resource Pools Monitoring chart, accessible through the Management Console Activity page. Click the tools icon on the top of the leftmost sidebar.

Permissions for monitoring and configuring resource pools

Only the database administrator can monitor and configure resource pools in Management Console.

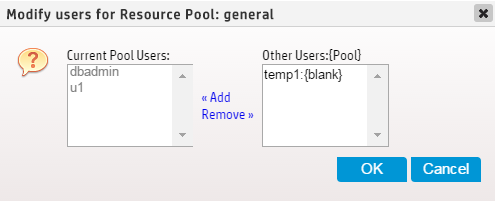
Modify resource pool parameters

1. On the Resource Pools Configuration page, choose a resource pool from the Resource Pools field. Parameter fields for that resource pool appear.
2. Use the parameter fields to view and modify parameters for the resource pool. Hover your cursor in the parameter field to display information about that parameter.
3. Click **Apply** to save your changes. A success message appears

Modify resource pool users

To add or remove resource pool users:

1. On the Resource Pools Configuration page, select a resource pool from the Resource Pools field.
2. Next to the Pool Users field, click **Add/Remove Pool Users** . The Modify Users for Resource Pool dialog appears.



3. The dialog displays users assigned to the resource pool in the Current Pool Users list. The Other Users list displays all other resource pool users are displayed, along with the pool to which they are currently assigned.
    1. To add users to the resource pool: Select the desired users from Other Users list and click **Add** .
    2. To remove users from the resource pool: Select the users to be removed from the Current Pool Users list and click **Remove** .
4. Click **Apply** to save your changes. A success message appears.

Create and remove resource pools

Database administrators can use MC to create resource pools and assign resource pool users, and remove user-generated resource pools.

To create a resource pool:

1. On the Resource Pools Configuration page, click **Create Pool** . Fields pre-populated with pool parameter default values appear.
2. Enter the new resource pool's parameters in the fields.
3. Click **Create Pool** . A success message appears.

To remove a resource pool:

1. First, remove all users from the resource pool to be deleted. This can be done on the Resource Pool Configuration Page.

2. When all users have been removed from the resource pool, choose the resource pool from the Resource Pools field on the Resource Pool Configuration Page. Parameter fields for that resource pool appear.
3. Click **Remove Pool** . A Confirm dialog appears.
4. Click **OK** in the Confirm dialog. A success message appears.

See also

- CREATE RESOURCE POOL
- Monitoring resource pools with MC
- Monitoring resource pools

Monitoring database messages and alerts with MC

Management Console periodically checks system health and performance. When an important event occurs or system performance fluctuates beyond user- or system-defined limits, the Management Console generates a message to alert you about the state of your system. View and manage alerts in the Message Center .

Message alert notifications

Management Console uses multiple methods to communicate alert notifications to ensure that you are immediately aware of any changes to your database that might require attention. You receive message notifications by email , and you can view notifications using the following components:

- **Message envelope icon** : This icon is located by the **MC Help** icon, in the top-right of any database-specific page. Select this icon display the **Message Center** quick view, and perform archive, read, and delete message actions. For details about message actions and alerts, see Message center .
- **Unread Messages (This Week)** widget: On the database **Overview** tab, this widget is located in the quick stats sidebar. It displays unread, high-priority messages. Select the number (including 0) in the widget to go to the **Message Center** .

Pre-configured alerts

Management Console provides pre-configured alerts to provide system monitoring capabilities without manual setup. Each alert has a pre-configured threshold that defines the acceptable performance limit, and MC sends a message notification when the database exceeds that threshold.

By default, pre-configured alerts are not active and require minimal initial setup. For details on how to set pre-configured alert properties, see Alert configuration .

Node health

Vertica provides the following pre-configured alerts to monitor node health:

- Node CPU
- Node Memory
- Node Disk Usage
- Node Disk I/O
- Node CPU I/O Wait
- Node Reboot Rate
- Node State Change
- Node Catalog Memory

Network health

Vertica provides the Network I/O Error pre-configured alert to monitor network health.

Query

Vertica provides the following pre-configured alerts to monitor queries:

- Queued Query
- NumberFailed
- Query Number
- Spilled Query Number
- Retried Query Number
- Query Running Time

License status

Vertica provides the License Usage pre-configured alert to monitor the status of your Vertica license.

Resource pool

MC can send alerts when an individual resource pool reaches a specified state or usage level. For details about resource pool configuration parameters, see Built-in resource pools configuration .

Important
Default settings for resource pool alerts apply to the GENERAL pool only.

You can configure the MC to send the following resource pool alerts:

- **Queries Reaching the Max Allowed Execution Time** : Triggers an alert when the specified number of queries reach the RUNTIMECAP execution threshold for the resource pool. You cannot set this alert if the resource pool does not have a RUNTIMECAP threshold set, or if the resource pool has a secondary resource pool .
- **Queries With Resource Rejections** : Triggers an alert when the specified number of queries exceed a specified number of resource rejections within a set period of time.
- **Minimum Starting Resource Value** : Triggers an alert when the resource pool reaches the minimum amount of resources allocated for the MEMORYSIZE value.

  > **Note**
  > By default, you cannot set MEMORYSIZE for the GENERAL pool. The GENERAL pool must have at least 1GB of memory and it cannot be smaller than 25% of the entire system memory.

- **Maximum Allowed Resource Value** : Triggers an alert when the resource pool reaches the MAXMEMORYSIZE value.
- **Ended Query With Queue Time Exceeding Limit** : Triggers an alert when the specified number of completed queries were queued for a specified length of time within a timeframe.
- **Ended Query With Run Time Exceeding Limit** : Triggers an alert when the specified number of completed queries ran for a specified length of time within a timeframe.

Custom alerts

Create custom alerts to measure system performance metrics that are not monitored by the pre-configured alerts. Create a dynamic SQL query that triggers an alert when it returns any results. You can configure how often an alert is generated, the alert priority, and who receives email alerts. For example, you can create custom alerts that monitor the following:

- Failed logins within a configurable time period
- Idle Vertica user sessions using a configurable time limit
- Database node is DOWN

For details about creating and managing custom alerts, including a tutorial on how to create a custom alert that monitors failed logins, see Custom alerts .

Default notifications

Management Console generates the following messages about the database that appear only in the Message Center :

- Low disk space
- Read-only file system
- Loss of K-safety
- Current fault tolerance at critical level
- Too many ROS containers
- Change in node state
- Recovery error
- Recovery failure
- Recovery lock error
- Recovery projection retrieval error
- Refresh error
- Refresh lock error
- Workload analyzer operations
- Tuple Mover error
- Timer service task error
- Last Good Epoch (LGE) lag
- License size compliance
- License term compliance

Disk space check and cleanup

When the Management Console checks alerts, it generates a result set and saves it to disk. If you use aggressive alert configurations, the result set might use a large amount of disk space. By default, Vertica reserves 500MB of disk space to save result sets.

Vertica checks the free disk space 2 times each day and cleans alerts that are older than 7 days. If the available disk space is low, custom alerts are disabled. Notifications and emails are generated when an alert is disabled due to insufficient disk space.

The /opt/console/vconsole/config/console.properties file contains these settings. Edit the following values to configure the how the MC manages your disk space:

Property	Description
console.diskspace.threshold	The amount of disk space Vertica reserves to save result sets. **Default:** 500MB
customthreshold.alerts.toKeepInDays	The number of days that alerts are retained on disk. **Default:** 7

In this section

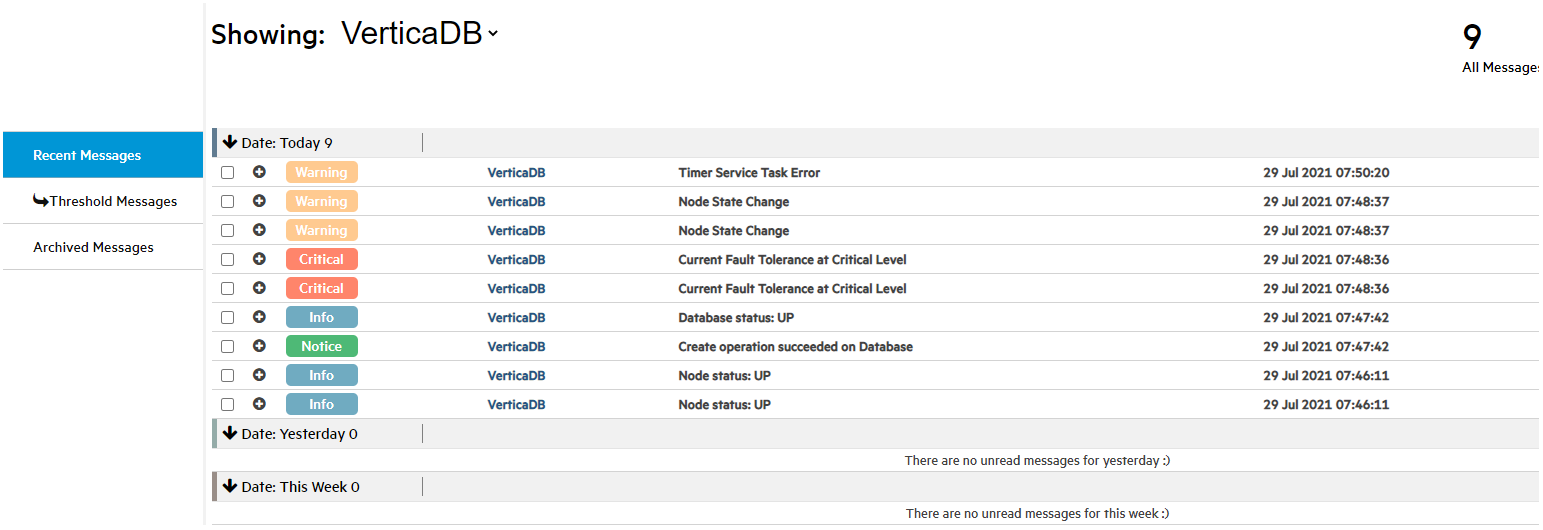
- Message center
- Alert configuration
- Setting up SMTP for email alerts
- Custom alerts
- Exporting MC-managed database messages and logs

Message center

The **Message Center** organizes system performance alerts to help you effectively monitor the state of your database. Pre-configured and custom alerts generate messages when the component they measure exceeds the specified threshold.

Access the **Message Center** in the following ways:

- On the MC Home page, select **Message Center** in the **MC Tools** section.
- Select the message envelope icon in the top-right of any database-specific page, then select **Message Center** in the quick view.
- On the database **Overview** tab, select the number (including 0) in the **Unread Messages (This Week)** widget.



The Message Center can retrieve up to 10,000 of the most recent messages generated for a database. By default, it lists up to 600 messages generated in the previous week. For details on how to retrieve messages that predate the previous week, see Date Filtering .

Note

To adjust the maximum number of messages listed in the Message Center, edit the messageCenter.maxEntries value in the /opt/vconsole/config/console.properties file from the command line. For example, the following value increases the number of alerts listed in the Message Center to 5000:

```
messageCenter.maxEntries=5000
```

Filter the messages grid

The Management Console provides options that filter the messages that populate the messages grid by database, keyword, message type, and date. Use one or more of these options to view only the messages that you want to manage.

Database filtering

Select **Showing** to list the databases that are associated with the logged in user account. Choose a specific database to view only messages for that database, or select **(All DBs)** to view and manage messages across all of your databases.

Keyword filtering

Use the search bar at the top-right of the screen to list messages that contain the entered text. For example, if you enter 29, the grid lists any messages that contain those characters within the message title or message details available when you click the plus (+) icon to expand the message row.

Message type filtering

After you select a value for **Showing** , the number of messages for that value are totaled and grouped by type near the top-right corner of the screen, below the search bar. These message types use the syslog standard to define severity levels. The MC message types are defined as follows:

- **All Messages** : Messages that are not archived or deleted, and are within messageCenter.maxEntries for the previous week.
- **High Priority** : Messages that you assigned a High Priority alert priority on the **Alerts** tab. These messages correspond to syslog levels 0 and 1.
- **Need Attention** : Critical or error messages that indicate that correspond to syslog levels 2 and 3.
- **Informational** : Warning, notice, informational, or debugging messages that correspond to syslog levels 4, 5, 6, and 7.

To populate the grid with only a single message type, select the number or message type description.

Date filtering

Select **Retrieve Older Messages** to enter **From** and **To** dates to list messages that were generated before the previous week. Vertica stores 10,000 of the most recent database messages so that you can retrieve older messages when necessary.

Message groups

The Message Center groups messages and notifications as **Recent Messages** , **Threshold Messages** , and **Archived Messages** . All message groups use the following priority levels and color codes:

- Critical (Red)
- Warning (Orange)
- Notice (Green)
- Info (Blue)

Recent messages and threshold messages

Recent Messages include the most recent messages generated within the previous week within the messageCenter.maxEntries value. Threshold messages include messages that are generated when the database exceeds a pre-configured, custom, or default alert threshold.

Recent Messages and Threshold Messages are listed using the message type, database name, a description of the message, and the date and time that the message was generated. Additionally, there are collapsible grid headings that group the alerts by **Today** , **Yesterday** , and **This Week** .

Archived messages

Archived messages are messages that you manually saved for later. When you select the check icon for an individual message or select **Archive All** , MC marks the message as read and archives it. Archived messages do not have the same date or messageCenter.maxEntries restrictions as **Recent Messages** or **Threshold Messages** .

The **Archived Messages** tab contains the following tools to refine search result filters:

- Use the **From** and **To** boxes above the grid to define a time frame for the archived message search. To combine multiple time frames, select the **Do not clear existing search results** checkbox.
- Sort or filter messages even further using the **Type** , **Database Name** , **Description** , and **Date** columns.

Message actions

Perform actions on all, multiple, or individual messages. To execute actions on all messages at the same time, use the **Select All** or **Select None** buttons near the top-right of the screen by the search bar.

Each message row has a checkbox so that you can perform actions on one or more messages simultaneously. Select the plus (+) icon to expand the message row and display the following message details:

- Summary
- Host IP
- Time of occurrence
- Number occurrences of this message
- Description

For additional information about each message, query EVENT_CONFIGURATIONS .

When you select the **X** icon or select multiple messages and click the **Delete Msgs** or **Delete all** buttons, the message is permanently deleted.

Alert configuration

Enable and customize settings for pre-configured or custom alerts for each database. For example, you can set the **Threshold Value** for **Node Disk Usage** to a 20% minimum or 80% maximum. If any node exceeds either of those thresholds, the MC generates a message and you receive a notification. Take action on alerts in the Message center .

To access the **Alerts** tab, log in to the Management Console, then select **Go to database** > **Settings** > **Alerts** .

Configurable settings

To configure any alert, you must toggle the switch on the left of the row to the on position. **Node State Change** is the only pre-configured alert turned on by default. By default, custom alerts are toggled off.

Pre-configued and custom alerts have the following settings:

- **Query variables** : Custom alerts only. Query variables are the variables that you added when you created the alert . You must provide a value for each variable. The query variable is not validated after you create the alert. If you update the variable to a value that results in an invalid query, the alert is silently disabled during the next execution.
  To view the original SQL query, hover the mouse over the alert name to display the alert in a tooltip.
- **Threshold Value** : Pre-configured alerts only. You can add a lower and upper limit on acceptable operating values for the component.
- **Check Interval** : This value determines how often Vertica executes the underlying SQL for the alert.
- **Alert Priority** : You can assign one of the following priority labels to determine how the alert is distributed:
  - **Alert** and **Critical** : Displays the message using the message alert notification mechanisms on the **Overview** page and creates a message in the **Message Center** .
  - **Warning** : Creates a message in the Message center .
- **Alert Email Recipients** : Configure email notifications for any alert priority when the alert is triggered. You must have SMPT configured . For details about adding email recipients to alerts, see Custom alerts .
- **Vertical ellipses** : For custom alerts and new resource pool alerts, click the vertical ellipses to delete the alert. You must have MC ADMIN role privileges to delete an alert.

Configuring resource pool alerts

Resource pool alerts use the same configurable settings as the other alerts, but you can set alerts per resource pool. By default, pre-configured alerts are set for the GENERAL pool only.

> **Note**
> Only users with MC ADMIN role privileges can add alerts for resource pools other than the GENERAL pool.

1. Log in to the Management Console and select **Databases** > **Settings** > **Alerts** .
2. In the top row labeled **Resource Pool** , select the blue box with a plus (+) symbol on the far right of the row. When you hover over the button, the button expands and displays **Add Resource Pool Alert +** .
   The **New Resource Pool Alert** window opens.
3. In **Alert Name** , choose the alert that you want to add to a resource pool.
4. In **Resource Pool** , choose the resource pool that you want to add this alert to.
5. When you are finished configuring the remaining settings, select **Create Alert** .

Edit **Check Interval** , **Alert Priority** , and Alert Email Recipients as you would other alerts. For guidance, see Configurable Settings or Custom alerts .

To delete the alert, select the vertical ellipses at the right of the row, and select **Delete** . You must have MC ADMIN role privileges to delete an alert.

Setting up SMTP for email alerts

Management Console can generate email notifications when your database exceeds high-priority alert thresholds. To receive email alerts, you must configure your SMTP settings in MC.

You must be an administrator to provide SMTP settings. To set up MC to send email:

1. Select the **Email Gateway** tab on the MC Settings page.
2. Provide the following information about your SMTP server:
   - **Email Server (Hostname)** : the hostname or IP of the SMTP server
   - **Port** : the port of the SMTP server
   - **Session Type** : the type of connection to use (e.g. SSL)
   - **SMTP Username (optional)** : the username credential for connecting to the server
   - **SMTP Password (optional)** : the password credential for connecting to the server
   - **Sender Address** : The sender address for your server when it sends email alerts
   - **Trust SSL Certificate** : Whether to automatically trust the SMTP server's certificate
3. Click **Test** at the top of the page. MC validates your SMTP settings and sends a test email to the inbox of the email alias you provided.
4. Verify that you successfully received the test email.
5. Click **Apply** at the top-right of the page to save the settings.

After you set up SMTP for email, you can configure MC to send high-priority threshold alerts through email. For details, see Alert configuration or Custom alerts .

Custom alerts

Create custom events-based alerts to track abnormalities and performance fluctuations for node health, queries, and resource pools using your own database schemas and tables. When the database triggers an active alert, you receive notifications according to the alert priority, and can take action in the Message center .

Creating a custom alert

You must have MC ADMIN role privileges to create a custom alert.

The following steps create a custom alert named **Failed logins within a X time period** to track the number of failed logins in the previous two hours, per user. This alert might indicate a possible distributed denial-of-service (DDoS) attack, or an application behaving inappropriately. The underlying SQL query uses a variable to create a dynamic threshold value that you can fine-tune after you create the alert.

1. Log in to the Management Console, then select **Go to database** > **Settings** > **Alerts** .
2. In the **Custom Alerts** row at the top of the page, click the blue box with a plus (+) symbol on the far right of the row. When you hover over the button, the button expands and displays **Create Custom Alert +** .
   The **Create custom alert** window displays.
3. In **Alert Name** , enter **Failed logins within X time period** .
4. In **SQL Query** , enter the following SQL query:

```
SELECT
  login_timestamp,
  user_name,
  node_name,
  client_hostname,
  reason
FROM
  login_failures
WHERE
  reason in ('INVALID USER', 'FAILED', 'INVALID DATABASE')
  AND login_timestamp > sysdate - INTERVAL '{{Time_Interval}}'
```

   The AND clause of the preceding query encloses the variable of type String named Time_Interval in double curly braces to represent a valid SQL syntax element.
   A variable is a dynamic value that is evaluated at runtime that you can configure after you save the alert. You can add up to 6 variables per custom alert. Variable names may consist of a maximum of 14 alpha-numeric characters, including underscores. Verify that the variable uses the correct data type and format. Variables of type String require single quotes around the curly braces.
   A SQL query triggers an alert if it returns one or more rows. Use the formatting or full screen buttons above and to the right of the **SQL Query** box as needed.
5. A box displays below the **SQL Query** box containing placeholder text that corresponds to each variable name. To test the alert, enter **2 Hours** in this box.
6. Select **Run Query** . The **Query Results** section displays any rows returned from your query. Alternatively, you might encounter one of the following issues:
   - If you use invalid SQL syntax, you receive an error.
   - If the query returns more than 5 columns or 100 rows, you receive a warning. Because every query result set is saved to disk, it is important to be aware of the size of your result set. For more information, see Monitoring database messages and alerts with MC .

Important
Queries do not timeout. A long-running query runs until it succeeds or returns a Query Error that indicates that there are insufficient resources to continue.

Select **Cancel Query** to stop long-running queries.

7. When you are satisfied with the query results, select **Create Alert** .
   The **Create custom alert** window closes and the alert you just created is listed in the **Custom Alerts** section on the **Alerts** page. When you point the mouse on the query name, the query is displayed in a tooltip. Under the query name, there are editable boxes that correspond to the variables you added when creating the alert.
8. In the **Time_Interval** variable box, enter **2 Hours** .
9. Select a value for **Check Interval** . The default setting is **10 minutes** . This value determines how often the query is executed.
10. Select a value for **Alert Priority** . By default, the alert is assigned the **Critical** value priority.
11. Optionally, select the **Manage Emails** icon under **Alert Email Recipients** to send an email alert to specific users when the alert is triggered. To register a user to receive email alerts, complete the steps in Setting up SMTP for email alerts .
12. Complete the following steps in the **Manage Email Recipient** window:
    1. To add an existing user to an alert, click the checkbox beside the existing MC user, or select the box at the top to add all. For non-existing MC users, enter their email address at the bottom of the window.
    2. In **Email Interval** , select how often the email is sent:
       - **Immediately**
       - One hour: The hour starts when you make the alert.
       - **One day** : Users receive the alert 24 hours after you create the alert.

       For example, if you select **One hour** , an email is sent every hour, even if the alert is triggered multiple times within the hour.
    3. Click **Save** .

After you create the alert, toggle it on or off using switch at the far left of the alert row.

Editing a custom alert

You must have MC ADMIN role privileges to edit a custom alert.

1. Log in to the Management Console, then select **Go to database** > **Settings** > **Alerts** .
2. In the **Custom Alerts** row at the top of the page, locate the custom alert that you want to edit.
3. Select the vertical ellipses, and select **Edit** .
   The **Edit custom alert** window opens and displays the previously saved values for the custom alert.
4. Edit the alert. You can edit any of the following alert properties:
   - Alert Name
   - SQL Query
   - Any variable values
5. Select **Run Query** . The **Query Results** section displays any rows returned from your query. Alternatively, you might encounter one of the following issues:
   - If you use invalid SQL syntax, you receive an error.
   - If the query returns more than 5 columns or 100 rows, you receive a warning. Because every query result set is saved to disk, it is important to be aware of the size of your result set. For more information, see Monitoring database messages and alerts with MC .

   Important
   Queries do not timeout. A long-running query runs until it succeeds or returns a Query Error that indicates that there are insufficient resources to continue.

   Select **Cancel Query** to stop long-running queries.

6. When you are satisfied with the query results, select **Update Alert** .

Deleting an alert

To delete a custom alert, select the vertical ellipses at the right of the row, and select **Delete** . You must have MC ADMIN role privileges to delete an alert.

Exporting MC-managed database messages and logs

You can export the contents of database messages, log details, query details, and MC user activity to a file.

Information comes directly from the MC interface. This means that if the last five minutes of vertica.log information displays on the interface, you can save that five minutes of data to a file, not the entire log. When you filter messages or logs, MC exports only the filtered results.

Depending on how you set your browser preferences, when you export messages you can view the output immediately or specify a location to save the file. System-generated file names include a timestamp for uniqueness.

The following table shows, by record type, the MC pages that contain content you can export, the name of the system-generated file, and what that file's output contains:

Message type	**Where you can export on MC**	**System-generated filename**	**Contents of exported file**
All db-related message types	**Message Center** page	vertica-alerts-< *timestamp* >sv	Exports messages in the Message Center to a .csv file. Message contents are saved under the following headings: • Create time • Severity • Database • Summary (of message) • Description (more details)
MC log files	**Diagnostics** page	mconsole-< *timestamp* >.log	Exports MC log search results from MC to a ** .log file under the following headings: • Time • Type (message severity) • Component (such as TM, Txn, Recover, and so on) • Message
Vertica logs	**Manage** page Double-click any node to get to the details and then click the VerticaLog tab	vertica-vertica-< *db* >-< *timestamp* >.log	Exports vertica log search results from MC to a .log file under the following headings: • Time • Type (message severity) • Component (such as TM, Txn, Recover, and so on) • Message
Agent logs	**Manage** page Click any node to get to the details and then click the AgentTools Log tab.	vertica-agent-< *db* >-< *timestamp* >.log	Exports agent log search results from MC to a .log file under the following headings: • Time • Type (message severity) • Component (such as TM, Txn, Recover, and so on) • Message
Query details	**Activity** page Click any query spike in the Queries graph to get to the Detail page.	vertica-querydetails-< *db* >-< *timestamp* >.dat	Exports query details for the database between < *timestamp* > and < *timestamp* > as a tab-delimited *.dat* file. Content is saved under the following headings: • Query type • Session ID • Node name • Started • Elapsed • User name • Request/Query

MC user activity	**Diagnostics** page Click the Audit Log task	vertica_audit< *timestamp* >.csv	Exports MC user-activity results to a .csv file. Content is saved under the following headings: • Time • MC User • Resource • Target User • Client IP • Activity

Monitoring MC user activity using audit log

When an MC user makes changes on the MC interface, whether to an MC-managed database or to the MC itself, their action generates a log entry that records a timestamp, the MC user name, the database and client host (if applicable), and the operation the user performed. You monitor user activity on the **Diagnostics > Audit Log** page.

MC records the following types of user operations:

- User log-on/log-off activities
- Database creation
- Database connection through the console interface
- Start/stop a database
- Remove a database from the console view
- Drop a database
- Database rebalance across the cluster
- License activity views on a database, as well as new license uploads
- Workload analyzer views on a database
- Database password changes
- Database settings changes (individual settings are tracked in the audit record)
- Syncing the database with the cluster (who clicked Sync on grid view)
- Query detail viewings of a database
- Closing sessions
- Node changes (add, start, stop, replace)
- User management (add, edit, enable, disable, delete)
- LDAP authentication (enable/disable)
- Management Console setting changes (individual settings are tracked in the audit record)
- SSL certificate uploads
- Message deletion and number deleted
- Console restart from the browser interface
- Factory reset from the browser interface
- Upgrade MC from the browser interface

Background cleanup of audit records

An internal MC job starts every day and, if required, clears audit records that exceed a specified timeframe and size. The default is 90 days and 2K in log size. MC clears whichever limit is first reached.

You can adjust the time and size limits by editing the following lines in the /opt/vconsole/config/console.properties file:

```
vertica.audit.maxDays=90vertica.audit.maxRecords=2000
```

Filter and export results

You can manipulate the output of the audit log by sorting column headings, scrolling through the log, refining your search to a specific date/time and you can export audit contents to a file.

If you want to export the log, see Exporting the user audit log .

If you perform a factory reset

If you perform a factory reset on MC's Diagnostics page (restore it to its pre-configured state), MC prompts you to export audit records before the reset occurs.

Monitoring external data sources with MC

By default, Management Console monitors a database using information from that database's Data Collector (DC) tables. MC can also monitor DC tables you have copied into Vertica tables, locally or remotely.

MC administrators provide mappings to local schemas or to an external database containing the corresponding DC data. MC can then render its charts and graphs from the new repository instead of from local DC tables. This offers the benefit of loading larger sets of data faster in MC, and retaining historical data long term.

> **Note**
> **Note:** MC also offers External Monitoring , which allows you to set up a Vertica storage database through the MC interface, then use Kafka to stream your data to the storage database. You can use the Data Source mapping process below if you prefer to set up your own alternative data source, or do not plan to use Kafka streaming.

Map an alternative data source

1. On the MC Settings page, navigate to the Data Source tab.
2. Select the database for which you are creating the data source mapping.
3. Choose the database user for which you want to create the mapping.
4. Set Repository Location to Local or Remote.
5. If Remote is selected, provide JDBC connection parameters for the remote database repository. Click **Validate Connection Properties** to confirm a successful connection.
6. Enter the schema mappings for v_internal and v_catalog. MC does not support mapping the v_monitor schema.
7. Input your table mappings in one of the following ways:
   - Click **Auto Discover** . MC retrieves the table mappings based on the database and schema mappings you provided.
   - Click **Manual Entry** . Manually input table mappings.
   - Click **Load Configurations** . If you previously saved a data source configuration for the database in a file, import the file to use that configuration for the currently selected user.
8. Optionally, click **Save Configurations** to export this configuration file. You can create a mapping for another database user with this configuration file later.
9. Click **Apply** to save and apply your configuration settings.

Reports using unmapped schemas

If a report in MC needs to access a locally stored schema or table that is unmapped, MC includes information from the local DC tables for that schema to complete the report.

For remote configurations, if a report depends on an unmapped schema or table, the entire report is run against the local DC tables. If the remote database is down when MC attempts to run a report against it, MC reruns the report against the local database.

When the MC runs a report, it records missing mappings in the MC log under the INFO severity level.

Monitoring depot activity with MC

The depot is a cache-like component on each node that downloads and stores local copies of table data. Queries that can access the data that they need on the depot, instead of fetching it from communal storage, generally execute much faster. If your database is in Eon Mode, you can use the Depot Activity page to view depot settings and evaluate how efficiently it handles queries and load activity.

To view depot settings and activity, navigate to **Database > Activity > Depot Activity Monitoring** . The Depot Activity page has the following tabs:

- At A Glance
- Depot Efficiency
- **Depot Content**
- Depot Pinning

In this section

- Why monitor the depot?
- Viewing depot activity
- Viewing depot efficiency
- Viewing depot content in MC
- Managing depot pinning policies

Why monitor the depot?

If you run an Eon Mode database on a cloud platform such as AWS, monitoring your depot in MC can help you tune performance and reduce expenses. MC can help address the following questions:

- How often do queries hit the depot versus the S3 bucket?
- Is the depot optimally sized?
- How many API calls are queries executing on the S3 bucket?
- What is the current depot usage on each node?
- Are projections and partitions tuned for best query performance?

To access depot monitoring capabilities: from the MC home page, navigate to **Database > Activity > Depot Activity Monitoring.** See Monitoring depot activity with MC .

How often do queries hit the depot versus the S3 bucket?

Queries run faster when they access node-based depot data rather than fetch it from communal storage. For details, see Query Depot Hits and Misses

Is the depot optimally sized?

To optimize your queries for speed, you might want to resize the depot to fit your query workload. This ensures that queries do not need to spend extra time fetching data from the communal repository on S3. The Eon meta-function ALTER_LOCATION_SIZE lets you change depot size on one node, all nodes in a subcluster, or all nodes in the database. The following statement resizes all depots in the database to 80MB:

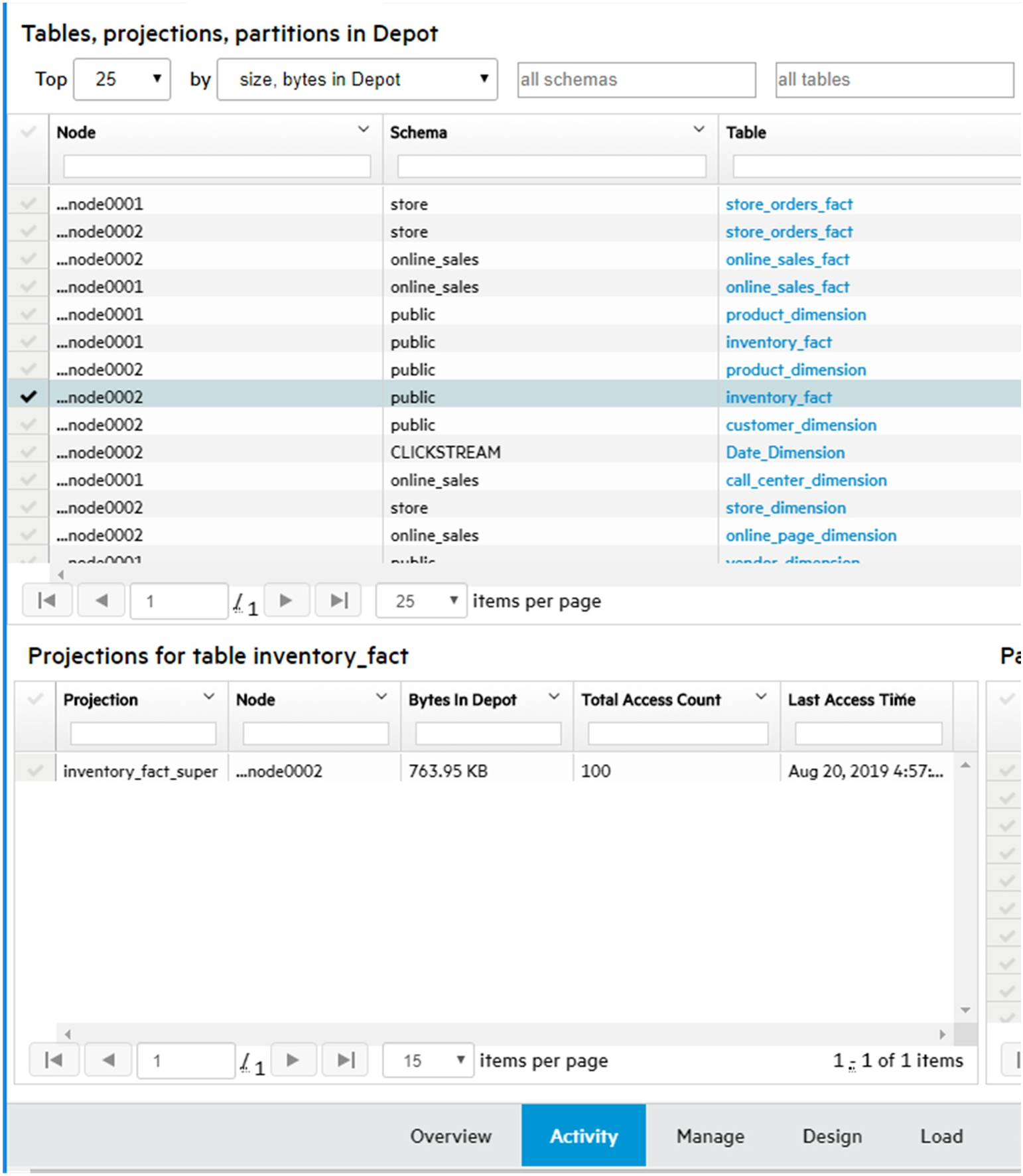
```
=> SELECT alter_location_size('depot', '','80%');
 alter_location_size
---------------------
 depotSize changed.
(1 row)
```

How many API calls on the SP3 bucket are related to queries?

On the Depot Activity Monitoring screen, in the Communal Storage Access Calls chart, MC displays how many of each type of API call your queries executed in a given timespan. To see details on which queries were running, click on any point on the chart.

What is the current depot usage on each node?

The Depot Content tab of the Depot Activity Monitoring page provides detailed information about how each table is using the depot space on the cluster nodes.



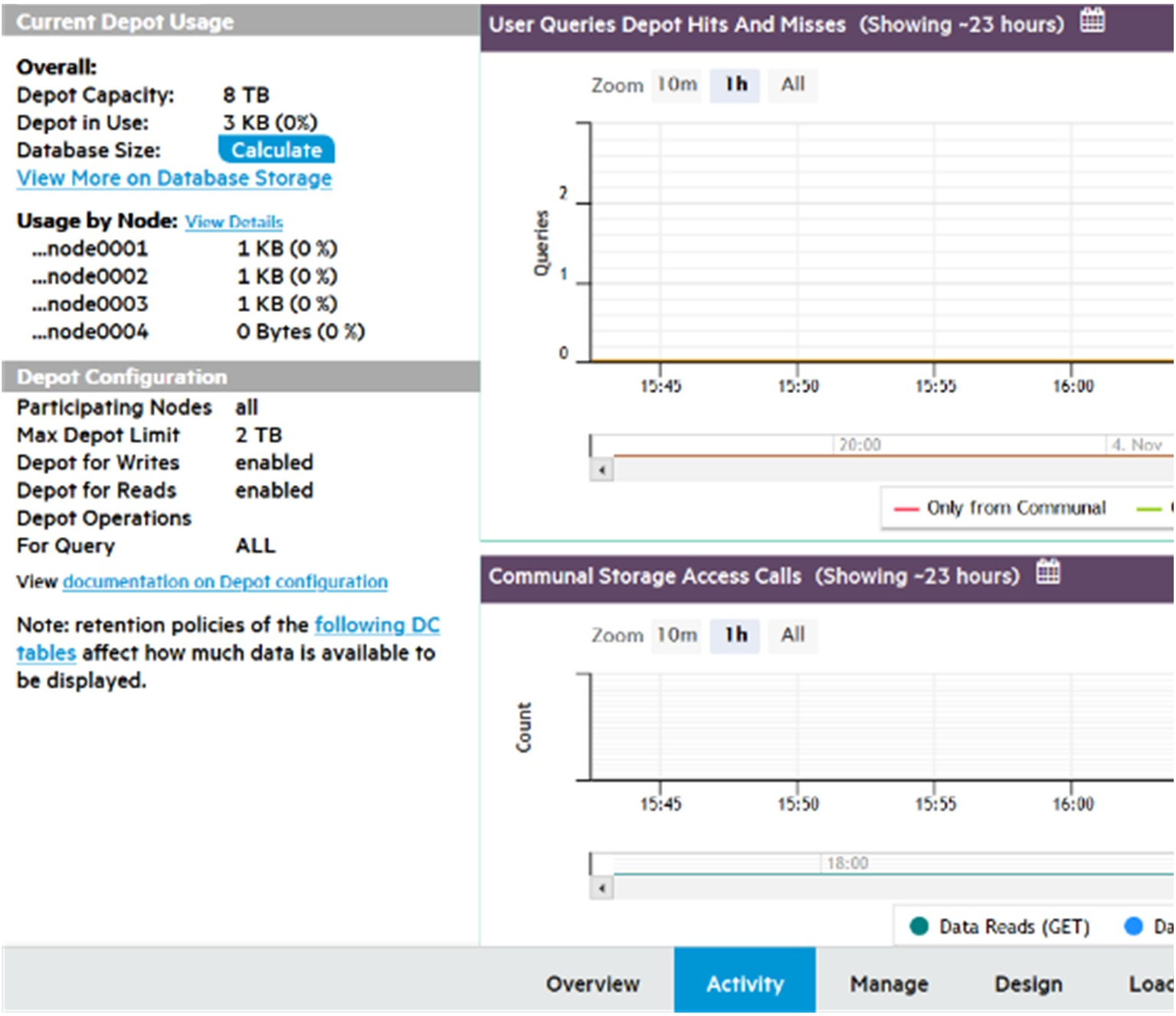
Are projections and partitions tuned for best query performance?

On the Depot Content tab, when you select a row, you are selecting the table depot content on a node. MC loads the details for that table for that node in the bottom section of the page, to show depot content for the selected table, broken down by either projections or partitions on a given node.

Viewing depot activity

The At A Glance screen provides a high level view of depot activity. The screen is divided into several sections:

- Current Depot Usage summarizes depot attributes and usage statistics.
- Depot Configuration shows how the depot is configured.
- User Queries Depot Hits and Misses shows how queries have interacted with the depot over time.
- Communal Storage Access Calls shows frequency of calls to communal storage.



Current depot usage

Displays a summary of depot attributes and usage statistics:

Overall

- **Depot Capacity:** Total depot capacity for all nodes in the database, added together.
- **Depot in Use:** Total depot space currently in use, on all nodes in the database added together.
- **Database Size:** Select **Calculate** to show the total size of the database, in GB.
- **View More on Database Storage:** Click to see the Storage View tab, with details on the storage for this database.

Usage by node

- Lists the number of bytes in the depot and percentage used, for each node in the database.
- **View More:** Click to display depot usage for individual nodes.

Depot configuration

Provides information about how the depot is configured:

- **Participating Nodes:** Number of nodes covered by these statistics.
- **Max Depot Limit:** Total amount of depot space on all participating nodes.
- **Depot for Writes:** Specifies whether the depot is Enabled or Disabled for write operations.
- **Depot for Reads:** Specifies whether the depot is Enabled or Disabled for read operations.
- **Depot Operations for Query:** Displays how system parameter DepotOperationsForQuery is set. This parameter specifies behavior when the depot does not contain queried file data, one of the following:
  - ALL (default): Fetch file data from communal storage, if necessary displace existing files by evicting them from the depot.
  - FETCHES : Fetch file data from communal storage only if space is available; otherwise, read the queried data directly from communal storage.
  - NONE : Do not fetch file data to the depot, read the queried data directly from communal storage.
- A link for querying internal DC tables, to obtain retention limits on depot activity such as Depot Reads.

User queries depot hits and misses

For optimal performance, the majority of queries should access data that is locally stored on the depot. To maximize depot access, make sure that your depot is large enough to accommodate frequently accessed data. Otherwise, Vertica must access communal storage more often to retrieve required data, which can significantly affect query performance.

User Queries Depot Hits and Misses helps you evaluate how queries have interacted with the depot over time.

- Color-coded graph lines show how many queries were accessing the depot or communal storage, or both, at any given time.
- The left y-axis indicates the number of queries.

Depot fetches and evictions

When a query fetches data from communal storage to a depot that lacks enough space for the new data, Vertica attempts to evict older data. The User Queries Depot Hits and Misses chart can help you monitor churn—that is, how many files are evicted from the depot, and how often:

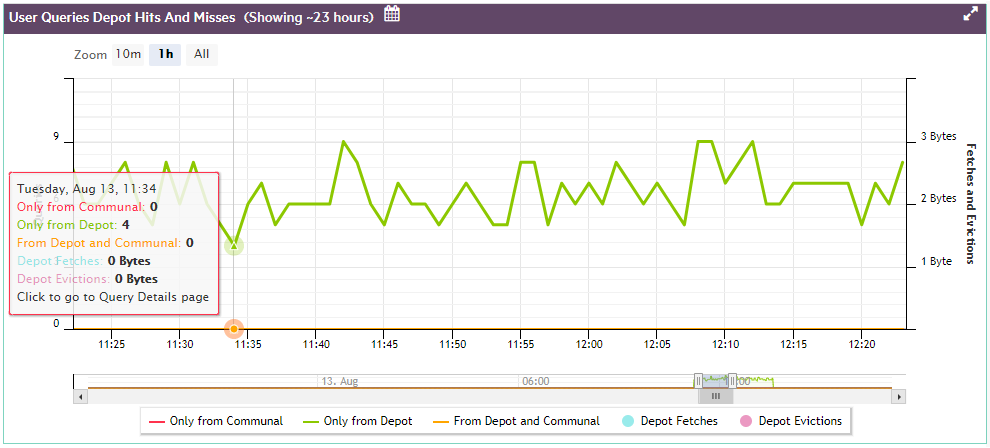
- Colored bars show the moments of depot fetches and evictions, as measured in megabytes.
- The right y-axis shows how much data was fetched or evicted.

If you observe that queries are consistently slower due to accessing communal storage, and notice the depot keeps experiencing frequent churn, it's likely that you need to increase depot size .

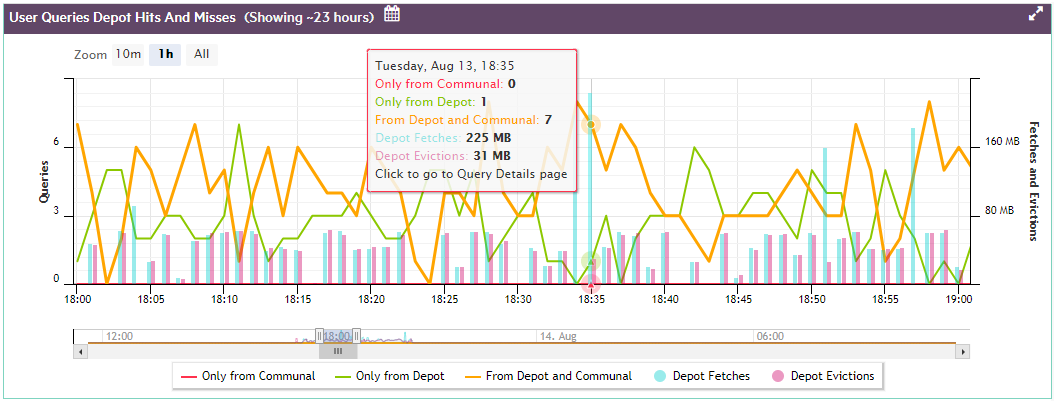
Depot query details

- Hover over a point on the query line to see details about the number of queries that ran.
- Hover over a Fetches or Evictions bar graph to see details about the number of bytes fetched or evicted.
- Click the line or bar to view the Query Details page, which provides information about every query that ran in the selected timespan.

The following example shows a depot size sufficient to run all queries in the depot:

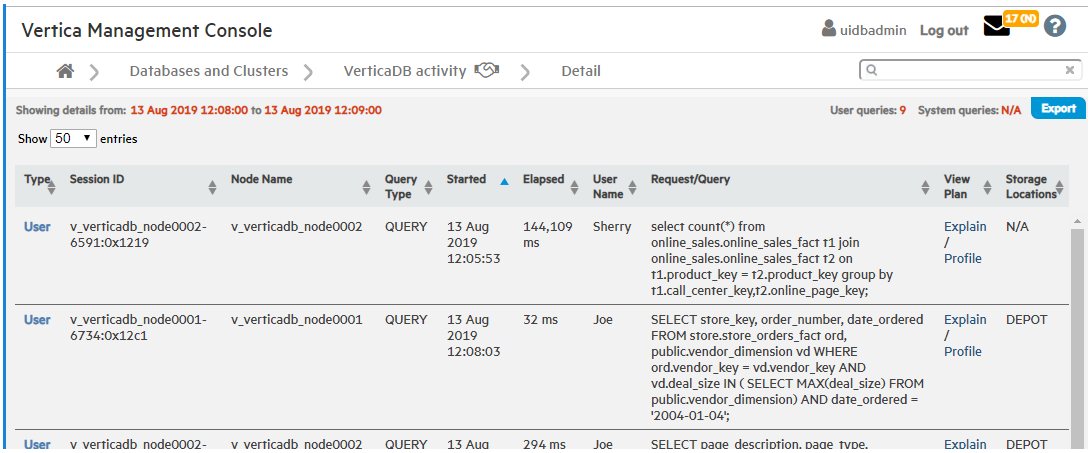


The next example shows what happens when the depot is too small for ongoing query activity, so a number of queries are forced to fetch their data from communal storage.



If you click on any point on the line, MC opens a Query Detail window that shows:

- All queries represented by that point
- Details for each query



Communal storage access calls

Shows how many communal storage access calls (for example, AWS S3 API calls) of each type your database has executed over a given time span, one of the following:

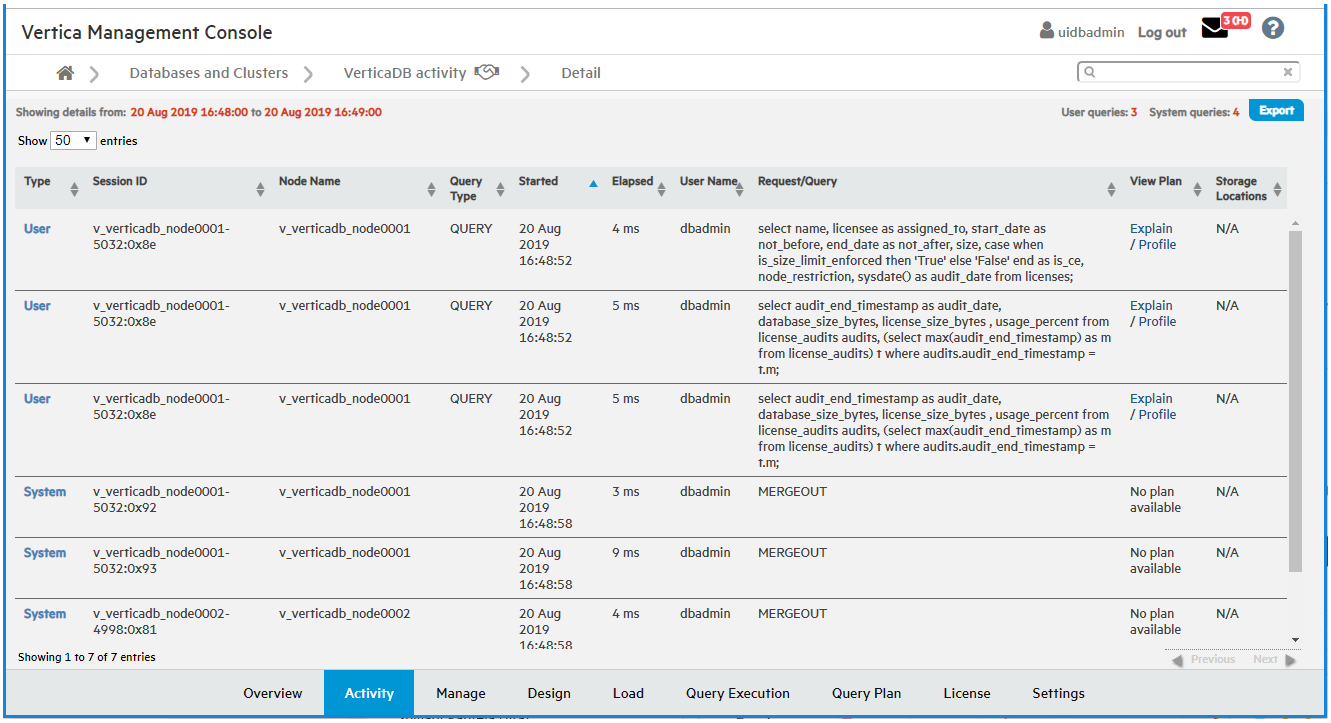
- Data Reads (GET)
- Data Writes (PUT
- Metadata Reads (LIST)
- Metadata Writes (POST, DELETE, COPY)

Hover over any point on the Communal Storage Access Calls chart, to view a summary of data about that point. For example, if your cluster is on AWS, the summary lists how many of each AWS S3 API call type were executed in the selected timespan.

Click on any point on the bar graph to view details on:

- All queries that ran during the selected timespan. These queries executed the API calls listed for that timespan on the Communal Storage Access Calls chart.
- Details on each query.

For example:



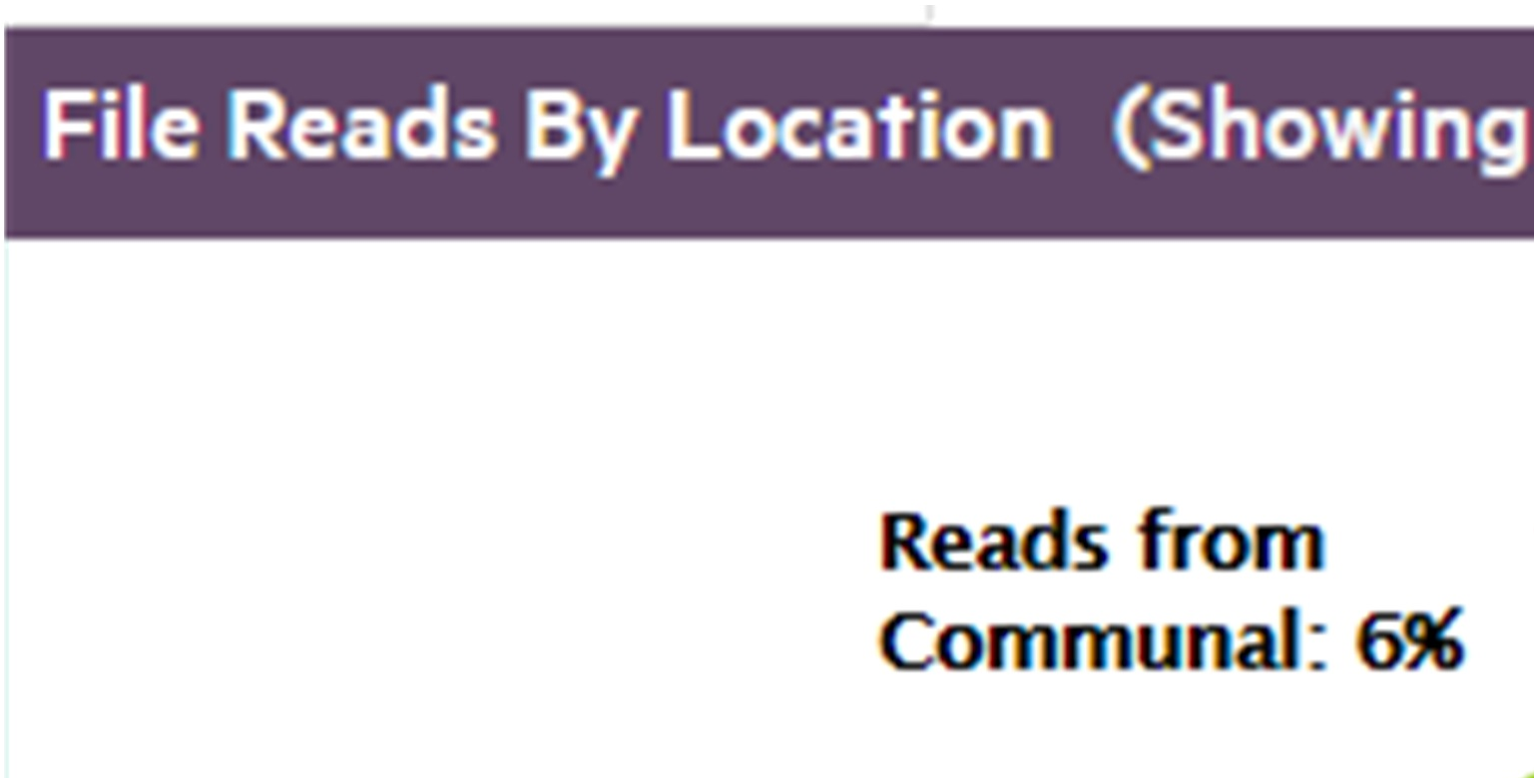
Viewing depot efficiency

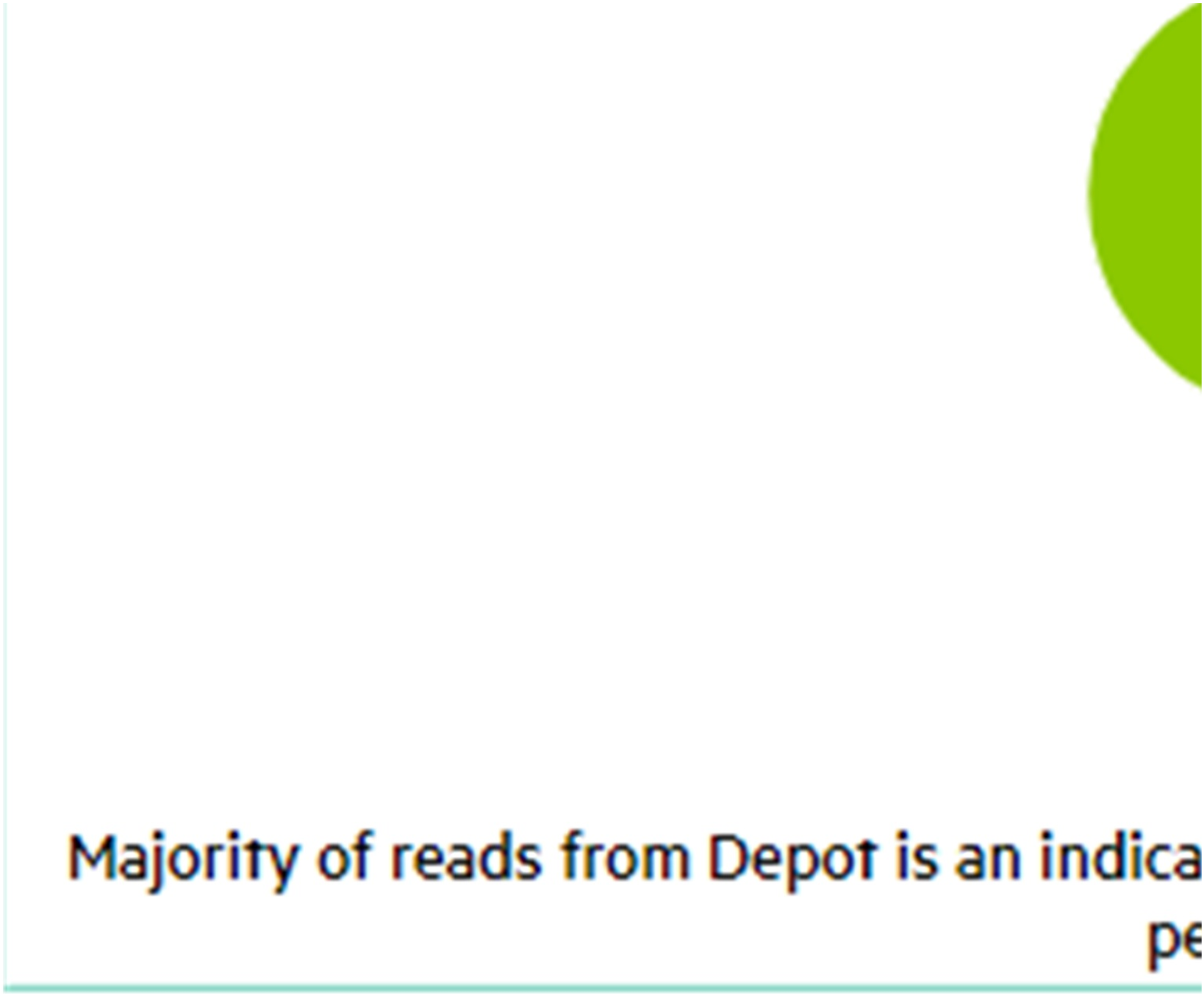
The Depot Efficiency tab provides several graphics that can help users quickly determine whether the depot is properly tuned.

- File reads by location
- Top 10 re-fetches in depot
- Depot pinning
- Number of tables in depot by age
- Number of tables in depot by access count
- Number of tables in depot by size

File reads by location

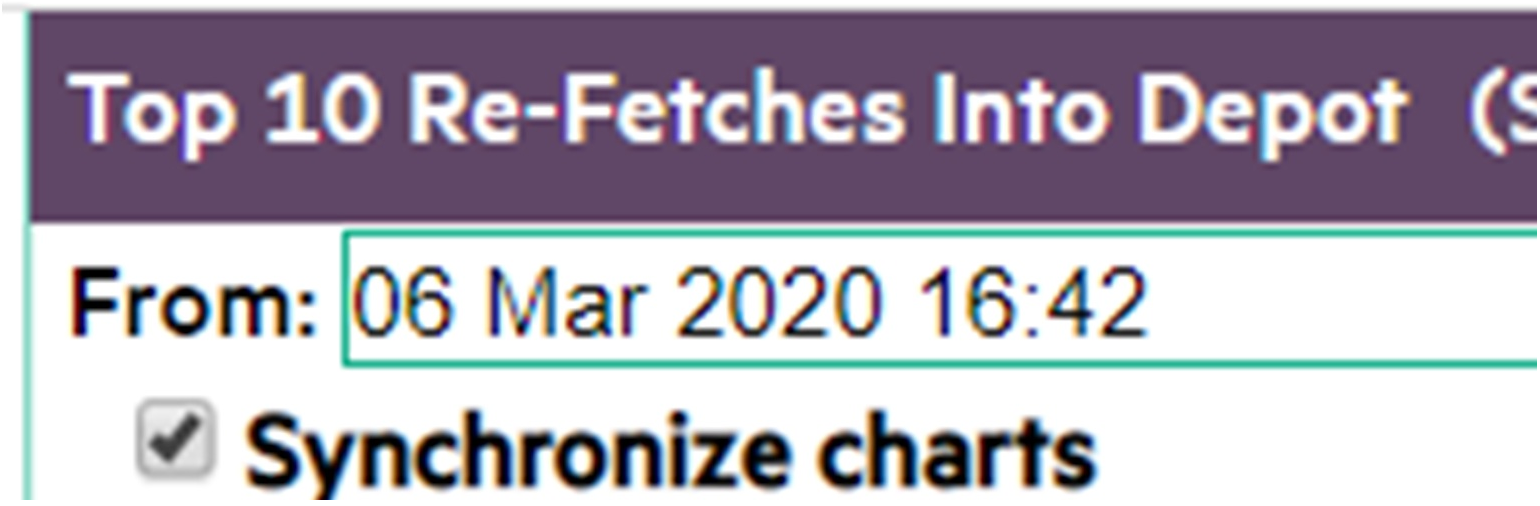
Shows the percentage of reads from depot and communal storage over the specified time span. In general, you want the majority of queries and other read operations to obtain their data from the depot rather than communal storage, as verified by the chart below.If communal storage shows a large percentage of file reads, it's likely that you need to increase the depot size.

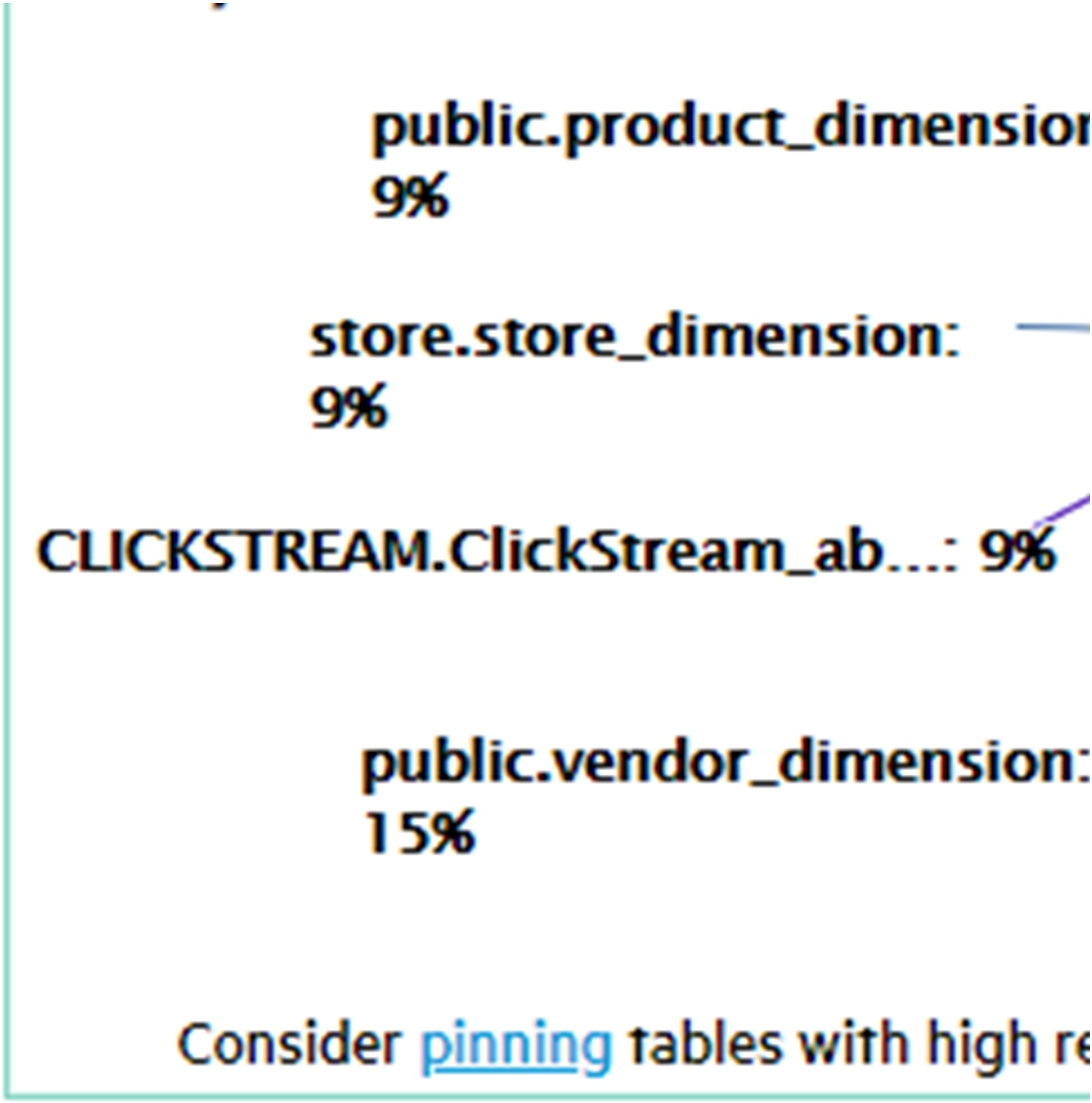




Top 10 re-fetches in depot

Vertica evicts data from the depot as needed to provide room for new data, and expedite request processing. Depot fetches and evictions are expected in a busy database. However, you generally want to avoid repeated evictions and fetches of the same table data. If this happens, consider increasing the depot size, or pinning the table or frequently accessed partitions to the depot.

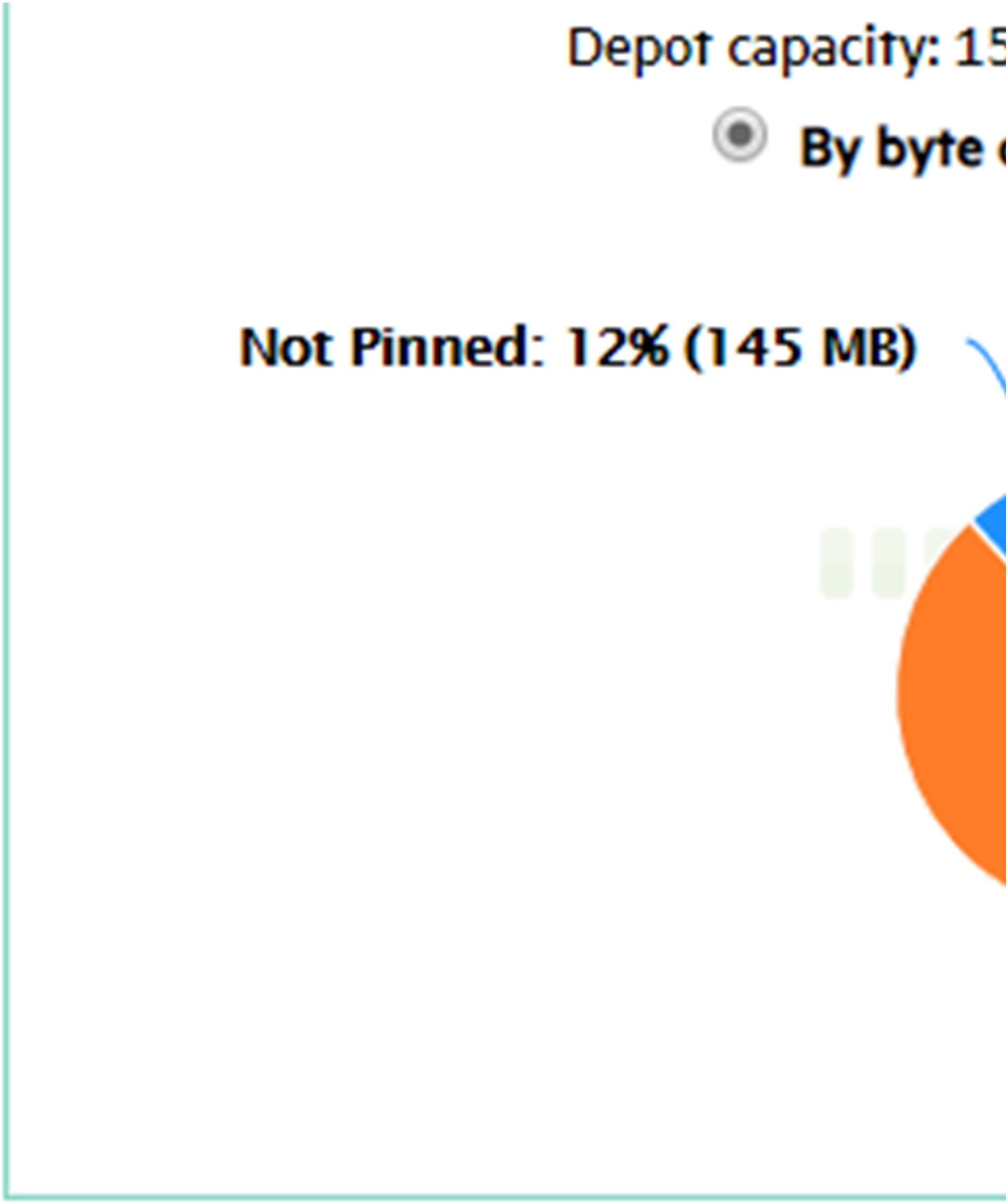




Depot pinning

It's often advisable to pin a table or table partitions whose data is frequently queried. Doing so reduces their exposure to eviction from the depot. However, you should also be careful not to use up too much depot storage with pinned data. If too much depot space is claimed by pinned objects (as shown below), the depot might be unable to handle load operations on unpinned objects.

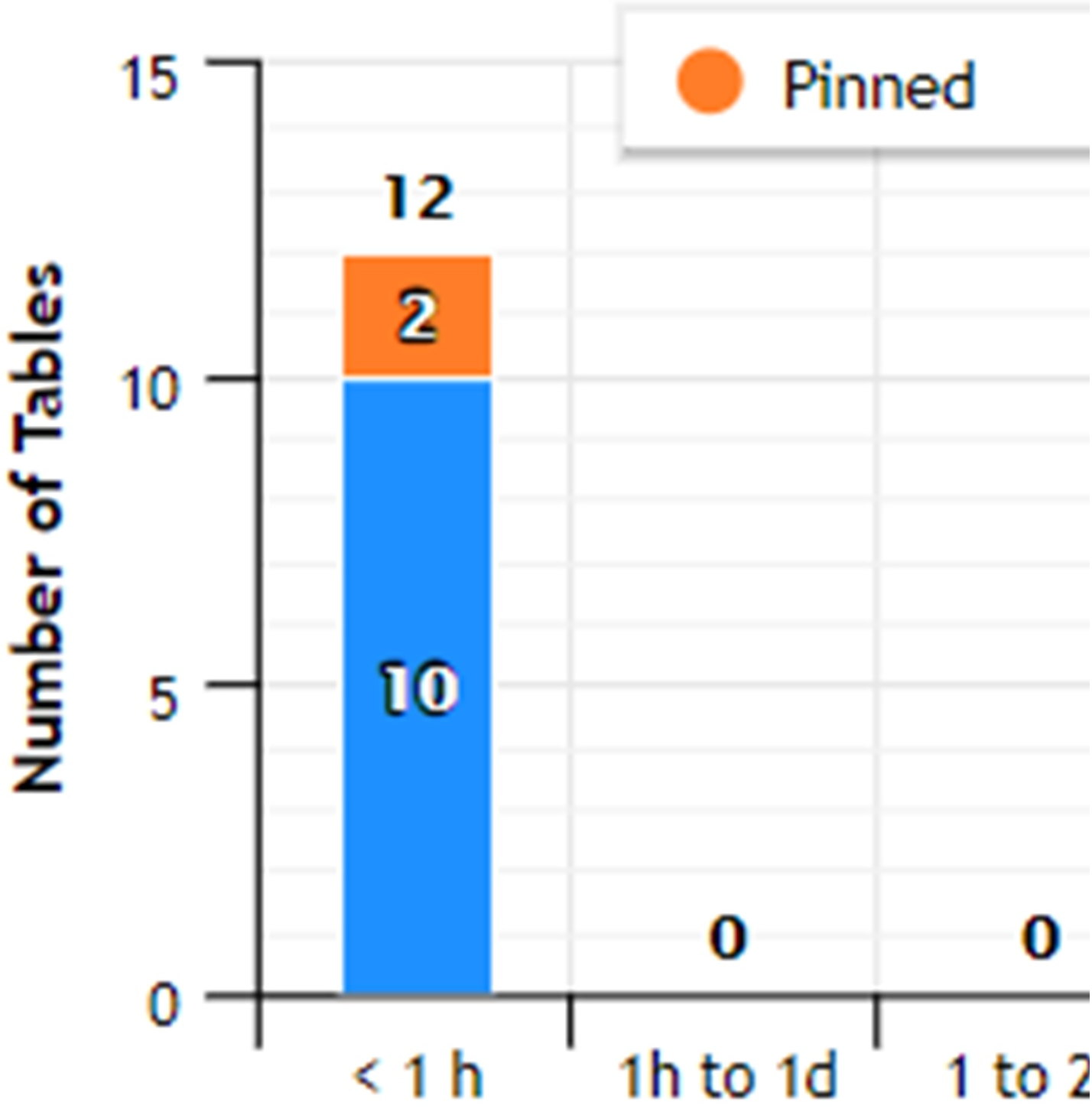




Number of tables in depot by age

Tables should typically reside in the depot for as long as their data are required. A short average lifespan of table residency might indicate frequent depot eviction, which can adversely impact overall performance. If this happens, consider increasing the depot size, or pinning frequently accessed table data.

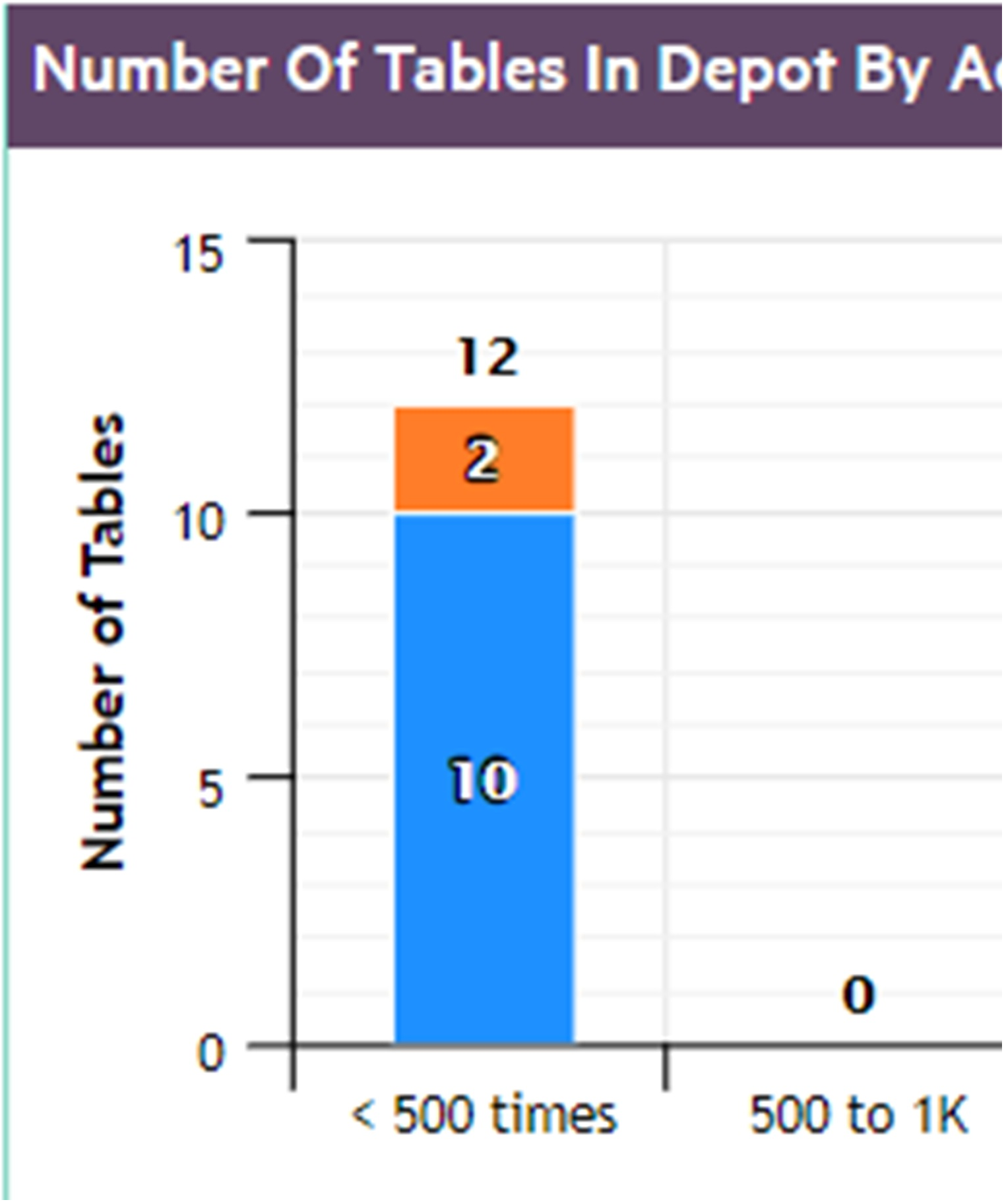
Number Of Tables In Depot By Ag



Large count of tables staying in Depot f
churn. This database w

Number of tables in depot by access count

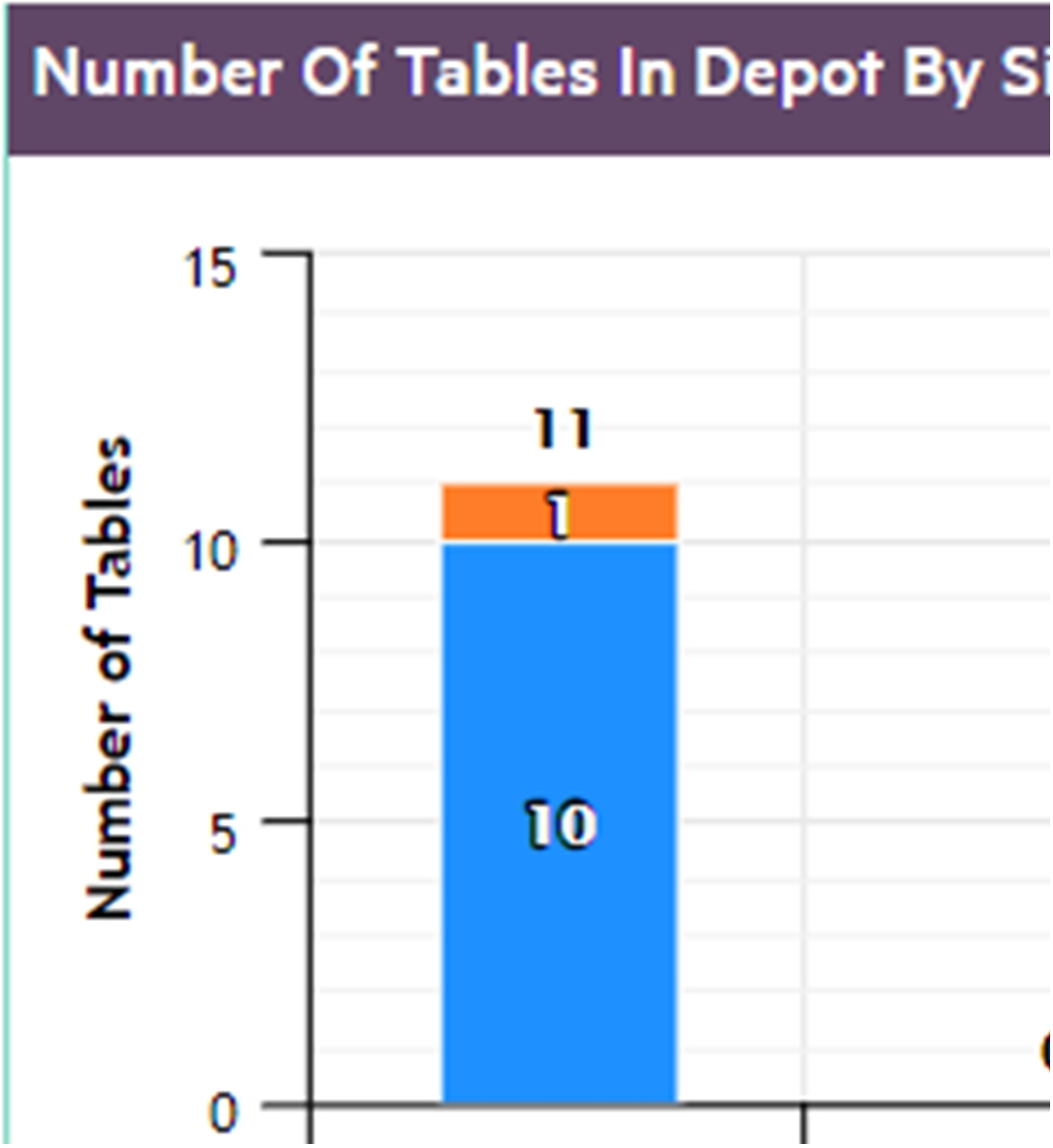
In general, the depot should largely be populated by tables that are frequently accessed, both pinned and unpinned.

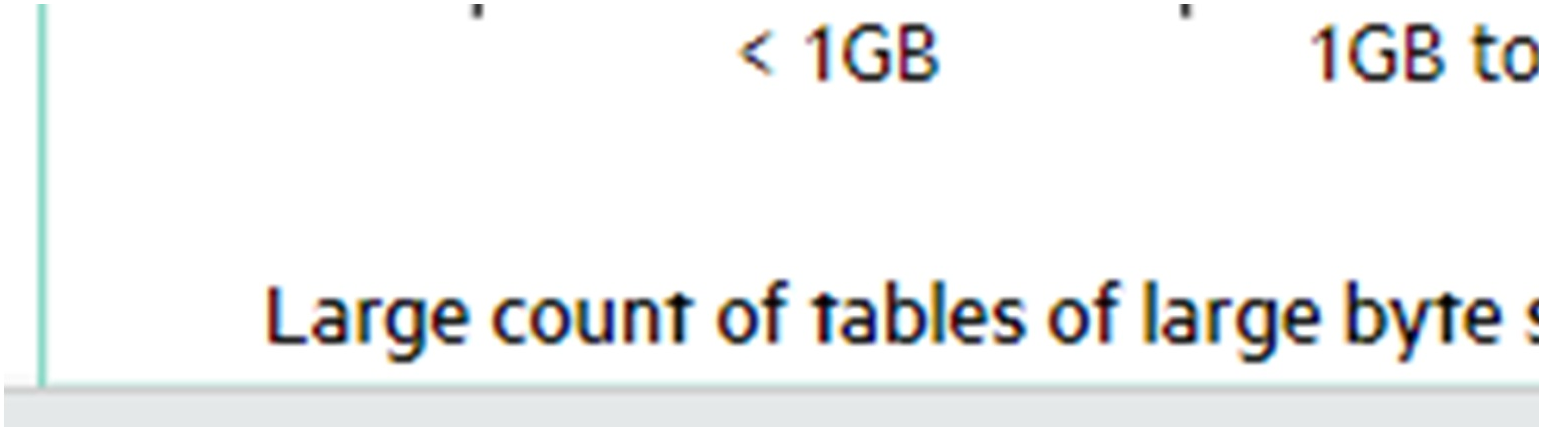


Large count of tables stayin

Number of tables in depot by size

It can be helpful to know the distribution of table sizes in the depot.





Viewing depot content in MC

You can view in detail how the nodes in your Eon database are using the depot:

- Display a list of tables with the largest amount of data in the depot.
- Use the filter fields to list the tables most frequently or most recently accessed in the depot.
- Display details about how frequently the projections and partitions for a specific table access the depot, and the last time the depot was accessed.

The **Depot Activity Monitoring > Depot Content** tab opens showing a default list of the top 25 tables in the database, as ranked by each table's total bytes in the depot. The list shows all the nodes for each of those top tables. The nodes are sorted solely based on most bytes in depot, so the nodes for a given table do not necessarily appear together.

Filter the list

You can use the filter fields above the table to focus the list more narrowly. The filters let you select:

- The number of top tables
- Whether the tables are selected by most bytes in depot, the highest number of times their depot was accessed, or the most recent last access time
- Tables in all schemas, or only in a specific schema
- All tables, or only a specific table
- All nodes, or only a specific node

In the Schema, Table, and Node filter fields, you can enter a text string to select all items whose names include that text string.

Select a node to see the breakdown of depot data in projections and partitions

Select a row in the top table. MC then loads the details to show how that table's depot content is distributed across the projections and the partitions for that table, that are on that node. The Projection and Partition panes show these details for the selected node:

- **Projection:** Number of bytes of data for the selected table that each projection has in the depot on the selected node.
- **Partition:** If the table is partitioned, this pane shows the number of bytes of data for the selected table that each partition has in the depot on the selected node.

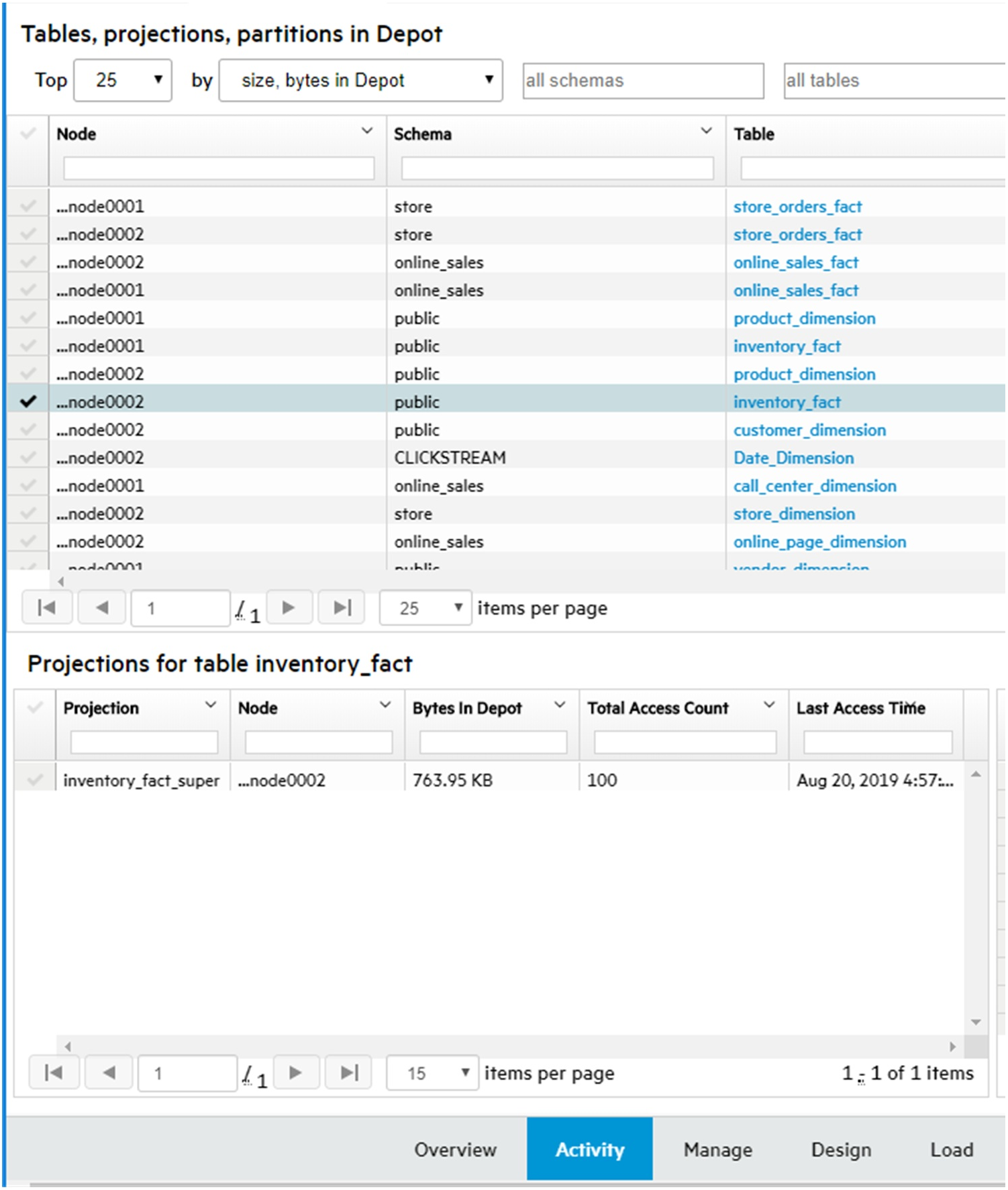
For each projection and each partition, MC also displays the total number of times, that the projection or partition has accessed the depot on that node, and the last access time.

For more information about projections, see Projections .

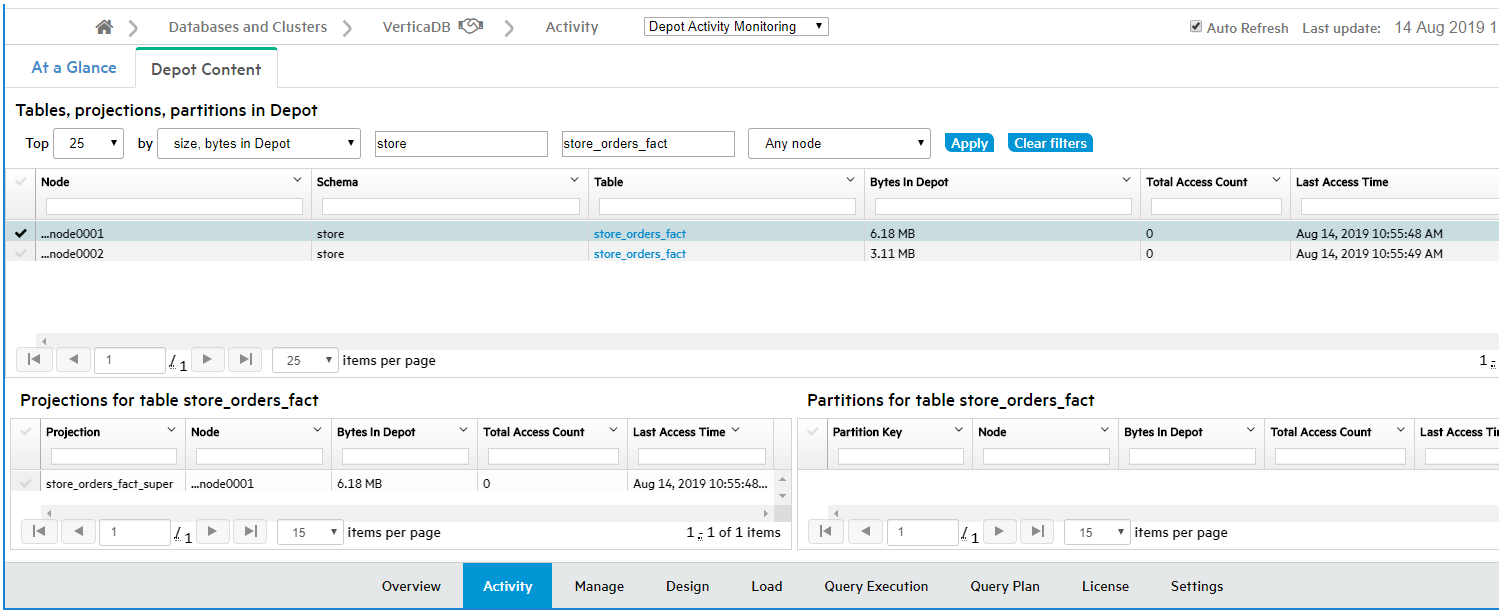
For more information about partitions, see Partitioning tables .

Steps to monitor depot content

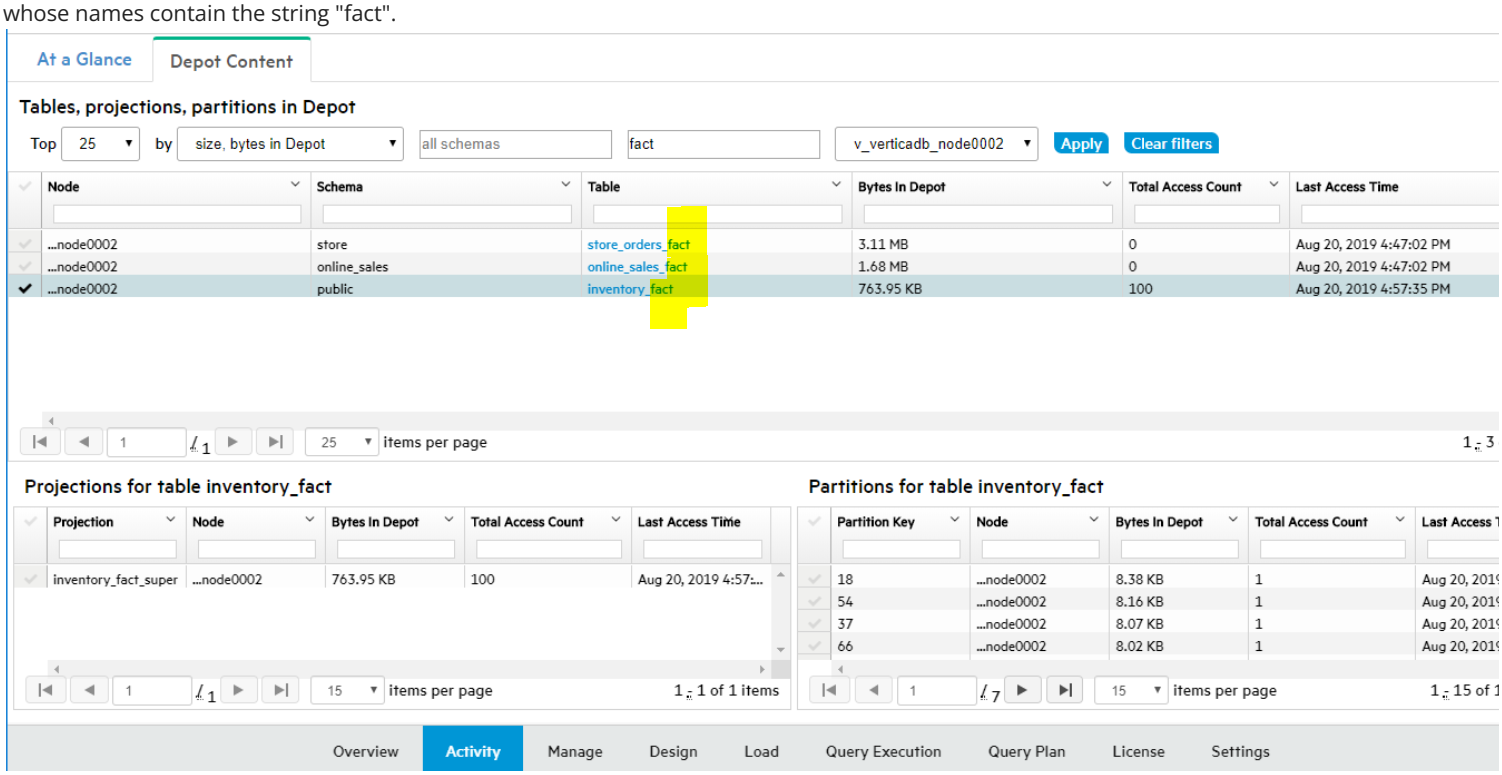
1. From the MC home page, open a database, select the **Activity** tab from the bottom menu, select **Depot Activity Monitoring** in the top selection box, and select the **Depot Content** tab. MC displays the top N tables (25 by default), ranked by the number of bytes of data each table has in the depot on all its nodes.



2. To narrow the list, use the filters at the top of the tab. You can show only the nodes in a certain schema and/or database, or display all the activity on a specific subgroup of nodes. Change the filters, then click **Apply** .



3. To select all items whose names contain a certain text string, enter that text string in a filter field. This example selects the nodes for the tables whose names contain the string "fact".



4. To display details on the projections and partitions for a specific table that are accessing the depot, select a row in the top pane of the **Depot Content** tab.

See also

[Monitoring depot activity with MC]

Managing depot pinning policies

Vertica [evicts] data from depots as needed to provide room for new data, and expedite request processing. You can pin database objects to reduce the risk of depot eviction. Two object types can be pinned: tables and table partitions.

The Depot Pinning tab lets you perform the following tasks:

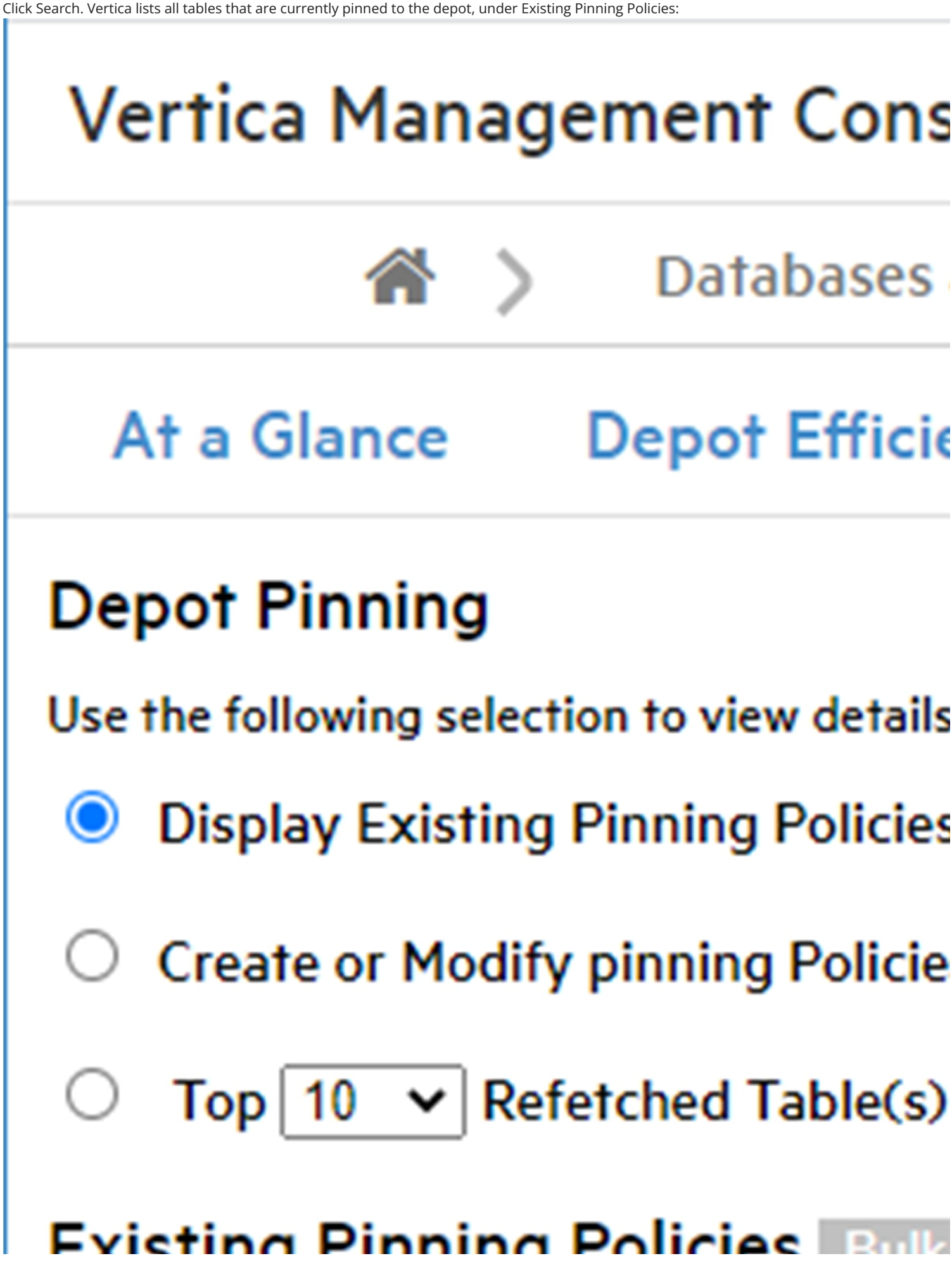
- [List and modify current pinning policies] .
- [Create] and [remove pinning policies] .
- [View tables that are frequently fetched from communal storage] .

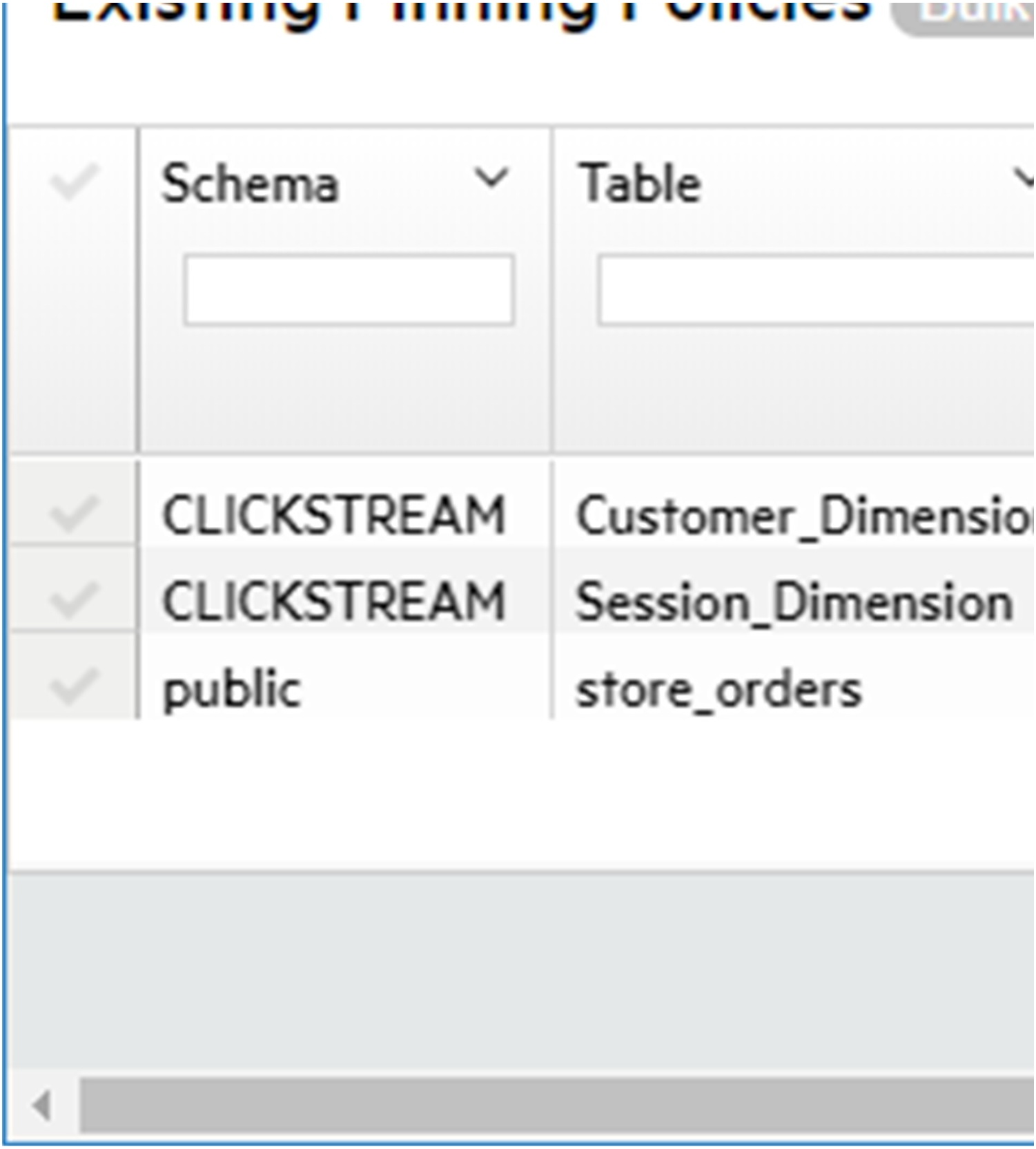
For details on pinning policies, see [Managing depot caching] .

Listing pinning policies

To list existing depot pinning policies:

1. Select Display Existing Pinning Policies.
2. Click Search. Vertica lists all tables that are currently pinned to the depot, under Existing Pinning Policies:





3. If desired, filter and sort the list of policies by setting the following fields:

Filter on:	**Set to:**
Schema	Full or partial name of the desired schema
Table	Full or partial name of the desired table
Policy Type	Table or Partition

Policy Scope	Database Level or subcluster name
Partitioned Table	Y (yes) or N (no)

Sort on:	**In ascending/descending order:**
Size in Depot	Absolute size (in MB) of cached data
% of Depot	Percentage of depot storage used by the cached data
Total Access Count	Number of times that cached data from this table has been queried or otherwise accessed
Last Access Time	Last time cached data of this table or partition was queried or otherwise accessed

Removing existing policies

You can also use the result set under Existing Pinning Policies to remove one or more policies.

To remove one or more table policies:

- From the policy list, select the check boxes of policies to remove.

> **Note**
> Policy Type of all selected policies must be set to Table.

- Click Bulk Remove Table Policies.

To remove a table's partition policies:

1. On the policy to remove, click Modify Policy.
2. In the Modify Pinning Policy dialog, perform one of the following actions:
   - Click Remove Policy on the desired policy.
   - Select the check boxes of one or more policies, then click Remove Selected Policies.
3. Click Close.

Creating pinning policies

You can create a policy that pins table data to a subcluster depot or to all database depots. You can specify the following policy types:

- Table: Pins all table data
- Partition: Pins the specified range of partition keys.

Find candidates for pinning

1. Select Create or Modify Pinning Policies.
2. Optionally filter the search by specifying a schema and the full or (for wildcard searches) partial name of a table.
3. Click Search.

> **Tip**
> To further refine and sort the result set, set one or more of the search fields above the table list .

You can use the filtered data to determine which tables or partitions are good candidates for depot pinning. For example, a high total access count relative to other tables (Total Access Count) might argue in favor of pinning. This can be evaluated against data storage requirements (% of Depot) and age of the cached data. For example, if pinned objects claim too much storage, a depot might be required to:

- Route large load operations directly to communal storage for processing.
- Increase frequency of evictions.
- Increase frequency of fetches from communal storage to handle queries on non-pinned objects.

All these cases can adversely affect overall database performance .

Tip
To minimize contention over depot usage, consider the following guidelines:

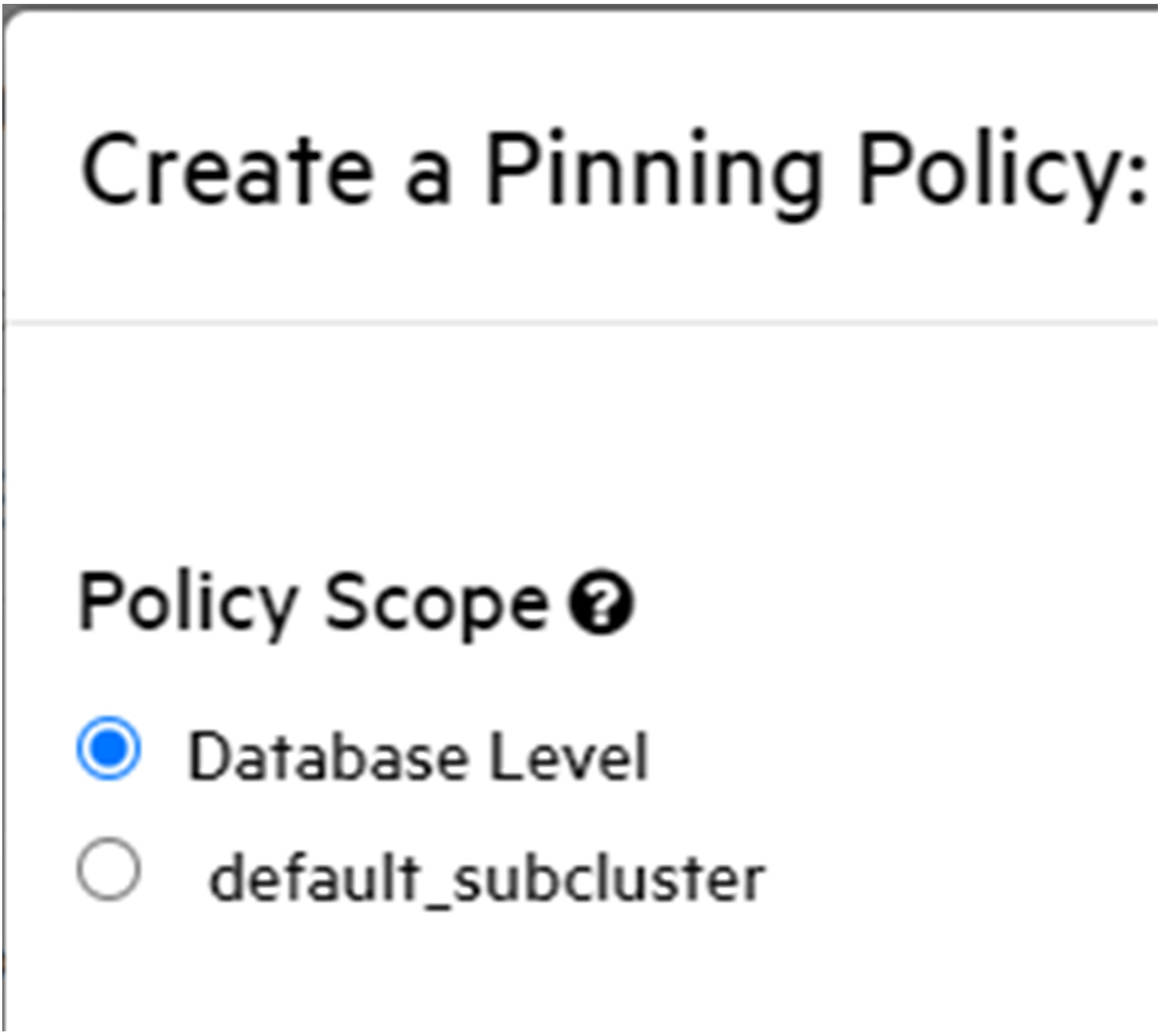
- Pin only those objects that are most active in DML operations and queries.
- Minimize the size of pinned data by setting policies at the smallest effective level—for example, pin only the data of a table's active partition .

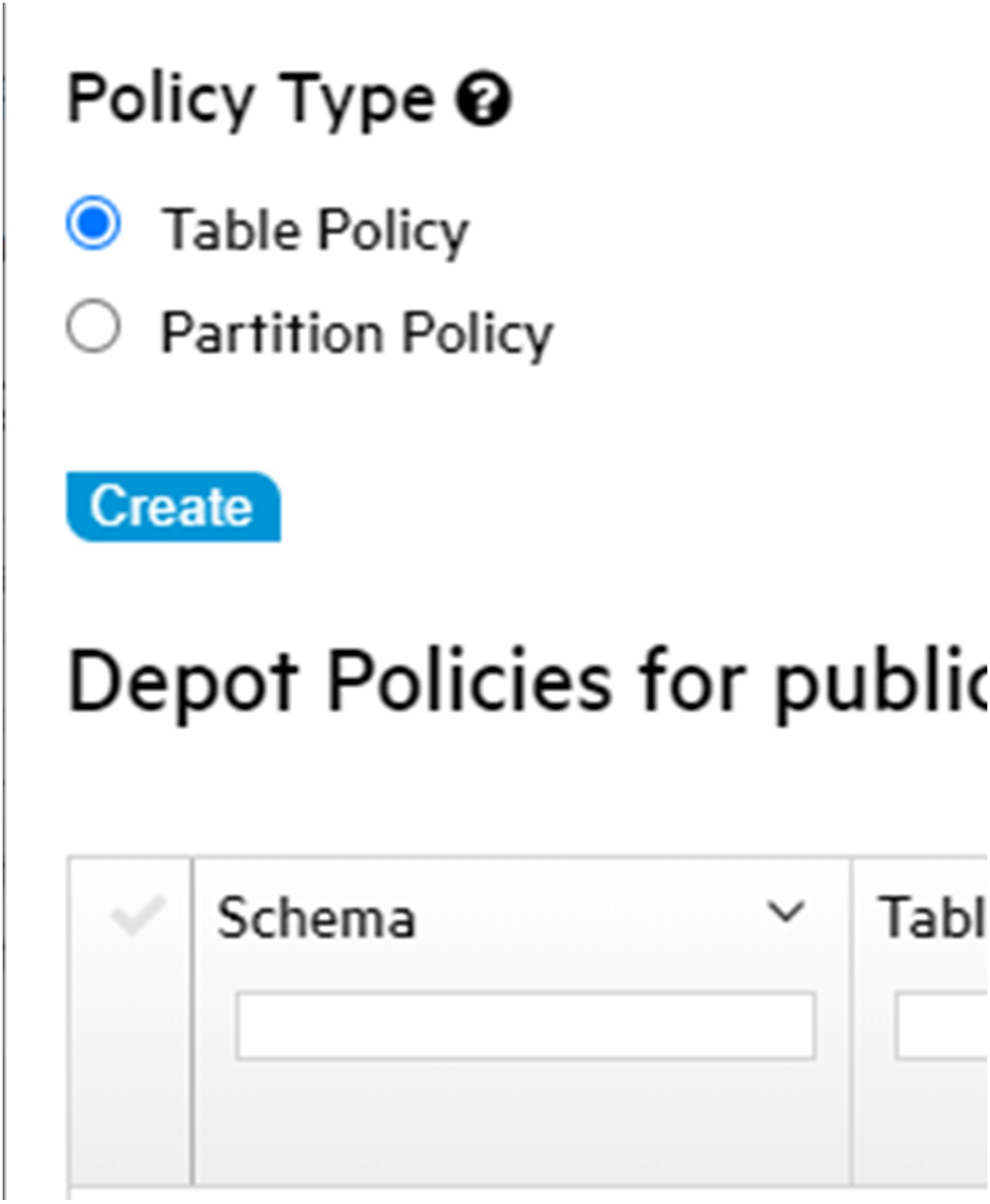
For details on how Vertica handles depot storage and turnover, see Managing depot caching .

Create a table or partition pinning policy

To create a pinning policy for a single table or table partition:

1. Under the Create or Modify Pinning Policies list , find the table to pin.
2. Click Create Policy. The Create a Pinning Policy dialog opens.
3. Select the desired policy scope, one of the following:
   - Database
   - An available subcluster
4. Select the desired policy type: Table Policy or Partition Policy
   Table Policy
   Click Create:





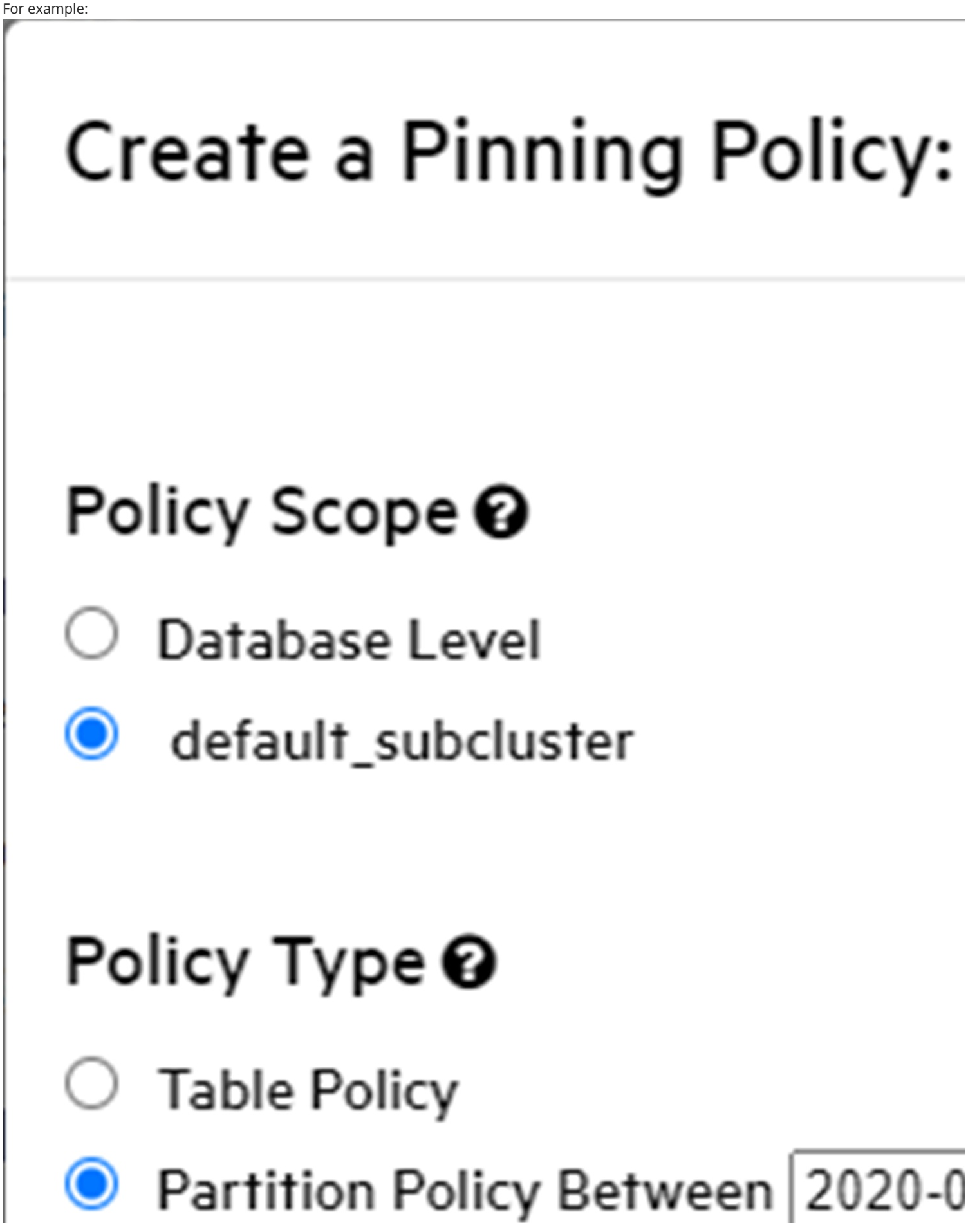
Partition Policy
(available only if the table is partitioned)

- Enter the minimum and maximum partition keys.

> **Note**

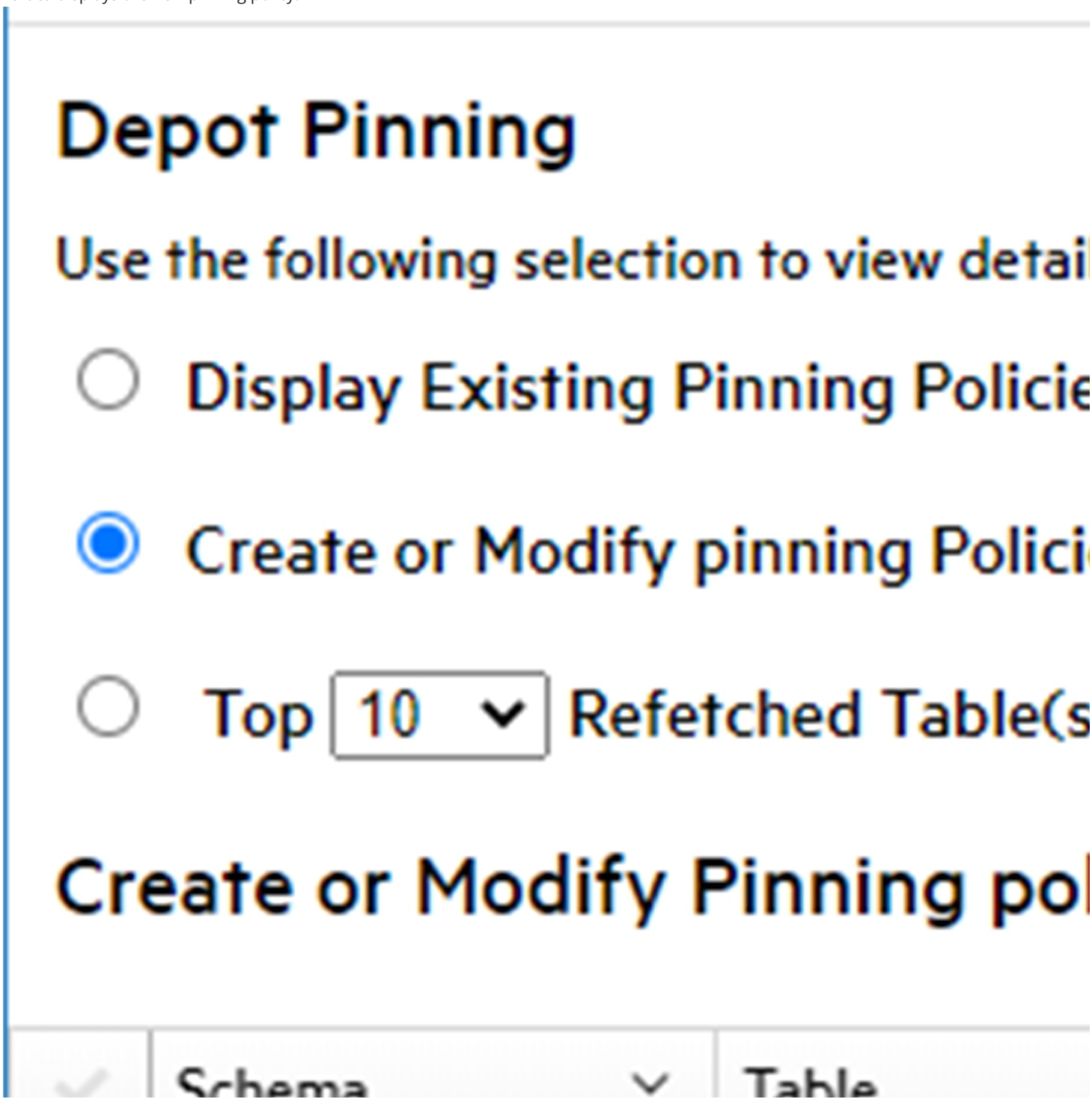
The MC shows a sample range of valid keys for this partition.

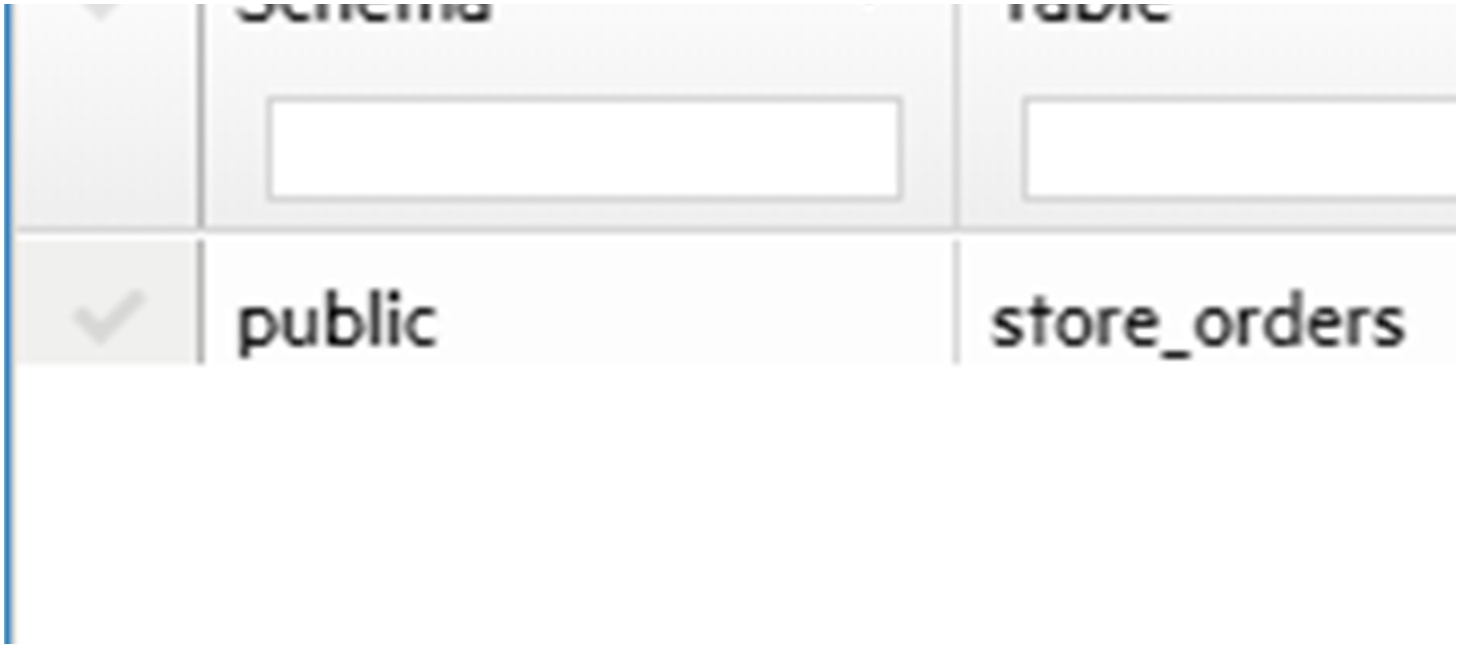
For example:





- Click Create.
  Vertica displays the new pinning policy:





- Optionally, add more partition-level policies on the same table by setting new partition keys.

5. When finished, click Close.

> **Note**
> If partition pinning policies on the same table specify overlapping key ranges, Vertica collates the partition ranges. For example, if you create two partition policies with key ranges of 1-3 and 2-4, Vertica creates a single pinning policy with a key range of 1-4.

Create pinning policies on multiple tables

To create a pinning policy on multiple tables:

1. On Create or Modify Pinning Policies, select the check boxes of the tables to pin.

   > **Note**
   > All checked tables must be unassigned to a pinning policy, as indicated by their Create Policy link.

2. Click Bulk Create Table Policies. The Bulk Create Table Policies dialog opens.
3. Select the desired policy scope, one of the following:
   - Database
   - subcluster (choose the desired subcluster)
4. Click Create, then click Close.

Removing a pinning policy

To remove an existing pinning policy:

1. On Create or Modify Pinning Policies, find the table with the policy to remove.
2. Click Modify Policy.
3. In the Modify Pinning Policy dialog, perform one of the following actions:
   - Click Remove Policy on the policy to remove.
   - Select the check boxes of one or more policies, then click Remove Selected Policies.
4. Click Close.

Remove pinning policies from multiple tables

To bulk-remove pinning policies from one or more tables:

1. On Create or Modify Pinning Policies, select the target table check boxes.

   > **Note**
   > All checked tables must comply with the following requirements:

- They must be assigned to a pinning policy as indicated by their Modify Policy link.
- Their pinning policy type must be set to Table.

2. Click Bulk Remove Table Policies. The Bulk Remove Table Policies dialog opens.
3. Click Remove, then click Close.

Viewing frequently fetched tables

You can query the depot for tables that are most frequently fetched from communal storage. This can help you quickly identify potential candidates for depot pinning:

1. Select Top *num* Refetched Tables(s) from Depot.
2. Specify the maximum number of results to return (by default 10), and the range of dates to query.

Tip
To further refine and sort the result set, set one or more of the search fields above the table list .

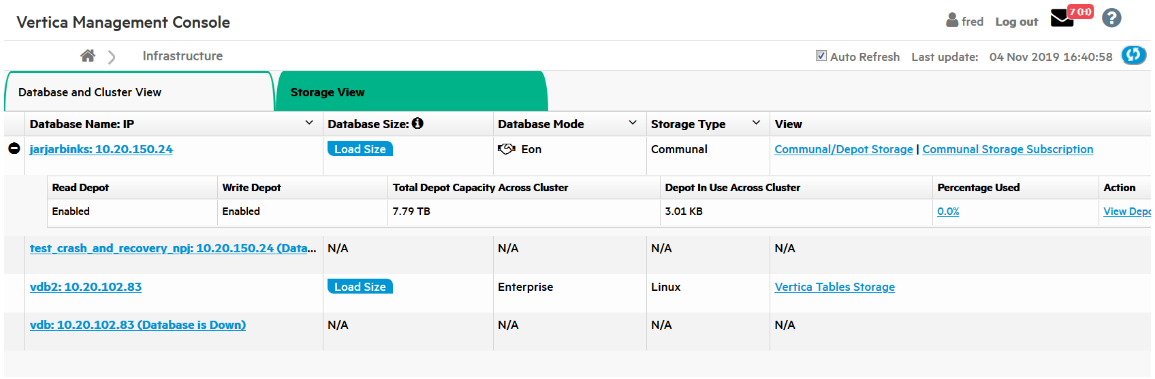
From the list, you can perform the following tasks:

- Create or remove pinning policies on individual tables and partitions by clicking on the desired action— Create Policy or Modify Policy .
- Select multiple tables and then remove their pinning policies. See Remove Pinning Policies from Multiple Tables .

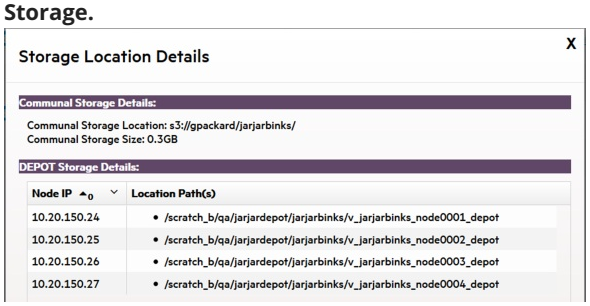
Monitoring depot storage with MC

To display detailed storage monitoring information for your Eon database:

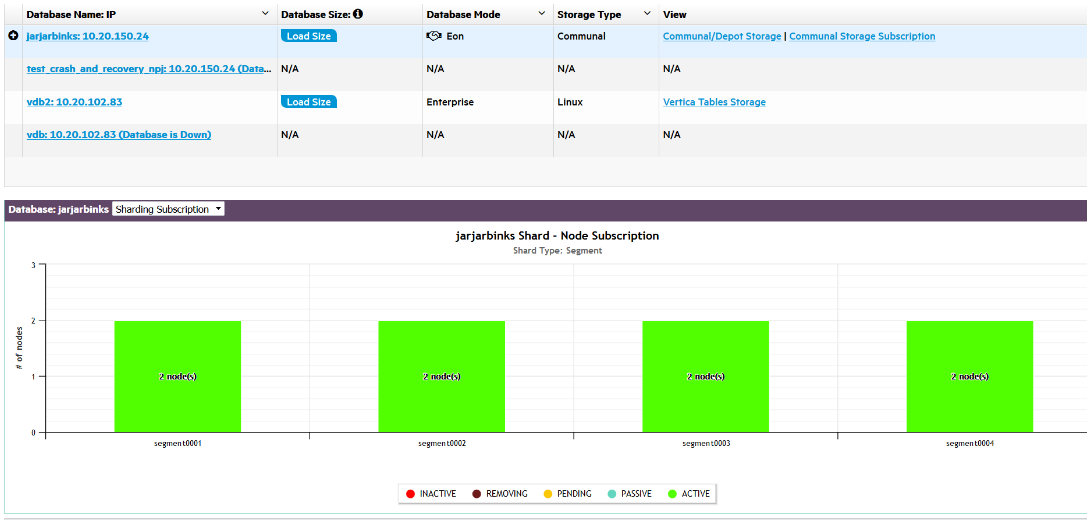
1. From the MC home page, select **View Your Infrastructure.**
2. On the Infrastructure page, select the **Storage View** tab.MC displays the **Storage View** screen, with details about the database storage and links to further detail screens:



3. To see the loaded size of the database, click **Load Size.**
4. To see communal storage details for the database, such as its location and size, and the IP addresses of the nodes, click **Communal/Depot Storage.**



5. To view the shard subscriptions for your Eon nodes, click **Communal Storage Subscription** . MC displays the shard type, how many nodes are subscribed to each shard, and the status of each shard subscription (Active, Inactive, Passive, Pending, Removing).

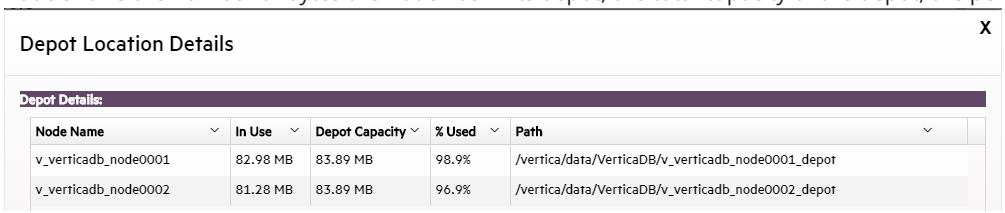


There are two views:

- **Sharding Subscription** displays how many nodes store each shard.
- **Node Subscription** displays how many shards are on each node.

Hover over a bar to display the details.

6. To display the depot details for all nodes in the database, click **View Depot Details by Nodes** . MC lists the nodes by node name, and for each node shows the number of bytes the node has in its depot, the total capacity of the depot, the percent used, and the path to the node's depot.



See also

Monitoring depot activity with MC

Extended monitoring

Enabling extended monitoring allows you to monitor a longer range of data through MC. This can offer insight into long-term trends in your database's health. MC can also continue to display your monitored database's dashboard while it is down.

Extended monitoring uses Kafka to stream monitoring data from your monitored databases to a single MC storage database. MC can query the storage database instead of your monitored database to render some of its charts, reducing impact on your monitored database's performance.

How extended monitoring works

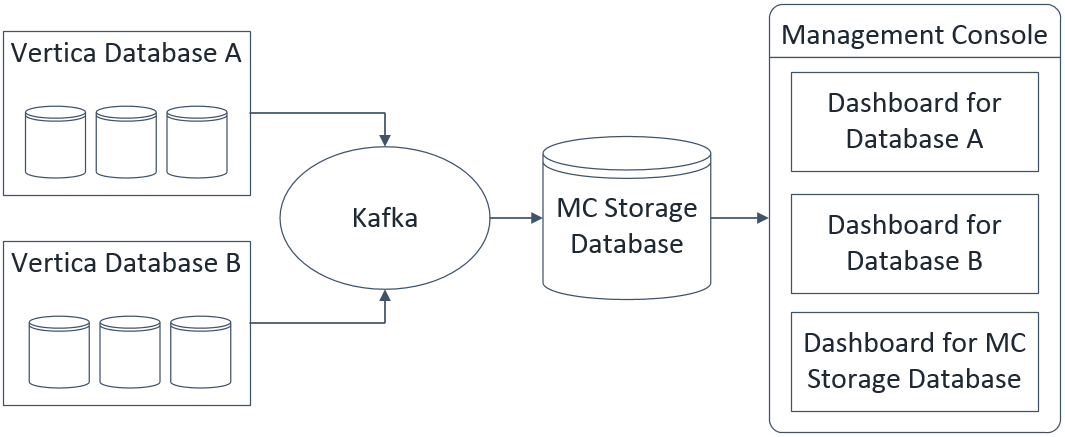
By default, MC monitors your database by querying it directly for monitoring data about system activities, performance, and resource utilization. Typically, the Data collector stores all monitoring data in data collector (DC) tables. However, DC tables have limited retention periods. See Data collector utility .

Extended monitoring stores your database's monitoring data in a dedicated storage database. Vertica streams data from your database's DC tables through Kafka servers to the storage database. To use extended monitoring, you must have access to a running Kafka server. For more how Vertica integrates with Kafka, see Apache Kafka integration .

After you set up and enable extended monitoring for a monitored database, MC renders several of your database's charts and graphs by querying the MC storage database instead of directly querying the database you are monitoring.

You can enable extended monitoring for any, or all, of your monitored databases. The MC storage database provides a single repository for monitoring data from every database that uses enabled extended monitoring.

In the following example, Kafka streams system data from two monitored databases to the storage database. MC uses the storage database to render individual dashboards for each monitored database. Be aware that MC always creates a dashboard that monitors the MC storage database.



Use extended monitoring

Important

Important: To use extended monitoring, OpenText recommends installing Management Console on a host without any other Vertica databases.

When a database has extended monitoring enabled, the MC charts that use the feature display a rocket ship icon in the corner. You can use these charts to access longer-term data about your database's health or performance.

To view historical information in these charts, click the calendar icon to specify the timeframe to display. For example, if your database has been down for several hours, your charts do not display recent activity in your database. You could use the timeframe filter in the System Bottlenecks chart to see unusual resource usage occurred in your database in the hour it went down.

You can view a history of the Kafka streaming jobs loading data into the storage database. MC displays these jobs on the Load tab of your storage database's dashboard. See Viewing load history .

Set up extended monitoring

To set up extended monitoring, see Managing the storage database and Managing extended monitoring on a database .

See also

- Managing the storage database
- Managing extended monitoring on a database
- Viewing load history
- Apache Kafka integration
- Data collector utility

In this section

- Managing the storage database
- Managing extended monitoring on a database
- Managing streaming services for extended monitoring

Managing the storage database

Extended Monitoring stores your Vertica database's monitoring data in a dedicated MC storage database.

To use Extended Monitoring, you must first set up the storage database and configure it for Kafka streaming. Then, turn on Extended Monitoring for any or all monitored databases.

MC automatically configures a schema for the storage database, named dcschema, which is synced with DC tables on your monitored databases.

> **Caution**
> Do not alter dcschema after MC has configured it. Altering it could cause the storage database to lose data or supply incorrect monitoring information to MC.

MC preparation

First verify that MC is not installed on the same host as a Vertica database. When Extended Monitoring is enabled, MC sharing a host with a production database can affect performance.

You must also increase the allocation of memory for the MC application server, as described in the next section. Tune the memory allocation options based on:

- The demands of your database.
- The amount of monitoring data you plan to view in MC charts at the same time.

For example, MC requires more memory to display a week of data in a chart.

Modify memory allocation

To modify memory allocation:

1. In Management Console, select the **Configuration** tab on the MC Settings page.
2. Modify the following fields under **Application Server JVM Settings** to increase the allocation of memory for the JVM:
   - **Initial Heap Size:** For Extended Monitoring, a minimum value of 2 GB is recommended. (The default value is 1 GB.)
   - **Maximum Heap Size:** For Extended Monitoring, a minimum value of 4 GB is recommended. (The default value is 2 GB.)
3. Click **Apply** at the top right of the page. A prompt appears to restart MC.
4. Click **OK** to restart MC and save your changes.

Storage database requirements

To set up storage for Extended Monitoring, your system must meet the following prerequisites:

- An available host, or available database whose Vertica version is the same version or a higher version of the database you plan to monitor.
- Configured MC for Extended Monitoring (See Prepare MC to Use Extended Monitoring .)
- Access to a deployed Kafka server (For details on installing Kafka, see the Apache Kafka site .)

Set up the storage database

To configure the storage database for Extended Monitoring, on the MC Settings page, select the MC Storage DB Setup tab. Modify the settings in each of the three areas:

1) Kafka Broker>

Enter the host name or IP addresses and ports for one or more of your deployed Kafka servers.

2) MC external storage database

Designate the storage database. You can create a new database or use an existing database.

- **Create a new database:** To create a new single node cluster on an available host using a Community Edition license of Vertica, choose this option. Doing so does not affect your normal Vertica license usage.
- **Use an existing database known to MC:** To designate a database you have already imported to MC, choose this option. If the schema 'dcschema' exists in the database, a dialog appears. Depending on your system needs, do one of the following:
  - To keep the existing schema's data, click **Append** . For example, if you have already used this database for Extended Monitoring storage and are reimporting it, you can use this option to retain its historical data for continued use.
  - To clear the existing schema from the database and create a fresh version of dcschema configured for Extended Monitoring storage, click **Remove** .

At the **Database name** prompt:

1. Select the database you want to use from the drop-down list.
2. To use that database for Extended Monitoring, click **Prepare MC Storage database** .

Advanced Streaming Options:

To change the value of the Scheduler Frame Duration, click **Advanced Streaming Options** . Management Console displays the **Streaming Options** window, which allows you to modify the Scheduler Frame Duration default that Management Console uses for Extended Monitoring..

The **Scheduler Frame Duration** is the amount of time given to the Kafka scheduler for each individual frame to process and run the COPY statements, after which KafkaSource terminates the COPY statements. Vertica must have enough time to complete COPY tasks within this duration.

If the frame duration is too small, you would see data loss, as the scheduler does not have sufficient time to process all the data. You may see errors or messages on Management Console's Load page for microbatches that are not able to process the data.

On the contrary, if the frame duration is too large, the scheduler will have too much time to process the incoming data and after it has finished processing the data, it might wait for the frame duration to expire. In this case, you may see some latency in the data getting processed. In addition, the charts in Management Console may not show the data in real time and may show some latency.

You can approximate the average available time per COPY using the following equation:

TimePerCopy=(FrameDuration Parallelism)/Microbatches*

This equation provides only a rough estimate. There are many factors that impact the amount of time that each COPY statement needs to run.

Vertica requires at least 100 milliseconds per COPY to function.

> **Note**
> The **Advanced Scheduler options** button is enabled when the Streaming is turned off. If Kafka Streaming is enabled, the **Advanced Scheduler options** button is disabled.

3) enable extended monitoring

Click **Select database(s) for extended monitoring** .

> **Note**
> For more information, see Managing extended monitoring on a database .

Restart the storage database

If you stop the storage database while streaming is enabled, streaming to the storage database stops automatically. You must re-enable streaming on the MC Storage DB Setup tab after you restart the storage database.

If streaming to the MC storage database is disabled while Extended Monitoring on your database is on, the Kafka retention policy determines how long streaming can remain disabled without data loss. See Managing streaming services for extended monitoring .

Discontinue the storage database

1. Select the Extended Monitoring tab in MC Settings.
2. Set Extended Monitoring for all databases to **OFF** .
3. Select the MC Storage DB Setup tab in MC Settings.
4. Click **Disable Streaming** in the MC External Storage Database section to de-activate your storage database.
5. Click **Remove** in the MC External Storage Database section to remove the MC Storage Database from MC.
6. Choose whether to keep or remove the data your storage database has collected:
   - **Keep Data:** Existing data will not removed. If you re-use this database for Extended Monitoring storage, you can choose to append new collected monitoring data to this existing data.
   - **Remove Data:** MC deletes its customized storage schema from the database.

Configure storage database memory usage

On the Resource Pools tab of the storage database, you can optionally increase the memory size of SYSQUERY and KAFKA_DEFAULT_POOL. For setting resource pool parameters in MC, see Configuring resource pools with MC .

- **SYSQUERY:** Reserved for temporary storage of intermediate results of queries against system monitoring and catalog tables. Default setting is 1G. For best performance for MC, set to 2G or higher.
- **KAFKA_DEFAULT_POOL:** Reserved for all queries executed by the Kafka scheduler. Default setting is 30%, which is the recommended setting. By default, queries spill to the general pool when they exceed the 30% memory size.

Manage disk space

The storage database uses a customized schema, named dcschema. You can monitor these tables on MC, using the Table Utilization chart on the storage database's Activity tab. The Table Utilization chart lists all the tables in dcschema and their details, such as row counts and column properties. You can sort by row count to determine if certain tables use more disk space on your storage database. See Monitoring table utilization and projections with MC .

You should regularly drop partitions from dcschema if you have limited disk space for the MC storage database. MC does not automatically drop partitions from the storage database. For more information on dropping partitions, see Dropping partitions .

The table dc_execution_engine_profiles is partitioned by day. Because this table typically contains the most rows, as a best practice you should drop partitions from this table more often. The following example shows how you can specify partition key 2016-08-22 to drop a partition from dc_execution_engine_profiles.

```
=> SELECT DROP_PARTITIONS
 ('dcschema.dc_execution_engine_profiles', 2016-08-2, 2016-08-22);
```

Other than dc_execution_engine_profiles, all other tables in dcschema are partitioned by week. The next example shows you how you can drop a partition from the table dc_cpu_aggregate_by_minute, specifying the thirty-fourth week of 2016.

```
=> SELECT DROP_PARTITION
 ('dcschema.dc_cpu_aggregate_by_minute', 201634, 201634);
```

Manage client sessions

By default Vertica allows 50 client sessions and an additional five administrator sessions per node. If you reach the limit on the storage database, MC switches back to default monitoring, and does not use Extended Monitoring data from the storage database.

You can optionally configure the maximum number of client sessions that can run on a single database cluster node on your MC storage database's Settings page:

1. On the storage database dashboard, click the **Settings** page.
2. Choose the **General** tab.
3. Enter a value in the **Maximum client sessions** field. Valid values are 0–1000.

For more details about managing client connections in MC, see Configuring Management Console .

See also

- Extended monitoring
- Managing extended monitoring on a database
- Viewing load history
- Create a private key file

Managing extended monitoring on a database

When you enable extended monitoring on your Vertica database, monitoring data from your database streams through Kafka servers to the MC storage database.

You can enable streaming for any or all databases that MC monitors.

Extended monitoring prerequisites

Before you can enable extended monitoring, your system must meet these prerequisites:

- The versions of MC and Vertica must match.
- Deployed Kafka server(s)
- Configured MC for extended monitoring (See Managing the storage database)
- Deployed MC storage database (See Managing the storage database)

Enable extended monitoring

1. Select the Extended Monitoring tab on MC Settings.

   The Extended Monitoring page displays all databases monitored by MC.
2. In the Memory Limit field for the database of your choice, set the maximum amount of memory the database can use for streaming monitoring data. For more about the memory limit, see Managing streaming services for extended monitoring .
3. In the Extended Monitoring column, select **ON** to enable streaming for the database of your choice.

   The database begins streaming its monitoring data to the Kafka server.

User access

When you change user permissions for a database using extended monitoring, the user access policy on the storage database does not automatically update. On the Extended Monitoring page, in the user access column for your database, click Refresh to sync the policy.

If you rename a Vertica user, you must re-map the user in MC Settings before refreshing the user access policy.

See also

- Extended monitoring
- Managing the storage database
- Viewing load history
- Apache Kafka integration

Managing streaming services for extended monitoring

When extended monitoring is enabled, Vertica streams data from your database through Kafka servers to the storage database.

For additional parameters that optimize the performance of Kafka with Vertica, see Kafka and Vertica configuration settings .

View streaming details in MC

Click the Load tab on your database's MC dashboard to see the Data Load Activity page. On this page, the Continuous tab displays details about all continuous loading jobs for extended monitoring. You can use this page to monitor whether your extended monitoring data is streaming successfully to the MC storage database.

See Viewing load history for more about the Data Load Activity page.

> **Tip**
> **Tip:** If you do not see loading jobs for extended monitoring, verify that you have selected **Show MC data collector monitoring streams** at the top of the Continuous tab.

Prevent data loss

The Memory Limit buffer allows you to restart the Kafka server without data loss. Vertica queues the streamed data until you restart the Kafka server. When the Kafka server remains down for an extended period of time, data loss occurs when the queue of streamed data exceeds the buffer. You set the buffer size on the Extended Monitoring tab when you enable extended monitoring for a database. See Managing extended monitoring on a database .

The Kafka retention policy determines when data loss occurs during the following scenarios:

- Restarting the MC storage database (see Managing the storage database)
- Disabling streaming on the MC storage database (see Managing the storage database)
- Restart a micro-batch (see Viewing load history)

The Kafka retention policy can allow you to restart these extended monitoring components without data loss. The Kafka server retains the data while the listed components are disabled. Data loss occurs when the streamed data exceeds the Kafka retention policy's log size or retention time limits. See the Apache Kafka documentation for how to configure the retention policy.

Changing the Kafka server

Be aware that when you change Kafka servers for extended monitoring on the MC Storage DB Setup page, you must disable all extended monitoring processes and re-configure the MC storage database. For storage database setup instructions, see Managing the storage database .

See also

- Managing extended monitoring on a database
- Viewing load history
- Kafka and Vertica configuration settings
- Apache Kafka integration

Troubleshooting with MC diagnostics

The Management Console **Diagnostics** page, which you access from the Home page, helps you resolve issues within the MC process, not the database.

What you can diagnose

- View Management Console logs, which you can sort by column headings, such as type, component, or message).
- Search messages for key words or phrases and search for log entries within a specific time frame.
- Export database messages to a file.
- Reset console parameters to their original configuration.

  > **Caution**
  > Reset removes all data (monitoring and configuration information) from storage and sets MC to factory settings .

- Restart the Management Console process . When the process completes, you are directed back to the login page.

In this section

- Viewing the MC log
- Exporting the user audit log
- Restarting MC
- Resetting MC to pre-configured state

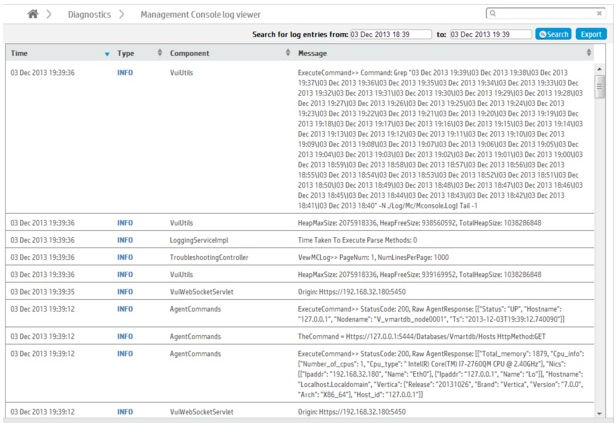
- Avoiding MC self-signed certificate expiration

Viewing the MC log

If you want to browse MC logs (not database logs), navigate to the **Diagnostics > MC Log** page.

This page provides a tabular view of the contents at /opt/vconsole/log/mc/mconsole.log , letting you more easily identify and troubleshoot issues related to MC.

You can sort log entries by clicking the column header and search within messages for key words, phrases, and log entries within a specific time frame. You can also export log messages to a file.



Exporting the user audit log

When an MC user makes changes on Management Console, whether to an MC-managed database or to the MC itself, their action generates a log entry that contains data you can export to a file.

If you perform an MC factory reset (restore MC to its pre-configured state), you automatically have the opportunity to export audit records before the reset occurs.

To manually export MC user activity

1. From the MC Home page, click **Diagnostics** and then click **Audit Log** .
2. On the Audit log viewer page, click **Export** and save the file to a location on the server.

To see what types of user operations the audit logger records, see Monitoring MC user activity using audit log .

Restarting MC

You might need to restart the MC web/application server for a number of reasons, such as after you change port assignments, use the MC interface to import a new SSL certificate, or if the MC interface or Vertica-related tasks become unresponsive.

Restarting MC requires ADMIN Role (MC) or SUPER Role (MC) privileges. For details about these roles, see Configuration privileges .

How to restart MC through the MC interface (using your browser)

1. Open a web browser and connect to MC as an administrator.
2. On MC's Home page, click **Diagnostics** .
3. Click **Restart Console** and then click OK to continue or Cancel to return to the Diagnostics page..

The MC process shuts down for a few seconds and automatically restarts. After the process completes, you are directed back to the sign-in page.

How to restart MC at the command line

If you are unable to connect to MC through a web browser for any reason, such as if the MC interface or Vertica-related tasks become unresponsive, you can run the vertica-consoled script with start, stop, or restart arguments.

Follow these steps to start, stop, or restart MC.

1. As root, open a terminal window on the server on which MC is installed.
2. Run the *vertica-consoled* script:

```
# /etc/init.d/vertica-consoled { stop | start | restart }
```

   For versions CentOS 7 and above, run:

```
# systemctl { stop | start | restart } vertica-consoled
```

Important
The systemctl function requires you to both start and stop services explicitly. If you kill or stop the vertica-consoled process without using systemctl stop , you cannot start the MC process again with the original systemctl start command. Instead, you must run systemctl stop vertica-consoled before running systemctl start vertica-consoled .

stop	Stops the MC application/web server.
start	Starts the MC application/web server. **Caution:** Use start only if you are certain MC is not already running. As a best practice, stop MC before you issue the start command.
restart	Restarts the MC application/web server. This process will report that the stop didn't work if MC is not already running.

How to restart MC on an AMI

You can use the following steps to restart an MC AMI instance.

1. SSH into the MC host as user dbadmin:

```
$ ssh -i example.pem dbadmin@52.xx.xx.xx
```

1. Run the *vertica-consoled* script using sudo:

```
# sudo /etc/init.d/vertica-consoled { stop | start | restart }
```

Starting over

If you need to return MC to its original state (a "factory reset"), see Resetting MC to pre-configured state .

Resetting MC to pre-configured state

If you decide to reset MC to its original, preconfigured state, you can do so on the **Diagnostics** page by clicking **Factory Reset** .

> **Tip**
> Consider trying one of the options described in Restarting MC first.

A factory reset removes all metadata (about a week's worth of database monitoring/configuration information and MC users) from storage and forces you to reconfigure MC again, as described in Configuring MC .

After you click Factory Reset, you have the chance to export audit records to a file by clicking Yes. If you click No (do not export audit records), the process begins. There is no undo.

Keep the following in mind concerning user accounts and the MC.

- When you first configure MC, during the configuration process you create an MC superuser (a Linux account). Issuing a Factory Reset on the MC does not create a new MC superuser, nor does it delete the existing MC superuser. When initializing after a Factory Reset, you must logon using the original MC superuser account.
- Note that, once MC is configured, you can add users that are specific to MC. Users created through the MC interface are MC specific. When you subsequently change a password through the MC, you only change the password for the specific MC user. Passwords external to MC (i.e., system Linux users and Vertica database passwords) remain unchanged.
  For information on MC users, refer to the sections, Creating an MC User and MC configuration privileges .

Avoiding MC self-signed certificate expiration

When you connect to MC through a client browser, Vertica assigns each HTTPS request a self-signed certificate, which includes a timestamp. To increase security and protect against password replay attacks, the timestamp is valid for several seconds only, after which it expires.

To avoid being blocked out of MC, synchronize time on the hosts in your Vertica cluster, and on the MC host if it resides on a dedicated server. To recover from loss or lack of synchronization, resync system time and the Network Time Protocol.

Uninstalling Management Console

The uninstall command shuts down Management Console (MC) and removes most of the files that MC installation script installed.

1. Log in to the target server as root.
2. Stop Management Console:

   ```
   # /etc/init.d/vertica-consoled stop
   ```

   For versions of Red Hat 7/CentOS 7 and above, use:

   ```
   # systemctl stop vertica-consoled
   ```

3. Look for previously-installed versions of MC and note the version:

   RPM

   ```
   # rpm -qa | grep vertica
   ```

   DEB

   ```
   # dpkg -l | grep vertica
   ```

4. Remove the package:

   RPM

   ```
   # rpm -e vertica-console
   ```

   DEB

   ```
   # dpkg -r vertica-console
   ```

5. Optionally, delete the MC directory and all subdirectories:

   ```
   # rm -rf /opt/vconsole
   ```

Administrator's guide

Welcome to the Vertica Administator's Guide. This document describes how to set up and maintain a Vertica Analytics Platform database.

Prerequisites

This document makes the following assumptions:

- You are familiar with the concepts discussed in Architecture .
- Performed the following procedures as described in Setup :
  - Constructed a hardware platform.
  - Installed Linux.
  - Installed Vertica and configured a cluster of hosts.

In this section

- Administration overview
- Managing licenses
- Configuring the database
- Database users and privileges
- Using the administration tools
- Operating the database
- Working with native tables
- Managing client connections
- Projections
- Partitioning tables
- Constraints
- Managing queries
- Transactions
- Vertica database locks
- Using text search
- Managing storage locations
- Analyzing workloads
- Managing the database
- Monitoring Vertica
- Backing up and restoring the database
- Failure recovery

- Collecting database statistics
- Using diagnostic tools
- Profiling database performance
- About locale
- Appendix: creating native binary format files

Administration overview

This document describes the functions performed by a Vertica database administrator (DBA). Perform these tasks using only the dedicated database administrator account that was created when you installed Vertica. The examples in this documentation set assume that the administrative account name is dbadmin.

- To perform certain cluster configuration and administration tasks, the DBA (users of the administrative account) must be able to supply the root password for those hosts. If this requirement conflicts with your organization's security policies, these functions must be performed by your IT staff.
- If you perform administrative functions using a different account from the account provided during installation, Vertica encounters file ownership problems.
- If you share the administrative account password, make sure that only one user runs the Administration tools at any time. Otherwise, automatic configuration propagation does not work correctly.
- The Administration Tools require that the calling user's shell be /bin/bash . Other shells give unexpected results and are not supported.

Managing licenses

You must license Vertica in order to use it. Vertica supplies your license in the form of one or more license files, which encode the terms of your license.

To prevent introducing special characters that invalidate the license, do not open the license files in an editor. Opening the file in this way can introduce special characters, such as line endings and file terminators, that may not be visible within the editor. Whether visible or not, these characters invalidate the license.

Applying license files

Be careful not to change the license key file in any way when copying the file between Windows and Linux, or to any other location. To help prevent applications from trying to alter the file, enclose the license file in an archive file (such as a .zip or .tar file). You should keep a back up of your license key file. OpenText recommends that you keep the backup in /opt/vertica.

After copying the license file from one location to another, check that the copied file size is identical to that of the one you received from Vertica.

In this section

- Obtaining a license key file
- Understanding Vertica licenses
- Installing or upgrading a license key
- Viewing your license status
- Viewing license compliance for Hadoop file formats
- Moving a cloud installation from by the hour (BTH) to bring your own license (BYOL)
- Auditing database size
- Monitoring database size for license compliance
- Managing license warnings and limits
- Exporting license audit results to CSV

Obtaining a license key file

Follow these steps to obtain a license key file:

1. Log in to the Software Entitlement Key site using your passport login information. If you do not have a passport login, create one.
2. On the Request Access page, enter your order number and select a role.
3. Enter your request access reasoning.
4. Click **Submit** .
5. After your request is approved, you will receive a confirmation email. On the site, click the **Entitlements** tab to see your Vertica software.
6. Under the Action tab, click **Activate** . You may select more than one product.
7. The License Activation page opens. Enter your Target Name.
8. Select you Vertica version and the quantity you want to activate.
9. Click **Next** .

10. Confirm your activation details and click Submit.
11. The Activation Results page displays. Follow the instructions in New Vertica license installations or Vertica license changes to complete your installation or upgrade.

Your Vertica Community Edition download package includes the Community Edition license, which allows three nodes and 1TB of data. The Vertica Community Edition license does not expire.

Understanding Vertica licenses

Vertica has flexible licensing terms. It can be licensed on the following bases:

- Term-based (valid until a specific date).
- Size-based (valid to store up to a specified amount of raw data).
- Both term- and size-based.
- Unlimited duration and data storage.
- Node-based with an unlimited number of CPUs and users (one node is a server acting as a single computer system, whether physical or virtual).
- A pay-as-you-go model where you pay for only the number of hours you use. This license is available on your cloud provider's marketplace.

Your license key has your licensing bases encoded into it. If you are unsure of your current license, you can view your license information from within Vertica .

> **Note**
> Vertica does not support license downgrades.

Community edition license

Vertica Community Edition (CE) is free and allows customers to cerate databases with the following limits:

- up to 3 of nodes
- up to 1 terabyte of data

Community Edition licenses cannot be installed co-located in a Hadoop infrastructure and used to query data stored in Hadoop formats.

As part of the CE license, you agree to the collection of some anonymous, non-identifying usage data. This data lets Vertica understand how customers use the product, and helps guide the development of new features. None of your personal data is collected. For details on what is collected, see the Community Edition End User License Agreement .

Vertica for SQL on Apache Hadoop license

Vertica for SQL on Apache Hadoop is a separate product with its own license. This documentation covers both products. Consult your license agreement for details about available features and limitations.

Installing or upgrading a license key

The steps you follow to apply your Vertica license key vary, depending on the type of license you are applying and whether you are upgrading your license.

In this section

- New Vertica license installations
- Vertica license changes

New Vertica license installations

Follow these steps to install a new Vertica license:

1. Copy the license key file you generated from the Software Entitlement Key site to your Administration host .
2. Ensure the license key's file permissions are set to 400 (read permissions).
3. Install Vertica as described in the Installing Vertica if you have not already done so. The interface prompts you for the license key file.
4. To install Community Edition, leave the default path blank and click **OK** . To apply your evaluation or Premium Edition license, enter the absolute path of the license key file you downloaded to your Administration Host and press **OK** . The first time you log in as the Database Superuser and run the Administration tools , the interface prompts you to accept the End-User License Agreement (EULA).

> **Note**
> If you installed Management Console , the MC administrator can point to the location of the license key during Management Console

configuration.

5. Choose **View EULA** .
6. Exit the EULA and choose **Accept EULA** to officially accept the EULA and continue installing the license, or choose **Reject EULA** to reject the EULA and return to the Advanced Menu.

Vertica license changes

If your license is expiring or you want your database to grow beyond your licensed data size, you must renew or upgrade your license. After you obtain your renewal or upgraded license key file, you can install it using Administration Tools or Management Console.

Upgrading does not require a new license unless you are increasing the capacity of your database. You can add-on capacity to your database using the Software Entitlement Key. You do not need uninstall and reinstall the license to add-on capacity.

Uploading or upgrading a license key using administration tools

1. Copy the license key file you generated from the Software Entitlement Key site to your Administration host .
2. Ensure the license key's file permissions are set to 400 (read permissions).
3. Start your database, if it is not already running.
4. In the Administration Tools, select Advanced > Upgrade License Key and click OK.
5. Enter the absolute path to your new license key file and click OK. The interface prompts you to accept the End-User License Agreement (EULA).
6. Choose View EULA.
7. Exit the EULA and choose Accept EULA to officially accept the EULA and continue installing the license, or choose Reject EULA to reject the EULA and return to the Advanced Menu.

Uploading or upgrading a license key using Management Console

1. From your database's Overview page in Management Console, click the License tab. The License page displays. You can view your installed licenses on this page.
2. Click Install New License at the top of the License page.
3. Browse to the location of the license key from your local computer and upload the file.
4. Click Apply at the top of the page. Management Console prompts you to accept the End-User License Agreement (EULA).
5. Select the check box to officially accept the EULA and continue installing the license, or click Cancel to exit.

> **Note**
> As soon as you renew or upgrade your license key from either your Administration host or Management Console, Vertica applies the license update. No further warnings appear.

Adding capacity

If you are adding capacity to your database, you do not need to uninstall and reinstall the license. Instead, you can install multiple licenses to increase the size of your database. This additive capacity only works for licenses with the same format, such as adding a Premium license capacity to an existing Premium license type. When you add capacity, the size of license will be the total of both licenses; the previous license is not overwritten. You cannot add capacity using two different license formats, such as adding Hadoop license capacity to an existing Premium license.

You can run the AUDIT() function to verify the license capacity was added on. The reflection of add-on capacity to your license will run during the automatic run of the audit function. If you want to see the immediate result of the add-on capacity, run the AUDIT() function to refresh.

> **Note**
> If you have an expired license, you must drop the expired license before you can continue to use Vertica. For more information, see DROP_LICENSE .

Viewing your license status

You can use several functions to display your license terms and current status.

Examining your license key

Use the DISPLAY_LICENSE SQL function to display the license information. This function displays the dates for which your license is valid (or Perpetual if your license does not expire) and any raw data allowance. For example:

```
=> SELECT DISPLAY_LICENSE();
              DISPLAY_LICENSE
-------------------------------------------------------
 Vertica Systems, Inc.
2007-08-03
Perpetual
500GB

(1 row)
```

You can also query the LICENSES system table to view information about your installed licenses. This table displays your license types, the dates for which your licenses are valid, and the size and node limits your licenses impose.

Alternatively, use the LICENSES table in Management Console. On your database Overview page, click the License tab to view information about your installed licenses.

Viewing your license compliance

If your license includes a raw data size allowance, Vertica periodically audits your database's size to ensure it remains compliant with the license agreement. If your license has a term limit, Vertica also periodically checks to see if the license has expired. You can see the result of the latest audits using the GET_COMPLIANCE_STATUS function.

```
=> select GET_COMPLIANCE_STATUS();
                    GET_COMPLIANCE_STATUS
---------------------------------------------------------------------------------
 Raw Data Size: 2.00GB +/- 0.003GB
 License Size : 4.000GB
 Utilization  : 50%
 Audit Time   : 2011-03-09 09:54:09.538704+00
 Compliance Status : The database is in compliance with respect to raw data size.
 License End Date: 04/06/2011
 Days Remaining: 28.59
(1 row)
```

To see how your ORC/Parquet data is affecting your license compliance, see Viewing license compliance for Hadoop file formats .

Viewing your license status through MC

Information about license usage is on the Settings page. See Monitoring database size for license compliance .

Viewing license compliance for Hadoop file formats

You can use the EXTERNAL_TABLE_DETAILS system table to gather information about all of your tables based on Hadoop file formats. This information can help you understand how much of your license's data allowance is used by ORC and Parquet-based data.

Vertica computes the values in this table at query time, so to avoid performance problems, restrict your queries to filter by table_schema, table_name, or source_format. These three columns are the only columns you can use in a predicate, but you may use all of the usual predicate operators.

```
=> SELECT * FROM EXTERNAL_TABLE_DETAILS
   WHERE source_format = 'PARQUET' OR source_format = 'ORC';
-[ RECORD 1 ]---------+----------------------------------------------------------------------------------------------------------------------------------------------------------------------
schema_oid            | 45035996273704978
table_schema          | public
table_oid             | 45035996273760390
table_name            | ORC_demo
source_format         | ORC
total_file_count      | 5
total_file_size_bytes | 789
source_statement      | COPY FROM 'ORC_demo/*' ORC
file_access_error     |
-[ RECORD 2 ]---------+----------------------------------------------------------------------------------------------------------------------------------------------------------------------
schema_oid            | 45035196277204374
table_schema          | public
table_oid             | 45035996274460352
table_name            | Parquet_demo
source_format         | PARQUET
total_file_count      | 3
total_file_size_bytes | 498
source_statement      | COPY FROM 'Parquet_demo/*' PARQUET
file_access_error     |
```

When computing the size of an external table, Vertica counts all data found in the location specified by the COPY FROM clause. If you have a directory that contains ORC and delimited files, for example, and you define your external table with "COPY FROM *" instead of "COPY FROM *.orc", this table includes the size of the delimited files. (You would probably also encounter errors when querying that external table.) When you query this table Vertica does not validate your table definition; it just uses the path to find files to report.

You can also use the AUDIT function to find the size of a specific table or schema. When using the AUDIT function on ORC or PARQUET external tables, the error tolerance and confidence level parameters are ignored. Instead, the AUDIT always returns the size of the ORC or Parquet files on disk.

```
=> select AUDIT('customers_orc');
  AUDIT
-----------
 619080883
(1 row)
```

Moving a cloud installation from by the hour (BTH) to bring your own license (BYOL)

Vertica offers two licensing options for some of the entries in the Amazon Web Services Marketplace and Google Cloud Marketplace:

- **Bring Your Own License (BYOL)** : a long-term license that you obtain through an online licensing portal . These deployments also work with a free Community Edition license. Vertica uses a community license automatically if you do not install a license that you purchased. (For more about Vertica licenses, see Managing licenses and Understanding Vertica licenses .)
- Vertica **by the Hour (BTH)** : a pay-as-you-go environment where you are charged an hourly fee for both the use of Vertica and the cost of the instances it runs on. The Vertica by the hour deployment offers an alternative to purchasing a term license. If you want to crunch large volumes of data within a short period of time, this option might work better for you. The BTH license is automatically applied to all clusters you create using a BTH MC instance.

If you start out with an hourly license, you can later decide to use a long-term license for your database. The support for an hourly versus a long-term license is built into the instances running your database. To move your database from an hourly license to a long-term license, you must create a new database cluster with a new set of instances.

To move from an hourly to a long-term license, follow these steps:

1. Purchase a BYOL license. Follow the process described in Obtaining a license key file .
2. Apply the new license to your database.
3. Shut down your database.
4. Create a new database cluster using a BYOL marketplace entry.
5. Revive your database onto the new cluster.

The exact steps you must take depend on your database mode and your preferred tool for managing your database:

- **Eon Mode databases:** Use either the command line or Management Console .
- **Enterprise Mode databases:** Use the command line to switch licenses.

Moving an Eon Mode database from BTH to BYOL using the command line

Follow these steps to move an Eon Mode database from an hourly to a long-term license.

Obtain a long-term BYOL license from the online licensing portal, described in Obtaining a license key file .Upload the license file to a node in your database. Note the absolute path in the node's filesystem, as you will need this later when installing the license.Connect to the node you uploaded the license file to in the previous step. Connect to your database using vsql and view the licenses table:

```
=> SELECT * FROM licenses;
```

Note the name of the hourly license listed in the NAME column, so you can check if it is still present later.

Install the license in the database using the INSTALL_LICENSE function with the absolute path to the license file you uploaded in step 2:

```
=> SELECT install_license('absolute path to BYOL license');
```

View the licenses table again:

```
=> SELECT * FROM licenses;
```

If only the new BYOL license appears in the table, skip to step 8. If the hourly license whose name you noted in step 4 is still in the table, copy the name and proceed to step 7.

Call the DROP_LICENSE function to drop the hourly license:

```
=> SELECT drop_license('hourly license name');
```

1. You will need the path for your cluster's communal storage in a later step. If you do not already know the path, you can find this information by executing this query:
   ```
   => SELECT location_path FROM V_CATALOG.STORAGE_LOCATIONS
     WHERE sharing_type = 'COMMUNAL';
   ```
2. Synchronize your database's metadata. See Synchronizing metadata .
3. Shut down the database by calling the SHUTDOWN function:
   ```
   => SELECT SHUTDOWN();
   ```
4. You now need to create a new BYOL cluster onto which you will revive your database. Deploy a new cluster including a new MC instance using a BYOL entry in the marketplace of your chosen cloud platform. See:
   - Deploy MC and AWS resources with a CloudFormation template
   - Deploy an MC instance in GCP for Eon Mode

   Important

   Your new BYOL cluster must have the same number of primary nodes as your existing hourly license cluster.
5. Revive your database onto the new cluster. For instructions, see Reviving an Eon Mode database cluster . Because you created the new cluster using a BYOL entry in the marketplace, the database uses the BYOL you applied earlier.
6. After reviving the database on your new BYOL cluster, terminate the instances for your hourly license cluster and MC. For instructions, see your cloud provider's documentation.

Moving an Eon Mode database from BTH to BYOL using the MC

Follow this procedure to move to BYOL and revive your database using MC:

1. Purchase a long-term BYOL license from the online licensing portal, following the steps detailed in Obtaining a license key file . Save the file to a location on your computer.
2. You now need to install the new license on your database. Log into MC and click your database in the Recent Databases list.
3. At the bottom of your database's Overview page, click the **License** tab.
4. Under the Installed Licenses list, note the name of the BTH license in the License Name column. You will need this later to check whether it is still present after installing the new long-term license.
5. In the ribbon at the top of the License History page, click the **Install New License** button. The Settings: License page opens.
6. Click the **Browse** button next to the **Upload a new license box** .
7. Locate the license file you obtained in step 1, and click **Open** .
8. Click the **Apply** button on the top right of the page.
9. Select the checkbox to agree to the EULA terms and click **OK** .

10. After Vertica installs the license, click the **Close** button.
11. Click the **License** tab at the bottom of the page.
12. If only the new long-term license appears in the **Installed Licenses** list, skip to Step 16. If the by-the-hour license also appears in the list, copy down its name from the **License Name** column.
13. You must drop the by-the-hour license before you can proceed. At the bottom of the page, click the **Query Execution** tab.
14. In the query editor, enter the following statement:
```
SELECT DROP_LICENSE('hourly license name');
```
15. Click **Execute Query** . The query should complete indicating that the license has been dropped.
16. You will need the path for your cluster's communal storage in a later step. If you do not already know the path, you can find this information by executing this query in the Query Execution tab:
```
SELECT location_path FROM V_CATALOG.STORAGE_LOCATIONS
  WHERE sharing_type = 'COMMUNAL';
```
17. Synchronize your database's metadata. See Synchronizing metadata .
18. You must now stop your by-the-hour database cluster. At the bottom of the page, click the **Manage** tab.
19. In the banner at the top of the page, click **Stop Database** and then click **OK** to confirm.
20. From the Amazon Web Services Marketplace or the Google Cloud Marketplace , deploy a new Vertica Management Console using a BYOL entry. Do not deploy a full cluster. You just need an MC deployment.
21. Log into your new MC instance and revive the database. See Reviving an Eon Mode database on AWS in MC for detailed instructions.
22. After reviving the database on your new environment, terminate the instances for your hourly license environment. To do so, on the AWS CloudFormation Stacks page , select the hourly environment's stack (its collection of AWS resources) and click **Actions** > **Delete Stack** .

Moving an Enterprise Mode database from hourly to BYOL using backup and restore

> **Note**
> Currently, AWS is the only platform supported for Enterprise Mode databases using hourly licenses.

In an Enterprise Mode database, follow this procedure to move to BYOL, and then back up and restore your database:

Obtain a long-term BYOL license from the online licensing portal, described in Obtaining a license key file .Upload the license file to a node in your database. Note the absolute path in the node's filesystem, as you will need this later when installing the license.Connect to the node you uploaded the license file to in the previous step. Connect to your database using vsql and view the licenses table:

```
=> SELECT * FROM licenses;
```

Note the name of the hourly license listed in the NAME column, so you can check if it is still present later.

Install the license in the database using the INSTALL_LICENSE function with the absolute path to the license file you uploaded in step 2:

```
=> SELECT install_license('absolute path to BYOL license');
```

View the licenses table again:

```
=> SELECT * FROM licenses;
```

If only the new BYOL license appears in the table, skip to step 8. If the hourly license whose name you noted in step 4 is still in the table, copy the name and proceed to step 7.

Call the DROP_LICENSE function to drop the hourly license:

```
=> SELECT drop_license('hourly license name');
```

1. Back up the database. See Backing up and restoring the database .
2. Deploy a new cluster for your database using one of the BYOL entries in the Amazon Web Services Marketplace .
3. Restore the database from the backup you created earlier. See Backing up and restoring the database . When you restore the database, it will use the BYOL you loaded earlier.
4. After restoring the database on your new environment, terminate the instances for your hourly license environment. To do so, on the AWS CloudFormation Stacks page , select the hourly environment's stack (its collection of AWS resources) and click **Actions** > **Delete Stack** .

After completing one of these procedures, see Viewing your license status to confirm the license drop and install were successful.

Auditing database size

You can use your Vertica software until columnar data reaches the maximum raw data size that your license agreement allows. Vertica periodically runs an audit of the columnar data size to verify that your database complies with this agreement. You can also run your own audits of database size with two functions:

- AUDIT : Estimates the raw data size of a database, schema, or table.
- AUDIT_FLEX : Estimates the size of one or more flexible tables in a database, schema, or projection.

The following two examples audit the database and one schema:

```
=> SELECT AUDIT('', 'database');
  AUDIT
----------
 76376696
(1 row)
```

```
=> SELECT AUDIT('online_sales', 'schema');
  AUDIT
----------
 35716504
(1 row)
```

Raw data size

AUDIT and AUDIT_FLEX use statistical sampling to estimate the raw data size of data stored in tables—that is, the uncompressed data that the database stores. For most data types, Vertica evaluates the raw data size as if the data were exported from the database in text format, rather than as compressed data. For details, see Evaluating Data Type Footprint .

By using statistical sampling, the audit minimizes its impact on database performance. The tradeoff between accuracy and performance impact is a small margin of error. Reports on your database size include the margin of error, so you can assess the accuracy of the estimate.

Data in ORC and Parquet-based external tables are also audited whether they are stored locally in the Vertica cluster's file system or remotely in S3 or on a Hadoop cluster. AUDIT always uses the file size of the underlying data files as the amount of data in the table. For example, suppose you have an external table based on 1GB of ORC files stored in HDFS. Then an audit of the table reports it as being 1GB in size.

> **Note**
> The Vertica audit does not verify that these files contain actual ORC or Parquet data. It just checks the size of the files that correspond to the external table definition.

Unaudited data

Table data that appears in multiple projections is counted only once. An audit also excludes the following data:

- Temporary table data.
- Data in SET USING columns.
- Non-columnar data accessible through external table definitions. Data in columnar formats such as ORC and Parquet count against your totals.
- Data that was deleted but not yet purged .
- Data stored in system and work tables such as monitoring tables, Data collector tables, and Database Designer tables.
- Delimiter characters.

Evaluating data type footprint

Vertica evaluates the footprint of different data types as follows:

- Strings and binary types—CHAR, VARCHAR, BINARY, VARBINARY—are counted as their actual size in bytes using UTF-8 encoding.
- Numeric data types are evaluated as if they were printed. Each digit counts as a byte, as does any decimal point, sign, or scientific notation. For example, -123.456 counts as eight bytes—six digits plus the decimal point and minus sign.
- Date/time data types are evaluated as if they were converted to text, including hyphens, spaces, and colons. For example, vsql prints a timestamp value of 2011-07-04 12:00:00 as 19 characters, or 19 bytes.
- Complex types are evaluated as the sum of the sizes of their component parts. An array is counted as the total size of all elements, and a ROW is counted as the total size of all fields.

Controlling audit accuracy

AUDIT can specify the level of an audit's error tolerance and confidence, by default set to 5 and 99 percent, respectively. For example, you can obtain a high level of audit accuracy by setting error tolerance and confidence level to 0 and 100 percent, respectively. Unlike estimating raw data size with statistical sampling, Vertica dumps all audited data to a raw format to calculate its size.

Caution
Vertica discourages database-wide audits at this level. Doing so can have a significant adverse impact on database performance.

The following example audits the database with 25% error tolerance:

```
=> SELECT AUDIT('', 25);
  AUDIT
----------
 75797126
(1 row)
```

The following example audits the database with 25% level of tolerance and 90% confidence level:

```
=> SELECT AUDIT('',25,90);
  AUDIT
----------
 76402672
(1 row)
```

Note
These accuracy settings have no effect on audits of external tables based on ORC or Parquet files. Audits of external tables based on these formats always use the file size of ORC or Parquet files.

Monitoring database size for license compliance

Your Vertica license can include a data storage allowance. The allowance can consist of data in columnar tables, flex tables, or both types of data. The AUDIT() function estimates the columnar table data size and any flex table materialized columns. The AUDIT_FLEX() function estimates the amount of __raw__ column data in flex or columnar tables. In regards to license data limits, data in __raw__ columns is calculated at 1/10th the size of structured data. Monitoring data sizes for columnar and flex tables lets you plan either to schedule deleting old data to keep your database in compliance with your license, or to consider a license upgrade for additional data storage.

Note
An audit of columnar data includes flex table real and materialized columns, but not __raw__ column data.

Viewing your license compliance status

Vertica periodically runs an audit of the columnar data size to verify that your database is compliant with your license terms. You can view the results of the most recent audit by calling the GET_COMPLIANCE_STATUS function.

```
=> select GET_COMPLIANCE_STATUS();
                    GET_COMPLIANCE_STATUS
---------------------------------------------------------------------------
 Raw Data Size: 2.00GB +/- 0.003GB
 License Size : 4.000GB
 Utilization  : 50%
 Audit Time   : 2011-03-09 09:54:09.538704+00
 Compliance Status : The database is in compliance with respect to raw data size.
 License End Date: 04/06/2011
 Days Remaining: 28.59
(1 row)
```

Periodically running GET_COMPLIANCE_STATUS to monitor your database's license status is usually enough to ensure that your database remains compliant with your license. If your database begins to near its columnar data allowance, you can use the other auditing functions described below to determine where your database is growing and how recent deletes affect the database size.

Manually auditing columnar data usage

You can manually check license compliance for all columnar data in your database using the AUDIT_LICENSE_SIZE function. This function performs the same audit that Vertica periodically performs automatically. The AUDIT_LICENSE_SIZE check runs in the background, so the function returns immediately. You can then query the results using GET_COMPLIANCE_STATUS .

> **Note**
> When you audit columnar data, the results include any flex table real and materialized columns, but not data in the __raw__ column. Materialized columns are virtual columns that you have promoted to real columns. Columns that you define when creating a flex table, or which you add with ALTER TABLE...ADD COLUMN statements are real columns. All __raw__ columns are real columns. However, since they consist of unstructured or semi-structured data, they are audited separately.

An alternative to AUDIT_LICENSE_SIZE is to use the AUDIT function to audit the size of the columnar tables in your entire database by passing an empty string to the function. This function operates synchronously, returning when it has estimated the size of the database.

```
=> SELECT AUDIT('');
  AUDIT
----------
 76376696
(1 row)
```

The size of the database is reported in bytes. The AUDIT function also allows you to control the accuracy of the estimated database size using additional parameters. See the entry for the AUDIT function for full details. Vertica does not count the AUDIT function results as an official audit. It takes no license compliance actions based on the results.

> **Note**
> The results of the AUDIT function do not include flex table data in __raw__ columns. Use the AUDIT_FLEX function to monitor data usage flex tables.

Manually auditing __raw__ column data

You can use the AUDIT_FLEX function to manually audit data usage for flex or columnar tables with a __raw__ column. The function calculates the encoded, compressed data stored in ROS containers for any __raw__ columns. Materialized columns in flex tables are calculated by the AUDIT function. The AUDIT_FLEX results do not include data in the __raw__ columns of temporary flex tables.

Targeted auditing

If audits determine that the columnar table estimates are unexpectedly large, consider schemas, tables, or partitions that are using the most storage. You can use the AUDIT function to perform targeted audits of schemas, tables, or partitions by supplying the name of the entity whose size you want to find. For example, to find the size of the online_sales schema in the VMart example database, run the following command:

```
=> SELECT AUDIT('online_sales');
  AUDIT
----------
 35716504
(1 row)
```

You can also change the granularity of an audit to report the size of each object in a larger entity (for example, each table in a schema) by using the granularity argument of the AUDIT function. See the AUDIT function.

Using Management Console to monitor license compliance

You can also get information about data storage of columnar data (for columnar tables and for materialized columns in flex tables) through the Management Console. This information is available in the database Overview page, which displays a grid view of the database's overall health.

- The needle in the license meter adjusts to reflect the amount used in megabytes.
- The grace period represents the term portion of the license.
- The Audit button returns the same information as the AUDIT() function in a graphical representation.
- The Details link within the License grid (next to the Audit button) provides historical information about license usage. This page also shows a progress meter of percent used toward your license limit.

Managing license warnings and limits

Term license warnings and expiration

The term portion of a Vertica license is easy to manage—you are licensed to use Vertica until a specific date. If the term of your license expires, Vertica alerts you with messages appearing in the Administration tools and vsql . For example:

```
=> CREATE TABLE T (A INT);
NOTICE 8723: Vertica license 432d8e57-5a13-4266-a60d-759275416eb2 is in its grace period; grace period expires in 28 days
HINT: Renew at https://softwaresupport.softwaregrp.com/
CREATE TABLE
```

Contact Vertica at https://softwaresupport.softwaregrp.com/ as soon as possible to renew your license, and then install the new license . After the grace period expires, Vertica stops processing DML queries and allows DDL queries with a warning message. If a license expires and one or more valid alternative licenses are installed, Vertica uses the alternative licenses.

Data size license warnings and remedies

If your Vertica columnar license includes a raw data size allowance, Vertica periodically audits the size of your database to ensure it remains compliant with the license agreement. For details of this audit, see Auditing database size . You should also monitor your database size to know when it will approach licensed usage. Monitoring the database size helps you plan to either upgrade your license to allow for continued database growth or delete data from the database so you remain compliant with your license. See Monitoring database size for license compliance for details.

If your database's size approaches your licensed usage allowance (above 75% of license limits), you will see warnings in the Administration tools , vsql , and Management Console. You have two options to eliminate these warnings:

- Upgrade your license to a larger data size allowance.
- Delete data from your database to remain under your licensed raw data size allowance. The warnings disappear after Vertica's next audit of the database size shows that it is no longer close to or over the licensed amount. You can also manually run a database audit (see Monitoring database size for license compliance for details).

If your database continues to grow after you receive warnings that its size is approaching your licensed size allowance, Vertica displays additional warnings in more parts of the system after a grace period passes. Use the GET_COMPLIANCE_STATUS function to check the status of your license.

If your Vertica premium edition database size exceeds your licensed limits

If your Premium Edition database size exceeds your licensed data allowance, all successful queries from ODBC and JDBC clients return with a status of SUCCESS_WITH_INFO instead of the usual SUCCESS. The message sent with the results contains a warning about the database size. Your ODBC and JDBC clients should be prepared to handle these messages instead of assuming that successful requests always return SUCCESS.

> **Note**
> These warnings for Premium Edition are in addition to any warnings you see in Administration Tools, vsql, and Management Console.

If your Vertica community edition database size exceeds 1 terabyte

If your Community Edition database size exceeds the limit of 1 terabyte, Vertica stops processing DML queries and allows DDL queries with a warning message.

To bring your database under compliance, you can choose to:

- Drop database tables. You can also consider truncating a table or dropping a partition. See TRUNCATE TABLE or DROP_PARTITIONS .
- Upgrade to Vertica Premium Edition (or an evaluation license).

Exporting license audit results to CSV

You can use admintools to audit a database for license compliance and export the results in CSV format, as follows:

```
admintools -t license_audit [--password=password] --database=database] [--file=csv-file] [--quiet]
```

where:

- *database* must be a running database. If the database is password protected, you must also supply the password.
- --file *csv-file* directs output to the specified file. If *csv-file* already exists, the tool returns an error message. If this option is unspecified, output is directed to stdout .
- --quiet specifies that the tool should run in quiet mode; if unspecified, status messages are sent to stdout .

Running the license_audit tool is equivalent to invoking the following SQL statements:

```
select audit('');
select audit_flex('');
select * from dc_features_used;
select * from v_catalog.license_audits;
select * from v_catalog.user_audits;
```

Audit results include the following information:

- Log of used Vertica features
- Estimated database size
- Raw data size allowed by your Vertica license
- Percentage of licensed allowance that the database currently uses
- Audit timestamps

The following truncated example shows the raw CSV output that license_audit generates:

```
FEATURES_USED
features_used,feature,date,sum
features_used,metafunction::get_compliance_status,2014-08-04,1
features_used,metafunction::bootstrap_license,2014-08-04,1
...

LICENSE_AUDITS
license_audits,database_size_bytes,license_size_bytes,usage_percent,audit_start_timestamp,audit_end_timestamp,confidence_level_percent,error_tolerance
percent,used_sampling,confidence_interval_lower_bound_bytes,confidence_interval_upper_bound_bytes,sample_count,cell_count,license_name
license_audits,808117909,536870912000,0.00150523690320551,2014-08-04 23:59:00.024874-04,2014-08-04 23:59:00.578419-
04,99,5,t,785472097,830763721,10000,174754646,vertica
...

USER_AUDITS
user_audits,size_bytes,user_id,user_name,object_id,object_type,object_schema,object_name,audit_start_timestamp,audit_end_timestamp,confidence_level_
ercent,error_tolerance_percent,used_sampling,confidence_interval_lower_bound_bytes,confidence_interval_upper_bound_bytes,sample_count,cell_count
user_audits,812489249,45035996273704962,dbadmin,45035996273704974,DATABASE,,VMart,2014-10-14 11:50:13.230669-04,2014-10-14
11:50:14.069057-04,99,5,t,789022736,835955762,10000,174755178

AUDIT_SIZE_BYTES
audit_size_bytes,now,audit
audit_size_bytes,2014-10-14 11:52:14.015231-04,810584417

FLEX_SIZE_BYTES
flex_size_bytes,now,audit_flex
flex_size_bytes,2014-10-14 11:52:15.117036-04,11850
```

Configuring the database

Before reading the topics in this section, you should be familiar with the material in Getting started and are familiar with creating and configuring a fully-functioning example database.

See also

- Security and authentication
- Implement locales for international data sets

In this section

- Configuration procedure
- Configuration parameter management
- Designing a logical schema
- Creating a database design

Configuration procedure

This section describes the tasks required to set up a Vertica database. It assumes that you have a valid license key file, installed the Vertica rpm package, and ran the installation script as described.

You complete the configuration procedure using:

- Administration tools
  If you are unfamiliar with Dialog-based user interfaces, read Using the administration tools interface before you begin. See also the Administration tools reference for details.
- vsql interactive interface
- Database Designer, described in Creating a database design

> **Note**
> You can also perform certain tasks using Management Console . Those tasks point to the appropriate topic.

Continuing configuring

Follow the configuration procedure sequentially as this section describes.

Vertica strongly recommends that you first experiment with creating and configuring a database .

You can use this generic configuration procedure several times during the development process, modifying it to fit your changing goals. You can omit steps such as preparing actual data files and sample queries, and run the Database Designer without optimizing for queries. For example, you can create, load, and query a database several times for development and testing purposes, then one final time to create and load the production database.

In this section

- Prepare disk storage locations
- Disk space requirements for Vertica
- Disk space requirements for Management Console
- Prepare the logical schema script
- Prepare data files
- Prepare load scripts
- Create an optional sample query script
- Create an empty database
- Create the logical schema
- Perform a partial data load
- Test the database
- Optimize query performance
- Complete the data load
- Test the optimized database
- Implement locales for international data sets
- Using time zones with Vertica
- Change transaction isolation levels

Prepare disk storage locations

You must create and specify directories in which to store your catalog and data files (physical schema). You can specify these locations when you install or configure the database, or later during database operations. Both the catalog and data directories must be owned by the database superuser .

The directory you specify for database catalog files (the catalog path) is used across all nodes in the cluster. For example, if you specify /home/catalog as the catalog directory, Vertica uses that catalog path on all nodes. The catalog directory should always be separate from any data file directories.

> **Note**
> Do not use a shared directory for more than one node. Data and catalog directories must be distinct for each node. Multiple nodes must not be allowed to write to the same data or catalog directory.

The data path you designate is also used across all nodes in the cluster. Specifying that data should be stored in /home/data, Vertica uses this path on all database nodes.

Do not use a single directory to contain both catalog and data files. You can store the catalog and data directories on different drives, which can be either on drives local to the host (recommended for the catalog directory) or on a shared storage location, such as an external disk enclosure or a SAN.

Before you specify a catalog or data path, be sure the parent directory exists on all nodes of your database. Creating a database in admintools also creates the catalog and data directories, but the parent directory must exist on each node.

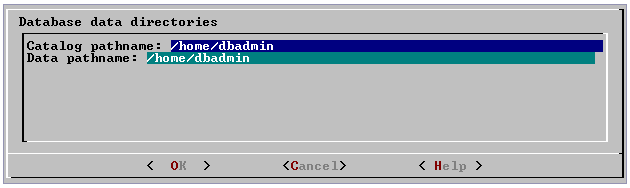
You do not need to specify a disk storage location during installation. However, you can do so by using the --data-dir parameter to the install_vertica script. See Specifying disk storage location during installation .

In this section

- Specifying disk storage location during database creation
- Specifying disk storage location on MC
- Configuring disk usage to optimize performance
- Using shared storage with Vertica
- Viewing database storage information
- Anti-virus scanning exclusions

Specifying disk storage location during database creation

When you invoke the Create Database command in the Administration tools , a dialog box allows you to specify the catalog and data locations. These locations must exist on each host in the cluster and must be owned by the database administrator.



When you click **OK** , Vertica automatically creates the following subdirectories:

```
catalog-pathname/database-name/node-name_catalog/data-pathname/database-name/node-name_data/
```

For example, if you use the default value (the database administrator's home directory) of /home/dbadmin for the Stock Exchange example database, the catalog and data directories are created on each node in the cluster as follows:

```
/home/dbadmin/Stock_Schema/stock_schema_node1_host01_catalog/home/dbadmin/Stock_Schema/stock_schema_node1_host01_data
```

Notes

- Catalog and data path names must contain only alphanumeric characters and cannot have leading space characters. Failure to comply with these restrictions will result in database creation failure.
- Vertica refuses to overwrite a directory if it appears to be in use by another database. Therefore, if you created a database for evaluation purposes, dropped the database, and want to reuse the database name, make sure that the disk storage location previously used has been completely cleaned up. See Managing storage locations for details.

Specifying disk storage location on MC

You can use the MC interface to specify where you want to store database metadata on the cluster in the following ways:

- When you configure MC the first time
- When you create new databases using on MC

See also

Configuring Management Console .

Configuring disk usage to optimize performance

Once you have created your initial storage location, you can add additional storage locations to the database later. Not only does this provide additional space, it lets you control disk usage and increase I/O performance by isolating files that have different I/O or access patterns. For example, consider:

- Isolating execution engine temporary files from data files by creating a separate storage location for temp space .
- Creating labeled storage locations and storage policies, in which selected database objects are stored on different storage locations based on measured performance statistics or predicted access patterns.

See also

Managing storage locations

Using shared storage with Vertica

If using shared SAN storage, ensure there is no contention among the nodes for disk space or bandwidth.

- Each host must have its own catalog and data locations. Hosts cannot share catalog or data locations.
- Configure the storage so that there is enough I/O bandwidth for each node to access the storage independently.

Viewing database storage information

You can view node-specific information on your Vertica cluster through the Management Console . See Monitoring Vertica Using Management Console for details.

Anti-virus scanning exclusions

You should exclude the Vertica catalog and data directories from anti-virus scanning. Certain anti-virus products have been identified as targeting Vertica directories, and sometimes lock or delete files in them. This can adversely affect Vertica performance and data integrity.

Identified anti-virus products include the following:

- ClamAV
- SentinelOne
- Sophos
- Symantec
- Twistlock

Important

This list is not comprehensive.

Disk space requirements for Vertica

In addition to actual data stored in the database, Vertica requires disk space for several data reorganization operations, such as mergeout and managing nodes in the cluster. For best results, Vertica recommends that disk utilization per node be no more than sixty percent (60%) for a K-Safe=1 database to allow such operations to proceed.

In addition, disk space is temporarily required by certain query execution operators, such as hash joins and sorts, in the case when they cannot be completed in memory (RAM). Such operators might be encountered during queries, recovery, refreshing projections, and so on. The amount of disk space needed (known as temp space) depends on the nature of the queries, amount of data on the node and number of concurrent users on the system. By default, any unused disk space on the data disk can be used as temp space. However, Vertica recommends provisioning temp space separate from data disk space.

See also

Configuring disk usage to optimize performance .

Disk space requirements for Management Console

You can install Management Console on any node in the cluster, so it has no special disk requirements other than disk space you allocate for your database cluster .

Prepare the logical schema script

Designing a logical schema for a Vertica database is no different from designing one for any other SQL database. Details are described more fully in Designing a logical schema .

To create your logical schema, prepare a SQL script (plain text file, typically with an extension of . sql) that:

1. Creates additional schemas (as necessary). See Using multiple schemas .
2. Creates the tables and column constraints in your database using the CREATE TABLE command.
3. Defines the necessary table constraints using the ALTER TABLE command.
4. Defines any views on the table using the CREATE VIEW command.

You can generate a script file using:

- A schema designer application.

- A schema extracted from an existing database.
- A text editor.
- One of the example database example-name_define_schema.sql scripts as a template. (See the example database directories in /opt/vertica/examples .)

In your script file, make sure that:

- Each statement ends with a semicolon.
- You use data types supported by Vertica, as described in the SQL Reference Manual.

Once you have created a database, you can test your schema script by executing it as described in Create the logical schema . If you encounter errors, drop all tables, correct the errors, and run the script again.

Prepare data files

Prepare two sets of data files:

- Test data files. Use test files to test the database after the partial data load. If possible, use part of the actual data files to prepare the test data files.
- Actual data files. Once the database has been tested and optimized, use your data files for your initial Data load .

How to name data files

Name each data file to match the corresponding table in the logical schema. Case does not matter.

Use the extension .tbl or whatever you prefer. For example, if a table is named Stock_Dimension , name the corresponding data file stock_dimension.tbl . When using multiple data files, append _nnn (where *nnn* is a positive integer in the range 001 to 999) to the file name. For example, stock_dimension.tbl_001 , stock_dimension.tbl_002 , and so on.

Prepare load scripts

> **Note**
> You can postpone this step if your goal is to test a logical schema design for validity.

Prepare SQL scripts to load data directly into physical storage using COPY on vsql , or through ODBC .

You need scripts that load:

- Large tables
- Small tables

Vertica recommends that you load large tables using multiple files. To test the load process, use files of 10GB to 50GB in size. This size provides several advantages:

- You can use one of the data files as a sample data file for the Database Designer .
- You can load just enough data to Perform a partial data load before you load the remainder.
- If a single load fails and rolls back, you do not lose an excessive amount of time.
- Once the load process is tested, for multi-terabyte tables, break up the full load in file sizes of 250–500GB.

See also

- Data load
- Using load scripts
- Distributing a load
- Constraint enforcement
- Handling messy data

> **Tip**
> You can use the load scripts included in the example databases as templates.

Create an optional sample query script

The purpose of a sample query script is to test your schema and load scripts for errors.

Include a sample of queries your users are likely to run against the database. If you don't have any real queries, just write simple SQL that collects counts on each of your tables. Alternatively, you can skip this step.

Create an empty database

Two options are available for creating an empty database:

- Using the Management Console . For details, see Creating a database using MC .
- Using Administration tools .

Although you can create more than one database (for example, one for production and one for testing), there can be only one active database for each installation of Vertica Analytic Database.

In this section

- Creating a database name and password
- Create a database using administration tools

Creating a database name and password

Database names

Database names must conform to the following rules:

- Be between 1-30 characters
- Begin with a letter
- Follow with any combination of letters (upper and lowercase), numbers, and/or underscores.

Database names are case sensitive; however, Vertica strongly recommends that you do not create databases with names that differ only in case. For example, do not create a database called mydatabase and another called MyDataBase .

Database passwords

Database passwords can contain letters, digits, and special characters listed in the next table. Passwords cannot include non-ASCII Unicode characters.

The allowed password length is between 0-100 characters. The database superuser can change a Vertica user's maximum password length using ALTER PROFILE .

You use Profiles to specify and control password definitions. For instance, a profile can define the maximum length, reuse time, and the minimum number or required digits for a password, as well as other details.

The following special (ASCII) characters are valid in passwords. Special characters can appear anywhere in a password string. For example, mypas$word or $mypassword are both valid, while ±mypassword is not. Using special characters other than the ones listed below can cause database instability.

- #
- ?
- =
- _
- '
-)
- (
- @
- \
- /
- !
- ,
- ~
- :
- %
- ;
- `
- ^

- +
- .
- -
- space
- &
- <
- >
- [
-]
- {
- }
- |
- *
- $
- "

See also

- Password guidelines
- ALTER PROFILE
- CREATE PROFILE
- DROP PROFILE

Create a database using administration tools

1. Run the Administration tools from your Administration host as follows:
   ```
   $ /opt/vertica/bin/admintools
   ```
   If you are using a remote terminal application, such as PuTTY or a Cygwin bash shell, see Notes for remote terminal users .
2. Accept the license agreement and specify the location of your license file. For more information see Managing licenses for more information. This step is necessary only if it is the first time you have run the Administration Tools
3. On the Main Menu, click **Configuration Menu** , and click **OK** .
4. On the Configuration Menu, click **Create Database** , and click **OK** .
5. Enter the name of the database and an optional comment, and click **OK** . See Creating a database name and password for naming guidelines and restrictions.
6. Establish the superuser password for your database.
   - To provide a password enter the password and click **OK** . Confirm the password by entering it again, and then click **OK** .
   - If you don't want to provide the password, leave it blank and click **OK** . If you don't set a password, Vertica prompts you to verify that you truly do not want to establish a superuser password for this database. Click **Yes** to create the database without a password or **No** to establish the password.

   Caution
   If you do not enter a password at this point, the superuser password is set to empty. Unless the database is for evaluation or academic purposes, Vertica strongly recommends that you enter a superuser password. See Creating a database name and password for guidelines.
7. Select the hosts to include in the database from the list of hosts specified when Vertica was installed (install_vertica -s), and click **OK** .
8. Specify the directories in which to store the data and catalog files, and click **OK** .

   Note
   Do not use a shared directory for more than one node. Data and catalog directories must be distinct for each node. Multiple nodes must not be allowed to write to the same data or catalog directory.
9. Catalog and data path names must contain only alphanumeric characters and cannot have leading spaces. Failure to comply with these restrictions results in database creation failure.
   For example:
   Catalog pathname: /home/dbadmin
   Data Pathname: /home/dbadmin
10. Review the **Current Database Definition** screen to verify that it represents the database you want to create, and then click **Yes** to proceed or **No**

to modify the database definition.

11. If you click **Yes** , Vertica creates the database you defined and then displays a message to indicate that the database was successfully created.

> **Note**
> For databases created with 3 or more nodes, Vertica automatically sets K-safety to 1 to ensure that the database is fault tolerant in case a node fails. For more information, see Failure recovery in the Administrator's Guide and MARK_DESIGN_KSAFE .

12. Click **OK** to acknowledge the message.

Create the logical schema

1. **Connect to the database** .
   In the Administration Tools Main Menu, click **Connect to Database** and click **OK** .
   See Connecting to the Database for details.
   The vsql welcome script appears:

```
Welcome to vsql, the Vertica Analytic Database interactive terminal.
Type:  \h or \? for help with vsql commands
     \g or terminate with semicolon to execute query
     \q to quit

=>
```

2. **Run the logical schema script**
   Using the \i meta-command in vsql to run the SQL logical schema script that you prepared earlier.
3. **Disconnect from the database**
   Use the \q meta-command in vsql to return to the Administration Tools.

Perform a partial data load

Vertica recommends that for large tables, you perform a partial data load and then test your database before completing a full data load. This load should load a representative amount of data.

1. **Load the small tables.**
   Load the small table data files using the SQL load scripts and data files you prepared earlier.
2. **Partially load the large tables.**
   Load 10GB to 50GB of table data for each table using the SQL load scripts and data files that you prepared earlier.

For more information about projections, see Projections .

Test the database

Test the database to verify that it is running as expected.

Check queries for syntax errors and execution times.

1. Use the vsql \timing meta-command to enable the display of query execution time in milliseconds.
2. Execute the SQL sample query script that you prepared earlier.
3. Execute several ad hoc queries.

Optimize query performance

Optimizing the database consists of optimizing for compression and tuning for queries. (See Creating a database design .)

To optimize the database, use the Database Designer to create and deploy a design for optimizing the database. See Using Database Designer to create a comprehensive design .

After you run the Database Designer, use the techniques described in Query optimization to improve the performance of certain types of queries.

> **Note**
> The database response time depends on factors such as type and size of the application query, database design, data size and data types stored, available computational power, and network bandwidth. Adding nodes to a database cluster does not necessarily improve the system response time for every query, especially if the response time is already short, e.g., less then 10 seconds, or the response time is not hardware bound.

Complete the data load

To complete the load:

1. Monitor system resource usage.
   Continue to run the top , free , and df utilities and watch them while your load scripts are running (as described in Monitoring Linux resource usage). You can do this on any or all nodes in the cluster. Make sure that the system is not swapping excessively (watch kswapd in top) or running out of swap space (watch for a large amount of used swap space in free).

   Note
   Vertica requires a dedicated server. If your loader or other processes take up significant amounts of RAM, it can result in swapping.

2. Complete the large table loads.
   Run the remainder of the large table load scripts.

Test the optimized database

Check query execution times to test your optimized design:

1. Use the vsql \timing meta-command to enable the display of query execution time in milliseconds.
   Execute a SQL sample query script to test your schema and load scripts for errors.

   Note
   Include a sample of queries your users are likely to run against the database. If you don't have any real queries, just write simple SQL that collects counts on each of your tables. Alternatively, you can skip this step.

2. Execute several ad hoc queries
   1. Run Administration tools and select Connect to Database .
   2. Use the \i meta-command to execute the query script; for example:

```
vmartdb=> \i vmart_query_03.sql  customer_name   | annual_income
------------------+---------------
 James M. McNulty |        999979
 Emily G. Vogel   |        999998
(2 rows)
Time: First fetch (2 rows): 58.411 ms. All rows formatted: 58.448 ms
vmartdb=> \i vmart_query_06.sql
 store_key | order_number | date_ordered
-----------+--------------+--------------
        45 |       202416 | 2004-01-04
       113 |        66017 | 2004-01-04
       121 |       251417 | 2004-01-04
        24 |       250295 | 2004-01-04
         9 |       188567 | 2004-01-04
       166 |        36008 | 2004-01-04
        27 |       150241 | 2004-01-04
       148 |       182207 | 2004-01-04
       198 |        75716 | 2004-01-04
(9 rows)
Time: First fetch (9 rows): 25.342 ms. All rows formatted: 25.383 ms
```

Once the database is optimized, it should run queries efficiently. If you discover queries that you want to optimize, you can modify and update the design incrementally .

Implement locales for international data sets

Locale specifies the user's language, country, and any special variant preferences, such as collation. Vertica uses locale to determine the behavior of certain string functions. Locale also determines the collation for various SQL commands that require ordering and comparison, such as aggregate GROUP BY and ORDER BY clauses, joins, and the analytic ORDER BY clause.

The default locale for a Vertica database is en_US@collation=binary (English US). You can define a new default locale that is used for all sessions on the database. You can also override the locale for individual sessions. However, projections are always collated using the default en_US@collation=binary collation, regardless of the session collation. Any locale-specific collation is applied at query time.

If you set the locale to null, Vertica sets the locale to en_US_POSIX . You can set the locale back to the default locale and collation by issuing the vsql meta-command \locale . For example:

Note

```
=> set locale to '';
INFO 2567:  Canonical locale: 'en_US_POSIX'
Standard collation: 'LEN'
English (United States, Computer)
SET
=> \locale en_US@collation=binary;
INFO 2567:  Canonical locale: 'en_US'
Standard collation: 'LEN_KBINARY'
English (United States)
=>  \locale
en_US@collation-binary;
```

You can set locale through ODBC , JDBC , and ADO.net .

ICU locale support

Vertica uses the ICU library for locale support; you must specify locale using the ICU locale syntax. The locale used by the database session is not derived from the operating system (through the LANG variable), so Vertica recommends that you set the LANG for each node running vsql, as described in the next section.

While ICU library services can specify collation, currency, and calendar preferences, Vertica supports only the collation component. Any keywords not relating to collation are rejected. Projections are always collated using the en_US@collation=binary collation regardless of the session collation. Any locale-specific collation is applied at query time.

The SET DATESTYLE TO ... command provides some aspects of the calendar, but Vertica supports only dollars as currency.

Changing DB locale for a session

This examples sets the session locale to Thai.

1. At the operating-system level for each node running vsql, set the LANG variable to the locale language as follows:
   ```
   export LANG=th_TH.UTF-8
   ```
   Note
   If setting the LANG= as shown does not work, the operating system support for locales may not be installed.
2. For each Vertica session (from ODBC/JDBC or vsql) set the language locale.
   From vsql:
   ```
   \locale th_TH
   ```
3. From ODBC/JDBC:
   ```
   "SET LOCALE TO th_TH;"
   ```
4. In PUTTY (or ssh terminal), change the settings as follows:
   ```
   settings > window > translation > UTF-8
   ```
5. Click **Apply** and then click **Save** .

All data loaded must be in UTF-8 format, not an ISO format, as described in Delimited data . Character sets like ISO 8859-1 (Latin1), which are incompatible with UTF-8, are not supported, so functions like SUBSTRING do not work correctly for multibyte characters. Thus, settings for locale should *not* work correctly. If the translation setting ISO-8859-11:2001 (Latin/Thai) works, the data is loaded incorrectly. To convert data correctly, use a utility program such as Linux iconv .

> **Note**
> The maximum length parameter for VARCHAR and CHAR data type refers to the number of octets (bytes) that can be stored in that field, not the number of characters. When using multi-byte UTF-8 characters, make sure to size fields to accommodate from 1 to 4 bytes per character, depending on the data.

See also

- Supported locales
- About locale
- SET LOCALE
- ICU User Guide

In this section

- Specify the default locale for the database
- Override the default locale for a session
- Server versus client locale settings

Specify the default locale for the database

After you start the database, the default locale configuration parameter, DefaultSessionLocale , sets the initial locale. You can override this value for individual sessions.

To set the locale for the database, use the configuration parameter as follows:

```
=> ALTER DATABASE DEFAULT SET DefaultSessionLocale = 'ICU-locale-identifier';
```

For example:

```
=> ALTER DATABASE DEFAULT SET DefaultSessionLocale = 'en_GB';
```

Override the default locale for a session

You can override the default locale for the current session in two ways:

- VSQL command \locale . For example:

  ```
  => \locale en_GBINFO:
  INFO 2567:  Canonical locale: 'en_GB'
  Standard collation: 'LEN'
  English (United Kingdom)
  ```

- SQL statement SET LOCALE . For example:

  ```
  => SET LOCALE TO en_GB;
  INFO 2567:  Canonical locale: 'en_GB'
  Standard collation: 'LEN'
  English (United Kingdom)
  ```

Both methods accept locale short and long forms. For example:

```
=> SET LOCALE TO LEN;
INFO 2567:  Canonical locale: 'en'
Standard collation: 'LEN'
English
```

```
=> \locale LEN
INFO 2567:  Canonical locale: 'en'
Standard collation: 'LEN'
English
```

See also

- SET LOCALE

Server versus client locale settings

Vertica differentiates database server locale settings from client application locale settings:

- Server locale settings only impact collation behavior for server-side query processing.
- Client applications verify that locale is set appropriately in order to display characters correctly.

The following sections describe best practices to ensure predictable results.

Server locale

The server session locale should be set as described in Specify the default locale for the database . If locales vary across different sessions, set the server locale at the start of each session from your client.

vsql client

- If the database does not have a default session locale, set the server locale for the session to the desired locale .
- The locale setting in the terminal emulator where the vsql client runs should be set to be equivalent to session locale setting on the server side (ICU locale). By doing so, the data is collated correctly on the server and displayed correctly on the client.
- All input data for vsql should be in UTF-8, and all output data is encoded in UTF-8
- Vertica does not support non UTF-8 encodings and associated locale values; .
- For instructions on setting locale and encoding, refer to your terminal emulator documentation.

ODBC clients

- ODBC applications can be either in ANSI or Unicode mode. If the user application is Unicode, the encoding used by ODBC is UCS-2. If the user application is ANSI, the data must be in single-byte ASCII, which is compatible with UTF-8 used on the database server. The ODBC driver converts UCS-2 to UTF-8 when passing to the Vertica server and converts data sent by the Vertica server from UTF-8 to UCS-2.
- If the user application is not already in UCS-2, the application must convert the input data to UCS-2, or unexpected results could occur. For example:
  - For non-UCS-2 data passed to ODBC APIs, when it is interpreted as UCS-2, it could result in an invalid UCS-2 symbol being passed to the APIs, resulting in errors.
  - The symbol provided in the alternate encoding could be a valid UCS-2 symbol. If this occurs, incorrect data is inserted into the database.
- If the database does not have a default session locale, ODBC applications should set the desired server session locale using SQLSetConnectAttr (if different from database wide setting). By doing so, you get the expected collation and string functions behavior on the server.

JDBC and ADO.NET clients

- JDBC and ADO.NET applications use a UTF-16 character set encoding and are responsible for converting any non-UTF-16 encoded data to UTF-16. The same cautions apply as for ODBC if this encoding is violated.
- The JDBC and ADO.NET drivers convert UTF-16 data to UTF-8 when passing to the Vertica server and convert data sent by Vertica server from UTF-8 to UTF-16.
- If there is no default session locale at the database level, JDBC and ADO.NET applications should set the correct server session locale by executing the SET LOCALE TO command in order to get the expected collation and string functions behavior on the server. For more information, see SET LOCALE .

Using time zones with Vertica

Vertica uses the public-domain tz database (time zone database), which contains code and data that represent the history of local time for locations around the globe. This database organizes time zone and daylight saving time data by partitioning the world into timezones whose clocks all agree on timestamps that are later than the POSIX Epoch (1970-01-01 00:00:00 UTC). Each timezone has a unique identifier. Identifiers typically follow the convention *area* / *location* , where *area* is a continent or ocean, and *location* is a specific location within the area—for example, Africa/Cairo, America/New_York, and Pacific/Honolulu.

Important

IANA acknowledge that 1970 is an arbitrary cutoff. They note the problems that face moving the cutoff earlier "due to the wide variety of local practices before computer timekeeping became prevalent." IANA's own description of the tz database suggests that users should regard historical dates and times, especially those that predate the POSIX epoch date, with a healthy measure of skepticism. For details, see Theory and pragmatics of

the tz code and data .

Vertica uses the TZ environment variable (if set) on each node for the default current time zone. Otherwise, Vertica uses the operating system time zone.

The TZ variable can be set by the operating system during login (see /etc/profile , /etc/profile.d , or /etc/bashrc) or by the user in .profile , .bashrc or .bash-profile . TZ must be set to the same value on each node when you start Vertica.

The following command returns the current time zone for your database:

```
=> SHOW TIMEZONE;
  name  |     setting
----------+------------------
 timezone | America/New_York
(1 row)
```

You can also set the time zone for a single session with SET TIME ZONE .

Conversion and storage of date/time data

There is no database default time zone. TIMESTAMPTZ (TIMESTAMP WITH TIMEZONE) data is converted from the current local time and stored as GMT/UTC (Greenwich Mean Time/Coordinated Universal Time).

When TIMESTAMPTZ data is used, data is converted back to the current local time zone, which might be different from the local time zone where the data was stored. This conversion takes into account daylight saving time (summer time), depending on the year and date to determine when daylight saving time begins and ends.

TIMESTAMP WITHOUT TIMEZONE data stores the timestamp as given, and retrieves it exactly as given. The current time zone is ignored. The same is true for TIME WITHOUT TIMEZONE. For TIME WITH TIMEZONE (TIMETZ), however, the current time zone setting is stored along with the given time, and that time zone is used on retrieval.

> **Note**
> Vertica recommends that you use TIMESTAMPTZ, not TIMETZ.

Querying data/time data

TIMESTAMPTZ uses the current time zone on both input and output, as in the following example:

```
=> CREATE TEMP TABLE s (tstz TIMESTAMPTZ);=> SET TIMEZONE TO 'America/New_York';
=> INSERT INTO s VALUES ('2009-02-01 00:00:00');
=> INSERT INTO s VALUES ('2009-05-12 12:00:00');
=> SELECT tstz AS 'Local timezone', tstz AT TIMEZONE 'America/New_York' AS 'America/New_York',
  tstz AT TIMEZONE 'GMT' AS 'GMT' FROM s;
    Local timezone     |  America/New_York   |         GMT
------------------------+---------------------+---------------------
 2009-02-01 00:00:00-05 | 2009-02-01 00:00:00 | 2009-02-01 05:00:00
 2009-05-12 12:00:00-04 | 2009-05-12 12:00:00 | 2009-05-12 16:00:00
(2 rows)
```

The -05 in the Local time zone column shows that the data is displayed in EST, while -04 indicates EDT. The other two columns show the TIMESTAMP WITHOUT TIMEZONE at the specified time zone.

The next example shows what happens if the current time zone is changed to GMT:

```
=> SET TIMEZONE TO 'GMT';=> SELECT tstz AS 'Local timezone', tstz AT TIMEZONE 'America/New_York' AS
  'America/New_York', tstz AT TIMEZONE 'GMT' as 'GMT' FROM s;
    Local timezone     |  America/New_York   |         GMT
------------------------+---------------------+---------------------
 2009-02-01 05:00:00+00 | 2009-02-01 00:00:00 | 2009-02-01 05:00:00
 2009-05-12 16:00:00+00 | 2009-05-12 12:00:00 | 2009-05-12 16:00:00
(2 rows)
```

The +00 in the Local time zone column indicates that TIMESTAMPTZ is displayed in GMT.

The approach of using TIMESTAMPTZ fields to record events captures the GMT of the event, as expressed in terms of the local time zone. Later, it allows for easy conversion to any other time zone, either by setting the local time zone or by specifying an explicit AT TIMEZONE clause.

The following example shows how TIMESTAMP WITHOUT TIMEZONE fields work in Vertica.

```
=> CREATE TEMP TABLE tnoz (ts TIMESTAMP);=> INSERT INTO tnoz VALUES('2009-02-01 00:00:00');
=> INSERT INTO tnoz VALUES('2009-05-12 12:00:00');
=> SET TIMEZONE TO 'GMT';
=> SELECT ts AS 'No timezone', ts AT TIMEZONE 'America/New_York' AS
  'America/New_York', ts AT TIMEZONE 'GMT' AS 'GMT' FROM tnoz;
     No timezone    |    America/New_York    |         GMT
--------------------+------------------------+------------------------
 2009-02-01 00:00:00 | 2009-02-01 05:00:00+00 | 2009-02-01 00:00:00+00
 2009-05-12 12:00:00 | 2009-05-12 16:00:00+00 | 2009-05-12 12:00:00+00
(2 rows)
```

The +00 at the end of a timestamp indicates that the setting is TIMESTAMP WITH TIMEZONE in GMT (the current time zone). The America/New_York column shows what the GMT setting was when you recorded the time, assuming you read a normal clock in the America/New_York time zone. What this shows is that if it is midnight in the America/New_York time zone, then it is 5 am GMT.

> **Note**
> 00:00:00 Sunday February 1, 2009 in America/New_York converts to 05:00:00 Sunday February 1, 2009 in GMT.

The GMT column displays the GMT time, assuming the input data was captured in GMT.

If you don't set the time zone to GMT, and you use another time zone, for example America/New_York, then the results display in America/New_York with a -05 and -04 , showing the difference between that time zone and GMT.

```
=> SET TIMEZONE TO 'America/New_York';
=> SHOW TIMEZONE;
   name   |     setting
----------+------------------
 timezone | America/New_York
(1 row)
=> SELECT ts AS 'No timezone', ts AT TIMEZONE 'America/New_York' AS
  'America/New_York', ts AT TIMEZONE 'GMT' AS 'GMT' FROM tnoz;
     No timezone    |    America/New_York    |         GMT
--------------------+------------------------+------------------------
 2009-02-01 00:00:00 | 2009-02-01 00:00:00-05 | 2009-01-31 19:00:00-05
 2009-05-12 12:00:00 | 2009-05-12 12:00:00-04 | 2009-05-12 08:00:00-04
(2 rows)
```

In this case, the last column is interesting in that it returns the time in New York, given that the data was captured in GMT.

See also

- TZ environment variable
- SET TIME ZONE
- Date/time data types

Change transaction isolation levels

By default, Vertica uses the READ COMMITTED isolation level for all sessions. You can change the default isolation level for the database or for a given session.

A transaction retains its isolation level until it completes, even if the session's isolation level changes during the transaction. Vertica internal processes (such as the Tuple Mover and refresh operations) and DDL operations always run at the SERIALIZABLE isolation level to ensure consistency.

Database isolation level

The configuration parameter TransactionIsolationLevel specifies the database isolation level, and is used as the default for all sessions. Use ALTER DATABASE to change the default isolation level.For example:

```
=> ALTER DATABASE DEFAULT SET TransactionIsolationLevel = 'SERIALIZABLE';
ALTER DATABASE
=> ALTER DATABASE DEFAULT SET TransactionIsolationLevel = 'READ COMMITTED';
ALTER DATABASE
```

Changes to the database isolation level only apply to future sessions. Existing sessions and their transactions continue to use their original isolation level.

Use SHOW CURRENT to view the database isolation level:

```
=> SHOW CURRENT TransactionIsolationLevel;
  level   |          name           |    setting
----------+-------------------------+----------------
 DATABASE | TransactionIsolationLevel | READ COMMITTED
(1 row)
```

Session isolation level

SET SESSION CHARACTERISTICS AS TRANSACTION changes the isolation level for a specific session. For example:

```
=> SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
```

Use SHOW to view the current session's isolation level:

```
=> SHOW TRANSACTION_ISOLATION;
```

See also

Transactions

Configuration parameter management

Vertica supports a wide variety of configuration parameters that affect many facets of database behavior. These parameters can be set with the appropriate ALTER statements at one or more levels, listed here in descending order of precedence:

1. User (ALTER USER)
2. Session (ALTER SESSION)
3. Node (ALTER NODE)
4. Database (ALTER DATABASE)

You can query system table CONFIGURATION_PARAMETERS to obtain the current settings for all user-accessible parameters. For example, the following query obtains settings for partitioning parameters: their current and default values, which levels they can be set at, and whether changes require a database restart to take effect:

```
=> SELECT parameter_name, current_value, default_value, allowed_levels, change_requires_restart
     FROM configuration_parameters  WHERE parameter_name ILIKE '%partitioncount%';
   parameter_name    | current_value | default_value | allowed_levels | change_requires_restart
---------------------+---------------+---------------+----------------+-------------------------
 MaxPartitionCount   | 1024          | 1024          | NODE, DATABASE | f
 ActivePartitionCount | 1             | 1             | NODE, DATABASE | f
(2 rows)
```

For details about individual configuration parameters grouped by category, see Configuration parameters .

Setting and clearing configuration parameters

You change specific configuration parameters with the appropriate ALTER statements; the same statements also let you reset configuration parameters to their default values. For example, the following ALTER statements change ActivePartitionCount at the database level from 1 to 2 , and DisablePartitionCount at the session level from 0 to 1:

```
=> ALTER DATABASE DEFAULT SET ActivePartitionCount = 2;
ALTER DATABASE
=> ALTER SESSION SET DisableAutopartition = 1;
ALTER SESSION
=> SELECT parameter_name, current_value, default_value FROM configuration_parameters
     WHERE parameter_name IN ('ActivePartitionCount', 'DisableAutopartition');
   parameter_name    | current_value | default_value
---------------------+---------------+---------------
 ActivePartitionCount | 2             | 1
 DisableAutopartition | 1             | 0
(2 rows)
```

You can later reset the same configuration parameters to their default values:

```
=> ALTER DATABASE DEFAULT CLEAR ActivePartitionCount;
ALTER DATABASE
=> ALTER SESSION CLEAR DisableAutopartition;
ALTER DATABASE
=> SELECT parameter_name, current_value, default_value FROM configuration_parameters
     WHERE parameter_name IN ('ActivePartitionCount', 'DisableAutopartition');
   parameter_name    | current_value | default_value
---------------------+---------------+---------------
 DisableAutopartition | 0             | 0
 ActivePartitionCount | 1             | 1
(2 rows)
```

Caution
Vertica is designed to operate with minimal configuration changes. Be careful to change configuration parameters according to documented guidelines.

In this section

- Viewing configuration parameter values

Viewing configuration parameter values

You can view active configuration parameter values in two ways:

- SHOW statements
- Query related system tables

SHOW statements

Use the following SHOW statements to view active configuration parameters:

- SHOW CURRENT : Returns settings of active configuration parameter values. Vertica checks settings at all levels, in the following ascending order of precedence:
  - session
  - node
  - database

  If no values are set at any scope, SHOW CURRENT returns the parameter's default value.
- SHOW DATABASE : Displays configuration parameter values set for the database.
- SHOW USER : Displays configuration parameters set for the specified user, and for all users.
- SHOW SESSION : Displays configuration parameter values set for the current session.
- SHOW NODE : Displays configuration parameter values set for a node.

If a configuration parameter requires a restart to take effect, the values in a SHOW CURRENT statement might differ from values in other SHOW statements. To see which parameters require restart, query the CONFIGURATION_PARAMETERS system table.

System tables

You can query several system tables for configuration parameters:

- SESSION_PARAMETERS returns session-scope parameters.

- CONFIGURATION_PARAMETERS returns parameters for all scopes: database, node, and session.
- USER_CONFIGURATION_PARAMETERS provides information about user-level configuration parameters that are in effect for database users.

Designing a logical schema

Designing a logical schema for a Vertica database is the same as designing for any other SQL database. A logical schema consists of objects such as schemas, tables, views and referential Integrity constraints that are visible to SQL users. Vertica supports any relational schema design that you choose.

In this section

- Using multiple schemas
- Tables in schemas

Using multiple schemas

Using a single schema is effective if there is only one database user or if a few users cooperate in sharing the database. In many cases, however, it makes sense to use additional schemas to allow users and their applications to create and access tables in separate namespaces. For example, using additional schemas allows:

- Many users to access the database without interfering with one another.
  Individual schemas can be configured to grant specific users access to the schema and its tables while restricting others.
- Third-party applications to create tables that have the same name in different schemas, preventing table collisions.

Unlike other RDBMS, a schema in a Vertica database is not a collection of objects bound to one user.

In this section

- Multiple schema examples
- Creating schemas
- Specifying objects in multiple schemas
- Setting search paths
- Creating objects that span multiple schemas

Multiple schema examples

This section provides examples of when and how you might want to use multiple schemas to separate database users. These examples fall into two categories: using multiple private schemas and using a combination of private schemas (i.e. schemas limited to a single user) and shared schemas (i.e. schemas shared across multiple users).

Using multiple private schemas

Using multiple private schemas is an effective way of separating database users from one another when sensitive information is involved. Typically a user is granted access to only one schema and its contents, thus providing database security at the schema level. Database users can be running different applications, multiple copies of the same application, or even multiple instances of the same application. This enables you to consolidate applications on one database to reduce management overhead and use resources more effectively. The following examples highlight using multiple private schemas.

Using multiple schemas to separate users and their unique applications

In this example, both database users work for the same company. One user (HRUser) uses a Human Resource (HR) application with access to sensitive personal data, such as salaries, while another user (MedUser) accesses information regarding company healthcare costs through a healthcare management application. HRUser should not be able to access company healthcare cost information and MedUser should not be able to view personal employee data.

To grant these users access to data they need while restricting them from data they should not see, two schemas are created with appropriate user access, as follows:

- HRSchema—A schema owned by HRUser that is accessed by the HR application.
- HealthSchema—A schema owned by MedUser that is accessed by the healthcare management application.

Using multiple schemas to support multitenancy

This example is similar to the last example in that access to sensitive data is limited by separating users into different schemas. In this case, however, each user is using a virtual instance of the same application.

An example of this is a retail marketing analytics company that provides data and software as a service (SaaS) to large retailers to help them determine which promotional methods they use are most effective at driving customer sales.

In this example, each database user equates to a retailer, and each user only has access to its own schema. The retail marketing analytics company provides a virtual instance of the same application to each retail customer, and each instance points to the user's specific schema in which to create and update tables. The tables in these schemas use the same names because they are created by instances of the same application, but they do not conflict because they are in separate schemas.

Example of schemas in this database could be:

- MartSchema—A schema owned by MartUser, a large department store chain.
- PharmSchema—A schema owned by PharmUser, a large drug store chain.

Using multiple schemas to migrate to a newer version of an application

Using multiple schemas is an effective way of migrating to a new version of a software application. In this case, a new schema is created to support the new version of the software, and the old schema is kept as long as necessary to support the original version of the software. This is called a "rolling application upgrade."

For example, a company might use a HR application to store employee data. The following schemas could be used for the original and updated versions of the software:

- HRSchema—A schema owned by HRUser, the schema user for the original HR application.
- V2HRSchema—A schema owned by V2HRUser, the schema user for the new version of the HR application.

Combining private and shared schemas

The previous examples illustrate cases in which all schemas in the database are private and no information is shared between users. However, users might want to share common data. In the retail case, for example, MartUser and PharmUser might want to compare their per store sales of a particular product against the industry per store sales average. Since this information is an industry average and is not specific to any retail chain, it can be placed in a schema on which both users are granted USAGE privileges .

Example of schemas in this database might be:

- MartSchema—A schema owned by MartUser, a large department store chain.
- PharmSchema—A schema owned by PharmUser, a large drug store chain.
- IndustrySchema—A schema owned by DBUser (from the retail marketing analytics company) on which both MartUser and PharmUser have USAGE privileges. It is unlikely that retailers would be given any privileges beyond USAGE on the schema and SELECT on one or more of its tables.

Creating schemas

You can create as many schemas as necessary for your database. For example, you could create a schema for each database user. However, schemas and users are not synonymous as they are in Oracle.

By default, only a superuser can create a schema or give a user the right to create a schema. (See GRANT (database) in the SQL Reference Manual.)

To create a schema use the CREATE SCHEMA statement, as described in the SQL Reference Manual.

Specifying objects in multiple schemas

Once you create two or more schemas, each SQL statement or function must identify the schema associated with the object you are referencing. You can specify an object within multiple schemas by:

- Qualifying the object name by using the schema name and object name separated by a dot. For example, to specify MyTable , located in Schema1 , qualify the name as Schema1.MyTable .
- Using a search path that includes the desired schemas when a referenced object is unqualified. By Setting search paths , Vertica will automatically search the specified schemas to find the object.

Setting search paths

Each user session has a search path of schemas. Vertica uses this search path to find tables and user-defined functions (UDFs) that are unqualified by their schema name. A session search path is initially set from the user's profile. You can change the session's search path at any time by calling SET SEARCH_PATH . This search path remains in effect until the next SET SEARCH_PATH statement, or the session ends.

Viewing the current search path

SHOW SEARCH_PATH returns the session's current search path. For example:

```
=> SHOW SEARCH_PATH;
   name    |                   setting
-----------+-------------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
```

Schemas are listed in descending order of precedence. The first schema has the highest precedence in the search order. If this schema exists, it is also defined as the current schema, which is used for tables that are created with unqualified names. You can identify the current schema by calling the function CURRENT_SCHEMA :

```
=> SELECT CURRENT_SCHEMA;
 current_schema
----------------
 public
(1 row)
```

Setting the user search path

A session search path is initially set from the user's profile. If the search path in a user profile is not set by CREATE USER or ALTER USER , it is set to the database default:

```
=> CREATE USER agent007;
CREATE USER
=> \c - agent007
You are now connected as user "agent007".
=> SHOW SEARCH_PATH;
   name    |                   setting
-----------+-------------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
```

$user resolves to the session user name—in this case, agent007 —and has the highest precedence. If a schema agent007 , exists, Vertica begins searches for unqualified tables in that schema. Also, calls to CURRENT_SCHEMA return this schema. Otherwise, Vertica uses public as the current schema and begins searches in it.

Use ALTER USER to modify an existing user's search path. These changes overwrite all non-system schemas in the search path, including $USER . System schemas are untouched. Changes to a user's search path take effect only when the user starts a new session; current sessions are unaffected.

Important

After modifying the user's search path, verify that the user has access privileges to all schemas that are on the updated search path.

For example, the following statements modify agent007 's search path, and grant access privileges to schemas and tables that are on the new search path:

```
=> ALTER USER agent007 SEARCH_PATH store, public;
ALTER USER
=> GRANT ALL ON SCHEMA store, public TO agent007;
GRANT PRIVILEGE
=> GRANT SELECT ON ALL TABLES IN SCHEMA store, public TO agent007;
GRANT PRIVILEGE
=> \c - agent007
You are now connected as user "agent007".
=> SHOW SEARCH_PATH;
   name    |                   setting
-----------+-------------------------------------------------
 search_path | store, public, v_catalog, v_monitor, v_internal
(1 row)
```

To verify a user's search path, query the system table USERS :

```
=> SELECT search_path FROM USERS WHERE user_name='agent007';
                search_path
----------------------------------------------------
 store, public, v_catalog, v_monitor, v_internal
(1 row)
```

To revert a user's search path to the database default settings, call ALTER USER and set the search path to DEFAULT . For example:

```
=> ALTER USER agent007 SEARCH_PATH DEFAULT;
ALTER USER
=> SELECT search_path FROM USERS WHERE user_name='agent007';
                 search_path
-----------------------------------------------------
 "$user", public, v_catalog, v_monitor, v_internal
(1 row)
```

Ignored search path schemas

Vertica only searches among existing schemas to which the current user has access privileges. If a schema in the search path does not exist or the user lacks access privileges to it, Vertica silently excludes it from the search. For example, if agent007 lacks SELECT privileges to schema public , Vertica silently skips this schema. Vertica returns with an error only if it cannot find the table anywhere on the search path.

Setting session search path

Vertica initially sets a session's search path from the user's profile. You can change the current session's search path with SET SEARCH_PATH . You can use SET SEARCH_PATH in two ways:

- Explicitly set the session search path to one or more schemas. For example:

```
=> \c - agent007
You are now connected as user "agent007".
dbadmin=> SHOW SEARCH_PATH;
   name    |                    setting
-----------+-----------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
(1 row)

=> SET SEARCH_PATH TO store, public;
SET
=> SHOW SEARCH_PATH;
   name    |                   setting
-----------+---------------------------------------------
 search_path | store, public, v_catalog, v_monitor, v_internal
(1 row)
```

- Set the session search path to the database default:

```
=> SET SEARCH_PATH TO DEFAULT;
SET
=> SHOW SEARCH_PATH;
   name    |                    setting
-----------+-----------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
(1 row)
```

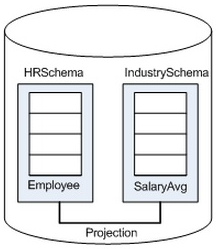
SET SEARCH_PATH overwrites all non-system schemas in the search path, including $USER . System schemas are untouched.

Creating objects that span multiple schemas

Vertica supports views that reference tables across multiple schemas. For example, a user might need to compare employee salaries to industry averages. In this case, the application queries two schemas:

- Shared schema IndustrySchema for salary averages

- Private schema HRSchema for company-specific salary information



Best Practice: When creating objects that span schemas, use qualified table names. This naming convention avoids confusion if the query path or table structure within the schemas changes at a later date.

Tables in schemas

In Vertica you can create persistent and temporary tables, through CREATE TABLE and CREATE TEMPORARY TABLE , respectively.

For detailed information on both types, see Creating Tables and Creating temporary tables .

Persistent tables

CREATE TABLE creates a table in the Vertica logical schema . For example:

```
CREATE TABLE vendor_dimension (
  vendor_key        INTEGER      NOT NULL PRIMARY KEY,
  vendor_name       VARCHAR(64),
  vendor_address    VARCHAR(64),
  vendor_city       VARCHAR(64),
  vendor_state      CHAR(2),
  vendor_region     VARCHAR(32),
  deal_size         INTEGER,
  last_deal_update  DATE

);
```

For detailed information, see Creating Tables.

Temporary tables

CREATE TEMPORARY TABLE creates a table whose data persists only during the current session. Temporary table data is never visible to other sessions.

Temporary tables can be used to divide complex query processing into multiple steps. Typically, a reporting tool holds intermediate results while reports are generated—for example, the tool first gets a result set, then queries the result set, and so on.

CREATE TEMPORARY TABLE can create tables at two scopes, global and local, through the keywords GLOBAL and LOCAL , respectively:

- GLOBAL (default): The table definition is visible to all sessions. However, table data is session-scoped.
- LOCAL : The table definition is visible only to the session in which it is created. When the session ends, Vertica automatically drops the table.

For detailed information, see Creating temporary tables .

Creating a database design

A *design* is a physical storage plan that optimizes query performance. Data in Vertica is physically stored in projections. When you initially load data into a table using INSERT, COPY (or COPY LOCAL), Vertica creates a default superprojection for the table. This superprojection ensures that all of the data is available for queries. However, these superprojections might not optimize database performance, resulting in slow query performance and low data compression.

To improve performance, create a design for your Vertica database that optimizes query performance and data compression. You can create a design in several ways:

- Use Database Designer , a toolthat recommends a design for optimal performance.
- Manually create a design
- Use Database Designer to create an initial design and then manually modify it.

Database Designer can help you minimize how much time you spend on manual database tuning. You can also use Database Designer to redesign the database incrementally as requirements such as workloads change over time.

Database Designer runs as a background process. This is useful if you have a large design that you want to run overnight. An active SSH session is not required, so design and deploy operations continue to run uninterrupted if the session ends.

> **Tip**
> Vertica recommends that you first globally optimize your database using the Comprehensive setting in Database Designer. If the performance of the comprehensive design is not adequate, you can design custom projections using an incremental design and manually, as described in Creating custom designs .

In this section

- About Database Designer
- How Database Designer creates a design
- Database Designer access requirements
- Logging projection data for Database Designer
- General design settings
- Building a design
- Resetting a design
- Deploying a design
- How to create a design
- Running Database Designer programmatically
- Creating custom designs

About Database Designer

Vertica Database Designer uses sophisticated strategies to create a design that provides excellent performance for ad-hoc queries and specific queries while using disk space efficiently.

During the design process, Database Designer analyzes the logical schema definition, sample data, and sample queries, and creates a physical schema (projections) in the form of a SQL script that you deploy automatically or manually. This script creates a minimal set of superprojections to ensure K-safety.

In most cases, the projections that Database Designer creates provide excellent query performance within physical constraints while using disk space efficiently.

General design options

When you run Database Designer, several general options are available:

- Create a comprehensive or incremental design.
- Optimize for query execution, load, or a balance of both.
- Require K-safety.
- Recommend unsegmented projections when feasible.
- Analyze statistics before creating the design.

Design input

Database Designer bases its design on the following information that you provide:

- **Design queries** that you typically run during normal database operations.
- **Design tables** that contain sample data.

Output

Database Designer yields the following output:

- Design script that creates the projections for the design in a way that meets the optimization objectives and distributes data uniformly across the cluster.
- Deployment script that creates and refreshes the projections for your design. For comprehensive designs, the deployment script contains commands that remove non-optimized projections. The deployment script includes the full design script.
- Backup script that contains SQL statements to deploy the design that existed on the system before deployment. This file is useful in case you need to revert to the pre-deployment design.

Design restrictions

Database Designer-generated designs:

- Exclude live aggregate or Top-K projections. You must create these manually. See CREATE PROJECTION .
- Do not sort, segment, or partition projections on LONG VARBINARY and LONG VARCHAR columns.
- Do not support operations on complex types.

Post-design options

While running Database Designer, you can choose to deploy your design automatically after the deployment script is created, or to deploy it manually, after you have reviewed and tested the design. Vertica recommends that you test the design on a non-production server before deploying the design to your production server.

How Database Designer creates a design

Design recommendations

Database Designer-generated designs can include the following recommendations:

- Sort buddy projections in the same order, which can significantly improve load, recovery, and site node performance. All buddy projections have the same base name so that they can be identified as a group.

  > **Note**
  > If you manually create projections, Database Designer recommends a buddy with the same sort order, if one does not already exist. By default, Database Designer recommends both super and non-super segmented projections with a buddy of the same sort order and segmentation.

- Accepts unlimited queries for a comprehensive design.
- Identifies similar design queries and assigns them a signature.
  For queries with the same signature, Database Designer weights the queries, depending on how many queries have that signature. It then considers the weighted query when creating a design.
- Recommends and creates projections in a way that minimizes data skew by distributing data uniformly across the cluster.
- Produces higher quality designs by considering UPDATE, DELETE, and SELECT statements.

Database Designer access requirements

By default, only users with the DBADMIN role can run Database Designer. Non-DBADMIN users can run Database Designer only if they are granted the necessary privileges and DBDUSER role, as described below. You can also enable users to run Database Designer on the Management Console (see Enabling Users to run Database Designer on Management Console).

1. Add a temporary folder to all cluster nodes with CREATE LOCATION :
   ```
   => CREATE LOCATION '/tmp/dbd' ALL NODES;
   ```
2. Grant the desired user CREATE privileges to create schemas on the current (DEFAULT) database, with GRANT DATABASE :
   ```
   => GRANT CREATE ON DATABASE DEFAULT TO dbd-user;
   ```
3. Grant the DBDUSER role to *dbd-user* with GRANT ROLE :
   ```
   => GRANT DBDUSER TO dbd-user;
   ```
4. On all nodes in the cluster, grant *dbd-user* access to the temporary folder with GRANT LOCATION :
   ```
   => GRANT ALL ON LOCATION '/tmp/dbd' TO dbd-user;
   ```
5. Grant *dbd-user* privileges on one or more database schemas and their tables, with GRANT SCHEMA and GRANT TABLE , respectively:
   ```
   => GRANT ALL ON SCHEMA this-schema[,...] TO dbd-user;
   => GRANT ALL ON ALL TABLES IN SCHEMA this-schema[,...] TO dbd-user;
   ```
6. Enable the DBDUSER role on *dbd-user* in one of the following ways:
   - As *dbd-user* , enable the DBDUSER role with SET ROLE :
     ```
     => SET ROLE DBDUSER;
     ```
   - As DBADMIN, automatically enable the DBDUSER role for *dbd-user* on each login, with ALTER USER :
     ```
     => ALTER USER dbd-user DEFAULT ROLE DBDUSER;
     ```

Important

When you grant the DBDUSER role, be sure to associate a resource pool with that user to manage resources while Database Designer runs.

Multiple users can run Database Designer concurrently without interfering with each other or exhausting cluster resources. When a user runs Database Designer, either with Management Console or programmatically, execution is generally contained by the user's resource pool, but might spill over into system resource pools for less-intensive tasks.

Enabling users to run Database Designer on Management Console

Users who are already granted the DBDUSER role and required privileges, as described above , can also be enabled to run Database Designer on Management Console :

1. Log in as a superuser to Management Console.
2. Click **MC Settings** .
3. Click **User Management** .
4. Specify an MC user:
   - To create an MC user, click **Add** .
   - To use an existing MC user, select the user and click **Edit** .
5. Next to the **DB access level** window, click **Add** .
6. In the **Add Permissions** window:
   1. From the **Choose a database** drop-down list, select the database on which to create a design.
   2. In the **Database username** field, enter the *dbd-user* user name that you created earlier .
   3. In the Database password field, enter the database password.
   4. In the **Restrict access** drop-down list, select the level of MC user for this user.
7. Click **OK** to save your changes.
8. Log out of the MC Super user account.

The MC user is now mapped to *dbd-user* . Log in as the MC user and use Database Designer to create an optimized design for your database.

DBDUSER capabilities and limitations

As a DBDUSER, the following constraints apply:

- Designs must set K-safety to be equal to system K-safety. If a design violates K-safety by lacking enough buddy projections for tables, the design does not complete.
- You cannot explicitly advance the ancient history mark (AHM)—for example, call MAKE_AHM_NOW —until after deploying the design.

When you create a design, you automatically have privileges to manipulate that design. Other tasks might require additional privileges:

Task	Required privileges
Submit design tables	• USAGE on the design table schema • OWNER on the design table
Submit a single design query	• EXECUTE on the design query
Submit a file of design queries	• READ privilege on the storage location that contains the query file • EXECUTE privilege on all queries in the file
Submit design queries from results of a user query	• EXECUTE privilege on the user queries • EXECUTE privilege on each design query retrieved from the results of the user query
Create design and deployment scripts	• WRITE privilege on the storage location of the design script • WRITE privilege on the storage location of the deployment script

Logging projection data for Database Designer

When you run Database Designer, the Optimizer proposes a set of ideal projections based on the options that you specify. When you deploy the design, Database Designer creates the design based on these projections. However, space or budget constraints may prevent Database Designer from creating all the proposed projections. In addition, Database Designer may not be able to implement the projections using ideal criteria.

To get information about the projections, first enable the Database Designer logging capability. When enabled, Database Designer stores information about the proposed projections in two Data Collector tables. After Database Designer deploys the design, these logs contain information about which proposed projections were actually created. After deployment, the logs contain information about:

- Projections that the Optimizer proposed
- Projections that Database Designer actually created when the design was deployed
- Projections that Database Designer created, but not with the ideal criteria that the Optimizer identified.
- The DDL used to create all the projections
- Column optimizations

If you do not deploy the design immediately, review the log to determine if you want to make any changes. If the design has been deployed, you can still manually create some of the projections that Database Designer did not create.

To enable the Database Designer logging capability, see Enabling logging for Database Designer .

To view the logged information, see Viewing Database Designer logs .

In this section

- Enabling logging for Database Designer
- Viewing Database Designer logs
- Database Designer logs: example data

Enabling logging for Database Designer

By default, Database Designer does not log information about the projections that the Optimizer proposed and the Database Designer deploys.

To enable Database Designer logging, enter the following command:

```
=> ALTER DATABASE DEFAULT SET DBDLogInternalDesignProcess = 1;
```

To disable Database Designer logging, enter the following command:

```
=> ALTER DATABASE DEFAULT SET DBDLogInternalDesignProcess = 0;
```

See also

- Logging projection data for Database Designer
- Viewing Database Designer logs

Viewing Database Designer logs

You can find data about the projections that Database Designer considered and deployed in two Data Collector tables:

- DC_DESIGN_PROJECTION_CANDIDATES
- DC_DESIGN_QUERY_PROJECTION_CANDIDATES

DC_DESIGN_PROJECTION_CANDIDATES

The DC_DESIGN_PROJECTION_CANDIDATES table contains information about all the projections that the Optimizer proposed. This table also includes the DDL that creates them. The i s_a_winner field indicates if that projection was part of the actual deployed design. To view the DC_DESIGN_PROJECTION_CANDIDATES table, enter:

```
=> SELECT *  FROM DC_DESIGN_PROJECTION_CANDIDATES;
```

DC_DESIGN_QUERY_PROJECTION_CANDIDATES

The DC_DESIGN_QUERY_PROJECTION_CANDIDATES table lists plan features for all design queries.

Possible features are:

- FULLY DISTRIBUTED JOIN
- MERGE JOIN
- GROUPBY PIPE
- FULLY DISTRIBUTED GROUPBY
- RLE PREDICATE
- VALUE INDEX PREDICATE
- LATE MATERIALIZATION

For all design queries, the DC_DESIGN_QUERY_PROJECTION_CANDIDATES table includes the following plan feature information:

- Optimizer path cost.
- Database Designer benefits.
- Ideal plan feature and its description, which identifies how the referenced projection should be optimized.
- If the design was deployed, the actual plan feature and its description is included in the table. This information identifies how the referenced projection was actually optimized.

Because most projections have multiple optimizations, each projection usually has multiple rows.To view the DC_DESIGN_QUERY_PROJECTION_CANDIDATES table, enter:

```
=> SELECT *  FROM DC_DESIGN_QUERY_PROJECTION_CANDIDATES;
```

To see example data from these tables, see Database Designer logs: example data .

Database Designer logs: example data

In the following example, Database Designer created the logs after creating a comprehensive design for the VMart sample database. The output shows two records from the DC_DESIGN_PROJECTION_CANDIDATES table.

The first record contains information about the customer_dimension_dbd_1_sort_$customer_gender$__$annual_income$ projection. The record includes the CREATE PROJECTION statement that Database Designer used to create the projection. The is_a_winner column is t , indicating that Database Designer created this projection when it deployed the design.

The second record contains information about the product_dimension_dbd_2_sort_$product_version$__$product_key$ projection. For this projection, the is_a_winner column is f . The Optimizer recommended that Database Designer create this projection as part of the design. However, Database Designer did not create the projection when it deployed the design. The log includes the DDL for the CREATE PROJECTION statement. If you want to add the projection manually, you can use that DDL. For more information, see Creating a design manually .

```
=> SELECT * FROM dc_design_projection_candidates;
-[ RECORD 1 ]--------+-----------------------------------------------------------------
time                 | 2014-04-11 06:30:17.918764-07
node_name            | v_vmart_node0001
session_id           | localhost.localdoma-931:0x1b7
user_id              | 45035996273704962
user_name            | dbadmin
design_id            | 45035996273705182
design_table_id      | 45035996273720620
projection_id        | 45035996273726626
iteration_number     | 1
projection_name      | customer_dimension_dbd_1_sort_$customer_gender$__$annual_income$
projection_statement | CREATE PROJECTION v_dbd_sarahtest_sarahtest."customer_dimension_dbd_1_
           sort_$customer_gender$__$annual_income$"
(
customer_key ENCODING AUTO,
customer_type ENCODING AUTO,
customer_name ENCODING AUTO,
customer_gender ENCODING RLE,
title ENCODING AUTO,
household_id ENCODING AUTO,
customer_address ENCODING AUTO,
customer_city ENCODING AUTO,
customer_state ENCODING AUTO,
customer_region ENCODING AUTO,
marital_status ENCODING AUTO,
customer_age ENCODING AUTO,
number_of_children ENCODING AUTO,
annual_income ENCODING AUTO,
occupation ENCODING AUTO,
largest_bill_amount ENCODING AUTO,
store_membership_card ENCODING AUTO,
customer_since ENCODING AUTO,
deal_stage ENCODING AUTO,
deal_size ENCODING AUTO,
last_deal_update ENCODING AUTO
```

)
AS
SELECT customer_key,
customer_type,
customer_name,
customer_gender,
title,
household_id,
customer_address,
customer_city,
customer_state,
customer_region,
marital_status,
customer_age,
number_of_children,
annual_income,
occupation,
largest_bill_amount,
store_membership_card,
customer_since,
deal_stage,
deal_size,
last_deal_update
FROM public.customer_dimension
ORDER BY customer_gender,
annual_income
UNSEGMENTED ALL NODES;
is_a_winner | t
-[RECORD 2]--------+---
time | 2014-04-11 06:30:17.961324-07
node_name | v_vmart_node0001
session_id | localhost.localdoma-931:0x1b7
user_id | 45035996273704962
user_name | dbadmin
design_id | 45035996273705182
design_table_id | 45035996273720624
projection_id | 45035996273726714
iteration_number | 1
projection_name | product_dimension_dbd_2_sort_$product_version$__$product_key$
projection_statement | CREATE PROJECTION v_dbd_sarahtest_sarahtest."product_dimension_dbd_2_
sort_$product_version$__$product_key$"
(
product_key ENCODING AUTO,
product_version ENCODING RLE,
product_description ENCODING AUTO,
sku_number ENCODING AUTO,
category_description ENCODING AUTO,
department_description ENCODING AUTO,
package_type_description ENCODING AUTO,
package_size ENCODING AUTO,
fat_content ENCODING AUTO,
diet_type ENCODING AUTO,
weight ENCODING AUTO,
weight_units_of_measure ENCODING AUTO,
shelf_width ENCODING AUTO,
shelf_height ENCODING AUTO,
shelf_depth ENCODING AUTO,
product_price ENCODING AUTO,
product_cost ENCODING AUTO,
lowest_competitor_price ENCODING AUTO,
highest_competitor_price ENCODING AUTO,

```
average_competitor_price ENCODING AUTO,
discontinued_flag ENCODING AUTO
)
AS
SELECT product_key,
product_version,
product_description,
sku_number,
category_description,
department_description,
package_type_description,
package_size,
fat_content,
diet_type,
weight,
weight_units_of_measure,
shelf_width,
shelf_height,
shelf_depth,
product_price,
product_cost,
lowest_competitor_price,
highest_competitor_price,
average_competitor_price,
discontinued_flag
FROM public.product_dimension
ORDER BY product_version,
product_key
UNSEGMENTED ALL NODES;
is_a_winner          | f
.
.
.
```

The next example shows the contents of two records in the DC_DESIGN_QUERY_PROJECTION_CANDIDATES. Both of these rows apply to projection id 45035996273726626.

In the first record, the Optimizer recommends that Database Designer optimize the customer_gender column for the GROUPBY PIPE algorithm.

In the second record, the Optimizer recommends that Database Designer optimize the public.customer_dimension table for late materialization. Late materialization can improve the performance of joins that might spill to disk.

```
=> SELECT * FROM dc_design_query_projection_candidates;
-[ RECORD 1 ]----------------+-----------------------------------------------------------
time                         | 2014-04-11 06:30:17.482377-07
node_name                    | v_vmart_node0001
session_id                   | localhost.localdoma-931:0x1b7
user_id                      | 45035996273704962
user_name                    | dbadmin
design_id                    | 45035996273705182
design_query_id              | 3
iteration_number             | 1
design_table_id              | 45035996273720620
projection_id                | 45035996273726626
ideal_plan_feature           | GROUP BY PIPE
ideal_plan_feature_description | Group-by pipelined on column(s) customer_gender
dbd_benefits                 | 5
opt_path_cost                | 211
-[ RECORD 2 ]----------------+-----------------------------------------------------------
time                         | 2014-04-11 06:30:17.48276-07
node_name                    | v_vmart_node0001
session_id                   | localhost.localdoma-931:0x1b7
user_id                      | 45035996273704962
user_name                    | dbadmin
design_id                    | 45035996273705182
design_query_id              | 3
iteration_number             | 1
design_table_id              | 45035996273720620
projection_id                | 45035996273726626
ideal_plan_feature           | LATE MATERIALIZATION
ideal_plan_feature_description | Late materialization on table public.customer_dimension
dbd_benefits                 | 4
opt_path_cost                | 669
.
.
.
```

You can view the actual plan features that Database Designer implemented for the projections it created. To do so, query the V_INTERNAL.DC_DESIGN_QUERY_PROJECTIONS table:

```
=> select * from v_internal.dc_design_query_projections;
-[ RECORD 1 ]-------------------+-----------------------------------------------------------
time                          | 2014-04-11 06:31:41.19199-07
node_name                       | v_vmart_node0001
session_id                      | localhost.localdoma-931:0x1b7
user_id                         | 45035996273704962
user_name                       | dbadmin
design_id                       | 45035996273705182
design_query_id                 | 1
projection_id                   | 2
design_table_id                 | 45035996273720624
actual_plan_feature             | RLE PREDICATE
actual_plan_feature_description | RLE on predicate column(s) department_description
dbd_benefits                    | 2
opt_path_cost                   | 141
-[ RECORD 2 ]-------------------+-----------------------------------------------------------
time                          | 2014-04-11 06:31:41.192292-07
node_name                       | v_vmart_node0001
session_id                      | localhost.localdoma-931:0x1b7
user_id                         | 45035996273704962
user_name                       | dbadmin
design_id                       | 45035996273705182
design_query_id                 | 1
projection_id                   | 2
design_table_id                 | 45035996273720624
actual_plan_feature             | GROUP BY PIPE
actual_plan_feature_description | Group-by pipelined on column(s) fat_content
dbd_benefits                    | 5
opt_path_cost                   | 155
```

General design settings

Before you run Database Designer, you must provide specific information on the design to create.

Design name

All designs that you create with Database Designer must have unique names that conform to the conventions described in Identifiers , and are no more than 32 characters long (16 characters if you use Database Designer in Administration Tools or Management Console).

The design name is incorporated into the names of files that Database Designer generates, such as its deployment script. This can help you differentiate files that are associated with different designs.

Design type

Database Designer can create two distinct design types: comprehensive or incremental.

Comprehensive design

A comprehensive design creates an initial or replacement design for all the tables in the specified schemas. Create a comprehensive design when you are creating a new database.

To help Database Designer create an efficient design, load representative data into the tables before you begin the design process. When you load data into a table, Vertica creates an unoptimized superprojection so that Database Designer has projections to optimize. If a table has no data, Database Designer cannot optimize it.

Optionally, supply Database Designer with representative queries that you plan to use so Database Designer can optimize the design for them. If you do not supply any queries, Database Designer creates a generic optimization of the superprojections that minimizes storage, with no query-specific projections.

During a comprehensive design, Database Designer creates deployment scripts that:

- Create projections to optimize query performance.
- Create replacement buddy projections when Database Designer changes the encoding of existing projections that it decides to keep.

Incremental design

After you create and deploy a comprehensive database design, your database is likely to change over time in various ways. Consider using Database Designer periodically to create incremental designs that address these changes. Changes that warrant an incremental design can include:

- Significant data additions or updates
- New or modified queries that you run regularly
- Performance issues with one or more queries
- Schema changes

Optimization objective

Database Designer can optimize the design for one of three objectives:

- **Load** : Designs that are optimized for loads minimize database size, potentially at the expense of query performance.
- **Query** : Designs that are optimized for query performance. These designs typically favor fast query execution over load optimization, and thus result in a larger storage footprint.
- **Balanced** : Designs that are balanced between database size and query performance.

A fully optimized query has an optimization ratio of 0.99. Optimization ratio is the ratio of a query's benefits achieved in the design produced by the Database Designer to that achieved in the ideal plan. The optimization ratio is set in the OptRatio parameter in designer.log .

Design tables

Database Designer needs one or more tables with a moderate amount of sample data—approximately 10 GB—to create optimal designs. Design tables with large amounts of data adversely affect Database Designer performance. Design tables with too little data prevent Database Designer from creating an optimized design. If a design table has no data, Database Designer ignores it.

> **Note**
> If you drop a table after adding it to the design, Database Designer cannot build or deploy the design.

Design queries

A database design that is optimized for query performance requires a set of representative queries, or *design queries* . Design queries are required for incremental designs, and optional for comprehensive designs. You list design queries in a SQL file that you supply as input to Database Designer. Database Designer checks the validity of the queries when you add them to your design, and again when it builds the design. If a query is invalid, Database Designer ignores it.

If you use Management Console to create a database design, you can submit queries either from an input file or from the system table QUERY_REQUESTS . For details, see Creating a design manually .

The maximum number of design queries depends on the design type: ≤200 queries for a comprehensive design, ≤100 queries for an incremental design. Optionally, you can assign weights to the design queries that signify their relative importance. Database Designer uses those weights to prioritize the queries in its design.

Segmented and unsegmented projections

When creating a comprehensive design, Database Designer creates projections based on data statistics and queries. It also reviews the submitted design tables to decide whether projections should be segmented (distributed across the cluster nodes) or unsegmente d (replicated on all cluster nodes).

By default, Database Designer recommends only segmented projections. You can enable Database Designer to recommend unsegmented projections . In this case, Database Designer recommends segmented superprojections for large tables when deploying to multi-node clusters, and unsegmented superprojections for smaller tables.

Database Designer uses the following algorithm to determine whether to recommend unsegmented projections. Assuming that *largest-row-count* equals the number of rows in the design table with the largest number of rows, Database Designer recommends unsegmented projections if any of the following conditions is true:

- *largest-row-count* < 1,000,000 AND *number-table-rows* ≤ *10%-largest-row-count*
- *largest-row-count* ≥ 10,000,000 AND *number-table-rows* ≤ *1%-largest-row-count*
- 1,000,000 ≤ *largest-row-count* < 10,000,000 AND *number-table-rows* ≤ 100,000

Database Designer does not segment projections on:

- Single-node clusters
- LONG VARCHAR and LONG VARBINARY columns

For more information, see High availability with projections .

Statistics analysis

By default, Database Designer analyzes statistics for design tables when they are added to the design. Accurate statistics help Database Designer optimize compression and query performance.

Analyzing statistics takes time and resources. If you are certain that design table statistics are up to date, you can specify to skip this step and avoid the overhead otherwise incurred.

For more information, see Collecting Statistics .

Building a design

After you have created design tables and loaded data into them, and then specified the parameters you want Database Designer to use when creating the physical schema, direct Database Designer to create the scripts necessary to build the design.

> **Note**
> You cannot stop a running database if Database Designer is building a database design.

When you build a database design, Vertica generates two scripts:

- **Deployment script** : *design-name* _deploy.sql—Contains the SQL statements that create projections for the design you are deploying, deploy the design, and drop unused projections. When the deployment script runs, it creates the optimized design. For details about how to run this script and deploy the design, see Deploying a Design .
- **Design script** : *design-name* _design.sql—Contains the CREATE PROJECTION statements that Database Designeruses to create the design. Review this script to make sure you are happy with the design.
  The design script is a subset of the deployment script. It serves as a backup of the DDL for the projections that the deployment script creates.

When you create a design using Management Console:

- If you submit a large number of queries to your design and build it right immediately, a timing issue could cause the queries not to load before deployment starts. If this occurs, you might see one of the following errors:
  - No queries to optimize for
  - No tables to design projections for

  To accommodate this timing issue, you may need to reset the design, check the **Queries** tab to make sure the queries have been loaded, and then rebuild the design. Detailed instructions are in:
  - Using the wizard to create a design
  - Creating a design manually
- The scripts are deleted when deployment completes. To save a copy of the deployment script after the design is built but before the deployment completes, go to the **Output** window and copy and paste the SQL statements to a file.

Resetting a design

You must reset a design when:

- You build a design and the output scripts described in Building a Design are not created.
- You build a design but Database Designer cannot complete the design because the queries it expects are not loaded.

Resetting a design discards all the run-specific information of the previous Database Designer build, but retains its configuration (design type, optimization objectives, K-safety, etc.) and tables and queries.

After you reset a design, review the design to see what changes you need to make. For example, you can fix errors, change parameters, or check for and add additional tables or queries. Then you can rebuild the design.

You can only reset a design in Management Console or by using the DESIGNER_RESET_DESIGN function.

Deploying a design

After running Database Designer to generate a deployment script, Vertica recommends that you test your design on a non-production server before you deploy it to your production server.

Both the design and deployment processes run in the background. This is useful if you have a large design that you want to run overnight. Because an active SSH session is not required, the design/deploy operations continue to run uninterrupted, even if the session is terminated.

> **Note**
> You cannot stop a running database if Database Designer is building or deploying a database design.

Database Designer runs as a background process. Multiple users can run Database Designer concurrently without interfering with each other or using up all the cluster resources. However, if multiple users are deploying a design on the same tables at the same time, Database Designer may not be able to complete the deployment. To avoid problems, consider the following:

- Schedule potentially conflicting Database Designer processes to run sequentially overnight so that there are no concurrency problems.
- Avoid scheduling Database Designer runs on the same set of tables at the same time.

There are two ways to deploy your design:

- Deploying Designs Using Database Designer
- Deploying Designs Manually

In this section

- Deploying designs using Database Designer
- Deploying designs manually

Deploying designs using Database Designer

OpenText recommends that you run Database Designer and deploy optimized projections right after loading your tables with sample data because Database Designer provides projections optimized for the current state of your database.

If you choose to allow Database Designer to automatically deploy your script during a comprehensive design and are running Administrative Tools, Database Designer creates a backup script of your database's current design. This script helps you re-create the design of projections that may have been dropped by the new design. The backup script is located in the output directory you specified during the design process.

If you choose not to have Database Designer automatically run the deployment script (for example, if you want to maintain projections from a pre-existing deployment), you can manually run the deployment script later. See Deploying designs manually .

To deploy a design while running Database Designer, do one of the following:

- In Management Console, select the design and click **Deploy Design** .
- In the Administration Tools, select **Deploy design** in the **Design Options** window.

If you are running Database Designer programmatically, use DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY and set the deploy parameter to 'true'.

Once you have deployed your design, query the DEPLOY_STATUS system table to see the steps that the deployment took:

```
vmartdb=> SELECT * FROM V_MONITOR.DEPLOY_STATUS;
```

Deploying designs manually

If you choose not to have Database Designer deploy your design at design time, you can deploy the design later using the deployment script:

1. Make sure that the target database contains the same tables and projections as the database where you ran Database Designer. The database should also contain sample data.
2. To deploy the projections to a test or production environment, execute the deployment script in vsql with the meta-command \i as follows, where *design-name* is the name of the database design:
   ```
   => \i design-name_deploy.sql
   ```
3. For a K-safe database, call Vertica meta-function GET_PROJECTIONS on tables of the new projections. Check the output to verify that all projections have enough buddies to be identified as safe.
4. If you create projections for tables that already contains data, call REFRESH or START_REFRESH to update new projections. Otherwise, these projections are not available for query processing.
5. Call MAKE_AHM_NOW to set the Ancient History Mark (AHM) to the most recent epoch.
6. Call DROP PROJECTION on projections that are no longer needed, and would otherwise waste disk space and reduce load speed.
7. Call ANALYZE_STATISTICS on all database projections:
   ```
   => SELECT ANALYZE_STATISTICS ('');
   ```
   This function collects and aggregates data samples and storage information from all nodes on which a projection is stored, and then writes statistics into the catalog.

How to create a design

There are three ways to create a design using Database Designer:

- From Management Console, open a database and select the **Design** page at the bottom of the window.
  For details about using Management Console to create a design, see Creating a database design in Management Console
- Programmatically, using the techniques described in About Running Database Designer Programmatically . To run Database Designer programmatically, you must be a DBADMIN or have been granted the DBDUSER role and enabled that role.
- From the Administration Tools menu, by selecting **Configuration Menu > Run Database Designer** . You must be a DBADMIN user to run Database Designer from the Administration Tools.
  For details about using Administration Tools to create a design, see Using administration tools to create a design .

The following table shows what Database Designer capabilities are available in each tool:

Database Designer Capability	**Management Console**	**Running Database Designer Programmatically**	**Administrative Tools**
Create design	Yes	Yes	Yes
Design name length (# of characters)	16	32	16
Build design (create design and deployment scripts)	Yes	Yes	Yes
Create backup script			Yes
Set design type (comprehensive or incremental)	Yes	Yes	Yes
Set optimization objective	Yes	Yes	Yes
Add design tables	Yes	Yes	Yes
Add design queries file	Yes	Yes	Yes
Add single design query		Yes	
Use query repository	Yes	Yes	
Set K-safety	Yes	Yes	Yes
Analyze statistics	Yes	Yes	Yes
Require all unsegmented projections	Yes	Yes	
View event history	Yes	Yes	
Set correlation analysis mode (Default = 0)		Yes	

In this section

- Using administration tools to create a design

Using administration tools to create a design

To use the Administration Tools interface to create an optimized design for your database, you must be a DBADMIN user. Follow these steps:

1. Log in as the dbadmin user and start Administration Tools.
2. From the main menu, start the database for which you want to create a design. The database must be running before you can create a design for it.
3. On the main menu, select **Configuration Menu** and click **OK** .
4. On the Configuration Menu, select **Run Database Designer** and click **OK** .
5. On the **Select a database to design** window, enter the name of the database for which you are creating a design and click **OK** .

6. On the **Enter the directory for Database Designer output** window, enter the full path to the directory to contain the design script, deployment script, backup script, and log files, and click **OK** .
   For information about the scripts, see Building a design .
7. On the **Database Designer** window, enter a name for the design and click **OK** .
8. On the **Design Type** window, choose which type of design to create and click **OK** .
   For details, see Design Types .
9. The **Select schema(s) to add to query search path** window lists all the schemas in the database that you selected. Select the schemas that contain representative data that you want Database Designer to consider when creating the design and click **OK** .
   For details about choosing schema and tables to submit to Database Designer, see Design Tables with Sample Data .
10. On the **Optimization Objectives** window, select the objective you want for the database optimization:
    - **Optimize with Queries**
    - **Update statistics**
    - **Deploy design**
11. The final window summarizes the choices you have made and offers you two choices:
    - **Proceed** with building the design, and deploying it if you specified to deploy it immediately. If you did not specify to deploy, you can review the design and deployment scripts and deploy them manually, as described in Deploying designs manually .
    - **Cancel** the design and go back to change some of the parameters as needed.
12. Creating a design can take a long time.To cancel a running design from the Administration Tools window, enter **Ctrl+C** .

To create a design for the VMart example database, see Using Database Designer to create a comprehensive design in Getting Started.

Running Database Designer programmatically

Vertica provides a set of meta-functions that enable programmatic access to Database Designer functionality. Run Database Designer programmatically to perform the following tasks:

- Optimize performance on tables that you own.
- Create or update a design without requiring superuser or DBADMIN intervention.
- Add individual queries and tables, or add data to your design, and then rerun Database Designer to update the design based on this new information.
- Customize the design.
- Use recently executed queries to set up your database to run Database Designer automatically on a regular basis.
- Assign each design query a *query weight* that indicates the importance of that query in creating the design. Assign a higher weight to queries that you run frequently so that Database Designer prioritizes those queries in creating the design.

For more details about Database Designer functions, see Database Designer function categories .

In this section

- Database Designer function categories
- Workflow for running Database Designer programmatically
- Privileges for running Database Designer functions
- Resource pool for Database Designer users

Database Designer function categories

Database Designer functions perform the following operations, generally performed in the following order:

1. Create a design .
2. Set design properties .
3. Populate a design .
4. Create design and deployment scripts .
5. Get design data .
6. Clean up .

Important

You can also use meta-function DESIGNER_SINGLE_RUN , which encapsulates all of these steps with a single call. The meta-function iterates over all queries within a specified timespan, and returns with a design ready for deployment.

For detailed information, see Workflow for running Database Designer programmatically . For information on required privileges, see Privileges for running Database Designer functions

> **Caution**
> Before running Database Designer functions on an existing schema, back up the current design by calling EXPORT_CATALOG .

Create a design

DESIGNER_CREATE_DESIGN directs Database Designer to create a design.

Set design properties

The following functions let you specify design properties:

- DESIGNER_SET_DESIGN_TYPE : Specifies whether the design is comprehensive or incremental.
- DESIGNER_DESIGN_PROJECTION_ENCODINGS : Analyzes encoding in the specified projections and creates a script that implements encoding recommendations.
- DESIGNER_SET_DESIGN_KSAFETY : Sets the K-safety value for a comprehensive design.
- DESIGNER_SET_OPTIMIZATION_OBJECTIVE : Specifies whether the design optimizes for query or load performance.
- DESIGNER_SET_PROPOSE_UNSEGMENTED_PROJECTIONS : Enables inclusion of unsegmented projections in the design.

Populate a design

The following functions let you add tables and queries to your Database Designer design:

- DESIGNER_ADD_DESIGN_TABLES : Adds the specified tables to a design.
- DESIGNER_ADD_DESIGN_QUERY , DESIGNER_ADD_DESIGN_QUERIES , DESIGNER_ADD_DESIGN_QUERIES_FROM_RESULTS : Adds queries to the design and weights them.

Create design and deployment scripts

The following functions populate the Database Designer workspace and create design and deployment scripts. You can also analyze statistics, deploy the design automatically, and drop the workspace after the deployment:

- DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY : Populates the design and creates design and deployment scripts.
- DESIGNER_WAIT_FOR_DESIGN : Waits for a currently running design to complete.

Reset a design

DESIGNER_RESET_DESIGN discards all the run-specific information of the previous Database Designer build or deployment of the specified design but retains its configuration.

Get design data

The following functions display information about projections and scripts that the Database Designer created:

- DESIGNER_OUTPUT_ALL_DESIGN_PROJECTIONS : Sends to standard output DDL statements that define design projections.
- DESIGNER_OUTPUT_DEPLOYMENT_SCRIPT : Sends to standard output a design's deployment script.

Cleanup

The following functions cancel any running Database Designer operation or drop a Database Designer design and all its contents:

- DESIGNER_CANCEL_POPULATE_DESIGN : Cancels population or deployment operation for the specified design if it is currently running.
- DESIGNER_DROP_DESIGN : Removes the schema associated with the specified design and all its contents.
- DESIGNER_DROP_ALL_DESIGNS : Removes all Database Designer-related schemas associated with the current user.

Workflow for running Database Designer programmatically

The following example shows the steps you take to create a design by running Database Designer programmatically.

Important
Before running Database Designer functions on an existing schema, back up the current design by calling function EXPORT_CATALOG .

Before you run this example, you should have the DBDUSER role, and you should have enabled that role using the SET ROLE DBDUSER command:

1. Create a table in the public schema:

```
=> CREATE TABLE T(
  x INT,
  y INT,
  z INT,
  u INT,
  v INT,
  w INT PRIMARY KEY
  );
```

2. Add data to the table:

```
\! perl -e 'for ($i=0; $i<100000; ++$i)   {printf("%d, %d, %d, %d, %d, %d\n", $i/10000, $i/100, $i/10, $i/2, $i, $i);}'
  | vsql -c "COPY T FROM STDIN DELIMITER ',' DIRECT;"
```

3. Create a second table in the public schema:

```
=> CREATE TABLE T2(
  x INT,
  y INT,
  z INT,
  u INT,
  v INT,
  w INT PRIMARY KEY
  );
```

4. Copy the data from table T1 to table T2 and commit the changes:

```
=> INSERT /*+DIRECT*/ INTO T2 SELECT * FROM T;
=> COMMIT;
```

5. Create a new design:

```
=> SELECT DESIGNER_CREATE_DESIGN('my_design');
```

This command adds information to the DESIGNS system table in the V_MONITOR schema.

6. Add tables from the public schema to the design :

```
=> SELECT DESIGNER_ADD_DESIGN_TABLES('my_design', 'public.t');
=> SELECT DESIGNER_ADD_DESIGN_TABLES('my_design', 'public.t2');
```

These commands add information to the DESIGN_TABLES system table.

7. Create a file named queries.txt in /tmp/examples , or another directory where you have READ and WRITE privileges. Add the following two queries in that file and save it. Database Designer uses these queries to create the design:

```
SELECT DISTINCT T2.u FROM T JOIN T2 ON T.z=T2.z-1 WHERE T2.u > 0;
SELECT DISTINCT w FROM T;
```

8. Add the queries file to the design and display the results—the numbers of accepted queries, non-design queries, and unoptimizable queries:

```
=> SELECT DESIGNER_ADD_DESIGN_QUERIES
    ('my_design',
    '/tmp/examples/queries.txt',
    'true'
    );
```

The results show that both queries were accepted:

```
Number of accepted queries                  =2
Number of queries referencing non-design tables =0
Number of unsupported queries                =0
Number of illegal queries                   =0
```

The DESIGNER_ADD_DESIGN_QUERIES function populates the DESIGN_QUERIES system table.

9. Set the design type to **comprehensive** . (This is the default.) A comprehensive design creates an initial or replacement design for all the design tables:

```
=> SELECT DESIGNER_SET_DESIGN_TYPE('my_design', 'comprehensive');
```

10. Set the optimization objective to **query** . This setting creates a design that focuses on faster query performance, which might recommend additional projections. These projections could result in a larger database storage footprint:

```
=> SELECT DESIGNER_SET_OPTIMIZATION_OBJECTIVE('my_design', 'query');
```

11. Create the design and save the design and deployment scripts in /tmp/examples , or another directory where you have READ and WRITE privileges. The following command:
    - Analyzes statistics
    - Doesn't deploy the design.
    - Doesn't drop the design after deployment.
    - Stops if it encounters an error.

```
=> SELECT DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY
  ('my_design',
   '/tmp/examples/my_design_projections.sql',
   '/tmp/examples/my_design_deploy.sql',
   'True',
   'False',
   'False',
   'False'
   );
```

    This command adds information to the following system tables:
    - DEPLOYMENT_PROJECTION_STATEMENTS
    - DEPLOYMENT_PROJECTIONS
    - OUTPUT_DEPLOYMENT_STATUS
12. Examine the status of the Database Designer run to see what projections Database Designer recommends. In the deployment_projection_name column:
    - rep indicates a replicated projection
    - super indicates a superprojection

      The deployment_status column is pending because the design has not yet been deployed.

      For this example, Database Designer recommends four projections:

```
=> \x
Expanded display is on.
=> SELECT * FROM OUTPUT_DEPLOYMENT_STATUS;
-[ RECORD 1 ]---------+---------------------------
deployment_id             | 45035996273795970
deployment_projection_id   | 1
deployment_projection_name | T_DBD_1_rep_my_design
deployment_status          | pending
error_message              | N/A
-[ RECORD 2 ]---------+---------------------------
deployment_id             | 45035996273795970
deployment_projection_id   | 2
deployment_projection_name | T2_DBD_2_rep_my_design
deployment_status          | pending
error_message              | N/A
-[ RECORD 3 ]---------+---------------------------
deployment_id             | 45035996273795970
deployment_projection_id   | 3
deployment_projection_name | T_super
deployment_status          | pending
error_message              | N/A
-[ RECORD 4 ]---------+---------------------------
deployment_id             | 45035996273795970
deployment_projection_id   | 4
deployment_projection_name | T2_super
deployment_status          | pending
error_message              | N/A
```

13. View the script /tmp/examples/my_design_deploy.sql to see how these projections are created when you run the deployment script. In this example, the script also assigns the encoding schemes RLE and COMMONDELTA_COMP to columns where appropriate.
14. Deploy the design from the directory where you saved it:

```
=> \i /tmp/examples/my_design_deploy.sql
```

15. Now that the design is deployed, delete the design:

```
=> SELECT DESIGNER_DROP_DESIGN('my_design');
```

Privileges for running Database Designer functions

Non-DBADMIN users with the DBDUSER role can run Database Designer f unctions . Two steps are required to enable users to run these functions:

1. A DBADMIN or superuser grants the user the DBDUSER role:
   ```
   => GRANT DBDUSER TO username;
   ```
   This role persists until the DBADMIN revokes it.
2. Before the DBDUSER can run Database Designer functions, one of the following must occur:
   - The user enables the DBDUSER role:
     ```
     => SET ROLE DBDUSER;
     ```
   - The superuser sets the user's default role to DBDUSER:
     ```
     => ALTER USER username DEFAULT ROLE DBDUSER;
     ```

General DBDUSER limitations

As a DBDUSER, the following restrictions apply:

- You can set a design's K-safety to a value less than or equal to system K-safety. You cannot change system K-safety.
- You cannot explicitly change the ancient history mark (AHM), even during design deployment.

Design dependencies and privileges

Individual design tasks are likely to have dependencies that require specific privileges:

Task	Required privileges
Add tables to a design	• USAGE privilege on the design table schema • OWNER privilege on the design table
Add a single design query to the design	• Privilege to execute the design query
Add a query file to the design	• Read privilege on the storage location that contains the query file • Privilege to execute all the queries in the file
Add queries from the result of a user query to the design	• Privilege to execute the user query • Privilege to execute each design query retrieved from the results of the user query
Create design and deployment scripts	• WRITE privilege on the storage location of the design script • WRITE privilege on the storage location of the deployment script

Resource pool for Database Designer users

When you grant a user the DBDUSER role, be sure to associate a resource pool with that user to manage resources during Database Designer runs. This allows multiple users to run Database Designer concurrently without interfering with each other or using up all cluster resources.

> **Note**
> When a user runs Database Designer, execution is mostly contained in the user's resource pool. However, Vertica might also use other system resource pools to perform less-intensive tasks.

Creating custom designs

Vertica strongly recommends that you use the physical schema design produced by Database Designer , which provides K-safety , excellent query performance, and efficient use of storage space. If any queries run less as efficiently than you expect, consider using the Database Designer incremental design process to optimize the database design for the query.

If the projections created by Database Designer still do not meet your needs, you can write custom projections, from scratch or based on projection designs created by Database Designer.

If you are unfamiliar with writing custom projections, start by modifying an existing design generated by Database Designer.

In this section

- Custom design process
- Planning your design
- Design fundamentals

Custom design process

To create a custom design or customize an existing one:

1. Plan the new design or modifications to an existing one. See Planning your design .
2. Create or modify projections. See Design fundamentals and CREATE PROJECTION for more detail.
3. Deploy projections to a test environment. See Writing and deploying custom projections .
4. Test and modify projections as needed.
5. After you finalize the design, deploy projections to the production environment.

Planning your design

The syntax for creating a design is easy for anyone who is familiar with SQL. As with any successful project, however, a successful design requires some initial planning. Before you create your first design:

- Become familiar with standard design requirements and plan your design to include them. See Design requirements .
- Determine how many projections you need to include in the design. See Determining the number of projections to use .
- Determine the type of compression and encoding to use for columns. See Architecture .
- Determine whether or not you want the database to be K-safe. Vertica recommends that all production databases have a minimum K-safety of one (K=1). Valid K-safety values are 0, 1, and 2. See Designing for K-safety .

In this section

- Design requirements
- Determining the number of projections to use
- Designing for K-safety
- Designing for segmentation

Design requirements

A physical schema design is a script that contains CREATE PROJECTION statements. These statements determine which columns are included in projections and how they are optimized.

If you use Database Designer as a starting point, it automatically creates designs that meet all fundamental design requirements. If you intend to create or modify designs manually, be aware that all designs must meet the following requirements:

- Every design must create at least one superprojection for every table in the database that is used by the client application. These projections provide complete coverage that enables users to perform ad-hoc queries as needed. They can contain joins and they are usually configured to maximize performance through sort order, compression, and encoding.
- Query-specific projections are optional. If you are satisfied with the performance provided through superprojections, you do not need to create additional projections. However, you can maximize performance by tuning for specific query work loads.
- Vertica recommends that all production databases have a minimum K-safety of one (K=1) to support high availability and recovery. (K-safety can be set to 0, 1, or 2.) See High availability with projections and Designing for K-safety .
- Vertica recommends that if you have more than 20 nodes, but small tables, do not create replicated projections. If you create replicated projections, the catalog becomes very large and performance may degrade. Instead, consider segmenting those projections.

Determining the number of projections to use

In many cases, a design that consists of a set of superprojections (and their buddies) provides satisfactory performance through compression and encoding. This is especially true if the sort orders for the projections have been used to maximize performance for one or more query predicates (WHERE clauses).

However, you might want to add additional query-specific projections to increase the performance of queries that run slowly, are used frequently, or are run as part of business-critical reporting. The number of additional projections (and their buddies) that you create should be determined by:

- Your organization's needs
- The amount of disk space you have available on each node in the cluster
- The amount of time available for loading data into the database

As the number of projections that are tuned for specific queries increases, the performance of these queries improves. However, the amount of disk space used and the amount of time required to load data increases as well. Therefore, you should create and test designs to determine the optimum number of projections for your database configuration. On average, organizations that choose to implement query-specific projections achieve optimal performance through the addition of a few query-specific projections.

Designing for K-safety

Vertica recommends that all production databases have a minimum K-safety of one (K=1). Valid K-safety values for production databases are 1 and 2. Non-production databases do not have to be K-safe and can be set to 0.

A K-safe database must have at least three nodes, as shown in the following table:

K-safety level	Number of required nodes
1	3+
2	5+

> **Note**
> Vertica only supports K-safety levels 1 and 2.

You can set K-safety to 1 or 2 only when the physical schema design meets certain redundancy requirements. See Requirements for a K-safe physical schema design .

Using Database Designer

To create designs that are K-safe, Vertica recommends that you use the Database Designer . When creating projections with Database Designer, projection definitions that meet K-safe design requirements are recommended and marked with a K-safety level. Database Designer creates a script that uses the MARK_DESIGN_KSAFE function to set the K-safety of the physical schema to 1. For example:

```
=> \i VMart_Schema_design_opt_1.sql
CREATE PROJECTION
CREATE PROJECTION
mark_design_ksafe
----------------------
Marked design 1-safe
(1 row)
```

By default, Vertica creates K-safe superprojections when database K-safety is greater than 0.

Monitoring K-safety

Monitoring tables can be accessed programmatically to enable external actions, such as alerts. You monitor the K-safety level by querying the SYSTEM table for settings in columns DESIGNED_FAULT_TOLERANCE and CURRENT_FAULT_TOLERANCE .

Loss of K-safety

When K nodes in your cluster fail, your database continues to run, although performance is affected. Further node failures could potentially cause the database to shut down if the failed node's data is not available from another functioning node in the cluster.

See also

K-safety in an Enterprise Mode database

In this section

- Requirements for a K-safe physical schema design
- Requirements for a physical schema design with no K-safety
- Designing segmented projections for K-safety
- Designing unsegmented projections for K-Safety

Requirements for a K-safe physical schema design

Database Designer automatically generates designs with a K-safety of 1 for clusters that contain at least three nodes. (If your cluster has one or two nodes, it generates designs with a K-safety of 0. You can modify a design created for a three-node (or greater) cluster, and the K-safe requirements are already set.

If you create custom projections, your physical schema design must meet the following requirements to be able to successfully recover the database in the event of a failure:

- Segmented projections must be segmented across all nodes. Refer to Designing for segmentation and Designing segmented projections for K-safety .
- Replicated projections must be replicated on all nodes. See Designing unsegmented projections for K-Safety .
- Segmented projections must have K+1 buddy projections—projections with identical columns and segmentation criteria, where corresponding segments are placed on different nodes.

You can use the MARK_DESIGN_KSAFE function to find out whether your schema design meets requirements for K-safety.

Requirements for a physical schema design with no K-safety

If you use Database Designer to generate an comprehensive design that you can modify and you do not want the design to be K-safe, set K-safety level to 0 (zero).

If you want to start from scratch, do the following to establish minimal projection requirements for a functioning database with no K-safety (K=0):

1. Define at least one superprojection for each table in the logical schema .
2. Replicate (define an exact copy of) each dimension table superprojection on each node .

Designing segmented projections for K-safety

Projections must comply with database K-safety requirements. In general, you must create buddy projections for each segmented projection, where the number of buddy projections is K+1. Thus, if system K-safety is set to 1, each projection segment must be duplicated by one buddy; if K-safety is set to 2, each segment must be duplicated by two buddies.

Automatic creation of buddy projections

You can use CREATE PROJECTION so it automatically creates the number of buddy projections required to satisfy K-safety, by including SEGMENTED BY ... ALL NODES . If CREATE PROJECTION specifies K-safety (KSAFE= *n*) , Vertica uses that setting; if the statement omits KSAFE , Vertica uses system K-safety.

In the following example, CREATE PROJECTION creates segmented projection ttt_p1 for table ttt . Because system K-safety is set to 1, Vertica requires a buddy projection for each segmented projection. The CREATE PROJECTION statement omits KSAFE , so Vertica uses system K-safety and creates two buddy projections: ttt_p1_b0 and ttt_p1_b1 :

```
=> SELECT mark_design_ksafe(1);

  mark_design_ksafe
----------------------
 Marked design 1-safe
(1 row)

=> CREATE TABLE ttt (a int, b int);
WARNING 6978:  Table "ttt" will include privileges from schema "public"
CREATE TABLE

=> CREATE PROJECTION ttt_p1 as SELECT * FROM ttt SEGMENTED BY HASH(a) ALL NODES;
CREATE PROJECTION

=> SELECT projection_name from projections WHERE anchor_table_name='ttt';
 projection_name
-----------------
 ttt_p1_b0
 ttt_p1_b1
(2 rows)
```

Vertica automatically names buddy projections by appending the suffix _b *n* to the projection base name—for example ttt_p1_b0 .

Manual creation of buddy projections

If you create a projection on a single node, and system K-safety is greater than 0, you must manually create the number of buddies required for K-safety. For example, you can create projection xxx_p1 for table xxx on a single node, as follows:

```
=> CREATE TABLE xxx (a int, b int);
WARNING 6978:  Table "xxx" will include privileges from schema "public"
CREATE TABLE

=> CREATE PROJECTION xxx_p1 AS SELECT * FROM xxx SEGMENTED BY HASH(a) NODES v_vmart_node0001;
CREATE PROJECTION
```

Because K-safety is set to 1, a single instance of this projection is not K-safe. Attempts to insert data into its anchor table xxx return with an error like this:

```
=> INSERT INTO xxx VALUES (1, 2);
ERROR 3586:  Insufficient projections to answer query
DETAIL:  No projections that satisfy K-safety found for table xxx
HINT:  Define buddy projections for table xxx
```

In order to comply with K-safety, you must create a buddy projection for projection xxx_p1 . For example:

```
=> CREATE PROJECTION xxx_p1_buddy AS SELECT * FROM xxx SEGMENTED BY HASH(a) NODES v_vmart_node0002;
CREATE PROJECTION
```

Table xxx now complies with K-safety and accepts DML statements such as INSERT :

```
VMart=> INSERT INTO xxx VALUES (1, 2);
 OUTPUT
--------
      1
(1 row)
```

See also

For general information about segmented projections and buddies, see Segmented projections . For information about designing for K-safety, see Designing for K-safety and Designing for segmentation .

Designing unsegmented projections for K-Safety

In many cases, dimension tables are relatively small, so you do not need to segment them. Accordingly, you should design a K-safe database so projections for its dimension tables are replicated without segmentation on all cluster nodes. You create these projections with a CREATE PROJECTION statement that includes the keywords UNSEGMENTED ALL NODES . These keywords specify to create identical instances of the projection on all cluster nodes.

The following example shows how to create an unsegmented projection for the table store.store_dimension :

```
=> CREATE PROJECTION store.store_dimension_proj (storekey, name, city, state)
           AS SELECT store_key, store_name, store_city, store_state
           FROM store.store_dimension
           UNSEGMENTED ALL NODES;
CREATE PROJECTION
```

Vertica uses the same name to identify all instances of the unsegmented projection—in this example, store.store_dimension_proj . The keyword ALL NODES specifies to replicate the projection on all nodes:

```
=> \dj store.store_dimension_proj
                    List of projections
 Schema |        Name         | Owner |       Node       | Comment
--------+---------------------+-------+------------------+---------
 store  | store_dimension_proj | dbadmin | v_vmart_node0001 |
 store  | store_dimension_proj | dbadmin | v_vmart_node0002 |
 store  | store_dimension_proj | dbadmin | v_vmart_node0003 |
(3 rows)
```

For more information about projection name conventions, see Projection naming .

Designing for segmentation

You segment projections using hash segmentation. Hash segmentation allows you to segment a projection based on a built-in hash function that provides even distribution of data across multiple nodes, resulting in optimal query execution. In a projection, the data to be hashed consists of one or more column values, each having a large number of unique values and an acceptable amount of skew in the value distribution. Primary key columns that meet the criteria could be an excellent choice for hash segmentation.

> **Note**
> For detailed information about using hash segmentation in a projection, see CREATE PROJECTION in the SQL Reference Manual.

When segmenting projections, determine which columns to use to segment the projection. Choose one or more columns that have a large number of unique data values and acceptable skew in their data distribution. Primary key columns are an excellent choice for hash segmentation. The columns must be unique across all the tables being used in a query.

Design fundamentals

Although you can write custom projections from scratch, Vertica recommends that you use Database Designer to create a design to use as a starting point. This ensures that you have projections that meet basic requirements.

In this section

- Writing and deploying custom projections
- Designing superprojections
- Sort order benefits
- Choosing sort order: best practices
- Prioritizing column access speed

Writing and deploying custom projections

Before you write custom projections, review the topics in Planning your design carefully. Failure to follow these considerations can result in non-functional projections.

To manually modify or create a projection:

1. Write a script with CREATE PROJECTION statements to create the desired projections.
2. Run the script in vsql with the meta-command \i .

> **Note**
> You must have a database loaded with a logical schema.

3. For a K-safe database, call Vertica meta-function GET_PROJECTIONS on tables of the new projections. Check the output to verify that all projections have enough buddies to be identified as safe.
4. If you create projections for tables that already contains data, call REFRESH or START_REFRESH to update new projections. Otherwise, these projections are not available for query processing.
5. Call MAKE_AHM_NOW to set the Ancient History Mark (AHM) to the most recent epoch.
6. Call DROP PROJECTION on projections that are no longer needed, and would otherwise waste disk space and reduce load speed.
7. Call ANALYZE_STATISTICS on all database projections:

```
=> SELECT ANALYZE_STATISTICS ('');
```

   This function collects and aggregates data samples and storage information from all nodes on which a projection is stored, and then writes statistics into the catalog.

Designing superprojections

Superprojections have the following requirements:

- They must contain every column within the table.
- For a K-safe design, superprojections must either be replicated on all nodes within the database cluster (for dimension tables) or paired with buddies and segmented across all nodes (for very large tables and medium large tables). See Projections and High availability with projections for an overview of projections and how they are stored. See Designing for K-safety for design specifics.

To provide maximum usability, superprojections need to minimize storage requirements while maximizing query performance. To achieve this, the sort order for columns in superprojections is based on storage requirements and commonly used queries.

Sort order benefits

Column sort order is an important factor in minimizing storage requirements, and maximizing query performance.

Minimize storage requirements

Minimizing storage saves on physical resources and increases performance by reducing disk I/O. You can minimize projection storage by prioritizing low-cardinality columns in its sort order. This reduces the number of rows Vertica stores and accesses to retrieve query results.

After identifying projection sort columns, analyze their data and choose the most effective encoding method. The Vertica optimizer gives preference to columns with run-length encoding (RLE), so be sure to use it whenever appropriate. Run-length encoding replaces sequences (runs) of identical values with a single pair that contains the value and number of occurrences. Therefore, it is especially appropriate to use it for low-cardinality columns whose run length is large.

Maximize query performance

You can facilitate query performance through column sort order as follows:

- Where possible, sort order should prioritize columns with the lowest cardinality.
- Do not sort projections on columns of type LONG VARBINARY and LONG VARCHAR.

See also

Choosing sort order: best practices

Choosing sort order: best practices

When choosing sort orders for your projections, Vertica has several recommendations that can help you achieve maximum query performance, as illustrated in the following examples.

Combine RLE and sort order

When dealing with predicates on low-cardinality columns, use a combination of RLE and sorting to minimize storage requirements and maximize query performance.

Suppose you have a students table contain the following values and encoding types:

Column	# of Distinct Values	Encoded With
gender	2 (M or F)	RLE
pass_fail	2 (P or F)	RLE
class	4 (freshman, sophomore, junior, or senior)	RLE
name	10000 (too many to list)	Auto

You might have queries similar to this one:

```
SELECT name FROM studentsWHERE gender = 'M' AND pass_fail = 'P' AND class = 'senior';
```

The fastest way to access the data is to work through the low-cardinality columns with the smallest number of distinct values before the high-cardinality columns. The following sort order minimizes storage and maximizes query performance for queries that have equality restrictions on gender , class , pass_fail , and name . Specify the ORDER BY clause of the projection as follows:

```
ORDER BY students.gender, students.pass_fail, students.class, students.name
```

In this example, the gender column is represented by two RLE entries, the pass_fail column is represented by four entries, and the class column is represented by 16 entries, regardless of the cardinality of the students table. Vertica efficiently finds the set of rows that satisfy all the predicates, resulting in a huge reduction of search effort for RLE encoded columns that occur early in the sort order. Consequently, if you use low-cardinality columns in local predicates, as in the previous example, put those columns early in the projection sort order, in increasing order of distinct cardinality (that is, in increasing order of the number of distinct values in each column).

If you sort this table with student.class first, you improve the performance of queries that restrict only on the student.class column, and you improve the compression of the student.class column (which contains the largest number of distinct values), but the other columns do not compress as well. Determining which projection is better depends on the specific queries in your workload, and their relative importance.

Storage savings with compression decrease as the cardinality of the column increases; however, storage savings with compression increase as the number of bytes required to store values in that column increases.

Maximize the advantages of RLE

To maximize the advantages of RLE encoding, use it only when the average run length of a column is greater than 10 when sorted. For example, suppose you have a table with the following columns, sorted in order of cardinality from low to high:

```
address.country, address.region, address.state, address.city, address.zipcode
```

The zipcode column might not have 10 sorted entries in a row with the same zip code, so there is probably no advantage to run-length encoding that column, and it could make compression worse. But there are likely to be more than 10 countries in a sorted run length, so applying RLE to the country column can improve performance.

Put lower cardinality column first for functional dependencies

In general, put columns that you use for local predicates (as in the previous example) earlier in the join order to make predicate evaluation more efficient. In addition, if a lower cardinality column is uniquely determined by a higher cardinality column (like city_id uniquely determining a state_id), it is always better to put the lower cardinality, functionally determined column earlier in the sort order than the higher cardinality column.

For example, in the following sort order, the Area_Code column is sorted before the Number column in the customer_info table:

```
ORDER BY = customer_info.Area_Code, customer_info.Number, customer_info.Address
```

In the query, put the Area_Code column first, so that only the values in the Number column that start with 978 are scanned.

```
=> SELECT AddressFROM customer_info WHERE Area_Code='978' AND Number='9780123457';
```

Sort for merge joins

When processing a join, the Vertica optimizer chooses from two algorithms:

- **Merge join** —If both inputs are pre-sorted on the join column, the optimizer chooses a merge join, which is faster and uses less memory.
- **Hash join** —Using the hash join algorithm, Vertica uses the smaller (inner) joined table to build an in-memory hash table on the join column. A hash join has no sort requirement, but it consumes more memory because Vertica builds a hash table with the values in the inner table. The optimizer chooses a hash join when projections are not sorted on the join columns.

If both inputs are pre-sorted, merge joins do not have to do any pre-processing, making the join perform faster. Vertica uses the term sort-merge join to refer to the case when at least one of the inputs must be sorted prior to the merge join. Vertica sorts the inner input side but only if the outer input side is already sorted on the join columns.

To give the Vertica query optimizer the option to use an efficient merge join for a particular join, create projections on both sides of the join that put the join column first in their respective projections. This is primarily important to do if both tables are so large that neither table fits into memory. If all tables that a table will be joined to can be expected to fit into memory simultaneously, the benefits of merge join over hash join are sufficiently small that it probably isn't worth creating a projection for any one join column.

Sort on columns in important queries

If you have an important query, one that you run on a regular basis, you can save time by putting the columns specified in the WHERE clause or the GROUP BY clause of that query early in the sort order.

If that query uses a high-cardinality column such as Social Security number, you may sacrifice storage by placing this column early in the sort order of a projection, but your most important query will be optimized.

Sort columns of equal cardinality by size

If you have two columns of equal cardinality, put the column that is larger first in the sort order. For example, a CHAR(20) column takes up 20 bytes, but an INTEGER column takes up 8 bytes. By putting the CHAR(20) column ahead of the INTEGER column, your projection compresses better.

Sort foreign key columns first, from low to high distinct cardinality

Suppose you have a fact table where the first four columns in the sort order make up a foreign key to another table. For best compression, choose a sort order for the fact table such that the foreign keys appear first, and in increasing order of distinct cardinality. Other factors also apply to the design of projections for fact tables, such as partitioning by a time dimension, if any.

In the following example, the table inventory stores inventory data, and product_key and warehouse_key are foreign keys to the product_dimension and warehouse_dimension tables:

```
=> CREATE TABLE inventory (
 date_key INTEGER NOT NULL,
 product_key INTEGER NOT NULL,
 warehouse_key INTEGER NOT NULL,
 ...
);
=> ALTER TABLE inventory
   ADD CONSTRAINT fk_inventory_warehouse FOREIGN KEY(warehouse_key)
   REFERENCES warehouse_dimension(warehouse_key);
ALTER TABLE inventory
   ADD CONSTRAINT fk_inventory_product FOREIGN KEY(product_key)
   REFERENCES product_dimension(product_key);
```

The inventory table should be sorted by warehouse_key and then product, since the cardinality of the warehouse_key column is probably lower that the cardinality of the product_key .

Prioritizing column access speed

If you measure and set the performance of storage locations within your cluster, Vertica uses this information to determine where to store columns based on their rank. For more information, see Setting storage performance .

How columns are ranked

Vertica stores columns included in the projection sort order on the fastest available storage locations. Columns not included in the projection sort order are stored on slower disks. Columns for each projection are ranked as follows:

- Columns in the sort order are given the highest priority (numbers > 1000).
- The last column in the sort order is given the rank number 1001.
- The next-to-last column in the sort order is given the rank number 1002, and so on until the first column in the sort order is given 1000 + # of sort columns.
- The remaining columns are given numbers from 1000–1, starting with 1000 and decrementing by one per column.

Vertica then stores columns on disk from the highest ranking to the lowest ranking. It places highest-ranking columns on the fastest disks and the lowest-ranking columns on the slowest disks.

Overriding default column ranking

You can modify which columns are stored on fast disks by manually overriding the default ranks for these columns. To accomplish this, set the ACCESSRANK keyword in the column list. Make sure to use an integer that is not already being used for another column. For example, if you want to give a column the fastest access rank, use a number that is significantly higher than 1000 + the number of sort columns. This allows you to enter more columns over time without bumping into the access rank you set.

The following example sets column store_key 's access rank to 1500:

```
CREATE PROJECTION retail_sales_fact_p (
    store_key ENCODING RLE ACCESSRANK 1500,
    pos_transaction_number ENCODING RLE,
    sales_dollar_amount,
    cost_dollar_amount )
AS SELECT
    store_key,
    pos_transaction_number,
    sales_dollar_amount,
    cost_dollar_amount
FROM store.store_sales_fact
ORDER BY store_key
SEGMENTED BY HASH(pos_transaction_number) ALL NODES;
```

Database users and privileges

Database users should only have access to the database resources that they need to perform their tasks. For example, most users should be able to read data but not modify or insert new data. A smaller number of users typically need permission to perform a wider range of database tasks—for example, create and modify schemas, tables, and views. A very small number of users can perform administrative tasks, such as rebalance nodes on a cluster, or start or stop a database. You can also let certain users extend their own privileges to other users.

Client authentication controls what database objects users can access and change in the database. You specify access for specific users or roles with GRANT statements.

In this section

- Database users
- Database roles
- Database privileges
- Access policies

Database users

Every Vertica database has one or more users. When users connect to a database, they must log on with valid credentials (username and password) that a superuser defined in the database.

Database users own the objects they create in a database, such as tables, procedures, and storage locations.

> **Note**
> By default, users have the right to create temporary tables in a database.

In this section

- Types of database users
- Creating a database user
- User-level configuration parameters
- Locking user accounts
- Setting and changing user passwords

Types of database users

In a Vertica database, there are three types of users:

- Database administrator (DBADMIN)
- Object owner
- Everyone else (PUBLIC)

> **Note**
> External to a Vertica database, an MC administrator can create users through the Management Console and grant them database access. See Users in Management Console for details.

In this section

- Database administration user
- Object owner
- PUBLIC user

Database administration user

On installation, a new Vertica database automatically contains a user with superuser privileges . Unless explicitly named during installation , this user is identified as dbadmin . This user cannot be dropped and has the following irrevocable roles:

- DBADMIN
- DBDUSER
- PSEUDOSUPERUSER

With these roles, the dbadmin user can perform all database operations. This user can also create other users with administrative privileges.

Important
Do not confuse the dbadmin user with the DBADMIN role . The DBADMIN role is a set of privileges that can be assigned to one or more users.

The Vertica documentation often references the dbadmin user as a superuser. This reference is unrelated to Linux superusers.

Creating additional database administrators

As the dbadmin user, you can create other users with the same privileges:

1. Create a user:
```
=> CREATE USER DataBaseAdmin2;
CREATE USER
```
2. Grant the appropriate roles to new user DataBaseAdmin2 :
```
=> GRANT dbduser, dbadmin, pseudosuperuser to DataBaseAdmin2;
GRANT ROLE
```
User DataBaseAdmin2 now has the same privileges granted to the original dbadmin user.
3. As DataBaseAdmin2 , enable your assigned roles with SET ROLE :
```
=> \c - DataBaseAdmin2;
You are now connected to database "VMart" as user "DataBaseAdmin2".
=> SET ROLE dbadmin, dbduser, pseudosuperuser;
SET ROLE
```
4. Confirm the roles are enabled:
```
=> SHOW ENABLED ROLES;
name          | setting
------------------------------------------------
enabled roles | dbduser, dbadmin, pseudosuperuser
```

Object owner

An object owner is the user who creates a particular database object and can perform any operation on that object. By default, only an owner (or a superuser) can act on a database object. In order to allow other users to use an object, the owner or superuser must grant privileges to those users using one of the GRANT statements .

> **Note**
> Object owners are PUBLIC users for objects that other users own.

See Database privileges for more information.

PUBLIC user

All non-DBA (superuser) or object owners are PUBLIC users.

> **Note**
> Object owners are PUBLIC users for objects that other users own.

Newly-created users do not have access to schema PUBLIC by default. Make sure to GRANT USAGE ON SCHEMA PUBLIC to all users you create.

See also

- PUBLIC

Creating a database user

To create a database user:

1. From vsql , connect to the database as a superuser.
2. Issue the CREATE USER statement with optional parameters.
3. Run a series of GRANT statements to grant the new user privileges.

To create a user on MC, see Creating an MC user in Management Console

New user privileges

By default, new database users have the right to create temporary tables in the database.

New users do not have access to schema PUBLIC by default. Be sure to call GRANT USAGE ON SCHEMA PUBLIC to all users you create.

Modifying users

You can change information about a user, such as his or her password, by using the ALTER USER statement. If you want to configure a user to not have any password authentication, you can set the empty password '' in CREATE USER or ALTER USER statements, or omit the IDENTIFIED BY parameter in CREATE USER .

Example

The following series of commands add user Fred to a database with password 'password. The second command grants USAGE privileges to Fred on the public schema:

```
=> CREATE USER Fred IDENTIFIED BY 'password';
=> GRANT USAGE ON SCHEMA PUBLIC to Fred;
```

User names created with double-quotes are case sensitive. For example:

```
=> CREATE USER "FrEd1";
```

In the above example, the logon name must be an exact match. If the user name was created without double-quotes (for example, FRED1), then the user can log on as FRED1 , FrEd1 , fred1 , and so on.

See also

- Granting and revoking privileges
- Granting database roles

User-level configuration parameters

ALTER USER lets you set user-level configuration parameters on individual users. These settings override database- or session-level settings on the same parameters. For example, the following ALTER USER statement sets DepotOperationsForQuery for users Yvonne and Ahmed to FETCHES, thus overriding the default setting of ALL:

```
=> SELECT user_name, parameter_name, current_value, default_value FROM user_configuration_parameters
   WHERE user_name IN('Ahmed', 'Yvonne') AND parameter_name = 'DepotOperationsForQuery';
 user_name |     parameter_name      | current_value | default_value
-----------+-------------------------+---------------+---------------
 Ahmed     | DepotOperationsForQuery | ALL           | ALL
 Yvonne    | DepotOperationsForQuery | ALL           | ALL
(2 rows)

=> ALTER USER Ahmed SET DepotOperationsForQuery='FETCHES';
ALTER USER
=> ALTER USER Yvonne SET DepotOperationsForQuery='FETCHES';
ALTER USER
```

Identifying user-level parameters

To identify user-level configuration parameters, query the allowed_levels column of system table CONFIGURATION_PARAMETERS. For example, the following query identifies user-level parameters that affect depot usage:

```
n=> SELECT parameter_name, allowed_levels, default_value, current_level, current_value
    FROM configuration_parameters WHERE allowed_levels ilike '%USER%' AND parameter_name ilike '%depot%';
     parameter_name      |     allowed_levels      | default_value | current_level | current_value
-------------------------+-------------------------+---------------+---------------+---------------
 UseDepotForReads        | SESSION, USER, DATABASE | 1             | DEFAULT       | 1
 DepotOperationsForQuery | SESSION, USER, DATABASE | ALL           | DEFAULT       | ALL
 UseDepotForWrites       | SESSION, USER, DATABASE | 1             | DEFAULT       | 1
(3 rows)
```

Viewing user parameter settings

You can obtain user settings in two ways:

- Query system table USER_CONFIGURATION_PARAMETERS :

```
=> SELECT * FROM user_configuration_parameters;
 user_name |      parameter_name       | current_value | default_value
-----------+---------------------------+---------------+---------------
 Ahmed     | DepotOperationsForQuery   | FETCHES       | ALL
 Yvonne    | DepotOperationsForQuery   | FETCHES       | ALL
 Yvonne    | LoadSourceStatisticsLimit | 512           | 256
(3 rows)
```

- Use SHOW USER :

```
=> SHOW USER Yvonne PARAMETER ALL;
 user |         parameter         | setting
--------+---------------------------+---------
 Yvonne | DepotOperationsForQuery   | FETCHES
 Yvonne | LoadSourceStatisticsLimit | 512
(2 rows)

=> SHOW USER ALL PARAMETER ALL;
 user |         parameter         | setting
--------+---------------------------+---------
 Yvonne | DepotOperationsForQuery   | FETCHES
 Yvonne | LoadSourceStatisticsLimit | 512
 Ahmed  | DepotOperationsForQuery   | FETCHES
(3 rows)
```

Locking user accounts

As a superuser, you can manually lock and unlock a database user account with ALTER USER...ACCOUNT LOCK and ALTER USER...ACCOUNT UNLOCK , respectively. For example, the following command prevents user Fred from logging in to the database:

```
=> ALTER USER Fred ACCOUNT LOCK;
=> \c - Fred
FATAL 4974: The user account "Fred" is locked
HINT: Please contact the database administrator
```

The following example unlocks access to Fred's user account:

```
=> ALTER USER Fred ACCOUNT UNLOCK;|
=> \c - Fred
You are now connected as user "Fred".
```

Locking new accounts

CREATE USER can specify to lock a new account. Like any locked account, it can be unlocked with ALTER USER...ACCOUNT UNLOCK .

```
=> CREATE USER Bob ACCOUNT LOCK;
CREATE USER
```

Locking accounts for failed login attempts

A user's profile can specify to lock an account after a certain number of failed login attempts.

Setting and changing user passwords

As a superuser, you can set any user's password when you create that user with CREATE USER , or later with ALTER USER . Non-superusers can also change their own passwords with ALTER USER . One exception applies: users who are added to the Vertica database with the LDAPLink service cannot change their passwords with ALTER USER .

You can also give a user a pre-hashed password if you provide its associated salt. The salt must be a hex string. This method bypasses password complexity requirements .

To view password hashes and salts of existing users, see the PASSWORDS system table.

Changing a user's password has no effect on their current session.

Setting user passwords in VSQL

In this example, the user 'Bob' is created with the password 'mypassword.'

```
=> CREATE USER Bob IDENTIFIED BY 'mypassword';
CREATE USER
```

The password is then changed to 'Orca.'

```
=> ALTER USER Bob IDENTIFIED BY 'Orca' REPLACE 'mypassword';
ALTER USER
```

In this example, the user 'Alice' is created with a pre-hashed password and salt.

```
=> CREATE USER Alice IDENTIFIED BY
'sha512e0299de83ecfaa0b6c9cbb1feabfbe0b3c82a1495875cd9ec1c4b09016f09b42c1'
SALT '465a4aec38a85d6ecea5a0ac8f2d36d8';
```

Setting user passwords in Management Console

On Management Console, users with ADMIN or IT privileges can reset a user's non-LDAP password:

1. Sign in to Management Console and navigate to **MC Settings** > **User management** .
2. Click to select the user to modify and click **Edit** .
3. Click **Edit password** and enter the new password twice.
4. Click **OK** and then **Save** .

Database roles

A role is a collection of privileges that can be granted to one or more users or other roles. Roles help you grant and manage sets of privileges for various categories of users, rather than grant those privileges to each user individually.

For example, several users might require administrative privileges. You can grant these privileges to them as follows:

1. Create an administrator role with CREATE ROLE :
   ```
   CREATE ROLE administrator;
   ```
2. Grant the role to the appropriate users.
3. Grant the appropriate privileges to this role with one or more GRANT statements. You can later add and remove privileges as needed. Changes in role privileges are automatically propagated to the users who have that role.

After users are assigned roles, they can either enable those roles themselves, or you can automatically enable their roles for them.

In this section

- Predefined database roles
- Role hierarchy
- Creating and dropping roles
- Granting privileges to roles
- Granting database roles
- Revoking database roles
- Enabling roles
- Enabling roles automatically
- Viewing user roles

Predefined database roles

Vertica has the following predefined roles:

- DBADMIN
- PSEUDOSUPERUSER
- DBDUSER
- SYSMONITOR
- UDXDEVELOPER
- MLSUPERVISOR
- PUBLIC

Automatic role grants

On installation, Vertica automatically grants and enables predefined roles as follows:

- The DBADMIN, PSEUDOSUPERUSER, and DBDUSER roles are irrevocably granted to the dbadmin user . These roles are always enabled for dbadmin , and can never be dropped.
- PUBLIC is granted to dbadmin , and to all other users as they are created. This role is always enabled and cannot be dropped or revoked.

Granting predefined roles

After installation, the dbadmin user and users with the PSEUDOSUPERUSER role can grant one or more predefined roles to any user or non-predefined role. For example, the following set of statements creates the userdba role and grants it the predefined role DBADMIN:

```
=> CREATE ROLE userdba;
CREATE ROLE
=> GRANT DBADMIN TO userdba WITH ADMIN OPTION;
GRANT ROLE
```

Users and roles that are granted a predefined role can extend that role to other users, if the original GRANT (Role) statement includes WITH ADMIN OPTION. One exception applies: if you grant a user the PSEUDOSUPERUSER role and omit WITH ADMIN OPTION, the grantee can grant any role, including all predefined roles, to other users.

For example, the userdba role was previously granted the DBADMIN role. Because the GRANT statement includes WITH ADMIN OPTION, users who are assigned the userdba role can grant the DBADMIN role to other users:

```
=> GRANT userdba TO fred;
GRANT ROLE
=> \c - fred
You are now connected as user "fred".
=> SET ROLE userdba;
SET
=> GRANT dbadmin TO alice;
GRANT ROLE
```

Modifying predefined Roles

Excluding SYSMONITOR, you can grant predefined roles privileges on individual database objects, such as tables or schemas. For example:

```
=> CREATE SCHEMA s1;
CREATE SCHEMA
=> GRANT ALL ON SCHEMA s1 to PUBLIC;
GRANT PRIVILEGE
```

You can grant PUBLIC any role, including predefined roles. For example:

```

=> CREATE ROLE r1;
CREATE ROLE
=> GRANT r1 TO PUBLIC;
GRANT ROLE
```

You cannot modify any other predefined role by granting another role to it. Attempts to do so return a rollback error:

```
=> CREATE ROLE r2;
CREATE ROLE
=> GRANT r2 TO PSEUDOSUPERUSER;
ROLLBACK 2347:  Cannot alter predefined role "pseudosuperuser"
```

In this section

- DBADMIN
- PSEUDOSUPERUSER
- DBDUSER
- SYSMONITOR
- UDXDEVELOPER
- MLSUPERVISOR
- PUBLIC

DBADMIN

The DBADMIN role is a predefined role that is assigned to the dbadmin user on database installation. Thereafter, the dbadmin user and users with the

PSEUDOSUPERUSER role can grant any role to any user or non-predefined role.

For example, superuser dbadmin creates role fred and grants fred the DBADMIN role:

```
=> CREATE USER fred;
CREATE USER
=> GRANT DBADMIN TO fred WITH ADMIN OPTION;
GRANT ROLE
```

After user fred enables its DBADMIN role , he can exercise his DBADMIN privileges by creating user alice . Because the GRANT statement includes WITH ADMIN OPTION , fred can also grant the DBADMIN role to user alice :

```
=> \c - fred
You are now connected as user "fred".
=> SET ROLE dbadmin;
SET
CREATE USER alice;
CREATE USER
=> GRANT DBADMIN TO alice;
GRANT ROLE
```

DBADMIN privileges

The following table lists privileges that are supported for the DBADMIN role:

- Create users and roles, and grant them roles and privileges
- Create and drop schemas
- View all system tables
- View and terminate user sessions
- Access all data created by any user

PSEUDOSUPERUSER

The PSEUDOSUPERUSER role is a predefined role that is automatically assigned to the dbadmin user on database installation. The dbadmin can grant this role to any user or non-predefined role. Thereafter, PSEUDOSUPERUSER users can grant any role, including predefined roles, to other users.

PSEUDOSUPERUSER privileges

Users with the PSEUDOSUPERUSER role are entitled to complete administrative privileges, which cannot be revoked. Role privileges include:

- Bypass all GRANT/REVOKE authorization
- Create schemas and tables
- Create users and roles, and grant privileges to them
- Modify user accounts—for example, set user account's passwords, and lock/unlock accounts.
- Create or drop a UDF library and function, or any external procedure

DBDUSER

The DBDUSER role is a predefined role that is assigned to the dbadmin user on database installation. The dbadmin and any PSEUDOSUPERUSER can grant this role to any user or non-predefined role. Users who have this role and enable it can call Database Designer functions from the command line.

> **Note**
> Non-DBADMIN users with the DBDUSER role cannot run Database Designer through Administration Tools. Only DBADMIN users can run Administration Tools.

Associating DBDUSER with resource pools

Be sure to associate a resource pool with the DBDUSER role, to facilitate resource management when you run Database Designer. Multiple users can run Database Designer concurrently without interfering with each other or exhausting all the cluster resources. Whether you run Database Designer programmatically or with Administration Tools, design execution is generally contained by the user's resource pool, but might spill over into system resource pools for less-intensive tasks.

SYSMONITOR

An organization's database administrator may have many responsibilities outside of maintaining Vertica as a DBADMIN user. In this case, as the DBADMIN you may want to delegate some Vertica administrative tasks to another Vertica user.

The DBADMIN can assign a delegate the SYSMONITOR role to grant access to system tables without granting full DBADMIN access.

The SYSMONITOR role provides the following privileges.

- View all system tables that are marked as monitorable. You can see a list of all the monitorable tables by issuing the statement:
  ```
  => SELECT * FROM system_tables WHERE is_monitorable='t';
  ```
- If WITH ADMIN OPTION was included when granting SYSMONITOR to the user or role, that user or role can then grant SYSMONITOR privileges to other users and roles.

Grant a SYSMONITOR role

To grant a user or role the SYSMONITOR role, you must be one of the following:

- a DBADMIN user
- a user assigned the SYSMONITOR who has the ADMIN OPTION

Use the GRANT (Role) SQL statement to assign a user the SYSMONITOR role. This example shows how to grant the SYSMONITOR role to user1 and includes administration privileges by using the WITH ADMIN OPTION parameter. The ADMIN OPTION grants the SYSMONITOR role administrative privileges:

```
=> GRANT SYSMONITOR TO user1 WITH ADMIN OPTION;
```

This example shows how to revoke the ADMIN OPTION from the SYSMONITOR role for user1:

```
=> REVOKE ADMIN OPTION for SYSMONITOR FROM user1;
```

Use CASCADE to revoke ADMIN OPTION privileges for all users assigned the SYSMONITOR role:

```
=> REVOKE ADMIN OPTION for SYSMONITOR FROM PUBLIC CASCADE;
```

Example

This example shows how to:

- Create a user
- Create a role
- Grant SYSMONITOR privileges to the new role
- Grant the role to the user

```
=> CREATE USER user1;
=> CREATE ROLE monitor;
=> GRANT SYSMONITOR TO monitor;
=> GRANT monitor TO user1;
```

Assign SYSMONITOR privileges

This example uses the user and role created in the Grant SYSMONITOR Role example and shows how to:

- Create a table called personal_data
- Log in as user1
- Grant user1 the monitor role. (You already granted the monitor SYSMONITOR privileges in the Grant a SYSMONITOR Role example.)
- Run a SELECT statement as user1

The results of the operations are based on the privilege already granted to user1.

```
=> CREATE TABLE personal_data (SSN varchar (256));
=> \c -user1;
=> SET ROLE monitor;
=> SELECT COUNT(*) FROM TABLES;
 COUNT
-------
 1
(1 row)
```

Because you assigned the SYSMONITOR role, user1 can see the number of rows in the Tables system table. In this simple example, there is only one table (personal_data) in the database so the SELECT COUNT returns one row. In actual conditions, the SYSMONITOR role would see all the tables in the database.

Check if a table is accessible by SYSMONITOR

To check if a system table can be accessed by a user assigned the SYSMONITOR role:

```
=> SELECT table_name, is_monitorable FROM system_tables WHERE table_name='table_name'
```

For example, the following statement shows that the CURRENT_SESSION system table is accessible by the SYSMONITOR:

```
=> SELECT table_name, is_monitorable FROM system_tables WHERE table_name='current_session';
  table_name    | is_monitorable
----------------+----------------
 current_session | t
(1 row)
```

UDXDEVELOPER

The UDXDEVELOPER role is a predefined role that enables users to create and replace user-defined libraries. The dbadmin can grant this role to any user or non-predefined role.

UDXDEVELOPER privileges

Users with the UDXDEVELOPER role can perform the following actions:

- CREATE LIBRARY
- If library owner or with the DROP privilege:
  - CREATE OR REPLACE LIBRARY
  - DROP LIBRARY
  - ALTER LIBRARY

To use the privileges of this role, you must explicitly enable it using SET ROLE .

Security considerations

A user with the UDXDEVELOPER role can create libraries and, therefore, can install any UDx function in the database. UDx functions run as the Linux user that owns the database, and therefore have access to resources that Vertica has access to.

A poorly-written function can degrade database performance. Give this role only to users you trust to use UDxs responsibly. You can limit the memory that a UDx can consume by running UDxs in fenced mode and by setting the FencedUDxMemoryLimitMB configuration parameter.

MLSUPERVISOR

The MLSUPERVISOR role is a predefined role to which all the ML-model management privileges of DBADMIN are delegated. An MLSUPERVISOR can manage all models in the V_CATALOG.MODELS table on behalf of a dbadmin . However, it cannot import/export models.

In the following example, user alice uses her MLSUPERVISOR privileges to reassign ownership of the model my_model from user bob to user nina :

```
=> \c - alice
You are now connected as user "alice".

=> SELECT model_name, schema_name, owner_name FROM models;
 model_name  | schema_name | owner_name
-------------+-------------+------------
 my_model    | public      | bob
 mylinearreg | myschema2   | alice
 (2 rows)

=> SET ROLE MLSUPERVISOR;

=> ALTER MODEL my_model OWNER to nina;

=> SELECT model_name, schema_name, owner_name FROM models;
 model_name  | schema_name | owner_name
-------------+-------------+------------
 my_model    | public      | nina
 mylinearreg | myschema2   | alice
 (2 rows)

=> DROP MODEL my_model;
```

MLSUPERVISOR privileges

The following privileges are supported for the MLSUPERVISOR role:

- ML-model management privileges of DBADMIN
- Management (USAGE, ALTER, DROP) of all models in V_CATALOG.MODELS

To use the privileges of this role, you must explicitly enable it using SET ROLE . Note that an MLSUPERVISOR cannot import/export models.

See also

- Model versioning
- REGISTER_MODEL
- CHANGE_MODEL_STATUS

PUBLIC

The PUBLIC role is a predefined role that is automatically assigned to all new users. It is always enabled and cannot be dropped or revoked. Use this role to grant all database users the same minimum set of privileges.

Like any role, the PUBLIC role can be granted privileges to individual objects and other roles. The following example grants the PUBLIC role INSERT and SELECT privileges on table publicdata . This enables all users to read data in that table and insert new data:

```
=> CREATE TABLE publicdata (a INT, b VARCHAR);
CREATE TABLE
=> GRANT INSERT, SELECT ON publicdata TO PUBLIC;
GRANT PRIVILEGE
=> CREATE PROJECTION publicdataproj AS (SELECT * FROM publicdata);
CREATE PROJECTION
=> \c - bob
You are now connected as user "bob".
=> INSERT INTO publicdata VALUES (10, 'Hello World');
OUTPUT
--------
      1
(1 row)
```

The following example grants PUBLIC the employee role, so all database users have employee privileges:

```
=> GRANT employee TO public;
GRANT ROLE
```

Important

The clause WITH ADMIN OPTION is invalid for any GRANT statement that specifies PUBLIC as grantee.

Role hierarchy

By granting roles to other roles, you can build a hierarchy of roles, where roles lower in the hierarchy have a narrow range of privileges, while roles higher in the hierarchy are granted combinations of roles and their privileges. When you organize roles hierarchically, any privileges that you add to lower-level roles are automatically propagated to the roles above them.

Creating hierarchical roles

The following example creates two roles, assigns them privileges, then assigns both roles to another role.

1. Create table applog :

```
=> CREATE TABLE applog (id int, sourceID VARCHAR(32), data TIMESTAMP, event VARCHAR(256));
```

2. Create the logreader role and grant it read-only privileges on table applog :

```
=> CREATE ROLE logreader;
CREATE ROLE
=> GRANT SELECT ON applog TO logreader;
GRANT PRIVILEGE
```

3. Create the logwriter role and grant it write privileges on table applog :

```
=> CREATE ROLE logwriter;
CREATE ROLE
=> GRANT INSERT, UPDATE ON applog to logwriter;
GRANT PRIVILEGE
```

4. Create the logadmin role and grant it DELETE privilege on table applog :

```
=> CREATE ROLE logadmin;
CREATE ROLE
=> GRANT DELETE ON applog to logadmin;
GRANT PRIVILEGE
```

5. Grant the logreader and logwriter roles to role logadmin :

```
=> GRANT logreader, logwriter TO logadmin;
```

6. Create user bob and grant him the logadmin role:

```
=> CREATE USER bob;
CREATE USER
=> GRANT logadmin TO bob;
GRANT PRIVILEGE
```

7. Modify user bob 's account so his logadmin role is automatically enabled on login:

```

=> ALTER USER bob DEFAULT ROLE logadmin;
ALTER USER
=> \c - bob
You are now connected as user "bob".
=> SHOW ENABLED_ROLES;
    name      | setting
--------------+----------
 enabled roles | logadmin
(1 row)
```

Enabling hierarchical roles

Only roles that are explicitly granted to a user can be enabled for that user. In the previous example, roles logreader or logwriter cannot be enabled for bob . They can only be enabled indirectly, by enabling logadmin .

Hierarchical role grants and WITH ADMIN OPTION

If one or more roles are granted to another role using WITH ADMIN OPTION , then users who are granted the 'higher' role inherit administrative access to the subordinate roles.

For example, you might modify the earlier grants of roles logreader and logwriter to logadmin as follows:

```
=> GRANT logreader, logwriter TO logadmin WITH ADMIN OPTION;
NOTICE 4617:  Role "logreader" was already granted to role "logadmin"
NOTICE 4617:  Role "logwriter" was already granted to role "logadmin"
GRANT ROLE
```

User bob , through his logadmin role, is now authorized to grant its two subordinate roles to other users—in this case, role logreader to user Alice :

```
=> \c - bob;
You are now connected as user "bob".
=> GRANT logreader TO Alice;
GRANT ROLE
=> \c - alice;
You are now connected as user "alice".
=> show available_roles;
     name       |  setting
----------------+-----------
 available roles | logreader
(1 row)
```

Note

Because the grant of the logadmin role to bob did not include WITH ADMIN OPTION , he cannot grant that role to alice :

```
=> \c - bob;
You are now connected as user "bob".
=> GRANT logadmin TO alice;
ROLLBACK 4925:  The role "logadmin" cannot be granted to "alice"
```

Creating and dropping roles

As a superuser with the DBADMIN or PSEUDOSUPERUSER role, you can create and drop roles with CREATE ROLE and DROP ROLE , respectively.

```
=> CREATE ROLE administrator;
CREATE ROLE
```

A new role has no privileges or roles granted to it. Only superusers can grant privileges and access to the role.

Dropping database roles with dependencies

If you try to drop a role that is granted to users or other roles Vertica returns a rollback message:

```
=> DROP ROLE administrator;
NOTICE:  User Bob depends on Role administrator
ROLLBACK:  DROP ROLE failed due to dependencies
DETAIL:  Cannot drop Role administrator because other objects depend on it
HINT:  Use DROP ROLE ... CASCADE to remove granted roles from the dependent users/roles
```

To force the drop operation, qualify the DROP ROL E statement with CASCADE :

```
=> DROP ROLE administrator CASCADE;
DROP ROLE
```

Granting privileges to roles

You can use GRANT statements to assign privileges to a role, just as you assign privileges to users. See Database privileges for information about which privileges can be granted.

Granting a privilege to a role immediately affects active user sessions. When you grant a privilege to a role, it becomes immediately available to all users with that role enabled.

The following example creates two roles and assigns them different privileges on the same table.

1. Create table applog :

```
=> CREATE TABLE applog (id int, sourceID VARCHAR(32), data TIMESTAMP, event VARCHAR(256));
```

2. Create roles logreader and logwriter :

```
=> CREATE ROLE logreader;
CREATE ROLE
=> CREATE ROLE logwriter;
CREATE ROLE
```

3. Grant read-only privileges on applog to logreader , and write privileges to logwriter :

```
=> GRANT SELECT ON applog TO logreader;
GRANT PRIVILEGE
=> GRANT INSERT ON applog TO logwriter;
GRANT PRIVILEGE
```

Revoking privileges from roles

Use REVOKE statements to revoke a privilege from a role. Revoking a privilege from a role immediately affects active user sessions. When you revoke a privilege from a role, it is no longer available to users who have the privilege through that role.

For example:

```
=> REVOKE INSERT ON applog FROM logwriter;
REVOKE PRIVILEGE
```

Granting database roles

You can assign one or more roles to a user or another role with GRANT (Role) :

```
GRANT role[,...] TO grantee[,...] [ WITH ADMIN OPTION ]
```

For example, you might create three roles— appdata , applogs , and appadmin —and grant appadmin to user bob :

```
=> CREATE ROLE appdata;
CREATE ROLE
=> CREATE ROLE applogs;
CREATE ROLE
=> CREATE ROLE appadmin;
CREATE ROLE
=> GRANT appadmin TO bob;
GRANT ROLE
```

Granting roles to another role

GRANT can assign one or more roles to another role. For example, the following GRANT statement grants roles appdata and applogs to role appadmin :

```
=> GRANT appdata, applogs TO appadmin;
 -- grant to other roles
GRANT ROLE
```

Because user bob was previously assigned the role appadmin , he now has all privileges that are granted to roles appdata and applogs .

When you grant one role to another role, Vertica checks for circular references. In the previous example, role appdata is assigned to the appadmin role. Thus, subsequent attempts to assign appadmin to appdata fail, returning with the following warning:

```
=> GRANT appadmin TO appdata;
WARNING:  Circular assignation of roles is not allowed
HINT:  Cannot grant appadmin to appdata
GRANT ROLE
```

Enabling roles

After granting a role to a user, the role must be enabled. You can enable a role for the current session:

```
=> SET ROLE appdata;
SET ROLE
```

You can also enable a role as part of the user's login, by modifying the user's profile with ALTER USER...DEFAULT ROLE :

```
=> ALTER USER bob DEFAULT ROLE appdata;
ALTER USER
```

For details, see Enabling roles and Enabling roles automatically .

Granting administrative privileges

You can delegate to non-superusers users administrative access to a role by qualifying the GRANT (Role) statement with the option WITH ADMIN OPTION . Users with administrative access can manage access to the role for other users, including granting them administrative access. In the following example, a superuser grants the appadmin role with administrative privileges to users bob and alice.

```
=> GRANT appadmin TO bob, alice WITH ADMIN OPTION;
GRANT ROLE
```

Now, both users can exercise their administrative privileges to grant the appadmin role to other users, or revoke it. For example, user bob can now revoke the appadmin role from user alice :

```
=> \connect - bob
You are now connected as user "bob".
=> REVOKE appadmin FROM alice;
REVOKE ROLE
```

Caution

As with all user privilege models, database superusers should be cautious when granting any user a role with administrative privileges. For example, if the database superuser grants two users a role with administrative privileges, either user can revoke that role from the other user.

Example

The following example creates a role called commenter and grants that role to user bob :

1. Create the comments table:
   ```
   => CREATE TABLE comments (id INT, comment VARCHAR);
   ```
2. Create the commenter role:
   ```
   => CREATE ROLE commenter;
   ```
3. Grant to commenter INSERT and SELECT privileges on the comments table:
   ```
   => GRANT INSERT, SELECT ON comments TO commenter;
   ```
4. Grant the commenter role to user bob .
   ```
   => GRANT commenter TO bob;
   ```
5. In order to access the role and its associated privileges, bob enables the newly-granted role for himself:
   ```
   => \c - bob
   => SET ROLE commenter;
   ```
6. Because bob has INSERT and SELECT privileges on the comments table, he can perform the following actions:

```
=> INSERT INTO comments VALUES (1, 'Hello World');
 OUTPUT
--------
      1
(1 row)
=> SELECT * FROM comments;
 id |   comment
----+-------------
  1 | Hello World
(1 row)
=> COMMIT;
COMMIT
```

7. Because bob 's role lacks DELETE privileges, the following statement returns an error:

```
=> DELETE FROM comments WHERE id=1;
ERROR 4367:  Permission denied for relation comments
```

See also

Granting database access to MC users

Revoking database roles

REVOKE (Role) can revoke roles from one or more grantees—that is, from users or roles:

```
REVOKE [ ADMIN OPTION FOR ] role[,...] FROM grantee[,...] [ CASCADE ]
```

For example, the following statement revokes the commenter role from user bob :

```
=> \c
You are now connected as user "dbadmin".
=> REVOKE commenter FROM bob;
REVOKE ROLE
```

Revoking administrative access from a role

You can qualify REVOKE (Role) with the clause ADMIN OPTION FOR . This clause revokes from the grantees the authority (granted by an earlier GRANT (Role)...WITH ADMIN OPTION statement) to grant the specified roles to other users or roles. Current roles for the grantees are unaffected.

The following example revokes user Alice's authority to grant and revoke the commenter role:

```
=> \c
You are now connected as user "dbadmin".
=> REVOKE ADMIN OPTION FOR commenter FROM alice;
REVOKE ROLE
```

Enabling roles

When you enable a role in a session, you obtain all privileges assigned to that role. You can enable multiple roles simultaneously, thereby gaining all privileges of those roles, plus any privileges that are already granted to you directly.

By default, only predefined roles are enabled automatically for users. Otherwise, on starting a session, you must explicitly enable assigned roles with the Vertica function SET ROLE .

For example, the dbadmin creates the logreader role and assigns it to user alice :

```
=> \c
You are now connected as user "dbadmin".
=> CREATE ROLE logreader;
CREATE ROLE
=> GRANT SELECT ON TABLE applog to logreader;
GRANT PRIVILEGE
=> GRANT logreader TO alice;
GRANT ROLE
```

User alice must enable the new role before she can view the applog table:

```
=> \c - alice
You are now connected as user "alice".
=> SELECT * FROM applog;
ERROR:  permission denied for relation applog
=> SET ROLE logreader;
SET
=> SELECT * FROM applog;
 id | sourceID |          data           |                  event
----+----------+-------------------------+-----------------------------------------
  1 | Loader   | 2011-03-31 11:00:38.494226 | Error: Failed to open source file
  2 | Reporter | 2011-03-31 11:00:38.494226 | Warning: Low disk space on volume /scratch-a
(2 rows)
```

Enabling all user roles

You can enable all roles available to your user account with SET ROLE ALL :

```
=> SET ROLE ALL;
SET
=> SHOW ENABLED_ROLES;
    name     |          setting
-------------+-----------------------------
 enabled roles | logreader, logwriter
(1 row)
```

Important
You can also enable user roles on login. For more information, see Enabling roles automatically .

Disabling roles

A user can disable all roles with SET ROLE NONE . This statement disables all roles for the current session, excluding predefined roles:

```
=> SET ROLE NONE;
=> SHOW ENABLED_ROLES;
    name     | setting
-------------+---------
 enabled roles |
(1 row)
```

Enabling roles automatically

By default, new users are assigned the PUBLIC , which is automatically enabled when a new session starts. Typically, other roles are created and users are assigned to them, but these are not automatically enabled. Instead, users must explicitly enable their assigned roles with each new session, with SET ROLE .

You can automatically enable roles for users in two ways:

- Enable roles for individual users on login
- Enable all roles for all users on login

Enable roles for individual users

After assigning roles to users, you can set one or more default roles for each user by modifying their profiles, with ALTER USER...DEFAULT ROLE .
User default roles are automatically enabled at the start of the user session. You should consider setting default roles for users if they typically rely on the privileges of those roles to carry out routine tasks.

Important
ALTER USER...DEFAULT ROLE overwrites previous default role settings.

The following example shows how to set regional_manager as the default role for user LilyCP :

```
=> \c
You are now connected as user "dbadmin".
=> GRANT regional_manager TO LilyCP;
GRANT ROLE
=> ALTER USER LilyCP DEFAULT ROLE regional_manager;
ALTER USER
=> \c - LilyCP
You are now connected as user "LilyCP".
=> SHOW ENABLED_ROLES;
    name      |     setting
--------------+------------------
 enabled roles | regional_manager
(1 row)
```

Enable all roles for all users

Configuration parameter EnableAllRolesOnLogin specifies whether to enable all roles for all database users on login. By default, this parameter is set to 0. If set to 1, Vertica enables the roles of all users when they log in to the database.

Clearing default roles

You can clear all default role assignments for a user with ALTER USER...DEFAULT ROLE NONE . For example:

```
=> ALTER USER fred DEFAULT ROLE NONE;
ALTER USER
=> SELECT user_name, default_roles, all_roles FROM users WHERE user_name = 'fred';
 user_name | default_roles | all_roles
-----------+---------------+-----------
 fred      |               | logreader
(1 row)
```

Viewing user roles

You can obtain information about roles in three ways:

- Verify specific role assignments with the function HAS_ROLE .
- View all available (granted) and enabled roles .
- Obtain comprehensive information about roles, the users assigned to them, and the privileges granted to those users and roles by querying system tables ROLES , USERS , AND GRANTS , respectively.

> **Note**
> System tables do not show whether a role is available to a user indirectly through other roles. Call HAS_ROLE to obtain that information.

Verifying role assignments

The function HAS_ROLE checks whether a Vertica role is granted to the specified user or role. Non-superusers can use this function to check their own role membership. Superusers can use it to determine role assignments for other users and roles. You can also use Management Console to check role assignments .

In the following example, a dbadmin user checks whether user MikeL is assigned the admnistrator role:

```
=> \c
You are now connected as user "dbadmin".
=> SELECT HAS_ROLE('MikeL', 'administrator');
 HAS_ROLE
----------
 t
(1 row)
```

User MikeL checks whether he has the regional_manager role:

```
=> \c - MikeL
You are now connected as user "MikeL".
=> SELECT HAS_ROLE('regional_manager');
 HAS_ROLE
----------
 f
(1 row)
```

The dbadmin grants the regional_manager role to the administrator role. On checking again, MikeL verifies that he now has the regional_manager role:

```
dbadmin=> \c
You are now connected as user "dbadmin".
dbadmin=> GRANT regional_manager to administrator;
GRANT ROLE
dbadmin=> \c - MikeL
You are now connected as user "MikeL".
dbadmin=> SELECT HAS_ROLE('regional_manager');
 HAS_ROLE
----------
 t
(1 row)
```

Viewing available and enabled roles

SHOW AVAILABLE ROLES lists all roles granted to you:

```
=> SHOW AVAILABLE ROLES;
      name       |           setting
-----------------+------------------------------
 available roles | logreader, logwriter
(1 row)
```

SHOW ENABLED ROLES lists the roles enabled in your session:

```
=> SHOW ENABLED ROLES;
     name      | setting
---------------+----------
 enabled roles | logreader
(1 row)
```

Querying system tables

You can query tables ROLES , USERS , AND GRANTS , either separately or joined, to obtain detailed information about user roles, users assigned to those roles, and the privileges granted explicitly to users and implicitly through roles.

The following query on ROLES returns the names of all roles users can access, and the roles granted (assigned) to those roles. An asterisk (*) appended to a role indicates that the user can grant the role to other users:

```
=> SELECT * FROM roles;
      name       | assigned_roles
-----------------+----------------
 public          |
 dbduser         |
 dbadmin         | dbduser*
 pseudosuperuser | dbadmin*
 logreader       |
 logwriter       |
 logadmin        | logreader, logwriter
(7 rows)
```

The following query on system table USERS returns all users with the DBADMIN role. An asterisk (*) appended to a role indicates that the user can grant the role to other users:

```
=> SELECT user_name, is_super_user, default_roles, all_roles FROM v_catalog.users WHERE all_roles ILIKE '%dbadmin%';
 user_name | is_super_user |           default_roles            |             all_roles
-----------+---------------+------------------------------------+-----------------------------------
 dbadmin   | t             | dbduser*, dbadmin*, pseudosuperuser* | dbduser*, dbadmin*, pseudosuperuser*
 u1        | f             |                                    | dbadmin*
 u2        | f             |                                    | dbadmin
(3 rows)
```

The following query on system table GRANTS returns the privileges granted to user Jane or role R1. An asterisk (*) appended to a privilege indicates that the user can grant the privilege to other users:

```
=> SELECT grantor,privileges_description,object_name,object_type,grantee FROM grants WHERE grantee='Jane' OR grantee='R1';
grantor | privileges_description | object_name | object_type  |  grantee
--------+------------------------+-------------+--------------+-----------
dbadmin | USAGE                  | general     | RESOURCEPOOL | Jane
dbadmin |                        | R1          | ROLE         | Jane
dbadmin | USAGE*                 | s1          | SCHEMA       | Jane
dbadmin | USAGE, CREATE*         | s1          | SCHEMA       | R1
(4 rows)
```

Database privileges

When a database object is created, such as a schema, table, or view, ownership of that object is assigned to the user who created it. By default, only the object's owner, and users with superuser privileges such as database administrators, have privileges on a new object. Only these users (and other users whom they explicitly authorize) can grant object privileges to other users

Privileges are granted and revoked by GRANT and REVOKE statements, respectively. The privileges that can be granted on a given object are specific to its type. For example, table privileges include SELECT, INSERT, and UPDATE, while library and resource pool privileges have USAGE privileges only. For a summary of object privileges, see Database object privileges .

Because privileges on database objects can come from several different sources like explicit grants, roles, and inheritance, privileges can be difficult to monitor. Use the GET_PRIVILEGES_DESCRIPTION meta-function to check the current user's effective privileges across all sources on a specified database object.

In this section

- Ownership and implicit privileges
- Inherited privileges
- Default user privileges
- Effective privileges
- Privileges required for common database operations
- Database object privileges
- Granting and revoking privileges
- Modifying privileges
- Viewing privileges granted on objects

Ownership and implicit privileges

All users have *implicit* privileges on the objects that they own. On creating an object, its owner automatically is granted all privileges associated with the object's type (see Database object privileges). Regardless of object type, the following privileges are inseparable from ownership and cannot be revoked, not even by the owner:

- Authority to grant all object privileges to other users, and revoke them
- ALTER (where applicable) and DROP
- Extension of privilege granting authority on their objects to other users, and revoking that authority

Object owners can revoke all non-implicit, or *ordinary* , privileges from themselves. For example, on creating a table, its owner is automatically granted all implicit and ordinary privileges:

Implicit table privileges	**Ordinary table privileges**

ALTER DROP	DELETE INSERT REFERENCES SELECT TRUNCATE UPDATE

If user Joan creates table t1 , she can revoke ordinary privileges UPDATE and INSERT from herself, which effectively makes this table read-only:

```
=> \c - Joan
You are now connected as user "Joan".
=> CREATE TABLE t1 (a int);
CREATE TABLE
=> INSERT INTO t1 VALUES (1);
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
=> REVOKE UPDATE, INSERT ON TABLE t1 FROM Joan;
REVOKE PRIVILEGE
=> INSERT INTO t1 VALUES (3);
ERROR 4367:  Permission denied for relation t1
=> SELECT * FROM t1;
 a
---
 1
(1 row)
```

Joan can subsequently restore UPDATE and INSERT privileges to herself:

```
=> GRANT UPDATE, INSERT on TABLE t1 TO Joan;
GRANT PRIVILEGE
dbadmin=> INSERT INTO t1 VALUES (3);
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
dbadmin=> SELECT * FROM t1;
 a
---
 1
 3
(2 rows)
```

Inherited privileges

You can manage inheritance of privileges at three levels:

- Database
- Schema
- Tables and views

By default, inherited privileges are enabled at the database level and disabled at the schema level. If privilege inheritance is enabled at both levels, *new* tables and views automatically inherit those privileges when they are created. You can also exclude inheritance from specific tables and views.

In this section

- Enabling database inheritance
- Enabling schema inheritance
- Setting privilege inheritance on tables and views
- Example usage: implementing inherited privileges

Enabling database inheritance

By default, inherited privileges are enabled at the database level, through configuration parameter disableinheritedprivileges . To enable inherited privileges:

```
=> ALTER DATABASE [database name] SET disableinheritedprivileges = 0;
```

To disable inherited privileges:

```
=> ALTER DATABASE [database name] SET disableinheritedprivileges = 1;
```

Enabling schema inheritance

Caution

Enabling inherited privileges with ALTER SCHEMA ... DEFAULT INCLUDE PRIVILEGES only affects *newly* created tables and views.

This setting does not affect already-existing tables and views .

By default, inherited privileges are disabled at the schema level. If inherited privileges are enabled for the database, you can enable inheritance of schema privileges by its tables and views, with CREATE SCHEMA and ALTER SCHEMA . Unless explicitly excluded, privileges granted on the schema are automatically inherited by all new tables and views in it.

For information about which tables and views inherit privileges from which schemas, see INHERITING_OBJECTS .

For information about which privileges each table or view inherits, see the INHERITED_PRIVILEGES .

Note

If inherited privileges are disabled for the database, enabling inheritance on its schemas has no effect. Attempts to do so return the following message:

```
Inherited privileges are globally disabled; schema parameter is set but has no effect.
```

Enabling inheritance of schema privileges has no effect on existing tables and views. You must explicitly set schema inheritance on them with ALTER TABLE and ALTER VIEW . You can also explicitly exclude tables and views from inheriting schema privileges with CREATE TABLE / ALTER TABLE , and CREATE VIEW / ALTER VIEW , respectively.

You can enable schema privilege inheritance during schema creation with the following statement:

```
=> CREATE SCHEMA s1 DEFAULT INCLUDE PRIVILEGES;
```

If the schema already exists, you can use ALTER SCHEMA to have all *newly created tables and views* inherit the privileges of the schema. Tables and views created on the schema before this statement are not affected:

```
=> ALTER SCHEMA s1 DEFAULT INCLUDE PRIVILEGES;
```

After enabling inherited privileges on a schema, you can grant privileges on it to users and roles with GRANT (schema) :

```
=> GRANT USAGE, CREATE, SELECT, INSERT ON SCHEMA S1 TO PUBLIC;
GRANT PRIVILEGE
```

See also

- Setting privilege inheritance on tables and views
- Granting and revoking privileges

Setting privilege inheritance on tables and views

Caution
Enabling inherited privileges with ALTER SCHEMA ... DEFAULT INCLUDE PRIVILEGES only affects *newly* created tables and views.

This setting does not affect already-existing tables and views .

If inherited privileges are enabled for the database and a schema, privileges granted to the schema are automatically granted to all new tables and views in it. You can also explicitly exclude tables and views from inheriting schema privileges.

For information about which tables and views inherit privileges from which schemas, see INHERITING_OBJECTS .

For information about which privileges each table or view inherits, see the INHERITED_PRIVILEGES .

Set privileges inheritance on tables and views

CREATE TABLE / ALTER TABLE and CREATE VIEW / ALTER VIEW can allow tables and views to inherit privileges from their parent schemas. For example, the following statements enable inheritance on schema s1, so new table s1.t1 and view s1.myview automatically inherit the privileges set on that schema as applicable:

```
=> CREATE SCHEMA s1 DEFAULT INCLUDE PRIVILEGES;
CREATE SCHEMA
=> GRANT USAGE, CREATE, SELECT, INSERT ON SCHEMA S1 TO PUBLIC;
GRANT PRIVILEGE
=> CREATE TABLE s1.t1 ( ID int, f_name varchar(16), l_name(24));
WARNING 6978:  Table "t1" will include privileges from schema "s1"
CREATE TABLE
=> CREATE VIEW s1.myview AS SELECT ID, l_name FROM s1.t1
WARNING 6978:  View "myview" will include privileges from schema "s1"
CREATE VIEW
```

Note
Both CREATE statements omit the clause INCLUDE SCHEMA PRIVILEGES, so they return a warning message that the new objects will inherit schema privileges. CREATE statements that include this clause do not return a warning message.

If the schema already exists, you can use ALTER SCHEMA to have all *newly created tables and views* inherit the privileges of the schema. Tables and views created on the schema before this statement, however, are not affected:

```
=> CREATE SCHEMA s2;
CREATE SCHEMA
=> CREATE TABLE s2.t22 ( a int );
CREATE TABLE
...
=> ALTER SCHEMA S2 DEFAULT INCLUDE PRIVILEGES;
ALTER SCHEMA
```

In this case, inherited privileges were enabled on schema s2 after it already contained table s2.t22. To set inheritance on this table and other existing tables and views, you must explicitly set schema inheritance on them with ALTER TABLE and ALTER VIEW :

```
=> ALTER TABLE s2.t22 INCLUDE SCHEMA PRIVILEGES;
```

Exclude privileges inheritance from tables and views

You can use CREATE TABLE / ALTER TABLE and CREATE VIEW / ALTER VIEW to prevent table and views from inheriting schema privileges.

The following example shows how to create a table that does not inherit schema privileges:

```
=> CREATE TABLE s1.t1 ( x int) EXCLUDE SCHEMA PRIVILEGES;
```

You can modify an existing table so it does not inherit schema privileges:

```
=> ALTER TABLE s1.t1 EXCLUDE SCHEMA PRIVILEGES;
```

Example usage: implementing inherited privileges

The following steps show how user Joe enables inheritance of privileges on a given schema so other users can access tables in that schema.

1. Joe creates schema schema1 , and creates table table1 in it:

```
=>\c - Joe
You are now connected as user Joe
=> CREATE SCHEMA schema1;
CRDEATE SCHEMA
=> CREATE TABLE schema1.table1 (id int);
CREATE TABLE
```

2. Joe grants USAGE and CREATE privileges on schema1 to Myra :

```
=> GRANT USAGE, CREATE ON SCHEMA schema1 to Myra;
GRANT PRIVILEGE
```

3. Myra queries schema1.table1 , but the query fails:

```
=>\c - Myra
You are now connected as user Myra
=> SELECT * FROM schema1.table1;
ERROR 4367: Permission denied for relation table1
```

4. Joe grants Myra SELECT ON SCHEMA privileges on schema1 :

```
=>\c - Joe
You are now connected as user Joe
=> GRANT SELECT ON SCHEMA schema1 to Myra;
GRANT PRIVILEGE
```

5. Joe uses ALTER TABLE to include SCHEMA privileges for table1 :

```
=> ALTER TABLE schema1.table1 INCLUDE SCHEMA PRIVILEGES;
ALTER TABLE
```

6. Myra 's query now succeeds:

```
=>\c - Myra
You are now connected as user Myra
=> SELECT * FROM schema1.table1;
id
---
(0 rows)
```

7. Joe modifies schema1 to include privileges so all tables created in schema1 inherit schema privileges:

```
=>\c - Joe
You are now connected as user Joe
=> ALTER SCHEMA schema1 DEFAULT INCLUDE PRIVILEGES;
ALTER SCHEMA
=> CREATE TABLE schema1.table2 (id int);
CREATE TABLE
```

8. With inherited privileges enabled, Myra can query table2 without Joe having to explicitly grant privileges on the table:

```
=>\c - Myra
You are now connected as user Myra
=> SELECT * FROM schema1.table2;
id
---
(0 rows)
```

Default user privileges

To set the minimum level of privilege for all users, Vertica has the special PUBLIC , which it grants to each user automatically. This role is automatically enabled, but the database administrator or a superuser can also grant higher privileges to users separately using GRANT statements.

Default privileges for MC users

Privileges on Management Console (MC) are managed through roles, which determine a user's access to MC and to MC-managed Vertica databases through the MC interface. MC privileges do not alter or override Vertica privileges or roles. See Users, roles, and privileges for details.

Effective privileges

A user's *effective privileges* on an object encompass privileges of all types, including:

- Implicit privileges through object ownership
- Explicit privileges through GRANT statements on objects
- Inherited privileges through privileges on objects with inheritance enabled

You can view your effective privileges on an object with the GET_PRIVILEGES_DESCRIPTION meta-function.

Privileges required for common database operations

This topic lists the required privileges for database objects in Vertica.

Unless otherwise noted, superusers can perform all operations shown in the following tables. Object owners always can perform operations on their own objects.

> **Note**
> Certain actions, such as setting another user's resource pool or selecting a view , depend on the effective privileges of other users. If that other user acquires these prerequisite privileges through a role, it must be a default role for the action to succeed.
>
> For more information on changing a user's default roles, see Enabling roles automatically .

Schemas

The PUBLIC schema is present in any newly-created Vertica database. Newly-created users must be granted access to this schema:

```
=> GRANT USAGE ON SCHEMA public TO user;
```

A database superuser must also explicitly grant new users CREATE privileges, as well as grant them individual object privileges so the new users can create or look up objects in the PUBLIC schema.

Operation	Required Privileges
CREATE SCHEMA	Database: CREATE
DROP SCHEMA	Schema: owner
ALTER SCHEMA	Database: CREATE

Tables

Operation	Required Privileges

CREATE TABLE	Schema: CREATE **Note** Referencing sequences in the CREATE TABLE statement requires the following privileges: • Sequence schema: USAGE • Sequence: SELECT
DROP TABLE	Schema: USAGE or schema owner
TRUNCATE TABLE	Schema: USAGE or schema owner
ALTER TABLE ADD/DROP/ RENAME/ALTER-TYPE COLUMN	Schema: USAGE
ALTER TABLE ADD/DROP CONSTRAINT	Schema: USAGE
ALTER TABLE PARTITION (REORGANIZE)	Schema: USAGE
ALTER TABLE RENAME	USAGE and CREATE privilege on the schema that contains the table
ALTER TABLE...SET SCHEMA	• New schema: CREATE • Old Schema: USAGE
SELECT	• Schema: USAGE • SELECT privilege on table
INSERT	• Table: INSERT • Schema: USAGE
DELETE	• Schema: USAGE • Table: DELETE, SELECT when executing DELETE that references table column values in a WHERE or SET clause
UPDATE	• Schema: USAGE • Table: UPDATE, SELECT when executing UPDATE that references table column values in a WHERE or SET clause
REFERENCES	• Schema: USAGE on schema that contains constrained table and source of foreign key • Table: REFERENCES to create foreign key constraints that reference this table
ANALYZE_STATISTICS ANALYZE_STATISTICS_PARTITION	• Schema: USAGE • Table: One of INSERT, DELETE, or UPDATE
DROP_STATISTICS	• Schema: USAGE • Table: One of INSERT, DELETE, or UPDATE
DROP_PARTITIONS	Schema: USAGE

Views

Operation	**Required Privileges**

CREATE VIEW	• Schema: CREATE on view schema, USAGE on schema with base objects • Base objects: SELECT
DROP VIEW	• Schema: USAGE or owner • View: Owner
SELECT	• Base table: View owner must have SELECT...WITH GRANT OPTION • Schema: USAGE • View: SELECT

Projections

Operation	Required Privileges
CREATE PROJECTION	• Anchor table: SELECT • Schema: USAGE and CREATE, or owner **Note** If a projection is implicitly created with the table, no additional privilege is needed other than privileges for table creation.
AUTO/DELAYED PROJECTION	On projections created during INSERT...SELECT or COPY operations: • Schema: USAGE • Anchor table: SELECT
ALTER PROJECTION	Schema: USAGE and CREATE
DROP PROJECTION	Schema: USAGE or owner

External procedures

Operation	Required Privileges
CREATE PROCEDURE (external)	Superuser
DROP PROCEDURE (external)	Superuser
EXECUTE	• Schema: USAGE • Procedure: EXECUTE

Stored procedures

Operation	Required Privileges
CREATE PROCEDURE (stored)	Schema: CREATE

Triggers

Operation	Required Privileges
CREATE TRIGGER	Superuser

Schedules

Operation	Required Privileges
CREATE SCHEDULE	Superuser

Libraries

Operation	Required Privileges
CREATE LIBRARY	Superuser
DROP LIBRARY	Superuser

User-defined functions

> **Note**
> The following table uses these abbreviations:
>
> - UDF = Scalar
> - UDT = Transform
> - UDAnF= Analytic
> - UDAF = Aggregate

Operation	Required Privileges
CREATE FUNCTION (SQL) CREATE FUNCTION (scalar) CREATE TRANSFORM FUNCTION CREATE ANALYTIC FUNCTION (UDAnF) CREATE AGGREGATE FUNCTION (UDAF)	• Schema: CREATE • Base library: USAGE (if applicable)
DROP FUNCTION DROP TRANSFORM FUNCTION DROP AGGREGATE FUNCTION DROP ANALYTIC FUNCTION	• Schema: USAGE privilege • Function: owner
ALTER FUNCTION (scalar) ...RENAME TO	Schema: USAGE and CREATE
ALTER FUNCTION (scalar) ...SET SCHEMA	• Old schema: USAGE • New Schema: CREATE
EXECUTE (SQL/UDF/UDT/ ADAF/UDAnF) function	• Schema: USAGE • Function: EXECUTE

Sequences

Operation	Required Privileges
CREATE SEQUENCE	Schema: CREATE
DROP SEQUENCE	Schema: USAGE or owner
ALTER SEQUENCE	Schema: USAGE and CREATE
ALTER SEQUENCE ...SET SCHEMA	• Old schema: USAGE • New schema: CREATE

CURRVAL NEXTVAL	• Sequence schema: USAGE • Sequence: SELECT

Resource pools

Operation	Required Privileges
CREATE RESOURCE POOL	Superuser
ALTER RESOURCE POOL	Superuser to alter: • MAXMEMORYSIZE • PRIORITY • QUEUETIMEOUT Non-superuser, UPDATE to alter: • PLANNEDCONCURRENCY • SINGLEINITIATOR • MAXCONCURRENCY
SET SESSION RESOURCE_POOL	• Resource pool: USAGE • Users can only change their own resource pool setting using ALTER USER syntax
DROP RESOURCE POOL	Superuser

Users/profiles/roles

Operation	Required Privileges
CREATE USER CREATE PROFILE CREATE ROLE	Superuser
ALTER USER ALTER PROFILE ALTER ROLE	Superuser
DROP USER DROP PROFILE DROP ROLE	Superuser

Object visibility

You can use one or a combination of vsql \d meta commands and SQL system tables to view objects on which you have privileges to view.

- Use \dn to view schema names and owners
- Use \dt to view all tables in the database, as well as the system table V_CATALOG.TABLES
- Use \dj to view projections showing the schema, projection name, owner, and node, as well as the system table V_CATALOG.PROJECTIONS

Operation	Required Privileges
Look up schema	Schema: At least one privilege

Look up object in schema or in system tables	• Schema: USAGE
	• At least one privilege on any of the following objects:
	  ◦ TABLE
	  ◦ VIEW
	  ◦ FUNCTION
	  ◦ PROCEDURE
	  ◦ SEQUENCE

Look up projection	All anchor tables: At least one privilege Schema (all anchor tables): USAGE
Look up resource pool	Resource pool: SELECT
Existence of object	Schema: USAGE

I/O operations

Operation	**Required Privileges**
CONNECT TO VERTICA DISCONNECT	None
EXPORT TO VERTICA	• Source table: SELECT • Source schema: USAGE • Destination table: INSERT • Destination schema: USAGE
COPY FROM VERTICA	• Source/destination schema: USAGE • Source table: SELECT • Destination table: INSERT
COPY FROM *file*	Superuser
COPY FROM STDIN	• Schema: USAGE • Table: INSERT
COPY LOCAL	• Schema: USAGE • Table: INSERT

Comments

Operation	**Required Privileges**

COMMENT ON { is one of }: • AGGREGATE FUNCTION • ANALYTIC FUNCTION • CONSTRAINT • FUNCTION • LIBRARY • NODE • PROJECTION • PROJECTION COLUMN • SCHEMA • SEQUENCE • TABLE • TABLE COLUMN • TRANSFORM FUNCTION • VIEW	Object owner or superuser

Transactions

Operation	Required Privileges
COMMIT	None
ROLLBACK	None
RELEASE SAVEPOINT	None
SAVEPOINT	None

Sessions

Operation	Required Privileges
SET { is one of }: • DATESTYLE • ESCAPE_STRING_WARNING • INTERVALSTYLE • LOCALE • ROLE • SEARCH_PATH • SESSION AUTOCOMMIT • SESSION CHARACTERISTICS • SESSION MEMORYCAP • SESSION RESOURCE POOL • SESSION RUNTIMECAP • SESSION TEMPSPACE • STANDARD_CONFORMING_STRINGS • TIMEZONE	None
SHOW { name | ALL }	None

Tuning operations

Operation	Required Privileges
PROFILE	Same privileges required to run the query being profiled
EXPLAIN	Same privileges required to run the query for which you use the EXPLAIN keyword

TLS configuration

Operation	Required Privileges
ALTER	ALTER privileges on the TLS Configuration
DROP	DROP privileges on the TLS Configuration
Add certificates to a TLS Configuration	USAGE on the certificate's private key

Cryptographic key

Operation	Required Privileges
Create a certificate from the key	USAGE privileges on the key
DROP	DROP privileges on the key

Certificate

Operation	Required Privileges
Add certificate to TLS Configuration	ALTER privileges on the TLS Configuration and one of the following: • USAGE privileges on the certificate • USAGE privileges on the certificate's private key
DROP	DROP privileges on the certificate's private key

Database object privileges

Privileges can be granted explicitly on most user-visible objects in a Vertica database, such as tables and models. For some objects such as projections, privileges are implicitly derived from other objects.

Explicitly granted privileges

The following table provides an overview of privileges that can be explicitly granted on Vertica database objects:

Database Object	Privileges											
	ALTER	DROP	CREATE	DELETE	EXECUTE	INSERT	READ	REFERENCES	SELECT	TEMP	TRUNCATE	UPDATI
Database			•							•		
Schema	!	!	•	!		!		!	!		!	!
Table	•	•		•		•		•	•		•	•
View	•	•							•			
Sequence	•	•							•			
Procedure					•							
User-defined function	•	•			•							
Model	•	•										
Library												

Resource Pool												
Storage Location							•					
Key	•	•										
TLS Configuration	•											

Implicitly granted privileges

Metadata privileges

Superusers have unrestricted access to all non- cryptographic database metadata. For non-superusers, access to the metadata of specific objects depends on their privileges on those objects:

Metadata	**User access**
Catalog objects: • Tables • Columns • Constraints • Sequences • External procedures • Projections • ROS containers	Users must possess USAGE privilege on the schema and any type of access (SELECT) or modify privilege on the object to see catalog metadata about the object. For internal objects such as projections and ROS containers, which have no access privileges directly associated with them, you must have the requisite privileges on the associated schema and tables to view their metadata. For example, to determine whether a table has any projection data, you must have USAGE on the table schema and SELECT on the table.
User sessions and functions, and system tables related to these sessions	Non-superusers can access information about their own (current) sessions only, using the following functions: • CURRENT_DATABASE • CURRENT_SCHEMA • CURRENT_USER / SESSION_USER • HAS_TABLE_PRIVILEGE

Projection privileges

Projections, which store table data, do not have an owner or privileges directly associated with them. Instead, the privileges to create, access, or alter a projection are derived from the privileges that are set on its anchor tables and respective schemas.

Cryptographic privileges

Unless they have ownership, superusers only have implicit DROP privileges on keys, certificates, and TLS Configurations. This allows superusers to see the existence of these objects in their respective system tables (CRYPTOGRAPHIC_KEYS , CERTIFICATES , and TLS_CONFIGURATIONS) and DROP them, but does not allow them to see the key or certificate texts.

For details on granting additional privileges, see GRANT (key) and GRANT (TLS configuration) .

Granting and revoking privileges

Vertica supports GRANT and REVOKE statements to control user access to database objects—for example, GRANT (schema) and REVOKE (schema) , GRANT (table) and REVOKE (table) , and so on. Typically, a superuser creates users and roles shortly after creating the database, and then uses GRANT statements to assign them privileges.

Where applicable, GRANT statements require USAGE privileges on the object schema. The following users can grant and revoke privileges:

- Superusers: all privileges on all database objects, including the database itself
- Non-superusers: all privileges on objects that they own
- Grantees of privileges that include WITH GRANT OPTION: the same privileges on that object

In the following example, a dbadmin (with superuser privileges) creates user Carol . Subsequent GRANT statements grant Carol schema and table privileges:

- CREATE and USAGE privileges on schema PUBLIC
- SELECT, INSERT, and UPDATE privileges on table public.applog . This GRANT statement also includes WITH GRANT OPTION . This enables Carol to grant the same privileges on this table to other users —in this case, SELECT privileges to user Tom :

```
=> CREATE USER Carol;
CREATE USER
=> GRANT CREATE, USAGE ON SCHEMA PUBLIC to Carol;
GRANT PRIVILEGE
=> GRANT SELECT, INSERT, UPDATE ON TABLE public.applog TO Carol WITH GRANT OPTION;
GRANT PRIVILEGE
=> GRANT SELECT ON TABLE public.applog TO Tom;
GRANT PRIVILEGE
```

In this section

- Superuser privileges
- Schema owner privileges
- Object owner privileges
- Granting privileges
- Revoking privileges
- Privilege ownership chains

Superuser privileges

A Vertica superuser is a database user—by default, named dbadmin —that is automatically created on installation. Vertica superusers have complete and irrevocable authority over database users, privileges, and roles.

Important
Vertica superusers are not the same as Linux superusers with (root) privileges.

Superusers can change the privileges of any user and role, as well as override any privileges that are granted by users with the PSEUDOSUPERUSER role. They can also grant and revoke privileges on any user-owned object, and reassign object ownership.

> **Note**
> A superuser always changes a user's privileges on an object on behalf of the object owner. Thus, the grantor setting in system table V_CATALOG.GRANTS always shows the object owner rather than the superuser who issued the GRANT statement.

Cryptographic privileges

For most catalog objects, superusers have all possible privileges. However, for keys , certificates , and TLS Configurations superusers only get DROP privileges by default and must be granted the other privileges by their owners. For details, see GRANT (key) and GRANT (TLS configuration) .

Superusers can see the existence of all keys , certificates , and TLS Configurations , but they cannot see the text of keys or certificates unless they are granted USAGE privileges.

See also

DBADMIN

Schema owner privileges

The schema owner is typically the user who creates the schema. By default, the schema owner has privileges to create objects within a schema. The owner can also alter the schema: reassign ownership, rename it, and enable or disable inheritance of schema privileges.

Schema ownership does not necessarily grant the owner access to objects in that schema. Access to objects depends on the privileges that are granted on them.

All other users and roles must be explicitly granted access to a schema by its owner or a superuser.

Object owner privileges

The database, along with every object in it, has an owner. The object owner is usually the person who created the object, although a superuser can alter ownership of objects, such as table and sequence.

Object owners must have appropriate schema privilege to access, alter, rename, move or drop any object it owns without any additional privileges.

An object owner can also:

- **Grant privileges on their own object to other users**
  The WITH GRANT OPTION clause specifies that a user can grant the permission to other users. For example, if user Bob creates a table, Bob can grant privileges on that table to users Ted, Alice, and so on.
- **Grant privileges to** roles
  Users who are granted the role gain the privilege.

Granting privileges

As described in Granting and revoking privileges , specific users grant privileges using the GRANT statement with or without the optional WITH GRANT OPTION, which allows the user to grant the same privileges to other users.

- A superuser can grant privileges on all object types to other users.
- A superuser or object owner can grant privileges to roles . Users who have been granted the role then gain the privilege.
- An object owner can grant privileges on the object to other users using the optional WITH GRANT OPTION clause.
- The user needs to have USAGE privilege on schema and appropriate privileges on the object.

When a user grants an explicit list of privileges, such as GRANT INSERT, DELETE, REFERENCES ON applog TO Bob :

- The GRANT statement succeeds only if all the roles are granted successfully. If any grant operation fails, the entire statement rolls back.
- Vertica will return ERROR if the user does not have grant options for the privileges listed.

When a user grants ALL privileges, such as GRANT ALL ON applog TO Bob , the statement always succeeds. Vertica grants all the privileges on which the grantor has the WITH GRANT OPTION and skips those privileges without the optional WITH GRANT OPTION.

For example, if the user Bob has delete privileges with the optional grant option on the applog table, only DELETE privileges are granted to Bob, and the statement succeeds:

```
=> GRANT DELETE ON applog TO Bob WITH GRANT OPTION;GRANT PRIVILEGE
```

For details, see the GRANT statements .

Revoking privileges

The following non-superusers can revoke privileges on an object:

- Object owner
- Grantor of the object privileges

The user also must have USAGE privilege on the object's schema.

For example, the following query on system table V_CATALOG.GRANTS shows that users u1 , u2 , and u3 have the following privileges on schema s1 and table s1.t1 :

```
=> SELECT object_type, object_name, grantee, grantor, privileges_description FROM v_catalog.grants
   WHERE object_name IN ('s1', 't1') AND grantee IN ('u1', 'u2', 'u3');
object_type | object_name | grantee | grantor |  privileges_description
------------+-------------+---------+---------+--------------------------
 SCHEMA     | s1          | u1      | dbadmin | USAGE, CREATE
 SCHEMA     | s1          | u2      | dbadmin | USAGE, CREATE
 SCHEMA     | s1          | u3      | dbadmin | USAGE
 TABLE      | t1          | u1      | dbadmin | INSERT*, SELECT*, UPDATE*
 TABLE      | t1          | u2      | u1      | INSERT*, SELECT*, UPDATE*
 TABLE      | t1          | u3      | u2      | SELECT*
(6 rows)
```

> **Note**
> The asterisks (*) on privileges under privileges_description indicate that the grantee can grant these privileges to other users.

In the following statements, u2 revokes the SELECT privileges that it granted on s1.t1 to u3 . Subsequent attempts by u3 to query this table return an error:

```
=> \c - u2
You are now connected as user "u2".
=> REVOKE SELECT ON s1.t1 FROM u3;
REVOKE PRIVILEGE
=> \c - u3
You are now connected as user "u2".
=> SELECT * FROM s1.t1;
ERROR 4367:  Permission denied for relation t1
```

Revoking grant option

If you revoke privileges on an object from a user, that user can no longer act as grantor of those same privileges to other users. If that user previously granted the revoked privileges to other users, the REVOKE statement must include the CASCADE option to revoke the privilege from those users too; otherwise, it returns with an error.

For example, user u2 can grant SELECT, INSERT, and UPDATE privileges, and grants those privileges to user u4 :

```
=> \c - u2
You are now connected as user "u2".
=> GRANT SELECT, INSERT, UPDATE on TABLE s1.t1 to u4;
GRANT PRIVILEGE
```

If you query V_CATALOG.GRANTS for privileges on table s1.t1 , it returns the following result set:

```
=> \ c
You are now connected as user "dbadmin".
=> SELECT object_type, object_name, grantee, grantor, privileges_description FROM v_catalog.grants
    WHERE object_name IN ('t1') ORDER BY grantee;
 object_type | object_name | grantee | grantor |                privileges_description
-------------+-------------+---------+---------+---------------------------------------------------------------
 TABLE       | t1          | dbadmin | dbadmin | INSERT*, SELECT*, UPDATE*, DELETE*, REFERENCES*, TRUNCATE*
 TABLE       | t1          | u1      | dbadmin | INSERT*, SELECT*, UPDATE*
 TABLE       | t1          | u2      | u1      | INSERT*, SELECT*, UPDATE*
 TABLE       | t1          | u4      | u2      | INSERT, SELECT, UPDATE
(3 rows)
```

Now, if user u1 wants to revoke UPDATE privileges from user u2 , the revoke operation must cascade to user u4 , who also has UPDATE privileges that were granted by u2 ; otherwise, the REVOKE statement returns with an error:

```
=> \c - u1
=> REVOKE update ON TABLE s1.t1 FROM u2;
ROLLBACK 3052:  Dependent privileges exist
HINT:  Use CASCADE to revoke them too
=> REVOKE update ON TABLE s1.t1 FROM u2 CASCADE;
REVOKE PRIVILEGE
=> \c
You are now connected as user "dbadmin".
=>  SELECT object_type, object_name, grantee, grantor, privileges_description FROM v_catalog.grants
    WHERE object_name IN ('t1') ORDER BY grantee;
 object_type | object_name | grantee | grantor |                privileges_description
-------------+-------------+---------+---------+---------------------------------------------------------------
 TABLE       | t1          | dbadmin | dbadmin | INSERT*, SELECT*, UPDATE*, DELETE*, REFERENCES*, TRUNCATE*
 TABLE       | t1          | u1      | dbadmin | INSERT*, SELECT*, UPDATE*
 TABLE       | t1          | u2      | u1      | INSERT*, SELECT*
 TABLE       | t1          | u4      | u2      | INSERT, SELECT
(4 rows)
```

You can also revoke grantor privileges from a user without revoking those privileges. For example, user u1 can prevent user u2 from granting INSERT privileges to other users, but allow user u2 to retain that privilege:

```
=> \c - u1
You are now connected as user "u1".
=> REVOKE GRANT OPTION FOR INSERT ON TABLE s1.t1 FROM U2 CASCADE;
REVOKE PRIVILEGE
```

Note

The REVOKE statement must include the CASCADE, because user u2 previously granted user u4 INSERT privileges on table s1.t1 . When you revoke u2 's ability to grant this privilege, that privilege must be removed from any its grantees—in this case, user u4 .

You can confirm results of the revoke operation by querying V_CATALOG.GRANTS for privileges on table s1.t1 :

```
=> \c
You are now connected as user "dbadmin".
=> SELECT object_type, object_name, grantee, grantor, privileges_description FROM v_catalog.grants
    WHERE object_name IN ('t1') ORDER BY grantee;
 object_type | object_name | grantee | grantor |                privileges_description
-------------+-------------+---------+---------+---------------------------------------------------------
 TABLE       | t1          | dbadmin | dbadmin | INSERT*, SELECT*, UPDATE*, DELETE*, REFERENCES*, TRUNCATE*
 TABLE       | t1          | u1      | dbadmin | INSERT*, SELECT*, UPDATE*
 TABLE       | t1          | u2      | u1      | INSERT, SELECT*
 TABLE       | t1          | u4      | u2      | SELECT
(4 rows)
```

The query results show:

- User u2 retains INSERT privileges on the table but can no longer grant INSERT privileges to other users (as indicated by absence of an asterisk).
- The revoke operation cascaded down to grantee u4 , who now lacks INSERT privileges.

See also

REVOKE (table)

Privilege ownership chains

The ability to revoke privileges on objects can cascade throughout an organization. If the grant option was revoked from a user, the privilege that this user granted to other users will also be revoked.

If a privilege was granted to a user or role by multiple grantors, to completely revoke this privilege from the grantee the privilege has to be revoked by each original grantor. The only exception is a superuser may revoke privileges granted by an object owner, with the reverse being true, as well.

In the following example, the SELECT privilege on table t1 is granted through a chain of users, from a superuser through User3.

- A superuser grants User1 CREATE privileges on the schema s1:

  ```
  => \c - dbadmin
  You are now connected as user "dbadmin".
  => CREATE USER User1;
  CREATE USER
  => CREATE USER User2;
  CREATE USER
  => CREATE USER User3;
  CREATE USER
  => CREATE SCHEMA s1;
  CREATE SCHEMA
  => GRANT USAGE on SCHEMA s1 TO User1, User2, User3;
  GRANT PRIVILEGE
  => CREATE ROLE reviewer;
  CREATE ROLE
  => GRANT CREATE ON SCHEMA s1 TO User1;
  GRANT PRIVILEGE
  ```

- User1 creates new table t1 within schema s1 and then grants SELECT WITH GRANT OPTION privilege on s1.t1 to User2:

```
=> \c - User1
You are now connected as user "User1".
=> CREATE TABLE s1.t1(id int, sourceID VARCHAR(8));
CREATE TABLE
=> GRANT SELECT on s1.t1 to User2 WITH GRANT OPTION;
GRANT PRIVILEGE
```

- User2 grants SELECT WITH GRANT OPTION privilege on s1.t1 to User3:

```
=> \c - User2
You are now connected as user "User2".
=> GRANT SELECT on s1.t1 to User3 WITH GRANT OPTION;
GRANT PRIVILEGE
```

- User3 grants SELECT privilege on s1.t1 to the reviewer role:

```
=> \c - User3
You are now connected as user "User3".
=> GRANT SELECT on s1.t1 to reviewer;
GRANT PRIVILEGE
```

Users cannot revoke privileges upstream in the chain. For example, User2 did not grant privileges on User1, so when User1 runs the following REVOKE command, Vertica rolls back the command:

```
=> \c - User2
You are now connected as user "User2".
=> REVOKE CREATE ON SCHEMA s1 FROM User1;
ROLLBACK 0:  "CREATE" privilege(s) for schema "s1" could not be revoked from "User1"
```

Users can revoke privileges indirectly from users who received privileges through a cascading chain, like the one shown in the example above. Here, users can use the CASCADE option to revoke privileges from all users "downstream" in the chain. A superuser or User1 can use the CASCADE option to revoke the SELECT privilege on table s1.t1 from all users. For example, a superuser or User1 can execute the following statement to revoke the SELECT privilege from all users and roles within the chain:

```
=> \c - User1
You are now connected as user "User1".
=> REVOKE SELECT ON s1.t1 FROM User2 CASCADE;
REVOKE PRIVILEGE
```

When a superuser or User1 executes the above statement, the SELECT privilege on table s1.t1 is revoked from User2, User3, and the reviewer role. The GRANT privilege is also revoked from User2 and User3, which a superuser can verify by querying the V_CATALOG.GRANTS system table.

```
=> SELECT * FROM grants WHERE object_name = 's1' AND grantee ILIKE 'User%';
 grantor | privileges_description | object_schema | object_name | grantee
---------+------------------------+---------------+-------------+---------
 dbadmin | USAGE                  |               | s1          | User1
 dbadmin | USAGE                  |               | s1          | User2
 dbadmin | USAGE                  |               | s1          | User3
(3 rows)
```

Modifying privileges

A superuser or object owner can use one of the ALTER statements to modify a privilege, such as changing a sequence owner or table owner . Reassignment to the new owner does not transfer grants from the original owner to the new owner; grants made by the original owner are dropped.

Viewing privileges granted on objects

You can view information about privileges, grantors, grantees, and objects by querying these system tables:

- GRANTS
- INHERITED_PRIVILEGES
- INHERITING_OBJECTS

An asterisk (*) appended to a privilege indicates that the user can grant the privilege to other users.

You can also view the effective privileges on a specified database object by using the GET_PRIVILEGES_DESCRIPTION meta-function.

Viewing explicitly granted privileges

To view explicitly granted privileges on objects, query the GRANTS table.

The following query returns the explicit privileges for the schema, myschema.

```
=> SELECT grantee, privileges_description FROM grants WHERE object_name='myschema';
 grantee | privileges_description
---------+------------------------
 Bob     | USAGE, CREATE
 Alice   | CREATE
(2 rows)
```

Viewing inherited privileges

To view which tables and views inherit privileges from which schemas, query the INHERITING_OBJECTS table.

The following query returns the tables and views that inherit their privileges from their parent schema, customers.

```
=> SELECT * FROM inheriting_objects WHERE object_schema='customers';
    object_id     |     schema_id     | object_schema |  object_name  | object_type
------------------+-------------------+---------------+---------------+-------------
 45035996273980908 | 45035996273980902 | customers     | cust_info     | table
 45035996273980984 | 45035996273980902 | customers     | shipping_info | table
 45035996273980980 | 45035996273980902 | customers     | cust_set      | view
(3 rows)
```

To view the specific privileges inherited by tables and views and information on their associated grant statements, query the INHERITED_PRIVILEGES table.

The following query returns the privileges that the tables and views inherit from their parent schema, customers.

```
=> SELECT object_schema,object_name,object_type,privileges_description,principal,grantor FROM inherited_privileges WHERE
object_schema='customers';
 object_schema |  object_name  | object_type |                    privileges_description                    | principal | grantor
---------------+---------------+-------------+--------------------------------------------------------------+-----------+---------
 customers     | cust_info     | Table       | INSERT*, SELECT*, UPDATE*, DELETE*, ALTER*, REFERENCES*, DROP*, TRUNCATE* | dbadmin   | dbadmin
 customers     | shipping_info | Table       | INSERT*, SELECT*, UPDATE*, DELETE*, ALTER*, REFERENCES*, DROP*, TRUNCATE* | dbadmin   | dbadmin
 customers     | cust_set      | View        | SELECT*, ALTER*, DROP*                                       | dbadmin   | dbadmin
 customers     | cust_info     | Table       | SELECT                                                       | Val       | dbadmin
 customers     | shipping_info | Table       | SELECT                                                       | Val       | dbadmin
 customers     | cust_set      | View        | SELECT                                                       | Val       | dbadmin
 customers     | cust_info     | Table       | INSERT                                                       | Pooja     | dbadmin
 customers     | shipping_info | Table       | INSERT                                                       | Pooja     | dbadmin
(8 rows)
```

Viewing effective privileges on an object

To view the current user's effective privileges on a specified database object, user the GET_PRIVILEGES_DESCRIPTION meta-function.

In the following example, user Glenn has set the REPORTER role and wants to check his effective privileges on schema s1 and table s1.articles .

- Table s1.articles inherits privileges from its schema (s1).
- The REPORTER role has the following privileges:
  - SELECT on schema s1
  - INSERT WITH GRANT OPTION on table s1.articles
- User Glenn has the following privileges:
  - UPDATE and USAGE on schema s1 .
  - DELETE on table s1.articles .

GET_PRIVILEGES_DESCRIPTION returns the following effective privileges for Glenn on schema s1 :

```
=> SELECT GET_PRIVILEGES_DESCRIPTION('schema', 's1');
  GET_PRIVILEGES_DESCRIPTION
--------------------------------
 SELECT, UPDATE, USAGE
(1 row)
```

GET_PRIVILEGES_DESCRIPTION returns the following effective privileges for Glenn on table s1.articles :

```
=> SELECT GET_PRIVILEGES_DESCRIPTION('table', 's1.articles');
  GET_PRIVILEGES_DESCRIPTION
-------------------------------
 INSERT*, SELECT, UPDATE, DELETE
(1 row)
```

See also

- Inherited privileges
- Database users and privileges

Access policies

CREATE ACCESS POLICY lets you create access policies on tables that specify how much data certain users and roles can query from those tables. Access policies typically prevent these users from viewing the data of specific columns and rows of a table. You can apply access policies to table columns and rows . If a table has access policies on both, Vertica filters row access policies first, then filters the column access policies.

You can create most access policies for any table type—columnar, external, or flex . (You cannot create column access policies on flex tables.) You can also create access policies on any column type, including joins.

In this section

- Creating column access policies
- Creating row access policies
- Access policies and DML operations
- Access policies and query optimization
- Managing access policies

Creating column access policies

CREATE ACCESS POLICY can create access policies on individual table columns, one policy per column. Each column access policy lets you specify, for different users and roles, various levels of access to the data of that column. The column access expression can also specify how to render column data for users and roles.

The following example creates an access policy on the customer_address column in the client_dimension table. This access policy gives non-superusers with the administrator role full access to all data in that column, but masks customer address data from all other users:

```
=> CREATE ACCESS POLICY ON public.customer_dimension FOR COLUMN customer_address
-> CASE
-> WHEN ENABLED_ROLE('administrator') THEN customer_address
-> ELSE '***************'
-> END ENABLE;
CREATE ACCESS POLICY
```

> **Note**
> Vertica roles are compatible with LDAP users. You do not need separate LDAP roles to use column access policies with LDAP users.

Vertica uses this policy to determine the access it gives to users MaxineT and MikeL, who are assigned employee and administrator roles, respectively. When these users query the customer_dimension table, Vertica applies the column access policy expression as follows:

```
=> \c - MaxineT;
You are now connected as user "MaxineT".
=> SET ROLE employee;
SET
=> SELECT customer_type, customer_name, customer_gender, customer_address, customer_city FROM customer_dimension;
 customer_type |      customer_name      | customer_gender | customer_address |  customer_city
---------------+-------------------------+-----------------+------------------+-----------------
 Individual    | Craig S. Robinson       | Male            | **************   | Fayetteville
 Individual    | Mark M. Kramer          | Male            | **************   | Joliet
 Individual    | Barbara S. Farmer       | Female          | **************   | Alexandria
 Individual    | Julie S. McNulty        | Female          | **************   | Grand Prairie
 ...

=> \c - MikeL
You are now connected as user "MikeL".
=> SET ROLE administrator;
SET
=> SELECT customer_type, customer_name, customer_gender, customer_address, customer_city FROM customer_dimension;
 customer_type |      customer_name      | customer_gender | customer_address |  customer_city
---------------+-------------------------+-----------------+------------------+-----------------
 Individual    | Craig S. Robinson       | Male            | 138 Alden Ave    | Fayetteville
 Individual    | Mark M. Kramer          | Male            | 311 Green St     | Joliet
 Individual    | Barbara S. Farmer       | Female          | 256 Cherry St    | Alexandria
 Individual    | Julie S. McNulty        | Female          | 459 Essex St     | Grand Prairie
 ...
```

Restrictions

The following limitations apply to access policies:

- A column can have only one access policy.
- Column access policies cannot be set on columns of complex types other than native arrays.
- Column access policies cannot be set for materialized columns on flex tables. While it is possible to set an access policy for the __raw__ column, doing so restricts access to the whole table.
- Row access policies are invalid on temporary tables and tables with aggregate projections.
- Access policy expressions cannot contain:
  - Subqueries
  - Aggregate functions
  - Analytic functions
  - User-defined transform functions (UDTF)
- If the query optimizer cannot replace a deterministic expression that involves only constants with their computed values, it blocks all DML operations such as INSERT .

Creating row access policies

CREATE ACCESS POLICY can create a single row access policy for a given table. This policy lets you specify for different users and roles various levels of access to table row data. When a user launches a query, Vertica evaluates the access policy's WHERE expression against all table rows. The query returns with only those rows where the expression evaluates to true for the current user or role.

For example, you might want to specify different levels of access to table store.store_store_sales for four roles:

- employee : Users with this role should only access sales records that identify them as the employee, in column employee_key . The following query shows how many sales records (in store.store_sales_fact) are associated with each user (in public.emp_dimension):

```
=> SELECT COUNT(sf.employee_key) AS 'Total Sales', sf.employee_key, ed.user_name FROM store.store_sales_fact sf
    JOIN emp_dimension ed ON sf.employee_key=ed.employee_key
    WHERE ed.job_title='Sales Associate' GROUP BY sf.employee_key, ed.user_name ORDER BY sf.employee_key

 Total Sales | employee_key |  user_name
-------------+--------------+-------------
         533 |          111 | LucasLC
         442 |          124 | JohnSN
         487 |          127 | SamNS
         477 |          132 | MeghanMD
         545 |          140 | HaroldON
         ...
         563 |         1991 | MidoriMG
         367 |         1993 | ThomZM
(318 rows)
```

- regional_manager : Users with this role (public.emp_dimension) should only access sales records for the sales region that they manage (store.store_dimension):

```
=> SELECT distinct sd.store_region, ed.user_name, ed.employee_key, ed.job_title FROM store.store_dimension sd
    JOIN emp_dimension ed ON sd.store_region=ed.employee_region WHERE ed.job_title = 'Regional Manager';
 store_region | user_name | employee_key |    job_title
--------------+-----------+--------------+------------------
 West         | JamesGD   |         1070 | Regional Manager
 South        | SharonDM  |         1710 | Regional Manager
 East         | BenOV     |          593 | Regional Manager
 MidWest      | LilyCP    |          611 | Regional Manager
 NorthWest    | CarlaTG   |         1058 | Regional Manager
 SouthWest    | MarcusNK  |          150 | Regional Manager
(6 rows)
```

- dbadmin and administrator : Users with these roles have unlimited access to all table data.

Given these users and the data associated with them, you can create a row access policy on store.store_store_sales that looks like this:

```
CREATE ACCESS POLICY ON store.store_sales_fact FOR ROWS WHERE
  (ENABLED_ROLE('employee')) AND (store.store_sales_fact.employee_key IN
    (SELECT employee_key FROM public.emp_dimension WHERE user_name=CURRENT_USER()))
  OR
  (ENABLED_ROLE('regional_manager')) AND (store.store_sales_fact.store_key IN
    (SELECT sd.store_key FROM store.store_dimension sd
    JOIN emp_dimension ed ON sd.store_region=ed.employee_region WHERE ed.user_name = CURRENT_USER()))
  OR ENABLED_ROLE('dbadmin')
  OR ENABLED_ROLE ('administrator')
ENABLE;
```

Important
In this example, the row policy limits access to a set of roles that are explicitly included in policy's WHERE expression. All other roles and users are implicitly denied access to the table data.

The following examples indicate the different levels of access that are available to users with the specified roles:

- dbadmin has access to all rows in store.store_sales_fact :

```
=> \c
You are now connected as user "dbadmin".
=> SELECT count(*) FROM store.store_sales_fact;
  count
---------
 5000000
(1 row)
```

- User LilyCP has the role of regional_manager , so she can access all sales data of the Midwest region that she manages:

```
=> \c - LilyCP;
You are now connected as user "LilyCP".
=> SET ROLE regional_manager;
SET
=> SELECT count(*) FROM store.store_sales_fact;
 count
--------
 782272
(1 row)
```

- User SamRJ has the role of employee , so he can access only the sales data that he is associated with:

```
=> \c - SamRJ;
You are now connected as user "SamRJ".
=> SET ROLE employee;
SET
=> SELECT count(*) FROM store.store_sales_fact;
 count
-------
   417
(1 row)
```

Restrictions

The following limitations apply to row access policies:

- A table can have only one row access policy.
- Row access policies are invalid on the following tables:
  - Tables with aggregate projections
  - Temporary tables
  - System tables
  - Views
- You cannot create directed queries on a table with a row access policy.

Access policies and DML operations

By default, Vertica abides by a rule that a user can only edit what they can see. That is, you must be able to view all rows and columns in the table in their original values (as stored in the table) and in their originally defined data types to perform actions that modify data on a table. For example, if a column is defined as VARCHAR(9) and an access policy on that column specifies the same column as VARCHAR(10), users using the access policy will be unable to perform the following operations:

- INSERT
- UPDATE
- DELETE
- MERGE
- COPY

You can override this behavior by specifying GRANT TRUSTED in a new or existing access policy. This option forces the access policy to defer entirely to explicit GRANT statements when assessing whether a user can perform the above operations.

You can view existing access policies with the ACCESS_POLICY system table.

Row access

On tables where a row access policy is enabled, you can only perform DML operations when the condition in the row access policy evaluates to TRUE. For example:

t1 appears as follows:

```
 A | B
---+---
 1 | 1
 2 | 2
 3 | 3
```

Create the following row access policy on t1:

```
=> CREATE ACCESS POLICY ON t1 for ROWS
WHERE enabled_role('manager')
OR
A<2
ENABLE;
```

With this policy enabled, the following behavior exists for users who want to perform DML operations:

- A user with the manager role can perform DML on all rows in the table, because the WHERE clause in the policy evaluates to TRUE.
- Users with non-manager roles can only perform a SELECT to return data in column A that has a value of less than two. If the access policy has to read the data in the table to confirm a condition, it does not allow DML operations.

Column access

On tables where a column access policy is enabled, you can perform DML operations if you can view the entire column in its originally defined type.

Suppose table t1 is created with the following data types and values:

```
=> CREATE TABLE t1 (A int, B int);
=> INSERT INTO t1 VALUES (1,2);
=> SELECT * FROM t1;
 A | B
---+---
 1 | 2
(1 row)
```

Suppose the following access policy is created, which coerces the data type of column A from INT to VARCHAR(20) at execution time.

```
=> CREATE ACCESS POLICY on t1 FOR column A A::VARCHAR(20) ENABLE;
Column "A" is of type int but expression in Access Policy is of type varchar(20). It will be coerced at execution time
```

In this case, u1 can view column A in its entirety, but because the active access policy doesn't specify column A's original data type, u1 cannot perform DML operations on column A.

```
=> \c - u1
You are now connected as user "u1".
=> SELECT A FROM t1;
 A
---
 1
(1 row)

=> INSERT INTO t1 VALUES (3);
ERROR 6538:  Unable to INSERT: "Access denied due to active access policy on table "t1" for column "A""
```

Overriding default behavior with GRANT TRUSTED

Specifying GRANT TRUSTED in an access policy overrides the default behavior ("users can only edit what they can see") and instructs the access policy to defer entirely to explicit GRANT statements when assessing whether a user can perform a DML operation.

Important

GRANT TRUSTED can allow users to make changes to tables even if the data is obscured by the access policy. In these cases, certain users who are constrained by the access policy but also have GRANTs on the table can make changes to the data that they cannot view or verify.

GRANT TRUSTED is useful in cases where the form the data is stored in doesn't match its semantically "true" form.

For example, when integrating with Voltage SecureData , a common use case is storing encrypted data with VoltageSecureProtect , where decryption is left to a case expression in an access policy that calls VoltageSecureAccess . In this case, while the decrypted form is intuitively understood to be the data's "true" form, it's still stored in the table in its encrypted form; users who can view the decrypted data wouldn't see the data as it was stored and therefore wouldn't be able to perform DML operations. You can use GRANT TRUSTED to override this behavior and allow users to perform these operations if they have the grants.

In this example, the customer_info table contains columns for the customer first and last name and SSN. SSNs are sensitive and access to it should be controlled, so it is encrypted with VoltageSecureProtect as it is inserted into the table:

```
=> CREATE TABLE customer_info(first_name VARCHAR, last_name VARCHAR, ssn VARCHAR);
=> INSERT INTO customer_info SELECT 'Alice', 'Smith', VoltageSecureProtect('998-42-4910' USING PARAMETERS format='ssn');
=> INSERT INTO customer_info SELECT 'Robert', 'Eve', VoltageSecureProtect('899-28-1303' USING PARAMETERS format='ssn');
=> SELECT * FROM customer_info;
 first_name | last_name |     ssn
------------+-----------+-------------
 Alice      | Smith     | 967-63-8030
 Robert     | Eve       | 486-41-3371
(2 rows)
```

In this system, the role "trusted_ssn" identifies privileged users for which Vertica will decrypt the values of the "ssn" column with VoltageSecureAccess. To allow these privileged users to perform DML operations for which they have grants, you might use the following access policy:

```
=> CREATE ACCESS POLICY ON customer_info FOR COLUMN ssn
 CASE WHEN enabled_role('trusted_ssn') THEN VoltageSecureAccess(ssn USING PARAMETERS format='ssn')
 ELSE ssn END
 GRANT TRUSTED
 ENABLE;
```

Again, note that GRANT TRUSTED allows *all* users with GRANTs on the table to perform the specified operations, including users without the "trusted_ssn" role.

Access policies and query optimization

Access policies affect the projection designs that the Vertica Database Designer produces, and the plans that the optimizer creates for query execution.

Projection designs

When Database Designer creates projections for a given table, it takes into account access policies that apply to the current user. The set of projections that Database Designer produces for the table are optimized for that user's access privileges, and other users with similar access privileges. However, these projections might be less than optimal for users with different access privileges. These differences might have some effect on how efficiently Vertica processes queries for the second group of users. When you evaluate projection designs for a table, choose a design that optimizes access for all authorized users.

Query rewrite

The Vertica optimizer enforces access policies by rewriting user queries in its query plan, which can affect query performance. For example, the clients table has row and column access policies, both enabled. When a user queries this table, the query optimizer produces a plan that rewrites the query so it includes both policies:

```
=> SELECT * FROM clients;
```

The query optimizer produces a query plan that rewrites the query as follows:

```
SELECT * FROM (
SELECT custID, password, CASE WHEN enabled_role('manager') THEN SSN ELSE substr(SSN, 8, 4) END AS SSN FROM clients
WHERE enabled_role('broker') AND
  clients.clientID IN (SELECT brokers.clientID FROM brokers WHERE broker_name = CURRENT_USER())
) clients;
```

Managing access policies

By default, you can only manage access policies on tables that you own. You can optionally restrict access policy management to superusers with the AccessPolicyManagementSuperuserOnly parameter (false by default):

```
=> ALTER DATABASE DEFAULT SET PARAMETER AccessPolicyManagementSuperuserOnly = 1;
ALTER DATABASE
```

You can view and manage access policies for tables in several ways:

- View access policies
- Modify the expression of an access policy
- Enable or disable access policies
- Copy access policies to another table

Viewing access policies

You can view access policies in two ways:

- Query system table ACCESS_POLICY . For example, the following query returns all access policies on table public.customer_dimension :

```
=> \x
=> SELECT policy_type, is_policy_enabled, table_name, column_name, expression FROM access_policy WHERE table_name =
'public.customer_dimension';
-[ RECORD 1 ]-----+-----------------------------------------------------------------------------------------
policy_type       | Column Policy
is_policy_enabled | Enabled
table_name        | public.customer_dimension
column_name       | customer_address
expression        | CASE WHEN enabled_role('administrator') THEN customer_address ELSE '**************' END
```

- Export table DDL from the database catalog with EXPORT_TABLES , EXPORT_OBJECTS , or EXPORT_CATALOG . For example:

```
=> SELECT export_tables('','customer_dimension');
                          export_tables
-------------------------------------------------------------------------------
CREATE TABLE public.customer_dimension
(
   customer_key int NOT NULL,
   customer_type varchar(16),
   customer_name varchar(256),
   customer_gender varchar(8),
   ...
   CONSTRAINT C_PRIMARY PRIMARY KEY (customer_key) DISABLED
);

CREATE ACCESS POLICY ON public.customer_dimension FOR COLUMN customer_address CASE WHEN enabled_role('administrator') THEN
customer_address ELSE '**************' END ENABLE;
```

Modifying access policy expression

ALTER ACCESS POLICY can modify the expression of an existing access policy. For example, you can modify the access policy in the earlier example by extending access to the dbadmin role:

```
=> ALTER ACCESS POLICY ON public.customer_dimension FOR COLUMN customer_address
   CASE WHEN enabled_role('dbadmin') THEN customer_address
      WHEN enabled_role('administrator') THEN customer_address
      ELSE '**************' END ENABLE;
ALTER ACCESS POLICY
```

Querying system table ACCESS_POLICY confirms this change:

```
=> SELECT policy_type, is_policy_enabled, table_name, column_name, expression FROM access_policy
  WHERE table_name = 'public.customer_dimension' AND column_name='customer_address';
-[ RECORD 1 ]-----+-----------------------------------------------------------------------------------------------------------------------------------------
policy_type       | Column Policy
is_policy_enabled | Enabled
table_name        | public.customer_dimension
column_name       | customer_address
expression        | CASE WHEN enabled_role('dbadmin') THEN customer_address WHEN enabled_role('administrator') THEN customer_address ELSE
'**************' END
```

Enabling and disabling access policies

Owners of a table can enable and disable its row and column access policies.

Row access policies

You enable and disable row access policies on a table:

```
ALTER ACCESS POLICY ON [schema.]table FOR ROWS { ENABLE | DISABLE }
```

The following examples disable and then re-enable the row access policy on table customer_dimension :

```
=> ALTER ACCESS POLICY ON customer_dimension FOR ROWS DISABLE;
ALTER ACCESS POLICY
=> ALTER ACCESS POLICY ON customer_dimension FOR ROWS ENABLE;
ALTER ACCESS POLICY
```

Column access policies

You enable and disable access policies on a table column as follows:

```
ALTER ACCESS POLICY ON [schema.]table FOR COLUMN column { ENABLE | DISABLE }
```

The following examples disable and then re-enable the same column access policy on customer_dimension.customer_address :

```
=> ALTER ACCESS POLICY ON public.customer_dimension FOR COLUMN customer_address DISABLE;
ALTER ACCESS POLICY
=> ALTER ACCESS POLICY ON public.customer_dimension FOR COLUMN customer_address ENABLE;
ALTER ACCESS POLICY
```

Copying access polices

You copy access policies from one table to another as follows. Non-superusers must have ownership of both the source and destination tables:

```
ALTER ACCESS POLICY ON [schema.]table { FOR COLUMN column | FOR ROWS } COPY TO TABLE table
```

When you create a copy of a table or move its contents with the following functions (but not CREATE TABLE AS SELECT or CREATE TABLE LIKE), the access policies of the original table are copied to the new/destination table:

- COPY_TABLE
- COPY_PARTITIONS_TO_TABLE
- MOVE_PARTITIONS_TO_TABLE
- SWAP_PARTITIONS_BETWEEN_TABLES

To copy access policies to another table, use ALTER ACCESS POLICY .

> **Note**
> If you rename a table with ALTER TABLE...RENAME TO , the access policies that were stored under the previous name are stored under the table's new name.

For example, you can copy a row access policy as follows:

```
=> ALTER ACCESS POLICY ON public.emp_dimension FOR ROWS COPY TO TABLE public.regional_managers_dimension;
```

The following statement copies the access policy on column employee_key from table public.emp_dimension to store.store_sales_fact :

```
=> ALTER ACCESS POLICY ON public.emp_dimension FOR COLUMN employee_key COPY TO TABLE store.store_sales_fact;
```

> **Note**
> The copied policy retains the source policy's enabled/disabled settings.

Using the administration tools

The Vertica Administration tools allow you to easily perform administrative tasks. You can perform most Vertica database administration tasks with Administration Tools.

Run Administration Tools using the Database Superuser account on the Administration host , if possible. Make sure that no other Administration Tools processes are running.

If the Administration host is unresponsive, run Administration Tools on a different node in the cluster. That node permanently takes over the role of Administration host.

In this section

- Running the administration tools
- First login as database administrator

- Using the administration tools interface
- Notes for remote terminal users
- Using administration tools help
- Distributing changes made to the administration tools metadata
- Administration tools and Management Console
- Administration tools reference

Running the administration tools

Administration tools, or "admintools," supports various commands to manage your database.

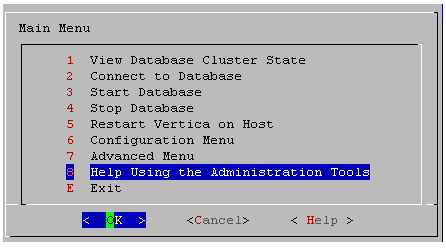
To run admintools, you must have SSH and local connections enabled for the dbadmin user.

Syntax

```
/opt/vertica/bin/admintools [--debug ][
    { -h | --help }
  | { -a | --help_all}
  | { -t | --tool } name_of_tool [options]
]
```

--debug	If you include this option, Vertica logs debug information. **Note** You can specify the debug option with or without naming a specific tool. If you specify debug with a specific tool, Vertica logs debug information during tool execution. If you do not specify a tool, Vertica logs debug information when you run tools through the admintools user interface.
-h --help	Outputs abbreviated help.
-a --help_all	Outputs verbose help, which lists all command-line sub-commands and options.
{ -t | --tool } *name_of_tool* [*options*]	Specifies the tool to run, where *name_of_tool* is one of the tools described in the help output, and *options* are one or more comma-delimited tool arguments. **Note** Enter admintools -h to see the list of tools available. Enter admintools -t *name_of_tool* --help to review a specific tool's options.

An unqualified admintools command displays the Main Menu dialog box.



If you are unfamiliar with this type of interface, read Using the administration tools interface .

Privileges

dbadmin user

First login as database administrator

The first time you log in as the Database Superuser and run the Administration Tools, the user interface displays.

1. In the end-user license agreement (EULA) window, type accept to proceed.
   A window displays, requesting the location of the license key file you downloaded from the Vertica website. The default path is /tmp/vlicense.dat .
2. Type the absolute path to your license key (for example,

Using the administration tools interface

The Vertica Administration Tools are implemented using Dialog, a graphical user interface that works in terminal (character-cell) windows.The interface responds to mouse clicks in some terminal windows, particularly local Linux windows, but you might find that it responds only to keystrokes. Thus, this section describes how to use the Administration Tools using only keystrokes.

> **Note**
> This section does not describe every possible combination of keystrokes you can use to accomplish a particular task. Feel free to experiment and to use whatever keystrokes you prefer.

Enter [return]

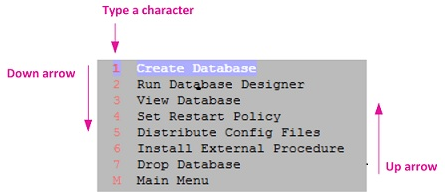
In all dialogs, when you are ready to run a command, select a file, or cancel the dialog, press the **Enter** key. The command descriptions in this section do not explicitly instruct you to press Enter.

OK - cancel - help

The OK, Cancel, and Help buttons are present on virtually all dialogs. Use the tab, space bar, or right and left arrow keys to select an option and then press Enter. The same keystrokes apply to dialogs that present a choice of Yes or No.

Menu dialogs

Some dialogs require that you choose one command from a menu. Type the alphanumeric character shown or use the up and down arrow keys to select a command and then press Enter.



List dialogs

In a list dialog, use the up and down arrow keys to highlight items, then use the space bar to select the items (which marks them with an X). Some list dialogs allow you to select multiple items. When you have finished selecting items, press Enter.

Form dialogs

In a form dialog (also referred to as a dialog box), use the tab key to cycle between **OK** , **Cancel** , **Help** , and the form field area. Once the cursor is in the form field area, use the up and down arrow keys to select an individual field (highlighted) and enter information. When you have finished entering information in all fields, press Enter.

Help buttons

Online help is provided in the form of text dialogs. If you have trouble viewing the help, see Notes for remote terminal users .

Notes for remote terminal users

The appearance of the graphical interface depends on the color and font settings used by your terminal window. The screen captures in this document were made using the default color and font settings in a PuTTy terminal application running on a Windows platform.

> **Note**
> If you are using a remote terminal application, such as PuTTy or a Cygwin bash shell, make sure your window is at least 81 characters wide and 23 characters high.

If you are using PuTTY, you can make the Administration Tools look like the screen captures in this document:

1. In a PuTTY window, right click the title area and select Change Settings.

2. Create or load a saved session.
3. In the Category dialog, click Window > Appearance.
4. In the Font settings, click the Change... button.
5. Select Font: Courier New: Regular Size: 10
6. Click Apply.

Repeat these steps for each existing session that you use to run the Administration Tools.

You can also change the translation to support UTF-8:

1. In a PuTTY window, right click the title area and select Change Settings.
2. Create or load a saved session.
3. In the Category dialog, click Window > Translation.
4. In the "Received data assumed to be in which character set" drop-down menu, select UTF-8.
5. Click Apply.

Using administration tools help

The **Help on Using the Administration Tools** command displays a help screen about using the Administration Tools.

Most of the online help in the Administration Tools is context-sensitive. For example, if you use up/down arrows to select a command, press tab to move to the Help button, and press return, you get help on the selected command.

In a menu dialog

1. Use the up and down arrow keys to choose the command for which you want help.
2. Use the Tab key to move the cursor to the Help button.
3. Press Enter (Return).

In a dialog box

1. Use the up and down arrow keys to choose the field on which you want help.
2. Use the Tab key to move the cursor to the Help button.
3. Press Enter (Return).

Scrolling

Some help files are too long for a single screen. Use the up and down arrow keys to scroll through the text.

Distributing changes made to the administration tools metadata

Administration Tools-specific metadata for a failed node will fall out of synchronization with other cluster nodes if you make the following changes:

- Modify the restart policy
- Add one or more nodes
- Drop one or more nodes.

When you restore the node to the database cluster, you can use the Administration Tools to update the node with the latest Administration Tools metadata:

1. Log on to a host that contains the metadata you want to transfer and start the Administration Tools. (See Using the administration tools .)
2. On the **Main Menu** in the Administration Tools, select **Configuration Menu** and click **OK** .
3. On the **Configuration Menu** , select **Distribute Config Files** and click **OK** .
4. Select **AdminTools Meta-Data** .
   The Administration Tools metadata is distributed to every host in the cluster.
5. Restart the database .

Administration tools and Management Console

You can perform most database administration tasks using the Administration Tools, but you have the additional option of using the more visual and dynamic Management Console .

The following table compares the functionality available in both interfaces. Continue to use Administration Tools and the command line to perform actions not yet supported by Management Console.

Vertica Functionality	Management Console	Administration Tools

Use a Web interface for the administration of Vertica	Yes	No
Manage/monitor one or more databases and clusters through a UI	Yes	No
Manage multiple databases on different clusters	Yes	Yes
View database cluster state	Yes	Yes
View multiple cluster states	Yes	No
Connect to the database	Yes	Yes
Start/stop an existing database	Yes	Yes
Stop/restart Vertica on host	Yes	Yes
Kill a Vertica process on host	No	Yes
Create one or more databases	Yes	Yes
View databases	Yes	Yes
Remove a database from view	Yes	No
Drop a database	Yes	Yes
Create a physical schema design (Database Designer)	Yes	Yes
Modify a physical schema design (Database Designer)	Yes	Yes
Set the restart policy	No	Yes
Roll back database to the Last Good Epoch	No	Yes
Manage clusters (add, replace, remove hosts)	Yes	Yes
Rebalance data across nodes in the database	Yes	Yes
Configure database parameters dynamically	Yes	No
View database activity in relation to physical resource usage	Yes	No
View alerts and messages dynamically	Yes	No
View current database size usage statistics	Yes	No
View database size usage statistics over time	Yes	No
Upload/upgrade a license file	Yes	Yes
Warn users about license violation on login	Yes	Yes
Create, edit, manage, and delete users/user information	Yes	No
Use LDAP to authenticate users with company credentials	Yes	Yes
Manage user access to MC through roles	Yes	No
Map Management Console users to a Vertica database	Yes	No

Enable and disable user access to MC and/or the database	Yes	No
Audit user activity on database	Yes	No
Hide features unavailable to a user through roles	Yes	No
Generate new user (non-LDAP) passwords	Yes	No

Management Console Provides some, but Not All of the Functionality Provided By the Administration Tools. MC Also Provides Functionality Not Available in the Administration Tools.

See also

- Monitoring Vertica Using Management Console

Administration tools reference

Administration Tools, or "admintools," uses the open-source vertica-python client to perform operations on the database.

The follow sections explain in detail all the steps you can perform with Vertica Administration Tools:

In this section

- Viewing database cluster state
- Connecting to the database
- Restarting Vertica on host
- Configuration menu options
- Advanced menu options
- Administration tools connection behavior and requirements
- Writing administration tools scripts

Viewing database cluster state

This tool shows the current state of the nodes in the database.

1. On the Main Menu, select **View Database Cluster State** , and click **OK** .
   The normal state of a running database is ALL UP. The normal state of a stopped database is ALL DOWN.
2. If some hosts are UP and some DOWN, restart the specific host that is down using **Restart Vertica on Host** from the Administration Tools, or you can start the database as described in Starting and Stopping the Database (unless you have a known node failure and want to continue in that state.)
   Nodes shown as INITIALIZING or RECOVERING indicate that Failure recovery is in progress.

Nodes in other states (such as NEEDS_CATCHUP) are transitional and can be ignored unless they persist.

See also

- Advanced menu options

Connecting to the database

This tool connects to a running database with vsql . You can use the Administration Tools to connect to a database from any node within the database while logged in to any user account with access privileges. You cannot use the Administration Tools to connect from a host that is not a database node. To connect from other hosts, run vsql as described in Connecting from the command line .

1. On the Main Menu, click **Connect to Database** , and then click **OK** .
2. Supply the database password if asked:

   ```
   Password:
   ```

   When you create a new user with the CREATE USER command, you can configure the password or leave it empty. You cannot bypass the password if the user was created with a password configured. You can change a user's password using the ALTER USER command.
   The Administration Tools connect to the database and transfer control to vsql .

```
Welcome to vsql, the Vertica Analytic Database interactive terminal.
Type:  \h or \? for help with vsql commands
     \g or terminate with semicolon to execute query
     \q to quit

=>
```

See Using vsql for more information.

> **Note**
> After entering your password, you may be prompted to change your password if it has expired. See Configuring client authentication for details of password security.

See also

- CREATE USER
- ALTER USER

Restarting Vertica on host

This tool restarts the Vertica process on one or more hosts in a running database. Use this tool if the Vertica process stopped or was killed on the host.

1. To view the current state nodes in cluster, on the Main Menu, select **View Database Cluster State** .
2. Click **OK** to return to the Main Menu.
3. If one or more nodes are down, select **Restart Vertica on Host** , and click **OK** .
4. Select the database that contains the host that you want to restart, and click **OK** .
5. Select the one or more hosts to restart, and click **OK** .
6. Enter the database password.
7. Select **View Database Cluster State** again to verify all nodes are up.

Configuration menu options

The Configuration Menu allows you to perform the following tasks:

In this section

- Creating a database
- Dropping a database
- Viewing a database
- Setting the restart policy
- Installing external procedure executable files

Creating a database

Use the procedures below to create either an Enterprise Mode or Eon Mode database with admintools. To create a database with an in-browser wizard in Management Console, see Creating a database using MC . For details about creating a database with admintools through the command line, see Writing administration tools scripts .

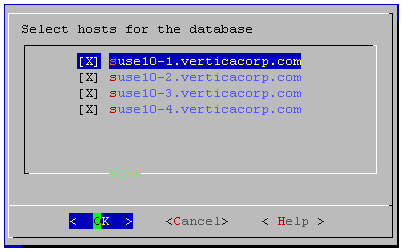
Create an Enterprise Mode database

1. On the Configuration Menu, click Create Database. Click OK.
2. Select Enterprise Mode as your database mode.
3. Enter the name of the database and an optional comment. Click OK.
4. Enter a password. See Creating a database name and password for rules.
   If you do not enter a password, you are prompted to confirm: Yes to enter a superuser password, No to create a database without one.

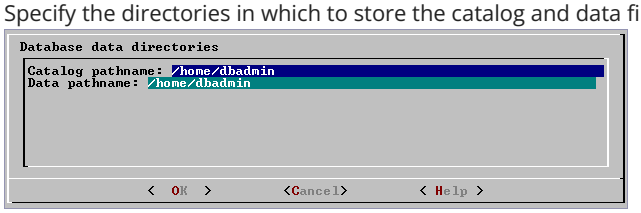
   > **Caution**
   > If you do not enter a password at this point, superuser password is set to empty. Unless the database is for evaluation or academic purposes, Vertica strongly recommends that you enter a superuser password.

5. If you entered a password, enter the password again.

6. Select the hosts to include in the database. The hosts in this list are the ones that were specified at installation time (install_vertica - s).



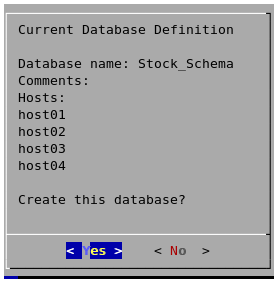
7. Specify the directories in which to store the catalog and data files.



> **Note**
> Catalog and data paths must contain only alphanumeric characters and cannot have leading space characters. Failure to comply with these restrictions could result in database creation failure.

> **Note**
> Do not use a shared directory for more than one node. Data and catalog directories must be distinct for each node. Multiple nodes must not be allowed to write to the same data or catalog directory.

8. Check the current database definition for correctness. Click Yes to proceed.



9. A message indicates that you have successfully created a database. Click OK.

You can also create an Enterprise Mode database using admintools through the command line, for example:

```
$ admintools -t create_db --data_path=/home/dbadmin --catalog_path=/home/dbadmin --database=verticadb --password=password --hosts=localhost
```

For more information, see Writing administration tools scripts .

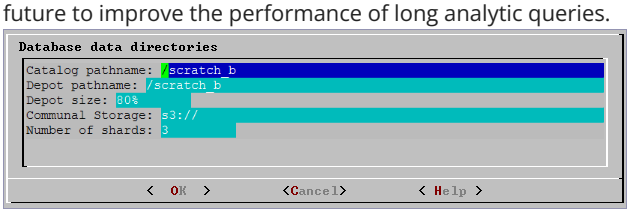
Create an Eon Mode database

> **Note**
> Currently, the admintools menu interface does not support creating an Eon Mode database on Google Cloud Platform. Use the MC or the admintools command line to create an Eon Mode database instead.

1. On the Configuration Menu, click Create Database. Click OK.
2. Select Eon Mode as your database mode.
3. Enter the name of the database and an optional comment. Click OK.
4. Enter a password. See Creating a database name and password for rules.
   AWS only: If you do not enter a password, you are prompted to confirm: Yes to enter a superuser password, No to create a database without one.

> **Caution**
> If you do not enter a password at this point, superuser password is set to empty. Unless the database is for evaluation or academic purposes, Vertica strongly recommends that you enter a superuser password.

5. If you entered a password, enter the password again.
6. Select the hosts to include in the database. The hosts in this list are those specified at installation time (install_vertica -s).
7. Specify the directories in which to store the catalog and depot, depot size, communal storage location, and number of shards.
   - **Depot Size:** Use an integer followed by %, K, G, or T. Default is 60% of the disk total disk space of the filesystem storing the depot.
   - **Communal Storage:** Use an existing Amazon S3 bucket in the same region as your instances. Specify a new subfolder name, which Vertica will dynamically create within the existing S3 bucket. For example, s3://existingbucket/newstorage1 . You can create a new subfolder within existing ones, but database creation will roll back if you do not specify any new subfolder name.
   - **Number of Shards:** Use a whole number. The default is equal to the number of nodes. For optimal performance, the number of shards should be no greater than 2x the number of nodes. When the number of nodes is greater than the number of shards (with ETS), the throughput of dashboard queries improves. When the number of shards exceeds the number of nodes, you can expand the cluster in the future to improve the performance of long analytic queries.



> **Note**
> Catalog and depot paths must contain only alphanumeric characters and cannot have leading space characters. Failure to comply with these restrictions could result in database creation failure.

8. Check the current database definition for correctness. Click Yes to proceed.
9. A message indicates that you successfully created a database. Click OK.

In on-premises, AWS, and Azure environments, you can create an Eon Mode database using admintools through the command line. For instructions specific to your environment, see Create a database in Eon Mode .

Dropping a database

This tool drops an existing database . Only the Database Superuser is allowed to drop a database.

1. Stop the database .
2. On the **Configuration Menu** , click **Drop Database** and then click **OK** .
3. Select the database to drop and click **OK** .
4. Click **Yes** to confirm that you want to drop the database.
5. Type **yes** and click **OK** to reconfirm that you really want to drop the database.
6. A message indicates that you have successfully dropped the database. Click **OK** .

When Vertica drops the database, it also automatically drops the node definitions that refer to the database . The following exceptions apply:

- Another database uses a node definition. If another database refers to any of these node definitions, none of the node definitions are dropped.
- A node definition is the only node defined for the host. (Vertica uses node definitions to locate hosts that are available for database creation, so removing the only node defined for a host would make the host unavailable for new databases.)

Viewing a database

This tool displays the characteristics of an existing database .

1. On the **Configuration Menu** , select **View Database** and click **OK** .
2. Select the database to view.
3. Vertica displays the following information about the database:
   - The name of the database.
   - The name and location of the log file for the database.
   - The hosts within the database cluster.
   - The value of the restart policy setting.
     Note: This setting determines whether nodes within a K-Safe database are restarted when they are rebooted. See Setting the restart policy .
   - The database port.
   - The name and location of the catalog directory.

Setting the restart policy

The Restart Policy enables you to determine whether or not nodes in a K-Safe database are automatically restarted when they are rebooted. Since this feature does not automatically restart nodes if the entire database is DOWN, it is not useful for databases that are not K-Safe.

To set the Restart Policy for a database:

1. Open the Administration Tools.
2. On the Main Menu, select **Configuration Menu** , and click **OK** .
3. In the Configuration Menu, select **Set Restart Policy** , and click **OK** .
4. Select the database for which you want to set the Restart Policy, and click **OK** .
5. Select one of the following policies for the database:
   - **Never** — Nodes are never restarted automatically.
   - **K-Safe** — Nodes are automatically restarted if the database cluster is still UP. This is the default setting.
   - **Always** — Node on a single node database is restarted automatically.

   > **Note**
   > Always does not work if a single node database was not shutdown cleanly or crashed.

6. Click **OK** .

Best practice for restoring failed hardware

Following this procedure will prevent Vertica from misdiagnosing missing disk or bad mounts as data corruptions, which would result in a time-consuming, full-node recovery.

If a server fails due to hardware issues, for example a bad disk or a failed controller, upon repairing the hardware:

1. Reboot the machine into runlevel 1, which is a root and console-only mode.
   Runlevel 1 prevents network connectivity and keeps Vertica from attempting to reconnect to the cluster.
2. In runlevel 1, validate that the hardware has been repaired, the controllers are online, and any RAID recover is able to proceed.

   > **Note**
   > You do not need to initialize RAID recover in runlevel 1; simply validate that it can recover.

3. Once the hardware is confirmed consistent, only then reboot to runlevel 3 or higher.

At this point, the network activates, and Vertica rejoins the cluster and automatically recovers any missing data. Note that, on a single-node database, if any files that were associated with a projection have been deleted or corrupted, Vertica will delete all files associated with that projection, which could result in data loss.

Installing external procedure executable files

1. Run the Administration tools .
   ```
   $ /opt/vertica/bin/adminTools
   ```
2. On the AdminTools **Main Menu** , click **Configuration Menu** , and then click **OK** .
3. On the **Configuration Menu** , click **Install External Procedure** and then click **OK** .
4. Select the database on which you want to install the external procedure.
5. Either select the file to install or manually type the complete file path, and then click **OK** .
6. If you are not the superuser, you are prompted to enter your password and click **OK** .
   The Administration Tools automatically create the *database-name* /procedures directory on each node in the database and installs the external procedure in these directories for you.
7. Click **OK** in the dialog that indicates that the installation was successful.

Advanced menu options

The Advanced Menu options allow you to perform the following tasks:

In this section

- Rolling back the database to the last good epoch

- Stopping Vertica on host
- Killing the Vertica process on host
- Upgrading a Vertica license key
- Managing clusters
- Getting help on administration tools
- Administration tools metadata

Rolling back the database to the last good epoch

Vertica provides the ability to roll the entire database back to a specific epoch primarily to assist in the correction of human errors during data loads or other accidental corruptions. For example, suppose that you have been performing a bulk load and the cluster went down during a particular COPY command. You might want to discard all epochs back to the point at which the previous COPY command committed and run the one that did not finish again. You can determine that point by examining the log files (see Monitoring the Log Files).

1. On the Advanced Menu, select **Roll Back Database to Last Good Epoch** .
2. Select the database to roll back. The database must be stopped.
3. Accept the suggested restart epoch or specify a different one.
4. Confirm that you want to discard the changes after the specified epoch.

The database restarts successfully.

Important

The default value of HistoryRetentionTime is 0, which means that Vertica only keeps historical data when nodes are down. This settings prevents the use of the Administration tools 'Roll Back Database to Last Good Epoch' option because the AHM remains close to the current epoch. Vertica cannot roll back to an epoch that precedes the AHM.

If you rely on the Roll Back option to remove recently loaded data, consider setting a day-wide window for removing loaded data. For example:

```
=> ALTER DATABASE DEFAULT SET HistoryRetentionTime = 86400;
```

Stopping Vertica on host

This command attempts to gracefully shut down the Vertica process on a single node.

> **Caution**
> Do not use this command to shut down the entire cluster. Instead, stop the database to perform a clean shutdown that minimizes data loss.

1. On the Advanced Menu, select **Stop Vertica on Host** and click **OK** .
2. Select the hosts to stop.
3. Confirm that you want to stop the hosts.
   If the command succeeds View Database Cluster State shows that the selected hosts are DOWN.
   If the command fails to stop any selected nodes, proceed to Killing Vertica Process on Host.

Killing the Vertica process on host

This command sends a kill signal to the Vertica process on a node.

> **Caution**
> Use this command only after you tried to stop the database and stop Vertica on a node and both were unsuccessful.

1. On the Advanced menu, select **Kill Vertica Process on Host** and click **OK** .
2. Select the hosts on which to kills the Vertica process.
3. Confirm that you want to stop the processes.
4. If the command succeeds, View Database Cluster State shows that the selected hosts are DOWN.

Upgrading a Vertica license key

The following steps are for licensed Vertica users. Completing the steps copies a license key file into the database. See Managing licenses for more information.

1. On the Advanced menu select **Upgrade License Key** . Click **OK** .
2. Select the database for which to upgrade the license key.
3. Enter the absolute pathname of your downloaded license key file (for example, /tmp/vlicense.dat). Click **OK** .
4. Click OK when you see a message indicating that the upgrade succeeded.

> **Note**
> If you are using Vertica Community Edition, follow the instructions in Vertica license changes for instructions to upgrade to a Vertica Premium Edition license key.

Managing clusters

Cluster Management lets you add, replace, or remove hosts from a database cluster. These processes are usually part of a larger process of adding , removing , or replacing a database node.

> **Note**
> View the database state to verify that it is running. See View Database Cluster State . If the database isn't running, restart it. See Starting the database .

Using cluster management

To use Cluster Management:

1. From the **Main Menu** , select Advanced Menu, and then click **OK** .
2. In the Advanced Menu, select **Cluster Management** , and then click **OK** .
3. Select one of the following, and then click **OK** .
   - **Add Hosts to Database:** See Adding Hosts to a Database .
   - **Re-balance Data:** See Rebalancing Data .
   - **Replace Host:** See Replacing Hosts .
   - **Remove Host from Database:** See Removing Hosts from a Database .

Getting help on administration tools

The **Help Using the Administration Tools** command displays a help screen about using the Administration Tools.

Most of the online help in the Administration Tools is context-sensitive. For example, if you up the use up/down arrows to select a command, press tab to move to the Help button, and press return, you get help on the selected command.

Administration tools metadata

The Administration Tools configuration data (metadata) contains information that databases need to start, such as the hostname/IP address of each participating host in the database cluster.

To facilitate hostname resolution within the Administration Tools, at the command line, and inside the installation utility, Vertica enforces all hostnames you provide through the Administration Tools to use IP addresses:

- **During installation**
  Vertica immediately converts any hostname you provide through command line options --hosts , -add-hosts or --remove-hosts to its IP address equivalent.
  - If you provide a hostname during installation that resolves to multiple IP addresses (such as in multi-homed systems), the installer prompts you to choose one IP address.
  - Vertica retains the name you give for messages and prompts only; internally it stores these hostnames as IP addresses.
- **Within the Administration Tools**
  All hosts are in IP form to allow for direct comparisons (for example db = database = database.example.com).
- **At the command line**
  Vertica converts any hostname value to an IP address that it uses to look up the host in the configuration metadata. If a host has multiple IP addresses that are resolved, Vertica tests each IP address to see if it resides in the metadata, choosing the first match. No match indicates that the host is not part of the database cluster.

Metadata is more portable because Vertica does not require the names of the hosts in the cluster to be exactly the same when you install or upgrade your database.

Administration tools connection behavior and requirements

The behavior of admintools when it connects to and performs operations on a database may vary based on your configuration. In particular, admintools considers its connection to other nodes, the status of those nodes, and the authentication method used by dbadmin.

Connection requirements and authentication

- admintools uses passwordless SSH connections between cluster hosts for most operations, which is configured or confirmed during installation with the install_vertica script
- For most situations, when issuing commands to the database, admintools prefers to uses its SSH connection to a target host and uses a localhost client connection to the Vertica database
- The incoming IP address determines the authentication method used. That is, a client connection may have different behavior from a local connection, which may be trusted by default
- dbadmin should have a local trust or password-based authentication method
- When deciding which host to use for multi-step operations, admintools prefers localhost, and then to reconnect to known-to-be-good nodes

K-safety support

The Administration Tools allow certain operations on a K-Safe database, even if some nodes are unresponsive.

The database must have been marked as K-Safe using the MARK_DESIGN_KSAFE function.

The following management functions within the Administration Tools are operational when some nodes are unresponsive.

> **Note**
> Vertica users can perform much of the below functionality using the Management Console interface. See Management Console and Administration Tools for details.

- View database cluster state
- Connect to database
- Start database (including manual recovery)
- Stop database
- Replace node (assuming node that is down is the one being replaced)
- View database parameters
- Upgrade license key

The following management functions within the Administration Tools require that all nodes be UP in order to be operational:

- Create database
- Run the Database Designer
- Drop database
- Set restart policy
- Roll back database to Last Good Epoch

Writing administration tools scripts

You can invoke most Administration Tools from the command line or a shell script.

Syntax

```
/opt/vertica/bin/admintools {
    { -h | --help }
  | { -a | --help_all}
  | { [--debug ] { -t | --tool } toolname [ tool-args ] }
}
```

> **Note**
> For convenience, add /opt/vertica/bin to your search path.

Parameters

-h -help	Outputs abbreviated help.
-a -help_all	Outputs verbose help, which lists all command-line sub-commands and options.
[debug] { -t | -tool } *toolname* [*args*]	Specifies the tool to run, where *toolname* is one of the tools listed in the help output described below, and *args* is one or more comma-delimited *toolname* arguments. If you include the debug option, Vertica logs debug information during tool execution.

Tools

To return a list of all available tools, enter admintools -h at a command prompt.

> **Note**
> To create a database or password, see Creating a database name and password for naming rules.

To display help for a specific tool and its options or commands, qualify the specified tool name with --help or -h , as shown in the example below:

```
$ admintools -t connect_db --help
Usage: connect_db [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to connect
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
```

To list all available tools and their commands and options in individual help text, enter admintools -a .

```
Usage:
    adminTools [-t | --tool] toolName [options]
Valid tools are:
                command_host
                connect_db
                create_db
                db_add_node
                db_add_subcluster
                db_remove_node
                db_remove_subcluster
                db_replace_node
                db_status
                distribute_config_files
                drop_db
                host_to_node
                install_package
                install_procedure
                kill_host
                kill_node
                license_audit
                list_allnodes
                list_db
                list_host
                list_node
                list_packages
                logrotate
                node_map
                re_ip
                rebalance_data
```

```
            rebalance_data
            restart_db
            restart_node
            restart_subcluster
            return_epoch
            revive_db
            set_restart_policy
            set_ssl_params
            show_active_db
            start_db
            stop_db
            stop_host
            stop_node
            stop_subcluster
            uninstall_package
            upgrade_license_key
            view_cluster
------------------------------------------------------------------------------
Usage: command_host [options]

Options:
  -h, --help            show this help message and exit
  -c CMD, --command=CMD
                        Command to run
  -F, --force           Provide the force cleanup flag. Only applies to start,
                        restart, condrestart. For other options it is ignored.
------------------------------------------------------------------------------
Usage: connect_db [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to connect
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
------------------------------------------------------------------------------
Usage: create_db [options]

Options:
  -h, --help            show this help message and exit
  -D DATA, --data_path=DATA
                        Path of data directory[optional] if not using compat21
  -c CATALOG, --catalog_path=CATALOG
                        Path of catalog directory[optional] if not using
                        compat21
  --compat21            (deprecated) Use Vertica 2.1 method using node names
                        instead of hostnames
  -d DB, --database=DB  Name of database to be created
  -l LICENSEFILE, --license=LICENSEFILE
                        Database license [optional]
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes [optional]
  -P POLICY, --policy=POLICY
                        Database restart policy [optional]
  -s HOSTS, --hosts=HOSTS
                        comma-separated list of hosts to participate in
                        database
  --shard-count=SHARD_COUNT
                        [Eon only] Number of shards in the database
  --communal-storage-location=COMMUNAL_STORAGE_LOCATION
                        [Eon only] Location of communal storage
  -x COMMUNAL_STORAGE_PARAMS, --communal-storage-params=COMMUNAL_STORAGE_PARAMS
                        [Eon only] Location of communal storage parameter file
```

```
                [Eon only] Location of communal storage parameter file
  --depot-path=DEPOT_PATH
                [Eon only] Path to depot directory
  --depot-size=DEPOT_SIZE
                [Eon only] Size of depot
  --force-cleanup-on-failure
                Force removal of existing directories on failure of
                command
  --force-removal-at-creation
                Force removal of existing directories before creating
                the database
  --timeout=NONINTERACTIVE_TIMEOUT
                set a timeout (in seconds) to wait for actions to
                complete ('never') will wait forever (implicitly sets
                -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                Setting this implies a timeout of 20 min.
------------------------------------------------------------------------
Usage: db_add_node [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of the database
  -s HOSTS, --hosts=HOSTS
                Comma separated list of hosts to add to database
  -p DBPASSWORD, --password=DBPASSWORD
                Database password in single quotes
  -a AHOSTS, --add=AHOSTS
                Comma separated list of hosts to add to database
  -c SCNAME, --subcluster=SCNAME
                Name of subcluster for the new node
  --timeout=NONINTERACTIVE_TIMEOUT
                set a timeout (in seconds) to wait for actions to
                complete ('never') will wait forever (implicitly sets
                -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                Setting this implies a timeout of 20 min.
  --compat21            (deprecated) Use Vertica 2.1 method using node names
                instead of hostnames
------------------------------------------------------------------------
Usage: db_add_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be modified
  -s HOSTS, --hosts=HOSTS
                Comma separated list of hosts to add to the subcluster
  -p DBPASSWORD, --password=DBPASSWORD
                Database password in single quotes
  -c SCNAME, --subcluster=SCNAME
                Name of the new subcluster for the new node
  --is-primary          Create primary subcluster
  --is-secondary        Create secondary subcluster
  --control-set-size=CONTROLSETSIZE
                Set the number of nodes that will run spread within
                the subcluster
  --like=CLONESUBCLUSTER
                Name of an existing subcluster from which to clone
                properties for the new subcluster
  --timeout=NONINTERACTIVE_TIMEOUT
                set a timeout (in seconds) to wait for actions to
                complete ('never') will wait forever (implicitly sets
```

```
                    complete ('never') will wait forever (implicitly sets
                    -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                    Setting this implies a timeout of 20 min.
------------------------------------------------------------------------------
Usage: db_remove_node [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be modified
  -s HOSTS, --hosts=HOSTS
                    Name of the host to remove from the db
  -p DBPASSWORD, --password=DBPASSWORD
                    Database password in single quotes
  --timeout=NONINTERACTIVE_TIMEOUT
                    set a timeout (in seconds) to wait for actions to
                    complete ('never') will wait forever (implicitly sets
                    -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                    Setting this implies a timeout of 20 min.
  --compat21            (deprecated) Use Vertica 2.1 method using node names
                    instead of hostnames
  --skip-directory-cleanup
                    Caution: this option will force you to do a manual
                    cleanup. This option skips directory deletion during
                    remove node. This is best used in a cloud environment
                    where the hosts being removed will be subsequently
                    discarded.
------------------------------------------------------------------------------
Usage: db_remove_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be modified
  -c SCNAME, --subcluster=SCNAME
                    Name of subcluster to be removed
  -p DBPASSWORD, --password=DBPASSWORD
                    Database password in single quotes
  --timeout=NONINTERACTIVE_TIMEOUT
                    set a timeout (in seconds) to wait for actions to
                    complete ('never') will wait forever (implicitly sets
                    -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                    Setting this implies a timeout of 20 min.
  --skip-directory-cleanup
                    Caution: this option will force you to do a manual
                    cleanup. This option skips directory deletion during
                    remove subcluster. This is best used in a cloud
                    environment where the hosts being removed will be
                    subsequently discarded.
------------------------------------------------------------------------------
Usage: db_replace_node [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of the database
  -o ORIGINAL, --original=ORIGINAL
                    Name of host you wish to replace
  -n NEWHOST, --new=NEWHOST
                    Name of the replacement host
  -p DBPASSWORD, --password=DBPASSWORD
                    Database password in single quotes
```

```
                    Database password in single quotes
  --timeout=NONINTERACTIVE_TIMEOUT
                    set a timeout (in seconds) to wait for actions to
                    complete ('never') will wait forever (implicitly sets
                    -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                    Setting this implies a timeout of 20 min.
------------------------------------------------------------------------------
Usage: db_status [options]

Options:
  -h, --help            show this help message and exit
  -s STATUS, --status=STATUS
                    Database status UP,DOWN or ALL(list running dbs -
                    UP,list down dbs - DOWN list all dbs - ALL
------------------------------------------------------------------------------
Usage: distribute_config_files
Sends admintools.conf from local host to all other hosts in the cluster

Options:
  -h, --help  show this help message and exit
------------------------------------------------------------------------------
Usage: drop_db [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Database to be dropped
------------------------------------------------------------------------------
Usage: host_to_node [options]

Options:
  -h, --help            show this help message and exit
  -s HOST, --host=HOST  comma separated list of hostnames which is to be
                    converted into its corresponding nodenames
  -d DB, --database=DB  show only node/host mapping for this database.
------------------------------------------------------------------------------
Usage: admintools -t install_package --package PACKAGE -d DB -p PASSWORD
Examples:
admintools -t install_package -d mydb -p 'mypasswd' --package default
    # (above) install all default packages that aren't currently installed

admintools -t install_package -d mydb -p 'mypasswd' --package default --force-reinstall
   # (above) upgrade (re-install) all default packages to the current version

admintools -t install_package -d mydb -p 'mypasswd' --package hcat
   # (above) install package hcat

See also: admintools -t list_packages

Options:
  -h, --help            show this help message and exit
  -d DBNAME, --dbname=DBNAME
                    database name
  -p PASSWORD, --password=PASSWORD
                    database admin password
  -P PACKAGE, --package=PACKAGE
                    specify package or 'all' or 'default'
  --force-reinstall     Force a package to be re-installed even if it is
                    already installed.
------------------------------------------------------------------------------
Usage: install_procedure [options]
```

```
Options:
  -h, --help            show this help message and exit
  -d DBNAME, --database=DBNAME
                        Name of database for installed procedure
  -f PROCPATH, --file=PROCPATH
                        Path of procedure file to install
  -p OWNERPASSWORD, --password=OWNERPASSWORD
                        Password of procedure file owner
------------------------------------------------------------------------------
Usage: kill_host [options]

Options:
  -h, --help            show this help message and exit
  -s HOSTS, --hosts=HOSTS
                        comma-separated list of hosts on which the vertica
                        process is to be killed using a SIGKILL signal
  --compat21            (deprecated) Use Vertica 2.1 method using node names
                        instead of hostnames
------------------------------------------------------------------------------
Usage: kill_node [options]

Options:
  -h, --help            show this help message and exit
  -s HOSTS, --hosts=HOSTS
                        comma-separated list of hosts on which the vertica
                        process is to be killed using a SIGKILL signal
  --compat21            (deprecated) Use Vertica 2.1 method using node names
                        instead of hostnames
------------------------------------------------------------------------------
Usage: license_audit --dbname DB_NAME [OPTIONS]
Runs audit and collects audit results.

Options:
  -h, --help            show this help message and exit
  -d DATABASE, --database=DATABASE
                        Name of the database to retrieve audit results
  -p PASSWORD, --password=PASSWORD
                        Password for database admin
  -q, --quiet           Do not print status messages.
  -f FILE, --file=FILE  Output results to FILE.
------------------------------------------------------------------------------
Usage: list_allnodes [options]

Options:
  -h, --help  show this help message and exit
------------------------------------------------------------------------------
Usage: list_db [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be listed
------------------------------------------------------------------------------
Usage: list_host [options]

Options:
  -h, --help  show this help message and exit
------------------------------------------------------------------------------
Usage: list_node [options]

Options:
  -h, --help            show this help message and exit
```

```
  -n NODENAME, --node=NODENAME
                        Name of the node to be listed
--------------------------------------------------------------------------------
Usage: admintools -t list_packages [OPTIONS]
Examples:
admintools -t list_packages                          # lists all available packages
admintools -t list_packages --package all            # lists all available packages
admintools -t list_packages --package default        # list all packages installed by default
admintools -t list_packages -d mydb --password 'mypasswd' # list the status of all packages in mydb

Options:
  -h, --help            show this help message and exit
  -d DBNAME, --dbname=DBNAME
                        database name
  -p PASSWORD, --password=PASSWORD
                        database admin password
  -P PACKAGE, --package=PACKAGE
                        specify package or 'all' or 'default'
--------------------------------------------------------------------------------
Usage: logrotateconfig [options]

Options:
  -h, --help            show this help message and exit
  -d DBNAME, --dbname=DBNAME
                        database name
  -r ROTATION, --rotation=ROTATION
                        set how often the log is rotated.[
                        daily|weekly|monthly ]
  -s MAXLOGSZ, --maxsize=MAXLOGSZ
                        set maximum log size before rotation is forced.
  -k KEEP, --keep=KEEP  set # of old logs to keep
--------------------------------------------------------------------------------
Usage: node_map [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  List only data for this database.
--------------------------------------------------------------------------------
Usage: re_ip [options]

Replaces the IP addresses of hosts and databases in a cluster, or changes the
control messaging mode/addresses of a database.

Options:
  -h, --help            show this help message and exit
  -f MAPFILE, --file=MAPFILE
                        A text file with IP mapping information. If the -O
                        option is not used, the command replaces the IP
                        addresses of the hosts in the cluster and all
                        databases for those hosts. In this case, the format of
                        each line in MAPFILE is: [oldIPaddress newIPaddress]
                        or [oldIPaddress newIPaddress, newControlAddress,
                        newControlBroadcast]. If the former,
                        'newControlAddress' and 'newControlBroadcast' would
                        set to default values. Usage: $ admintools -t re_ip -f
                        <mapfile>
  -O, --db-only         Updates the control messaging addresses of a database.
                        Also used for error recovery (when re_ip encounters
                        some certain errors, a mapfile is auto-generated).
                        Format of each line in MAPFILE: [NodeName
                        AssociatedNodeIPaddress, newControlAddress,
```

```
                    newControlBrodcast]. 'NodeName' and
                    'AssociatedNodeIPaddress' must be consistent with
                    admintools.conf. Usage: $ admintools -t re_ip -f
                    <mapfile> -O -d <db_name>
  -i, --noprompts       System does not prompt for the validation of the new
                    settings before performing the re_ip operation. Prompting is on
                    by default.
  -T, --point-to-point  Sets the control messaging mode of a database to
                    point-to-point. Usage: $ admintools -t re_ip -d
                    <db_name> -T
  -U, --broadcast       Sets the control messaging mode of a database to
                    broadcast. Usage: $ admintools -t re_ip -d <db_name>
                    -U
  -d DB, --database=DB  Name of a database. Required with the following
                    options: -O, -T, -U.
------------------------------------------------------------------------------
Usage: rebalance_data [options]

Options:
  -h, --help            show this help message and exit
  -d DBNAME, --dbname=DBNAME
                    database name
  -k KSAFETY, --ksafety=KSAFETY
                    specify the new k value to use
  -p PASSWORD, --password=PASSWORD
  --script              Don't re-balance the data, just provide a script for
                    later use.
------------------------------------------------------------------------------
Usage: restart_db [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be restarted
  -e EPOCH, --epoch=EPOCH
                    Epoch at which the database is to be restarted. If
                    'last' is given as argument the db is restarted from
                    the last good epoch.
  -p DBPASSWORD, --password=DBPASSWORD
                    Database password in single quotes
  -k, --allow-fallback-keygen
                    Generate spread encryption key from Vertica. Use under
                    support guidance only.
  --timeout=NONINTERACTIVE_TIMEOUT
                    set a timeout (in seconds) to wait for actions to
                    complete ('never') will wait forever (implicitly sets
                    -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                    Setting this implies a timeout of 20 min.
------------------------------------------------------------------------------
Usage: restart_node [options]

Options:
  -h, --help            show this help message and exit
  -s HOSTS, --hosts=HOSTS
                    comma-separated list of hosts to be restarted
  -d DB, --database=DB  Name of database whose node is to be restarted
  -p DBPASSWORD, --password=DBPASSWORD
                    Database password in single quotes
  --new-host-ips=NEWHOSTS
                    comma-separated list of new IPs for the hosts to be
                    restarted
```

```
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
  -F, --force           force the node to start and auto recover if necessary
  --compat21            (deprecated) Use Vertica 2.1 method using node names
                        instead of hostnames
  --waitfordown-timeout=WAITTIME
                        Seconds to wait until nodes to be restarted are down
------------------------------------------------------------------------------
Usage: restart_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database whose subcluster is to be restarted
  -c SCNAME, --subcluster=SCNAME
                        Name of subcluster to be restarted
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  -s NEWHOSTS, --hosts=NEWHOSTS
                        Comma separated list of new hosts to rebind to the
                        nodes
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
  -F, --force           Force the nodes in the subcluster to start and auto
                        recover if necessary
------------------------------------------------------------------------------
Usage: return_epoch [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database
  -p PASSWORD, --password=PASSWORD
                        Database password in single quotes
------------------------------------------------------------------------------
Usage: revive_db [options]

Options:
  -h, --help            show this help message and exit
  -s HOSTS, --hosts=HOSTS
                        comma-separated list of hosts to participate in
                        database
  -n NODEHOST, --node-and-host=NODEHOST
                        pair of nodename-hostname values delimited by "|" eg:
                        "v_testdb_node0001|10.0.0.1"Note: Each node-host pair
                        has to be specified as a new argument
  --communal-storage-location=COMMUNAL_STORAGE_LOCATION
                        Location of communal storage
  -x COMMUNAL_STORAGE_PARAMS, --communal-storage-params=COMMUNAL_STORAGE_PARAMS
                        Location of communal storage parameter file
  -d DBNAME, --database=DBNAME
                        Name of database to be revived
  --force               Force cleanup of existing catalog directory
  --display-only        Describe the database on communal storage, and exit
  --strict-validation   Print warnings instead of raising errors while
```

```
                    validating cluster_config.json
------------------------------------------------------------------------------
Usage: sandbox_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be modified
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  -c SCNAME, --subcluster=SCNAME
                        Name of subcluster to be sandboxed
  -b SBNAME, --sandbox=SBNAME
                        Name of the sandbox
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
------------------------------------------------------------------------------
Usage: set_restart_policy [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database for which to set policy
  -p POLICY, --policy=POLICY
                        Restart policy: ('never', 'ksafe', 'always')
------------------------------------------------------------------------------
Usage: set_ssl_params [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database whose parameters will be set
  -k KEYFILE, --ssl-key-file=KEYFILE
                        Path to SSL private key file
  -c CERTFILE, --ssl-cert-file=CERTFILE
                        Path to SSL certificate file
  -a CAFILE, --ssl-ca-file=CAFILE
                        Path to SSL CA file
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
------------------------------------------------------------------------------
Usage: show_active_db [options]

Options:
  -h, --help  show this help message and exit
------------------------------------------------------------------------------
Usage: start_db [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be started
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
  -F, --force           force the database to start at an epoch before data
```

```
                    consistency problems were detected.
  -U, --unsafe          Start database unsafely,  skipping recovery. Use under
                    support guidance only.
  -k, --allow-fallback-keygen
                    Generate spread encryption key from Vertica. Use under
                    support guidance only.
  -s HOSTS, --hosts=HOSTS
                    comma-separated list of hosts to be started
  --fast              Attempt fast startup on un-encrypted eon db. Fast
                    startup will use startup information from
                    cluster_config.json
--------------------------------------------------------------------------------
Usage: stop_db [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be stopped
  -p DBPASSWORD, --password=DBPASSWORD
                    Database password in single quotes
  -F, --force           Force the databases to shutdown, even if users are
                    connected.
  -z, --if-no-users     Only shutdown if no users are connected.
                    If any users are connected, exit with an error.
  -n DRAIN_SECONDS, --drain-seconds=DRAIN_SECONDS
                    Eon db only: seconds to wait for user connections to close.
                    Default value is 60 seconds.
                    When the time expires, connections will be forcibly closed
                    and the db will shut down.
  --timeout=NONINTERACTIVE_TIMEOUT
                    set a timeout (in seconds) to wait for actions to
                    complete ('never') will wait forever (implicitly sets
                    -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                    Setting this implies a timeout of 20 min.
--------------------------------------------------------------------------------
Usage: stop_host [options]

Options:
  -h, --help            show this help message and exit
  -s HOSTS, --hosts=HOSTS
                    comma-separated list of hosts on which the vertica
                    process is to be killed using a SIGTERM signal
  --compat21            (deprecated) Use Vertica 2.1 method using node names
                    instead of hostnames
--------------------------------------------------------------------------------
Usage: stop_node [options]

Options:
  -h, --help            show this help message and exit
  -s HOSTS, --hosts=HOSTS
                    comma-separated list of hosts on which the vertica
                    process is to be killed using a SIGTERM signal
  --compat21            (deprecated) Use Vertica 2.1 method using node names
                    instead of hostnames
--------------------------------------------------------------------------------
Usage: stop_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database whose subcluster is to be stopped
  -c SCNAME, --subcluster=SCNAME
```

```
                        Name of subcluster to be stopped
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  -n DRAIN_SECONDS, --drain-seconds=DRAIN_SECONDS
                        Seconds to wait for user connections to close.
                        Default value is 60 seconds.
                        When the time expires, connections will be forcibly closed
                        and the db will shut down.
  -F, --force           Force the subcluster to shutdown immediately,
                        even if users are connected.
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
------------------------------------------------------------------------------
Usage: uninstall_package [options]

Options:
  -h, --help            show this help message and exit
  -d DBNAME, --dbname=DBNAME
                        database name
  -p PASSWORD, --password=PASSWORD
                        database admin password
  -P PACKAGE, --package=PACKAGE
                        specify package or 'all' or 'default'
------------------------------------------------------------------------------
Usage: unsandbox_subcluster [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database to be modified
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes
  -c SCNAME, --subcluster=SCNAME
                        Name of subcluster to be un-sandboxed
  --timeout=NONINTERACTIVE_TIMEOUT
                        set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i)
  -i, --noprompts       do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
------------------------------------------------------------------------------
Usage: upgrade_license_key --database mydb --license my_license.key
upgrade_license_key --install --license my_license.key

Updates the vertica license.

Without '--install', updates the license used by the database and
the admintools license cache.

With '--install', updates the license cache in admintools that
is used for creating new databases.

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  Name of database. Cannot be used with --install.
  -l LICENSE, --license=LICENSE
                        Required - path to the license.
  -i, --install         When option is included, command will only update the
                        admintools license cache. Cannot be used with
```

```
                admintools license cache. Cannot be used with
                --database.
 -p PASSWORD, --password=PASSWORD
                Database password.
--------------------------------------------------------------------------------
Usage: view_cluster [options]

Options:
 -h, --help           show this help message and exit
 -x, --xpand           show the full cluster state, node by node
 -d DB, --database=DB  filter the output for a single database
```

Operating the database

This topic explains how to start and stop your Vertica database, and how to use the database index tool.

In this section

- Starting the database
- Stopping the database
- CRC and sort order check

Starting the database

You can start a database through one of the following:

- Management Console
- Administration Tools
- Command Line

Administration tools

You can start a database with the Vertica Administration Tools :

1. Open the Administration Tools and select View Database Cluster State to make sure all nodes are down and no other database is running.
2. Open the Administration Tools. See Using the administration tools for information about accessing the Administration Tools.
3. On the **Main Menu** , select **Start Database** ,and then select **OK** .
4. Select the database to start, and then click **OK** .

> **Caution**
> Start only one database at a time. If you start more than one database at a given time, results can be unpredictable: users are liable to encounter resource conflicts, or perform operations on the wrong database.

5. Enter the database password and click **OK** .
6. When prompted that the database started successfully, click **OK** .
7. Check the log files to make sure that no startup problems occurred.

Command line

You can start a database with the command line tool start_db :

```
$ /opt/vertica/bin/admintools -t start_db -d db-name
   [-p password]
   [-s host1[,...] | --hosts=host1[,...]]
   [--timeout seconds]
   [-i | --noprompts]
   [--fast]
   [-F | --force]
```

Option	Description
-d --database	Name of database to start.

-p --password	Required only during database creation, when you install a new license. If the license is valid, the option -p (or --password) is not required to start the database and is silently ignored. This is by design, as the database can only be started by the user who (as part of the verticadba UNIX user group) initially created the database or who has root or su privileges. If the license is invalid, Vertica uses the -p password argument to attempt to upgrade the license with the license file stored in /opt/vertica/config/share/license.key .
-s --hosts	(Eon Mode only) Comma delimited list of primary node host names or IP addresses. If you use this option, start_db attempts to start the database using just the nodes in the list. If omitted, start_db starts all database nodes. For details, see Start Just the Primary Nodes in an Eon Mode Database below.
--timeout	The number of seconds a timeout in seconds to await startup completion. If set to never , start_db never times out (implicitly sets -i)
-i --noprompts	Startup does not pause to await user input. Setting -i implies a timeout of 1200 seconds.
--fast	(Eon Mode only) Attempts fast startup on a database using startup information from cluster_config.json . This option can only be used with databases that do not use Spread encryption .
-F --force	Forces the database to start at an epoch before data consistency problems were detected.

The following example uses start_db to start a single-node database:

```
$ /opt/vertica/bin/admintools -t start_db -d VMart
Info:
no password specified, using none
Node Status: v_vmart_node0001: (DOWN)
Node Status: v_vmart_node0001: (DOWN)
Node Status: v_vmart_node0001: (DOWN)
Node Status: v_vmart_node0001: (DOWN)
Node Status: v_vmart_node0001: (DOWN)
Node Status: v_vmart_node0001: (DOWN)
Node Status: v_vmart_node0001: (DOWN)
Node Status: v_vmart_node0001: (DOWN)
Node Status: v_vmart_node0001: (UP)
Database VMart started successfully
```

Eon Mode database node startup

On starting an Eon Mode database, you can start all primary nodes, or a subset of them. In both cases, pass start_db the list of the primary nodes to start with the -s option.

The following requirements apply:

- Primary node hosts must already be up to start the database.
- The start_db tool cannot start stopped hosts such as cloud-based VMs. You must either manually start the hosts or use the MC to start the cluster.

The following example starts the three primary nodes in a six-node Eon Mode database:

```
$ admintools -t start_db -d verticadb -p 'password' \
  -s 10.11.12.10,10.11.12.20,10.11.12.30
   Starting nodes:
      v_verticadb_node0001 (10.11.12.10)
      v_verticadb_node0002 (10.11.12.20)
      v_verticadb_node0003 (10.11.12.30)
   Starting Vertica on all nodes. Please wait, databases with a large catalog may take a while to initialize.
   Node Status: v_verticadb_node0001: (DOWN) v_verticadb_node0002: (DOWN) v_verticadb_node0003: (DOWN)
   Node Status: v_verticadb_node0001: (DOWN) v_verticadb_node0002: (DOWN) v_verticadb_node0003: (DOWN)
   Node Status: v_verticadb_node0001: (DOWN) v_verticadb_node0002: (DOWN) v_verticadb_node0003: (DOWN)
   Node Status: v_verticadb_node0001: (DOWN) v_verticadb_node0002: (DOWN) v_verticadb_node0003: (DOWN)
   Node Status: v_verticadb_node0001: (DOWN) v_verticadb_node0002: (DOWN) v_verticadb_node0003: (DOWN)
   Node Status: v_verticadb_node0001: (DOWN) v_verticadb_node0002: (DOWN) v_verticadb_node0003: (DOWN)
   Node Status: v_verticadb_node0001: (UP) v_verticadb_node0002: (UP) v_verticadb_node0003: (UP)
Syncing catalog on verticadb with 2000 attempts.
Database verticadb: Startup Succeeded.  All Nodes are UP
```

After the database starts, the secondary subclusters are down. You can choose to start them as needed. See Starting a Subcluster .

Starting the database with a subset of primary nodes

As a best pratice, Vertica recommends that you always start an Eon Mode database with all primary nodes. Occasionally, you might be unable to start the hosts for all primary nodes. In that case, you might need to start the database with a subset of its primary nodes.

If start_db specifies a subset of database primary nodes, the following requirements apply:

- The nodes must comprise a quorum: at least 50% + 1 of all primary nodes in the cluster.
- Collectively, the nodes must provide coverage for all shards in communal storage. The primary nodes you use to start the database do not attempt to rebalance shard subscriptions while starting up.

If either or both of these conditions are not met, start_db returns an error. In the following example, start_db specifies three primary nodes in a database with nine primary nodes. The command returns an error that it cannot start the database with fewer than five primary nodes:

```
$ admintools -t start_db -d verticadb -p 'password' \
   -s 10.11.12.10,10.11.12.20,10.11.12.30
   Starting nodes:
      v_verticadb_node0001 (10.11.12.10)
      v_verticadb_node0002 (10.11.12.20)
      v_verticadb_node0003 (10.11.12.30)
Error: Quorum not satisfied for verticadb.
   3 < minimum 5  of 9 primary nodes.
Attempted to start the following nodes:
Primary
      v_verticadb_node0001 (10.11.12.10)
      v_verticadb_node0003 (10.11.12.30)
      v_verticadb_node0002 (10.11.12.20)
Secondary

 hint: you may want to start all primary nodes in the database
Database start up failed.  Cluster partitioned.
```

If you try to start the database with fewer than the full set of primary nodes and the cluster fails to start, Vertica processes might continue to run on some of the hosts. If so, subsequent attempts to start the database will return with an error like this:

```
Error: the vertica process for the database is running on the following hosts:
10.11.12.10
10.11.12.20
10.11.12.30
This may be because the process has not completed previous shutdown activities. Please wait and retry again.
Database start up failed.  Processes still running.
Database verticadb did not start successfully: Processes still running.. Hint: you may need to start all primary nodes.
```

Before you can start the database, you must stop the Vertica server process on the hosts listed in the error message, either with the admintools menus or the admintools command line's stop_host tool:

```
$ admintools -t stop_host -s 10.11.12.10,10.11.12.20,10.11.12.30
```

Stopping the database

There are many occasions when you must stop a database, for example, before an upgrade or performing various maintenance tasks. You can stop a running database through one of the following:

- Management Console
- Administration Tools interface
- Vertica functions
- Command line

You cannot stop a running database if any users are connected or Database Designer is building or deploying a database design.

Administration tools

To stop a running database with admintools:

1. Verify that all cluster nodes are up . If any nodes are down, identify and restart them .
2. Close all user sessions:
   - Identify all users with active sessions by querying the SESSIONS system table. Notify users of the impending shutdown and request them to shut down their sessions.
   - Prevent users from starting new sessions by temporarily resetting configuration parameter MaxClientSessions to 0:
     ```
     => ALTER DATABASE DEFAULT SET MaxClientSessions = 0;
     ```
   - Close all remaining user sessions with Vertica functions CLOSE_SESSION and CLOSE_ALL_SESSIONS .

   > **Note**
   > You can also force a database shutdown and block new sessions with the function SHUTDOWN .

3. Open Vertica Administration Tools .
4. From the Main Menu:
   - Select Stop Database
   - Click **OK**
5. Select the database to stop and click **OK** .
6. Enter the password (if asked) and click **OK** .
7. When prompted that database shutdown is complete, click **OK** .

Vertica functions

You can stop a database with the SHUTDOWN function. By default, the shutdown fails if any users are connected. To force a shutdown regardless of active user connections, call SHUTDOWN with an argument of true :

```
=> SELECT SHUTDOWN('true');
     SHUTDOWN
-------------------------
Shutdown: sync complete
(1 row)
```

In Eon Mode databases, you can stop subclusters with the SHUTDOWN_SUBCLUSTER and SHUTDOWN_WITH_DRAIN functions. SHUTDOWN_SUBCLUSTER shuts down subclusters immediately, whereas SHUTDOWN_WITH_DRAIN performs a graceful shutdown that drains client connections from subclusters before shutting them down. For more information, see Starting and stopping subclusters .

The following example demonstrates how you can shut down all subclusters in an Eon Mode database using SHUTDOWN_WITH_DRAIN:

```
=> SELECT SHUTDOWN_WITH_DRAIN('', 0);
NOTICE 0:  Begin shutdown of subcluster (default_subcluster, analytics)
              SHUTDOWN_WITH_DRAIN
-----------------------------------------------------------------------
Shutdown message sent to subcluster (default_subcluster, analytics)

(1 row)
```

Command line

You can stop a database with the admintools command stop_db :

```
$ $ admintools -t stop_db --help
Usage: stop_db [options]

Options:
  -h, --help            Show this help message and exit.
  -d DB, --database=DB  Name of database to be stopped.
  -p DBPASSWORD, --password=DBPASSWORD
                        Database password in single quotes.
  -F, --force           Force the database to shutdown, even if users are
                        connected.
  -z, --if-no-users     Only shutdown if no users are connected. If any users
                        are connected, exit with an error.
  -n DRAIN_SECONDS, --drain-seconds=DRAIN_SECONDS
                        Eon db only: seconds to wait for user connections to
                        close. Default value is 60 seconds. When the time
                        expires, connections will be forcibly closed and the
                        db will shut down.
  --timeout=NONINTERACTIVE_TIMEOUT
                        Set a timeout (in seconds) to wait for actions to
                        complete ('never') will wait forever (implicitly sets
                        -i).
  -i, --noprompts       Do not stop and wait for user input(default false).
                        Setting this implies a timeout of 20 min.
```

> **Note**
> You cannot use both the -z (--if-no-users) and -F (or --force) options in the same stop_db call.

stop_db behavior depends on whether it stops an Eon Mode or Enterprise Mode database.

Stopping an Eon Mode database

In Eon Mode databases, the default behavior of stop_db is to call SHUTDOWN_WITH_DRAIN to gracefully shut down all subclusters in the database. This graceful shutdown process drains client connections from subclusters before shutting them down.

The stop_db option -n (--drain-seconds) lets you specify the number of seconds to wait—by default, 60—before forcefully closing client connections and shutting down all subclusters. If you set a negative -n value, the subclusters are marked as draining but do not shut down until all active user sessions disconnect.

In the following example, the database initially has an active client session, but the session closes before the timeout limit is reached and the database shuts down:

```
$ admintools -t stop_db -d verticadb --password password --drain-seconds 200
Shutdown will use connection draining.
Shutdown will wait for all client sessions to complete, up to 200 seconds
Then it will force a shutdown.
Poller has been running for 0:00:00.000025 seconds since 2022-07-27 17:10:08.292919


-------------------------------------------------------------
client_sessions     |node_count         |node_names
-------------------------------------------------------------
0                   |5                  |v_verticadb_node0005,v_verticadb_node0006,v_verticadb_node0003,v_verticadb_node0...
1                   |1                  |v_verticadb_node0001
STATUS: vertica.engine.api.db_client.module is still running on 1 host: nodeIP as of 2022-07-27 17:10:18. See /opt/vertica/log/adminTools.log for full details.
Poller has been running for 0:00:11.371296 seconds since 2022-07-27 17:10:08.292919

...

-------------------------------------------------------------
client_sessions     |node_count         |node_names
-------------------------------------------------------------
0                   |5                  |v_verticadb_node0002,v_verticadb_node0004,v_verticadb_node0003,v_verticadb_node0...
1                   |1                  |v_verticadb_node0001
Stopping poller drain_status because it was canceled
Shutdown metafunction complete. Polling until database processes have stopped.
Database verticadb stopped successfully
```

If you use the -z (--if-no-users) option, the database shuts down immediately if there are no active user sessions. Otherwise, the stop_db command returns an error:

```
$ admintools -t stop_db -d verticadb --password password --if-no-users
Running shutdown metafunction. Not using connection draining

            Active session details
| Session id                     | Host Ip        | Connected User |
| ------- --                     | ---- --        | --------- ---- |
| v_verticadb_node0001-107720:0x257 | 192.168.111.31 | analyst        |
Database verticadb not stopped successfully for the following reason:
Shutdown did not complete. Message: Shutdown: aborting shutdown
Active sessions prevented shutdown.
Omit the option --if-no-users to close sessions. See stop_db --help.
```

You can use the -F (or --force) option to shut down all subclusters immediately, without checking for active user sessions or draining the subclusters.

Stopping an Enterprise Mode database

In Enterprise Mode databases, the default behavior of stop_db is to shut down the database only if there are no active sessions. If users are connected to the database, the command aborts with an error message and lists all active sessions. For example:

```
$ /opt/vertica/bin/admintools -t stop_db -d VMart
Info: no password specified, using none

        Active session details
| Session id                     | Host Ip       | Connected User |
| ------- --                     | ---- --       | --------- ---- |
| v_vmart_node0001-91901:0x162   | 10.20.100.247 | analyst        |
Database VMart not stopped successfully for the following reason:
Unexpected output from shutdown: Shutdown: aborting shutdown
NOTICE: Cannot shut down while users are connected
```

You can use the -F (or --force) option to override user connections and force a shutdown.

CRC and sort order check

As a superuser, you can run the Index tool on a Vertica database to perform two tasks:

- Run a cyclic redundancy check (CRC) on each block of existing data storage to check the data integrity of ROS data blocks.
- Check that the sort order in ROS containers is correct.

If the database is down, invoke the Index tool from the Linux command line. If the database is up, invoke from VSQL with Vertica meta-function RUN_INDEX_TOOL :

Operation	**Database down**	**Database up**
Run CRC	/opt/vertica/bin/vertica -D *catalog-path* -v	SELECT RUN_INDEX_TOOL ('checkcrc',...);
Check sort order	/opt/vertica/bin/vertica -D *catalog-path* -I	SELECT RUN_INDEX_TOOL ('checksort',...);

If invoked from the command line, the Index tool runs only on the current node. However, you can run the Index tool on multiple nodes simultaneously.

Result output

The Index tool writes summary information about its operation to standard output; detailed information on results is logged in one of two locations, depending on the environment where you invoke the tool:

Invoked from:	**Results written to:**
Linux command line	indextool.log in the database catalog directory
VSQL	vertica.log on the current node

For information about evaluating output for possible errors, see:

- Evaluating CRC errors
- Evaluating sort order errors

Optimizing performance

You can optimize meta-function performance by narrowing the scope of the operation to one or more projections, and specifying the number of threads used to execute the function. For details, see RUN_INDEX_TOOL .

In this section

- Evaluating CRC errors
- Evaluating sort order errors

Evaluating CRC errors

Vertica evaluates the CRC values in each ROS data block each time it fetches data disk to process a query. If CRC errors occur while fetching data, the following information is written to the vertica.log file:

```
CRC Check Failure Details:File Name:
File Offset:
Compressed size in file:
Memory Address of Read Buffer:
Pointer to Compressed Data:
Memory Contents:
```

The Event Manager is also notified of CRC errors, so you can use an SNMP trap to capture CRC errors:

```
"CRC mismatch detected on file <file_path>. File may be corrupted. Please check hardware and drivers."
```

If you run a query from vsql, ODBC, or JDBC, the query returns a FileColumnReader ERROR . This message indicates that a specific block's CRC does not match a given record as follows:

```
hint: Data file may be corrupt.  Ensure that all hardware (disk and memory) is working properly.
Possible solutions are to delete the file <pathname> while the node is down, and then allow the node
to recover, or truncate the table data.code: ERRCODE_DATA_CORRUPTED
```

Evaluating sort order errors

If ROS data is not sorted correctly in the projection's order, query results that rely on sorted data will be incorrect. You can use the Index tool to check the ROS sort order if you suspect or detect incorrect query results. The Index tool evaluates each ROS row to determine whether it is sorted correctly. If the check locates a row that is not in order, it writes an error message to the log file with the row number and contents of the unsorted row.

Reviewing errors

1. Open the indextool.log file. For example:
```
$ cd VMart/v_check_node0001_catalog
```
2. Look for error messages that include an OID number and the string Sort Order Violation . For example:
```
<INFO> ...on oid 45035996273723545: Sort Order Violation:
```
3. Find detailed information about the sort order violation string by running grep on indextool.log . For example, the following command returns the line before each string (-B1), and the four lines that follow (-A4):
```
[15:07:55][vertica-s1]: grep -B1 -A4 'Sort Order Violation:' /my_host/databases/check/v_check_node0001_catalog/indextool.log
2012-06-14 14:07:13.686 unknown:0x7fe1da7a1950 [EE] <INFO> An error occurred when running index tool thread on oid 45035996273723537:
Sort Order Violation:
Row Position: 624
Column Index: 0
Last Row: 2576000
This Row: 2575000
--
2012-06-14 14:07:13.687 unknown:0x7fe1dafa2950 [EE] <INFO> An error occurred when running index tool thread on oid 45035996273723545:
Sort Order Violation:
Row Position: 3
Column Index: 0
Last Row: 4
This Row: 2
--
```
4. Find the projection where a sort order violation occurred by querying system table STORAGE_CONTAINERS . Use a storage_oid equal to the OID value listed in indextool.log . For example:
```
=> SELECT * FROM storage_containers WHERE storage_oid = 45035996273723545;
```

Working with native tables

You can create two types of native tables in Vertica (ROS format), columnar and flexible. You can create both types as persistent or temporary. You can also create views that query a specific set of table columns.

The tables described in this section store their data in and are managed by the Vertica database. Vertica also supports external tables, which are defined in the database and store their data externally. For more information about external tables, see Working with external data .

In this section

- Creating tables
- Creating temporary tables
- Creating a table from other tables
- Immutable tables
- Disk quotas
- Managing table columns
- Altering table definitions
- Sequences
- Merging table data
- Removing table data
- Rebuilding tables
- Dropping tables

Creating tables

Use the CREATE TABLE statement to create a native table in the Vertica logical schema . You can specify the columns directly, as in the following example, or you can derive a table definition from another table using a LIKE or AS clause. You can specify constraints, partitioning, segmentation, and other factors. For details and restrictions, see the reference page.

The following example shows a basic table definition:

```
=> CREATE TABLE orders(
    orderkey    INT,
    custkey     INT,
    prodkey     ARRAY[VARCHAR(10)],
    orderprices ARRAY[DECIMAL(12,2)],
    orderdate   DATE
);
```

Table data storage

Unlike traditional databases that store data in tables, Vertica physically stores table data in projections , which are collections of table columns. Projections store data in a format that optimizes query execution. Similar to materialized views, they store result sets on disk rather than compute them each time they are used in a query.

In order to query or perform any operation on a Vertica table, the table must have one or more projections associated with it. For more information, see Projections .

Deriving a table definition from the data

You can use the INFER_TABLE_DDL function to inspect Parquet, ORC, JSON, or Avro data and produce a starting point for a table definition. This function returns a CREATE TABLE statement, which might require further editing. For columns where the function could not infer the data type, the function labels the type as unknown and emits a warning. For VARCHAR and VARBINARY columns, you might need to adjust the length. Always review the statement the function returns, but especially for tables with many columns, using this function can save time and effort.

Parquet, ORC, and Avro files include schema information, but JSON files do not. For JSON, the function inspects the raw data to produce one or more candidate table definitions. See the function reference page for JSON examples.

In the following example, the function infers a complete table definition from Parquet input, but the VARCHAR columns use the default size and might need to be adjusted:

```
=> SELECT INFER_TABLE_DDL('/data/people/*.parquet'
       USING PARAMETERS format = 'parquet', table_name = 'employees');
WARNING 9311:  This generated statement contains one or more varchar/varbinary columns which default to length 80
                INFER_TABLE_DDL
-------------------------------------------------------------------------------
 create table "employees"(
  "employeeID" int,
  "personal" Row(
    "name" varchar,
    "address" Row(
      "street" varchar,
      "city" varchar,
      "zipcode" int
    ),
    "taxID" int
  ),
  "department" varchar
 );
(1 row)
```

For Parquet files, you can use the GET_METADATA function to inspect a file and report metadata including information about columns.

See also

- Altering table definitions
- Creating temporary tables
- Creating a table from other tables
- Creating external tables

Creating temporary tables

CREATE TEMPORARY TABLE creates a table whose data persists only during the current session. Temporary table data is never visible to other sessions.

By default, all temporary table data is transaction-scoped—that is, the data is discarded when a COMMIT statement ends the current transaction. If CREATE TEMPORARY TABLE includes the parameter ON COMMIT PRESERVE ROWS , table data is retained until the current session ends.

Temporary tables can be used to divide complex query processing into multiple steps. Typically, a reporting tool holds intermediate results while reports are generated—for example, the tool first gets a result set, then queries the result set, and so on.

When you create a temporary table, Vertica automatically generates a default projection for it. For more information, see Auto-projections .

Global versus local tables

CREATE TEMPORARY TABLE can create tables at two scopes, global and local, through the keywords GLOBAL and LOCAL , respectively:

Global temporary tables	Vertica creates global temporary tables in the public schema. Definitions of these tables are visible to all sessions, and persist across sessions until they are explicitly dropped. Multiple users can access the table concurrently. Table data is session-scoped, so it is visible only to the session user, and is discarded when the session ends.
Local temporary tables	Vertica creates local temporary tables in the V_TEMP_SCHEMA namespace and inserts them transparently into the user's search path. These tables are visible only to the session where they are created. When the session ends, Vertica automatically drops the table and its data.

Data retention

You can specify whether temporary table data is transaction- or session-scoped:

- ON COMMIT DELETE ROWS (default): Vertica automatically removes all table data when each transaction ends.
- ON COMMIT PRESERVE ROWS : Vertica preserves table data across transactions in the current session. Vertica automatically truncates the table when the session ends.

> **Note**
> If you create a temporary table with ON COMMIT PRESERVE ROWS , you cannot add projections for that table if it contains data. You must first remove all data from that table with TRUNCATE TABLE .
>
> You can create projections for temporary tables created with ON COMMIT DELETE ROWS , whether populated with data or not. However, CREATE PROJECTION ends any transaction where you might have added data, so projections are always empty.

ON COMMIT DELETE ROWS

By default, Vertica removes all data from a temporary table, whether global or local, when the current transaction ends.

For example:

```
=> CREATE TEMPORARY TABLE tempDelete (a int, b int);
CREATE TABLE
=> INSERT INTO tempDelete VALUES(1,2);
 OUTPUT
--------
      1
(1 row)

=> SELECT * FROM tempDelete;
 a | b
---+---
 1 | 2
(1 row)

=> COMMIT;
COMMIT

=> SELECT * FROM tempDelete;
 a | b
---+---
(0 rows)
```

If desired, you can use DELETE within the same transaction multiple times, to refresh table data repeatedly.

ON COMMIT PRESERVE ROWS

You can specify that a temporary table retain data across transactions in the current session, by defining the table with the keywords ON COMMIT PRESERVE ROWS . Vertica automatically removes all data from the table only when the current session ends.

For example:

```
=> CREATE TEMPORARY TABLE tempPreserve (a int, b int) ON COMMIT PRESERVE ROWS;
CREATE TABLE
=> INSERT INTO tempPreserve VALUES (1,2);
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
=> SELECT * FROM tempPreserve;
 a | b
---+---
 1 | 2
(1 row)

=> INSERT INTO tempPreserve VALUES (3,4);
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
=> SELECT * FROM tempPreserve;
 a | b
---+---
 1 | 2
 3 | 4
(2 rows)
```

Eon restrictions

The following Eon Mode restrictions apply to temporary tables:

- K-safety of temporary tables is always set to 0, regardless of system K-safety. If a CREATE TEMPORARY TABLE statement sets *k-num* greater than 0, Vertica returns an warning.
- If subscriptions to the current session change, temporary tables in that session becomes inaccessible. Causes for session subscription changes include:
  - A node left the list of participating nodes.
  - A new node appeared in the list of participating nodes.
  - An active node changed for one or more shards.
  - A mergeout operation in the same session that is triggered by a user explicitly invoking DO_TM_TASK('mergeout') , or changing a column data type with ALTER TABLE...ALTER COLUMN .

    > **Note**
    > Background mergeout operations have no effect on session subscriptions.

Creating a table from other tables

You can create a table from other tables in two ways:

- Replicate an existing table through CREATE TABLE...LIKE .

- Create a table from a query through CREATE TABLE...AS .

Important
You can also copy one table to another with the Vertica function COPY_TABLE .

In this section

- Replicating a table
- Creating a table from a query

Replicating a table

You can create a table from an existing one using CREATE TABLE with the LIKE clause :

```
CREATE TABLE [schema.]table-name LIKE [schema.]existing-table
  [ {INCLUDING | EXCLUDING} PROJECTIONS ]
  [ {INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES ]
```

Creating a table with LIKE replicates the source table definition and any storage policy associated with it. It does not copy table data or expressions on columns.

Copying constraints

CREATE TABLE...LIKE copies all table constraints, with the following exceptions:

- Foreign key constraints.
- Any column that obtains its values from a sequence, including IDENTITY columns. Vertica copies the column values into the new table, but removes the original constraint. For example, the following table definition sets an IDENTITY constraint on column ID :

  ```
  CREATE TABLE public.Premium_Customer
  (
     ID IDENTITY ,
     lname varchar(25),
     fname varchar(25),
     store_membership_card int
  );
  ```

  The following CREATE TABLE...LIKE statement replicates this table as All_Customers . Vertica removes the IDENTITY constraint from All_Customers.ID , changing it to an integer column with a NOT NULL constraint:

  ```
  => CREATE TABLE All_Customers like Premium_Customer;
  CREATE TABLE
  => select export_tables('','All_Customers');
                       export_tables
  -----------------------------------------------------
  CREATE TABLE public.All_Customers
  (
     ID int NOT NULL,
     lname varchar(25),
     fname varchar(25),
     store_membership_card int
  );

  (1 row)
  ```

Including projections

You can qualify the LIKE clause with INCLUDING PROJECTIONS or EXCLUDING PROJECTIONS , which specify whether to copy projections from the source table:

- EXCLUDING PROJECTIONS (default): Do not copy projections from the source table.
- INCLUDING PROJECTIONS : Copy current projections from the source table. Vertica names the new projections according to Vertica naming conventions , to avoid name conflicts with existing objects.

Including schema privileges

You can specify default inheritance of schema privileges for the new table:

- EXCLUDE [SCHEMA] PRIVILEGES (default) disables inheritance of privileges from the schema
- INCLUDE [SCHEMA] PRIVILEGES grants the table the same privileges granted to its schema

For more information see Setting privilege inheritance on tables and views .

Restrictions

The following restrictions apply to the source table:

- It cannot have out-of-date projections.
- It cannot be a temporary table.

Example

1. Create the table states :

```
=> CREATE TABLE states (
   state char(2) NOT NULL, bird varchar(20), tree varchar (20), tax float, stateDate char (20))
   PARTITION BY state;
```

2. Populate the table with data:

```
INSERT INTO states VALUES ('MA', 'chickadee', 'american_elm', 5.675, '07-04-1620');
INSERT INTO states VALUES ('VT', 'Hermit_Thrasher', 'Sugar_Maple', 6.0, '07-04-1610');
INSERT INTO states VALUES ('NH', 'Purple_Finch', 'White_Birch', 0, '07-04-1615');
INSERT INTO states VALUES ('ME', 'Black_Cap_Chickadee', 'Pine_Tree', 5, '07-04-1615');
INSERT INTO states VALUES ('CT', 'American_Robin', 'White_Oak', 6.35, '07-04-1618');
INSERT INTO states VALUES ('RI', 'Rhode_Island_Red', 'Red_Maple', 5, '07-04-1619');
```

3. View the table contents:

```
=> SELECT * FROM states;


 state |        bird         |     tree     |  tax  |      stateDate
-------+---------------------+--------------+-------+----------------------
 VT    | Hermit_Thrasher     | Sugar_Maple  |     6 | 07-04-1610
 CT    | American_Robin      | White_Oak    |  6.35 | 07-04-1618
 RI    | Rhode_Island_Red    | Red_Maple    |     5 | 07-04-1619
 MA    | chickadee           | american_elm | 5.675 | 07-04-1620
 NH    | Purple_Finch        | White_Birch  |     0 | 07-04-1615
 ME    | Black_Cap_Chickadee | Pine_Tree    |     5 | 07-04-1615
(6 rows
```

4. Create a sample projection and refresh:

```
=> CREATE PROJECTION states_p AS SELECT state FROM states;

=> SELECT START_REFRESH();
```

5. Create a table like the states table and include its projections:

```
=> CREATE TABLE newstates LIKE states INCLUDING PROJECTIONS;
```

6. View projections for the two tables. Vertica has copied projections from states to newstates :

```
=> \dj
                                    List of projections
          Schema           |                Name                 |  Owner  |       Node       | Comment
-------------------------------+-----------------------------------------+---------+------------------+---------
 public                   | newstates_b0                        | dbadmin |                  |
 public                   | newstates_b1                        | dbadmin |                  |
 public                   | newstates_p_b0                      | dbadmin |                  |
 public                   | newstates_p_b1                      | dbadmin |                  |
 public                   | states_b0                           | dbadmin |                  |
 public                   | states_b1                           | dbadmin |                  |
 public                   | states_p_b0                         | dbadmin |                  |
 public                   | states_p_b1                         | dbadmin |                  |
```

7. View the table newstates , which shows columns copied from states :

```
=> SELECT * FROM newstates;


 state | bird | tree | tax | stateDate
-------+------+------+-----+-----------
(0 rows)
```

When you use the CREATE TABLE...LIKE statement, storage policy objects associated with the table are also copied. Data added to the new table use the same labeled storage location as the source table, unless you change the storage policy. For more information, see Working With Storage Locations .

See also

- Creating tables
- Creating temporary tables
- Creating external tables
- Creating a table from a query

Creating a table from a query

CREATE TABLE can specify an AS clause to create a table from a query, as follows:

```
CREATE [TEMPORARY] TABLE [schema.]table-name
   [ ( column-name-list ) ]
   [ {INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES ]
AS  [  /*+ LABEL */ ] [ AT epoch ] query [ ENCODED BY column-ref-list ]
```

Vertica creates a table from the query results and loads the result set into it. For example:

```
=> CREATE TABLE cust_basic_profile AS SELECT
    customer_key, customer_gender, customer_age, marital_status, annual_income, occupation
    FROM customer_dimension WHERE customer_age>18 AND customer_gender !='';
CREATE TABLE
=> SELECT customer_age, annual_income, occupation FROM cust_basic_profile
    WHERE customer_age > 23 ORDER BY customer_age;
 customer_age | annual_income |     occupation
--------------+---------------+--------------------
           24 |        469210 | Hairdresser
           24 |        140833 | Butler
           24 |        558867 | Lumberjack
           24 |        529117 | Mechanic
           24 |        322062 | Acrobat
           24 |        213734 | Writer
           ...
```

AS clause options

You can qualify an AS clause with one or both of the following options:

- LABEL hint that identifies a statement for profiling and debugging
- AT *epoch* clause to specify that the query return historical data

Labeling the AS clause

You can embed a LABEL hint in an AS clause in two places:

- Immediately after the keyword AS :
  ```
  CREATE TABLE myTable AS /*+LABEL myLabel*/...
  ```
- In the SELECT statement:
  ```
  CREATE TABLE myTable AS SELECT /*+LABEL myLabel*/
  ```

If the AS clause contains a LABEL hint in both places, the first label has precedence.

> **Note**

Labels are invalid for external tables.

Loading historical data

You can qualify a CREATE TABLE AS query with an AT *epoch* clause, to specify that the query return historical data, where *epoch* is one of the following:

- EPOCH LATEST : Return data up to but not including the current epoch. The result set includes data from the latest committed DML transaction.
- EPOCH *integer* : Return data up to and including the *integer* -specified epoch.
- TIME ' *timestamp* ' : Return data from the *timestamp* -specified epoch.

Note
These options are ignored if used to query temporary or external tables.

See Epochs for additional information about how Vertica uses epochs.

For details, see Historical queries .

Zero-width column handling

If the query returns a column with zero width, Vertica automatically converts it to a VARCHAR(80) column. For example:

```
=> CREATE TABLE example AS SELECT '' AS X;
CREATE TABLE
=> SELECT EXPORT_TABLES ('', 'example');
                    EXPORT_TABLES
-------------------------------------------------------------
CREATE TEMPORARY TABLE public.example
(
   X varchar(80)
);
```

Requirements and restrictions

- If you create a temporary table from a query, you must specify ON COMMIT PRESERVE ROWS in order to load the result set into the table. Otherwise, Vertica creates an empty table.
- If the query output has expressions other than simple columns, such as constants or functions, you must specify an alias for that expression, or list all columns in the column name list.
- You cannot use CREATE TABLE AS SELECT with a SELECT that returns values of complex types . You can, however, use CREATE TABLE LIKE .

See also

- Creating tables
- Creating temporary tables
- Creating external tables
- Replicating a table

Immutable tables

Many secure systems contain records that must be provably immune to change. Protective strategies such as row and block checksums incur high overhead. Moreover, these approaches are not foolproof against unauthorized changes, whether deliberate or inadvertent, by database administrators or other users with sufficient privileges.

Immutable tables are insert-only tables in which existing data cannot be modified, regardless of user privileges. Updating row values and deleting rows are prohibited. Certain changes to table metadata—for example, renaming table columns—are also prohibited, in order to prevent attempts to circumvent these restrictions. Flattened or external tables, which obtain their data from outside sources, cannot be set to be immutable.

You define an existing table as immutable with ALTER TABLE :

```
ALTER TABLE table SET IMMUTABLE ROWS;
```

Once set, table immutability cannot be reverted, and is immediately applied to all existing table data, and all data that is loaded thereafter. In order to modify the data of an immutable table, you must copy the data to a new table—for example, with COPY , CREATE TABLE...AS , or COPY_TABLE .

When you execute ALTER TABLE...SET IMMUTABLE ROWS on a table, Vertica sets two columns for that table in the system table TABLES . Together, these columns show when the table was made immutable:

- immutable_rows_since_timestamp : Server system time when immutability was applied. This is valuable for long-term timestamp retrieval and efficient comparison.
- immutable_rows_since_epoch : The epoch that was current when immutability was applied. This setting can help protect the table from attempts to pre-insert records with a future timestamp, so that row's epoch is less than the table's immutability epoch.

Enforcement

The following operations are prohibited on immutable tables:

- UPDATE
- DELETE
- ALTER TABLE...DROP COLUMN
- ALTER TABLE...RENAME COLUMN
- ALTER TABLE...ADD COLUMN where the new column is defined with a SET USING clause .

The following partition management functions are disallowed when the target table is immutable:

- COPY_PARTITIONS_TO_TABLE
- MOVE_PARTITIONS_TO_TABLE
- SWAP_PARTITIONS_BETWEEN_TABLES

Allowed operations

In general, you can execute any DML operation on an immutable table that does not affect existing row data—for example, add rows with COPY or INSERT . After you add data to an immutable table, it cannot be changed.

> **Tip**
> A table's immutability can render meaningless certain operations that are otherwise permitted on an immutable table. For example, you can add a column to an immutable table with ALTER TABLE...ADD COLUMN . However, all values in the new column are set to NULL (unless the column is defined with a DEFAULT value), and they cannot be updated.

Other allowed operations fall generally into two categories:

- Changes to a table's DDL that have no effect on its data:
  - ALTER TABLE...SET SCHEMA : Change the table schema.
  - ALTER TABLE...OWNER TO : Transfer table ownership.
  - ALTER TABLE...ALTER COLUMN...SET DATATYPE : Change a column's data type. Column data type changes are allowed only if they have no effect on the column's stored data.
- Block operations on multiple table rows, or the entire table:
  - DROP_PARTITIONS
  - TRUNCATE TABLE
  - DROP TABLE

Disk quotas

By default, schemas and tables are limited only by available disk space and license capacity. You can set disk quotas for schemas or individual tables, for example, to support multi-tenancy. Setting, modifying, or removing a disk quota requires superuser privileges.

Most user operations that increase storage size enforce disk quotas. A table can temporarily exceed its quota during some operations such as recovery. If you lower a quota below the current usage, no data is lost but you cannot add more. Treat quotas as advisory, not as hard limits

A schema quota, if set, must be larger than the largest table quota within it.

A disk quota is a string composed of an integer and a unit of measure (K, M, G, or T), such as '15G' or '1T'. Do not use a space between the number and the unit. No other units of measure are supported.

To set a quota at creation time, use the DISK_QUOTA option for CREATE SCHEMA or CREATE TABLE :

```
=> CREATE SCHEMA internal DISK_QUOTA '10T';
CREATE SCHEMA

=> CREATE TABLE internal.sales (...) DISK_QUOTA '5T';
CREATE TABLE

=> CREATE TABLE internal.leads (...) DISK_QUOTA '12T';
ROLLBACK 0:  Table can not have a greater disk quota than its Schema
```

To modify, add, or remove a quota on an existing schema or table, use ALTER SCHEMA or ALTER TABLE :

```
=> ALTER SCHEMA internal DISK_QUOTA '20T';
ALTER SCHEMA

=> ALTER TABLE internal.sales DISK_QUOTA SET NULL;
ALTER TABLE
```

You can set a quota that is lower than the current usage. The ALTER operation succeeds, the schema or table is temporarily over quota, and you cannot perform operations that increase data usage.

Data that is counted

In Eon Mode, disk usage is an aggregate of all space used by all shards for the schema or table. This value is computed for primary subscriptions only.

In Enterprise Mode, disk usage is the sum space used by all storage containers on all nodes for the schema or table. This sum excludes buddy projections but includes all other projections.

Disk usage is calculated based on compressed size.

When quotas are applied

Quotas, if present, affect most DML and ILM operations, including:

- COPY , INSERT , UPDATE , and MERGE
- Adding or refreshing columns (see ALTER COLUMN)
- REFRESH and START_REFRESH
- Subset restore of a table , if it would exceed the schema quota

The following example shows a failure caused by exceeding a table's quota:

```
=> CREATE TABLE stats(score int) DISK_QUOTA '1k';
CREATE TABLE

=> COPY stats FROM STDIN;
1
2
3
4
5
\.
ERROR 0: Disk Quota Exceeded for the Table object public.stats
HINT: Delete data and PURGE or increase disk quota at the table level
```

DELETE does not free space, because deleted data is still preserved in the storage containers. The delete vector that is added by a delete operation does not count against a quota, so deleting is a quota-neutral operation. Disk space for deleted data is reclaimed when you purge it; see Removing table data .

Some uncommon operations, such as ADD COLUMN , RESTORE , and SWAP PARTITION , can create new storage containers during the transaction. These operations clean up the extra locations upon completion, but while the operation is in progress, a table or schema could exceed its quota. If you get disk-quota errors during these operations, you can temporarily increase the quota, perform the operation, and then reset it.

Quotas do not affect recovery, rebalancing, or Tuple Mover operations.

Monitoring

The DISK_QUOTA_USAGES system table shows current disk usage for tables and schemas that have quotas. This table does not report on objects that do not have quotas.

You can use this table to monitor usage and make decisions about adjusting quotas:

```
=> SELECT * FROM DISK_QUOTA_USAGES;
   object_oid     | object_name | is_schema | total_disk_usage_in_bytes | disk_quota_in_bytes
------------------+-------------+-----------+---------------------------+---------------------
 45035996273705100 | s           | t         |                       307 |               10240
 45035996273705104 | public.t    | f         |                       614 |                1024
 45035996273705108 | s.t         | f         |                       307 |                2048
(3 rows)
```

Managing table columns

After you define a table, you can use ALTER TABLE to modify existing table columns. You can perform the following operations on a column:

In this section

- Renaming columns
- Changing scalar column data type
- Adding a new field to a complex type column
- Defining column values

Renaming columns

You rename a column with ALTER TABLE as follows:

```
ALTER TABLE [schema.]table-name  RENAME [ COLUMN ] column-name TO new-column-name
```

The following example renames a column in the Retail.Product_Dimension table from Product_description to Item_description :

```
=> ALTER TABLE Retail.Product_Dimension
   RENAME COLUMN Product_description TO Item_description;
```

If you rename a column that is referenced by a view, the column does not appear in the result set of the view even if the view uses the wild card (*) to represent all columns in the table. Recreate the view to incorporate the column's new name.

Changing scalar column data type

In general, you can change a column's data type with ALTER TABLE if doing so does not require storage reorganization. After you modify a column's data type, data that you load conforms to the new definition.

The sections that follow describe requirements and restrictions associated with changing a column with a scalar (primitive) data type. For information on modifying complex type columns, see Adding a new field to a complex type column .

Supported data type conversions

Vertica supports conversion for the following data types:

Data Types	**Supported Conversions**
Binary	Expansion and contraction.
Character	All conversions between CHAR, VARCHAR, and LONG VARCHAR.
Exact numeric	All conversions between the following numeric data types: integer data types—INTEGER, INT, BIGINT, TINYINT, INT8, SMALLINT—and NUMERIC values of scale <=18 and precision 0. You cannot modify the scale of NUMERIC data types; however, you can change precision in the ranges (0-18), (19-37), and so on.

Collection	The following conversions are supported: • Collection of one element type to collection of another element type, if the source element type can be coerced to the target element type. • Between arrays and sets. • Collection type to the same type (array to array or set to set), to change bounds or binary size. For details, see Changing Collection Columns .

Unsupported data type conversions

Vertica does not allow data type conversion on types that require storage reorganization:

- Boolean
- DATE/TIME
- Approximate numeric type
- BINARY to VARBINARY and vice versa

You also cannot change a column's data type if the column is one of the following:

- Primary key
- Foreign key
- Included in the SEGMENTED BY clause of any projection for that table.

You can work around some of these restrictions. For details, see Working with column data conversions .

In this section

- Changing column width
- Working with column data conversions

Changing column width

You can expand columns within the same class of data type. Doing so is useful for storing larger items in a column. Vertica validates the data before it performs the conversion.

In general, you can also reduce column widths within the data type class. This is useful to reclaim storage if the original declaration was longer than you need, particularly with strings. You can reduce column width only if the following conditions are true:

- Existing column data is no greater than the new width.
- All nodes in the database cluster are up.

Otherwise, Vertica returns an error and the conversion fails. For example, if you try to convert a column from varchar(25) to varchar(10) Vertica allows the conversion as long as all column data is no more than 10 characters.

In the following example, columns y and z are initially defined as VARCHAR data types, and loaded with values 12345 and 654321 , respectively. The attempt to reduce column z 's width to 5 fails because it contains six-character data. The attempt to reduce column y 's width to 5 succeeds because its content conforms with the new width:

```
=> CREATE TABLE t (x int, y VARCHAR, z VARCHAR);
CREATE TABLE
=> CREATE PROJECTION t_p1 AS SELECT * FROM t SEGMENTED BY hash(x) ALL NODES;
CREATE PROJECTION
=> INSERT INTO t values(1,'12345','654321');
 OUTPUT
--------
      1
(1 row)

=> SELECT * FROM t;
 x |   y   |   z
---+-------+--------
 1 | 12345 | 654321
(1 row)

=> ALTER TABLE t ALTER COLUMN z SET DATA TYPE char(5);
ROLLBACK 2378:  Cannot convert column "z" to type "char(5)"
HINT:  Verify that the data in the column conforms to the new type
=> ALTER TABLE t ALTER COLUMN y SET DATA TYPE char(5);
ALTER TABLE
```

Changing collection columns

If a column is a collection data type, you can use ALTER TABLE to change either its bounds or its maximum binary size. These properties are set at table creation time and can then be altered.

You can make a collection bounded, setting its maximum number of elements, as in the following example.

```
=> ALTER TABLE test.t1 ALTER COLUMN arr SET DATA TYPE array[int,10];
ALTER TABLE

=> \d test.t1
                                List of Fields by Tables
 Schema | Table | Column |      Type       | Size | Default | Not Null | Primary Key | Foreign Key
--------+-------+--------+-----------------+------+---------+----------+-------------+-------------
  test  |  t1   | arr    | array[int8, 10] |   80 |         | f        | f           |
(1 row)
```

Alternatively, you can set the binary size for the entire collection instead of setting bounds. Binary size is set either explicitly or from the DefaultArrayBinarySize configuration parameter. The following example creates an array column from the default, changes the default, and then uses ALTER TABLE to change it to the new default.

```
=> SELECT get_config_parameter('DefaultArrayBinarySize');
 get_config_parameter
----------------------
 100
(1 row)

=> CREATE TABLE test.t1 (arr array[int]);
CREATE TABLE

=> \d test.t1
                                List of Fields by Tables
 Schema | Table | Column |      Type       | Size | Default | Not Null | Primary Key | Foreign Key
--------+-------+--------+-----------------+------+---------+----------+-------------+-------------
  test  |  t1   | arr    | array[int8](96) |   96 |         | f        | f           |
(1 row)

=> ALTER DATABASE DEFAULT SET DefaultArrayBinarySize=200;
ALTER DATABASE

=> ALTER TABLE test.t1 ALTER COLUMN arr SET DATA TYPE array[int];
ALTER TABLE

=> \d test.t1
                                List of Fields by Tables
 Schema | Table | Column |      Type       | Size | Default | Not Null | Primary Key | Foreign Key
--------+-------+--------+-----------------+------+---------+----------+-------------+-------------
  test  |  t1   | arr    | array[int8](200)|  200 |         | f        | f           |
(1 row)
```

Alternatively, you can set the binary size explicitly instead of using the default value.

```
=> ALTER TABLE test.t1 ALTER COLUMN arr SET DATA TYPE array[int](300);
```

Purging historical data

You cannot reduce a column's width if Vertica retains any historical data that exceeds the new width. To reduce the column width, first remove that data from the table:

1. Advance the AHM to an epoch more recent than the historical data that needs to be removed from the table.
2. Purge the table of all historical data that precedes the AHM with the function PURGE_TABLE .

For example, given the previous example, you can update the data in column t.z as follows:

```
=> UPDATE t SET z = '54321';
 OUTPUT
--------
      1
(1 row)

=> SELECT * FROM t;
 x |   y   |   z
---+-------+-------
 1 | 12345 | 54321
(1 row)
```

Although no data in column z now exceeds 5 characters, Vertica retains the history of its earlier data, so attempts to reduce the column width to 5 return an error:

```
=> ALTER TABLE t ALTER COLUMN z SET DATA TYPE char(5);
ROLLBACK 2378:  Cannot convert column "z" to type "char(5)"
HINT:  Verify that the data in the column conforms to the new type
```

You can reduce the column width by purging the table's historical data as follows:

```
=> SELECT MAKE_AHM_NOW();
       MAKE_AHM_NOW
-------------------------------
 AHM set (New AHM Epoch: 6350)
(1 row)

=> SELECT PURGE_TABLE('t');
                                          PURGE_TABLE
-----------------------------------------------------------------------------------------------------------------------
 Task: purge operation
(Table: public.t) (Projection: public.t_p1_b0)
(Table: public.t) (Projection: public.t_p1_b1)

(1 row)

=> ALTER TABLE t ALTER COLUMN z SET DATA TYPE char(5);
ALTER TABLE
```

Working with column data conversions

Vertica conforms to the SQL standard by disallowing certain data conversions for table columns. However, you sometimes need to work around this restriction when you convert data from a non-SQL database. The following examples describe one such workaround, using the following table:

```
=> CREATE TABLE sales(id INT, price VARCHAR) UNSEGMENTED ALL NODES;
CREATE TABLE
=> INSERT INTO sales VALUES (1, '$50.00');
 OUTPUT
--------
      1
(1 row)

=> INSERT INTO sales VALUES (2, '$100.00');
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
=> SELECT * FROM SALES;
 id |  price
----+---------
  1 | $50.00
  2 | $100.00
(2 rows)
```

To convert the price column's existing data type from VARCHAR to NUMERIC, complete these steps:

1. Add a new column for temporary use . Assign the column a NUMERIC data type, and derive its default value from the existing price column.
2. Drop the original price column .
3. Rename the new column to the original column .

Add a new column for temporary use

1. Add a column temp_price to table sales . You can use the new column temporarily, setting its data type to what you want (NUMERIC), and deriving its default value from the price column. Cast the default value for the new column to a NUMERIC data type and query the table:

```
=> ALTER TABLE sales ADD COLUMN temp_price NUMERIC(10,2) DEFAULT
SUBSTR(sales.price, 2)::NUMERIC;
ALTER TABLE

=> SELECT * FROM SALES;
 id |  price  | temp_price
----+---------+------------
  1 | $50.00  |      50.00
  2 | $100.00 |     100.00
(2 rows)
```

2. Use ALTER TABLE to drop the default expression from the new column temp_price . Vertica retains the values stored in this column:

```
=> ALTER TABLE sales ALTER COLUMN temp_price DROP DEFAULT;
ALTER TABLE
```

Drop the original price column

Drop the extraneous price column. Before doing so, you must first advance the AHM to purge historical data that would otherwise prevent the drop operation:

1. Advance the AHM:

```
=> SELECT MAKE_AHM_NOW();
        MAKE_AHM_NOW
-------------------------------
 AHM set (New AHM Epoch: 6354)
(1 row)
```

2. Drop the original price column:

```
=> ALTER TABLE sales DROP COLUMN price CASCADE;
ALTER COLUMN
```

Rename the new column to the original column

You can now rename the temp_price column to price :

1. Use ALTER TABLE to rename the column:

```
=> ALTER TABLE sales RENAME COLUMN temp_price to price;
```

2. Query the sales table again:

```
=> SELECT * FROM sales;
 id | price
----+--------
  1 |  50.00
  2 | 100.00
(2 rows)
```

Adding a new field to a complex type column

You can add new fields to columns of complex types (any combination or nesting of arrays and structs) in native tables. To add a field to an existing table's column, use a single ALTER TABLE statement.

Requirements and restrictions

The following are requirements and restrictions associated with adding a new field to a complex type column:

- New fields can only be added to rows/structs.
- The new type definition must contain all of the existing fields in the complex type column. Dropping existing fields from the complex type is not allowed. All of the existing fields in the new type must exactly match their definitions in the old type.This requirement also means that existing fields cannot be renamed.
- New fields can only be added to columns of native (non-external) tables.
- New fields can be added at any level within a nested complex type. For example, if you have a column defined as ROW(id INT, name ROW(given_name VARCHAR(20), family_name VARCHAR(20)) , you can add a middle_name field to the nested ROW.
- New fields can be of any type, either complex or primitive.
- Blank field names are not allowed when adding new fields. Note that blank field names in complex type columns are allowed when creating the table. Vertica automatically assigns a name to each unnamed field.
- If you change the ordering of existing fields using ALTER TABLE, the change affects existing data in addition to new data. This means it is possible

to reorder existing fields.

- When you call ALTER COLUMN ... SET DATA TYPE to add a field to a complex type column, Vertica will place an O lock on the table preventing DELETE, UPDATE, INSERT, and COPY statements from accessing the table and blocking SELECT statements issued at SERIALIZABLE isolation level, until the operation completes.
- Performance is slower when adding a field to an array element than when adding a field to an element not nested in an array.

Examples

Adding a field

Consider a company storing customer data:

```
=> CREATE TABLE customers(id INT, name VARCHAR, address ROW(street VARCHAR, city VARCHAR, zip INT));
CREATE TABLE
```

The company has just decided to expand internationally, so now needs to add a country field:

```
=> ALTER TABLE customers ALTER COLUMN address
SET DATA TYPE ROW(street VARCHAR, city VARCHAR, zip INT, country VARCHAR);
ALTER TABLE
```

You can view the table definition to confirm the change:

```
=> \d customers
List of Fields by Tables
 Schema |   Table   | Column  |                          Type                          | Size | Default | Not Null | Primary Key | Foreign Key
--------+-----------+---------+--------------------------------------------------------+------+---------+----------+-------------+-------------
 public | customers | id      | int                                                    |    8 |         | f        | f           |
 public | customers | name    | varchar(80)                                            |   80 |         | f        | f           |
 public | customers | address | ROW(street varchar(80),city varchar(80),zip int,country varchar(80)) |   -1 |         | f        | f           |
(3 rows)
```

You can also see that the country field remains null for existing customers:

```
=> SELECT * FROM customers;
 id | name |                                address
----+------+-----------------------------------------------------------------------------
  1 | mina | {"street":"1 allegheny square east","city":"hamden","zip":6518,"country":null}
(1 row)
```

Common error messages

While you can add one or more fields with a single ALTER TABLE statement, existing fields cannot be removed. The following example throws an error because the city field is missing:

```
=> ALTER TABLE customers ALTER COLUMN address SET DATA TYPE ROW(street VARCHAR, state VARCHAR, zip INT, country VARCHAR);
ROLLBACK 2377:  Cannot convert column "address" from "ROW(varchar(80),varchar(80),int,varchar(80))" to type
"ROW(varchar(80),varchar(80),int,varchar(80))"
```

Similarly, you cannot alter the type of an existing field. The following example will throw an error because the zip field's type cannot be altered:

```
=> ALTER TABLE customers ALTER COLUMN address SET DATA TYPE ROW(street VARCHAR, city VARCHAR, zip VARCHAR, country VARCHAR);
ROLLBACK 2377:  Cannot convert column "address" from "ROW(varchar(80),varchar(80),int,varchar(80))" to type
"ROW(varchar(80),varchar(80),varchar(80),varchar(80))"
```

Additional properties

A complex type column's field order follows the order specified in the ALTER command, allowing you to reorder a column's existing fields. The following example reorders the fields of the address column:

```
=> ALTER TABLE customers ALTER COLUMN address
SET DATA TYPE ROW(street VARCHAR, country VARCHAR, city VARCHAR, zip INT);
ALTER TABLE
```

The table definition shows the address column's fields have been reordered:

```
=> \d customers
List of Fields by Tables
 Schema |   Table   | Column  |                          Type                          | Size | Default | Not Null | Primary Key | Foreign Key
--------+-----------+---------+-------------------------------------------------------------+------+---------+----------+-------------+-------------
 public | customers | id      | int                                                    |    8 |         | f        | f           |
 public | customers | name    | varchar(80)                                            |   80 |         | f        | f           |
 public | customers | address | ROW(street varchar(80),country varchar(80),city varchar(80),zip int) |   -1 |         | f        | f           |
 (3 rows)
```

Note that you cannot add new fields with empty names. When creating a complex table, however, you can omit field names, and Vertica automatically assigns a name to each unnamed field:

```
=> CREATE TABLE products(name VARCHAR, description ROW(VARCHAR));
CREATE TABLE
```

Because the field created in the description column has not been named, Vertica assigns it a default name. This default name can be checked in the table definition:

```
=> \d products
List of Fields by Tables
 Schema |  Table   |   Column    |        Type         | Size | Default | Not Null | Primary Key | Foreign Key
--------+----------+-------------+---------------------+------+---------+----------+-------------+-------------
 public | products | name        | varchar(80)         |   80 |         | f        | f           |
 public | products | description | ROW(f0 varchar(80)) |   -1 |         | f        | f           |
(2 rows)
```

Above, we see that the VARCHAR field in the description column was automatically assigned the name f0 . When adding new fields, you must specify the existing Vertica-assigned field name:

```
=> ALTER TABLE products ALTER COLUMN description
SET DATA TYPE ROW(f0 VARCHAR(80), expanded_description VARCHAR(200));
ALTER TABLE
```

Defining column values

You can define a column so Vertica automatically sets its value from an expression through one of the following clauses:

- DEFAULT
- SET USING
- DEFAULT USING

DEFAULT

The DEFAULT option sets column values to a specified value. It has the following syntax:

```
DEFAULT default-expression
```

Default values are set when you:

- Load new rows into a table, for example, with INSERT or COPY . Vertica populates DEFAULT columns in new rows with their default values. Values in existing rows, including columns with DEFAULT expressions, remain unchanged.
- Execute UPDATE on a table and set the value of a DEFAULT column to DEFAULT :

  ```
  => UPDATE table-name SET column-name=DEFAULT;
  ```

- Add a column with a DEFAULT expression to an existing table. Vertica populates the new column with its default values when it is added to the table.

  > **Note**
  > Altering an existing table column to specify a DEFAULT expression has no effect on existing values in that column. Vertica applies the DEFAULT expression only on new rows when they are added to the table, through load operations such as INSERT and COPY. To refresh all values in a column with the column's DEFAULT expression, update the column as shown above.

Restrictions

DEFAULT expressions cannot specify volatile functions with ALTER TABLE...ADD COLUMN. To specify volatile functions, use CREATE TABLE or ALTER TABLE...ALTER COLUMN statements.

SET USING

The SET USING option sets the column value to an expression when the function REFRESH_COLUMNS is invoked on that column. This option has the following syntax:

```
SET USING using-expression
```

This approach is useful for large denormalized (flattened) tables, where multiple columns get their values by querying other tables.

Restrictions

SET USING has the following restrictions:

- Volatile functions are not allowed.
- The expression cannot specify a sequence.
- Vertica limits the use of several meta-functions that copy table data: COPY_TABLE , COPY_PARTITIONS_TO_TABLE , MOVE_PARTITIONS_TO_TABLE , and SWAP_PARTITIONS_BETWEEN_TABLES :
  - If the source and target tables both have SET USING columns, the operation is permitted only if each source SET USING column has a corresponding target SET USING column.
  - If only the source table has SET USING columns, SWAP_PARTITIONS_BETWEEN_TABLES is disallowed.
  - If only the target table has SET USING columns, the operation is disallowed.

DEFAULT USING

The DEFAULT USING option sets DEFAULT and SET USING constraints on a column, equivalent to using DEFAULT and SET USING separately with the same expression on the same column. It has the following syntax:

```
DEFAULT USING expression
```

For example, the following column definitions are effectively identical:

```
=> ALTER TABLE public.orderFact ADD COLUMN cust_name varchar(20)
   DEFAULT USING (SELECT name FROM public.custDim WHERE (custDim.cid = orderFact.cid));
```

```
=> ALTER TABLE public.orderFact ADD COLUMN cust_name varchar(20)
   DEFAULT (SELECT name FROM public.custDim WHERE (custDim.cid = orderFact.cid))
   SET USING (SELECT name FROM public.custDim WHERE (custDim.cid = orderFact.cid));
```

DEFAULT USING supports the same expressions as SET USING and is subject to the same restrictions .

Supported expressions

DEFAULT and SET USING generally support the same expressions. These include:

- Queries
- Other columns in the same table
- Literals (constants)
- All operators supported by Vertica
- The following categories of functions:
  - Null-handling
  - User-defined scalar
  - System information
  - String
  - Mathematical
  - Formatting

Expression restrictions

The following restrictions apply to DEFAULT and SET USING expressions:

- The return value data type must match or be cast to the column data type.
- The expression must return a value that conforms to the column bounds. For example, a column that is defined as a VARCHAR(1) cannot be set to a default string of abc .
- In a temporary table, DEFAULT and SET USING do not support subqueries. If you try to create a temporary table where DEFAULT or SET USING use subquery expressions, Vertica returns an error.

- A column's SET USING expression cannot specify another column in the same table that also sets its value with SET USING. Similarly, a column's DEFAULT expression cannot specify another column in the same table that also sets its value with DEFAULT, or whose value is automatically set to a sequence . However, a column's SET USING expression can specify another column that sets its value with DEFAULT.

  > **Note**
  > You can set a column's DEFAULT expression from another column in the same table that sets its value with SET USING. However, the DEFAULT column is typically set to NULL , as it is only set on load operations that initially set the SET USING column to NULL .

- DEFAULT and SET USING expressions only support one SELECT statement; attempts to include multiple SELECT statements in the expression return an error. For example, given table t1 :

```
=> SELECT * FROM t1;
 a |    b
---+---------
 1 | hello
 2 | world
(2 rows)
```

  Attempting to create table t2 with the following DEFAULT expression returns with an error:

```
=> CREATE TABLE t2 (aa int, bb varchar(30) DEFAULT (SELECT 'I said ')||(SELECT b FROM t1 where t1.a = t2.aa));
ERROR 9745:  Expressions with multiple SELECT statements cannot be used in 'set using' query definitions
```

Disambiguating predicate columns

If a SET USING or DEFAULT query expression joins two columns with the same name, the column names must include their table names. Otherwise, Vertica assumes that both columns reference the dimension table, and the predicate always evaluates to true.

For example, tables orderFact and custDim both include column cid. Flattened table orderFact defines column cust_name with a SET USING query expression. Because the query predicate references columns cid from both tables, the column names are fully qualified:

```
=> CREATE TABLE public.orderFact
(
   ...
   cid int REFERENCES public.custDim(cid),
   cust_name varchar(20) SET USING (
      SELECT name FROM public.custDim WHERE (custDIM.cid = orderFact.cid)),
   ...
)
```

Examples

Derive a column's default value from another column

1. Create table t with two columns, date and state , and insert a row of data:

```
=> CREATE TABLE t (date DATE, state VARCHAR(2));
CREATE TABLE
=> INSERT INTO t VALUES (CURRENT_DATE, 'MA');
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMMIT
SELECT * FROM t;
    date    | state
------------+-------
 2017-12-28 | MA
(1 row)
```

2. Use ALTER TABLE to add a third column that extracts the integer month value from column date :

```
=> ALTER TABLE t ADD COLUMN month INTEGER DEFAULT date_part('month', date);
ALTER TABLE
```

3. When you query table t , Vertica returns the number of the month in column date :

```
=> SELECT * FROM t;
   date    | state | month
------------+-------+-------
 2017-12-28 | MA    |    12
(1 row)
```

Update default column values

1. Update table t by subtracting 30 days from date :

```
=> UPDATE t SET date = date-30;
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
=> SELECT * FROM t;
   date    | state | month
------------+-------+-------
 2017-11-28 | MA    |    12
(1 row)
```

   The value in month remains unchanged.

2. Refresh the default value in month from column date :

```
=> UPDATE t SET month=DEFAULT;
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
=> SELECT * FROM t;
   date    | state | month
------------+-------+-------
 2017-11-28 | MA    |    11
(1 row)
```

Derive a default column value from user-defined scalar function

This example shows a user-defined scalar function that adds two integer values. The function is called add2ints and takes two arguments.

1. Develop and deploy the function, as described in Scalar functions (UDSFs) .
2. Create a sample table, t1 , with two integer columns:

```
=> CREATE TABLE t1 ( x int, y int );
CREATE TABLE
```

3. Insert some values into t1:

```
=> insert into t1 values (1,2);
OUTPUT
--------
      1
(1 row)
=> insert into t1 values (3,4);
 OUTPUT
--------
      1
(1 row)
```

4. Use ALTER TABLE to add a column to t1 , with the default column value derived from the UDSF add2ints :

```
alter table t1 add column z int default add2ints(x,y);
ALTER TABLE
```

5. List the new column:

```
select z from t1;
 z
---
 3
 7
(2 rows)
```

Table with a SET USING column that queries another table for its values

1. Define tables t1 and t2 . Column t2.b is defined to get its data from column t1.b , through the query in its SET USING clause:

```
=> CREATE TABLE t1 (a INT PRIMARY KEY ENABLED, b INT);
CREATE TABLE

=> CREATE TABLE t2 (a INT, alpha VARCHAR(10),
    b INT SET USING (SELECT t1.b FROM t1 WHERE t1.a=t2.a))
    ORDER BY a SEGMENTED BY HASH(a) ALL NODES;
CREATE TABLE
```

   Important
   The definition for table t2 includes SEGMENTED BY and ORDER BY clauses that exclude SET USING column b . If these clauses are omitted, Vertica creates an auto-projection for this table that specifies column b in its SEGMENTED BY and ORDER BY clauses . Inclusion of a SET USING column in any projection's segmentation or sort order prevents function REFRESH_COLUMNS from populating this column. Instead, it returns with an error.

   For details on this and other restrictions, see REFRESH_COLUMNS .

2. Populate the tables with data:

```
=> INSERT INTO t1 VALUES(1,11),(2,22),(3,33),(4,44);
=> INSERT INTO t2 VALUES (1,'aa'),(2,'bb');
=> COMMIT;
COMMIT
```

3. View the data in table t2 : Column in SET USING column b is empty, pending invocation of Vertica function REFRESH_COLUMNS:

```
=> SELECT * FROM t2;
 a | alpha | b
---+-------+---
 1 | aa    |
 2 | bb    |
(2 rows)
```

4. Refresh the column data in table t2 by calling function REFRESH_COLUMNS:

```
=> SELECT REFRESH_COLUMNS ('t2','b', 'REBUILD');
     REFRESH_COLUMNS
---------------------------
 refresh_columns completed
(1 row)
```

   In this example, REFRESH_COLUMNS is called with the optional argument REBUILD. This argument specifies to replace all data in SET USING column b . It is generally good practice to call REFRESH_COLUMNS with REBUILD on any new SET USING column. For details, see REFRESH_COLUMNS .

5. View data in refreshed column b , whose data is obtained from table t1 as specified in the column's SET USING query:

```
=> SELECT * FROM t2 ORDER BY a;
 a | alpha | b
---+-------+----
 1 | aa    | 11
 2 | bb    | 22
(2 rows)
```

Expressions with correlated subqueries

DEFAULT and SET USING expressions support subqueries that can obtain values from other tables, and use those with values in the current table to compute column values. The following example adds a column gmt_delivery_time to fact table customer_orders . The column specifies a DEFAULT expression to set values in the new column as follows:

1. Calls meta-function NEW_TIME , which performs the following tasks:
   - Uses customer keys in customer_orders to query the customers dimension table for customer time zones.
   - Uses the queried time zone data to convert local delivery times to GMT.
2. Populates the gmt_delivery_time column with the converted values.

```
=> CREATE TABLE public.customers(
   customer_key int,
   customer_name varchar(64),
   customer_address varchar(64),
   customer_tz varchar(5),
   ...);

=> CREATE TABLE public.customer_orders(
   customer_key int,
   order_number int,
   product_key int,
   product_version int,
   quantity_ordered int,
   store_key int,
   date_ordered date,
   date_shipped date,
   expected_delivery_date date,
   local_delivery_time timestamptz,
   ...);

=> ALTER TABLE customer_orders ADD COLUMN gmt_delivery_time timestamp
  DEFAULT NEW_TIME(customer_orders.local_delivery_time,
    (SELECT c.customer_tz FROM customers c WHERE (c.customer_key = customer_orders.customer_key)),
    'GMT');
```

Altering table definitions

You can modify a table's definition with ALTER TABLE , in response to evolving database schema requirements. Changing a table definition is often more efficient than staging data in a temporary table, consuming fewer resources and less storage.

For information on making column-level changes, see Managing table columns . For details about changing and reorganizing table partitions, see Partitioning existing table data .

In this section

- Adding table columns
- Dropping table columns
- Altering constraint enforcement
- Renaming tables
- Moving tables to another schema
- Changing table ownership

Adding table columns

You add a column to a persistent table with ALTER TABLE..ADD COLUMN :

```
ALTER TABLE
...
ADD COLUMN [IF NOT EXISTS] column datatype
 [column-constraint]
 [ENCODING encoding-type]
 [PROJECTIONS (projections-list) | ALL PROJECTIONS ]
```

Note

> Before you add columns to a table, verify that all its superprojections are up to date.

Table locking

When you use ADD COLUMN to alter a table, Vertica takes an O lock on the table until the operation completes. The lock prevents DELETE, UPDATE, INSERT, and COPY statements from accessing the table. The lock also blocks SELECT statements issued at SERIALIZABLE isolation level, until the operation completes.

Adding a column to a table does not affect K-safety of the physical schema design.

You can add columns when nodes are down.

Adding new columns to projections

When you add a column to a table, Vertica automatically adds the column to superprojections of that table. The ADD..COLUMN clause can also specify to add the column to one or more non-superprojections, with one of these options:

- PROJECTIONS (*projections-list*): Adds the new column to one or more projections of this table, specified as a comma-delimted list of projection base names . Vertica adds the column to all buddies of each projection. The projection list cannot include projections with pre-aggregated data such as live aggregate projections; otherwise, Vertica rolls back the ALTER TABLE statement.
- ALL PROJECTIONS adds the column to all projections of this table, excluding projections with pre-aggregated data.

For example, the store_orders table has two projections—superprojection store_orders_super , and user-created projection store_orders_p . The following ALTER TABLE..ADD COLUMN statement adds column expected_ship_date to the store_orders table. Because the statement omits the PROJECTIONS option, Vertica adds the column only to the table's superprojection:

```
=> ALTER TABLE public.store_orders ADD COLUMN expected_ship_date date;
ALTER TABLE
=> SELECT projection_column_name, projection_name FROM projection_columns WHERE table_name ILIKE 'store_orders'
    ORDER BY projection_name , projection_column_name;
 projection_column_name |  projection_name
------------------------+--------------------
 order_date             | store_orders_p_b0
 order_no               | store_orders_p_b0
 ship_date              | store_orders_p_b0
 order_date             | store_orders_p_b1
 order_no               | store_orders_p_b1
 ship_date              | store_orders_p_b1
 expected_ship_date     | store_orders_super
 order_date             | store_orders_super
 order_no               | store_orders_super
 ship_date              | store_orders_super
 shipper                | store_orders_super
(11 rows)
```

The following ALTER TABLE...ADD COLUMN statement includes the PROJECTIONS option. This specifies to include projection store_orders_p in the add operation. Vertica adds the new column to this projection and the table's superprojection:

```
=> ALTER TABLE public.store_orders ADD COLUMN delivery_date date PROJECTIONS (store_orders_p);
=> SELECT projection_column_name, projection_name FROM projection_columns WHERE table_name ILIKE 'store_orders'
   ORDER BY projection_name, projection_column_name;
 projection_column_name |  projection_name
------------------------+--------------------
 delivery_date          | store_orders_p_b0
 order_date             | store_orders_p_b0
 order_no               | store_orders_p_b0
 ship_date              | store_orders_p_b0
 delivery_date          | store_orders_p_b1
 order_date             | store_orders_p_b1
 order_no               | store_orders_p_b1
 ship_date              | store_orders_p_b1
 delivery_date          | store_orders_super
 expected_ship_date     | store_orders_super
 order_date             | store_orders_super
 order_no               | store_orders_super
 ship_date              | store_orders_super
 shipper                | store_orders_super
(14 rows)
```

Updating associated table views

Adding new columns to a table that has an associated view does not update the view's result set, even if the view uses a wildcard (*) to represent all table columns. To incorporate new columns, you must recreate the view .

Dropping table columns

ALTER TABLE...DROP COLUMN drops the specified table column and the ROS containers that correspond to the dropped column:

```
ALTER TABLE [schema.]table DROP [ COLUMN ] [IF EXISTS] column [CASCADE | RESTRICT]
```

After the drop operation completes, data backed up from the current epoch onward recovers without the column. Data recovered from a backup that precedes the current epoch re-add the table column. Because drop operations physically purge object storage and catalog definitions (table history) from the table, AT EPOCH (historical) queries return nothing for the dropped column.

The altered table retains its object ID.

> **Note**
> Drop column operations can be fast because these catalog-level changes do not require data reorganization, so Vertica can quickly reclaim disk storage.

Restrictions

- You cannot drop or alter a primary key column or a column that participates in the table partitioning clause.
- You cannot drop the first column of any projection sort order, or columns that participate in a projection segmentation expression.
- In Enterprise Mode, all nodes must be up. This restriction does not apply to Eon mode.
- You cannot drop a column associated with an access policy. Attempts to do so produce the following error:
  ERROR 6482: Failed to parse Access Policies for table "t1"

Using CASCADE to force a drop

If the table column to drop has dependencies, you must qualify the DROP COLUMN clause with the CASCADE option. For example, the target column might be specified in a projection sort order. In this and other cases, DROP COLUMN...CASCADE handles the dependency by reorganizing catalog definitions or dropping a projection. In all cases, CASCADE performs the minimal reorganization required to drop the column.

Use CASCADE to drop a column with the following dependencies:

Dropped column dependency	CASCADE behavior

Any constraint	Vertica drops the column when a FOREIGN KEY constraint depends on a UNIQUE or PRIMARY KEY constraint on the referenced columns.
Specified in projection sort order	Vertica truncates projection sort order up to and including the projection that is dropped without impact on physical storage for other columns and then drops the specified column. For example if a projection's columns are in sort order (a,b,c), dropping column b causes the projection's sort order to be just (a), omitting column (c).
Specified in a projection segmentation expression	The column to drop is integral to the projection definition. If possible, Vertica drops the projection as long as doing so does not compromise K-safety; otherwise, the transaction rolls back.
Referenced as default value of another column	See Dropping a Column Referenced as Default , below.

Dropping a column referenced as default

You might want to drop a table column that is referenced by another column as its default value. For example, the following table is defined with two columns, a and b :, where b gets its default value from column a :

```
=> CREATE TABLE x (a int) UNSEGMENTED ALL NODES;
CREATE TABLE
=> ALTER TABLE x ADD COLUMN b int DEFAULT a;
ALTER TABLE
```

In this case, dropping column a requires the following procedure:

1. Remove the default dependency through ALTER COLUMN..DROP DEFAULT :
   ```
   => ALTER TABLE x ALTER COLUMN b DROP DEFAULT;
   ```
2. Create a replacement superprojection for the target table if one or both of the following conditions is true:
   - The target column is the table's first sort order column. If the table has no explicit sort order, the default table sort order specifies the first table column as the first sort order column. In this case, the new superprojection must specify a sort order that excludes the target column.
   - If the table is segmented, the target column is specified in the segmentation expression. In this case, the new superprojection must specify a segmentation expression that excludes the target column.

   Given the previous example, table x has a default sort order of (a,b). Because column a is the table's first sort order column, you must create a replacement superprojection that is sorted on column b :
   ```
   => CREATE PROJECTION x_p1 as select * FROM x ORDER BY b UNSEGMENTED ALL NODES;
   ```
3. Run START_REFRESH :
   ```
   => SELECT START_REFRESH();
            START_REFRESH
   ----------------------------------------
    Starting refresh background process.

   (1 row)
   ```
4. Run MAKE_AHM_NOW :
   ```
   => SELECT MAKE_AHM_NOW();
         MAKE_AHM_NOW
   -------------------------------
    AHM set (New AHM Epoch: 1231)
   (1 row)
   ```
5. Drop the column:
   ```
   => ALTER TABLE x DROP COLUMN a CASCADE;
   ```

Vertica implements the CASCADE directive as follows:

- Drops the original superprojection for table x (x_super).
- Updates the replacement superprojection x_p1 by dropping column a .

Examples

The following series of commands successfully drops a BYTEA data type column:

```
=> CREATE TABLE t (x BYTEA(65000), y BYTEA, z BYTEA(1));
CREATE TABLE
=> ALTER TABLE t DROP COLUMN y;
ALTER TABLE
=> SELECT y FROM t;
ERROR 2624:  Column "y" does not exist
=> ALTER TABLE t DROP COLUMN x RESTRICT;
ALTER TABLE
=> SELECT x FROM t;
ERROR 2624:  Column "x" does not exist
=> SELECT * FROM t;
 z
---
(0 rows)
=> DROP TABLE t CASCADE;
DROP TABLE
```

The following series of commands tries to drop a FLOAT(8) column and fails because there are not enough projections to maintain K-safety.

```
=> CREATE TABLE t (x FLOAT(8),y FLOAT(08));
CREATE TABLE
=> ALTER TABLE t DROP COLUMN y RESTRICT;
ALTER TABLE
=> SELECT y FROM t;
ERROR 2624:  Column "y" does not exist
=> ALTER TABLE t DROP x CASCADE;
ROLLBACK 2409:  Cannot drop any more columns in t
=> DROP TABLE t CASCADE;
```

Altering constraint enforcement

ALTER TABLE...ALTER CONSTRAINT can enable or disable enforcement of primary key , unique , and check constraints. You must qualify this clause with the keyword ENABLED or DISABLED :

- ENABLED enforces the specified constraint.
- DISABLED disables enforcement of the specified constraint.

For example:

```
ALTER TABLE public.new_sales ALTER CONSTRAINT C_PRIMARY ENABLED;
```

For details, see Constraint enforcement .

Renaming tables

ALTER TABLE...RENAME TO renames one or more tables. Renamed tables retain their original OIDs.

You rename multiple tables by supplying two comma-delimited lists. Vertica maps the names according to their order in the two lists. Only the first list can qualify table names with a schema. For example:

```
=> ALTER TABLE S1.T1, S1.T2 RENAME TO U1, U2;
```

The RENAME TO parameter is applied atomically: all tables are renamed, or none of them. For example, if the number of tables to rename does not match the number of new names, none of the tables is renamed.

> **Caution**
> If a table is referenced by a view, renaming it causes the view to fail, unless you create another table with the previous name to replace the renamed table.

Using rename to swap tables within a schema

You can use ALTER TABLE...RENAME TO to swap tables within the same schema, without actually moving data. You cannot swap tables across schemas.

The following example swaps the data in tables T1 and T2 through intermediary table temp :

1. t1 to temp
2. t2 to t1
3. temp to t2

```
=> DROP TABLE IF EXISTS temp, t1, t2;
DROP TABLE
=> CREATE TABLE t1 (original_name varchar(24));
CREATE TABLE
=> CREATE TABLE t2 (original_name varchar(24));
CREATE TABLE
=> INSERT INTO t1 VALUES ('original name t1');
 OUTPUT
--------
      1
(1 row)

=> INSERT INTO t2 VALUES ('original name t2');
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
=> ALTER TABLE t1, t2, temp RENAME TO temp, t1, t2;
ALTER TABLE
=> SELECT * FROM t1, t2;
  original_name   |  original_name
------------------+------------------
 original name t2 | original name t1
(1 row)
```

Moving tables to another schema

ALTER TABLE...SET SCHEMA moves a table from one schema to another. Vertica automatically moves all projections that are anchored to the source table to the destination schema. It also moves all IDENTITY columns to the destination schema.

Moving a table across schemas requires that you have USAGE privileges on the current schema and CREATE privileges on destination schema. You can move only one table between schemas at a time. You cannot move temporary tables across schemas.

Name conflicts

If a table of the same name or any of the projections that you want to move already exist in the new schema, the statement rolls back and does not move either the table or any projections. To work around name conflicts:

1. Rename any conflicting table or projections that you want to move.
2. Run ALTER TABLE...SET SCHEMA again.

> **Note**
> Vertica lets you move system tables to system schemas. Moving system tables could be necessary to support designs created through the Database Designer .

Example

The following example moves table T1 from schema S1 to schema S2 . All projections that are anchored on table T1 automatically move to schema S2 :

```
=> ALTER TABLE S1.T1 SET SCHEMA S2;
```

Changing table ownership

As a superuser or table owner, you can reassign table ownership with ALTER TABLE...OWNER TO , as follows:

```
ALTER TABLE [schema.]table-name OWNER TO owner-name
```

Changing table ownership is useful when moving a table from one schema to another. Ownership reassignment is also useful when a table owner leaves the company or changes job responsibilities. Because you can change the table owner, the tables won't have to be completely rewritten, you can avoid loss in productivity.

Changing table ownership automatically causes the following changes:

- Grants on the table that were made by the original owner are dropped and all existing privileges on the table are revoked from the previous owner. Changes in table ownership has no effect on schema privileges.
- Ownership of dependent IDENTITY sequences are transferred with the table. However, ownership does not change for named sequences created with CREATE SEQUENCE . To transfer ownership of these sequences, use ALTER SEQUENCE .
- New table ownership is propagated to its projections.

Example

In this example, user Bob connects to the database, looks up the tables, and transfers ownership of table t33 from himself to user Alice.

```
=> \c - Bob
You are now connected as user "Bob".
=> \d
 Schema |  Name  | Kind  |  Owner  | Comment
--------+--------+-------+---------+---------
 public | applog | table | dbadmin |
 public | t33    | table | Bob     |
(2 rows)
=> ALTER TABLE t33 OWNER TO Alice;
ALTER TABLE
```

When Bob looks up database tables again, he no longer sees table t33 :

```
=> \d                List of tables
              List of tables
 Schema |  Name  | Kind  |  Owner  | Comment
--------+--------+-------+---------+---------
 public | applog | table | dbadmin |
(1 row)
```

When user Alice connects to the database and looks up tables, she sees she is the owner of table t33 .

```
=> \c - Alice
You are now connected as user "Alice".
=> \d
            List of tables
 Schema | Name | Kind  | Owner | Comment
--------+------+-------+-------+---------
 public | t33  | table | Alice |
(2 rows)
```

Alice or a superuser can transfer table ownership back to Bob. In the following case a superuser performs the transfer.

```
=> \c - dbadmin
You are now connected as user "dbadmin".
=> ALTER TABLE t33 OWNER TO Bob;
ALTER TABLE
=> \d
                List of tables
 Schema |   Name   | Kind |  Owner  | Comment
--------+----------+-------+---------+---------
 public | applog   | table | dbadmin |
 public | comments | table | dbadmin |
 public | t33      | table | Bob     |
 s1     | t1       | table | User1   |
(4 rows)
```

You can also query system table TABLES to view table and owner information. Note that a change in ownership does not change the table ID.

In the below series of commands, the superuser changes table ownership back to Alice and queries the TABLES system table.

```
=> ALTER TABLE t33 OWNER TO Alice;
ALTER TABLE
=> SELECT table_schema_id, table_schema, table_id, table_name, owner_id, owner_name FROM tables;
  table_schema_id  | table_schema |     table_id      | table_name |     owner_id      | owner_name
-------------------+--------------+-------------------+------------+-------------------+------------
 45035996273704968 | public       | 45035996273713634 | applog     | 45035996273704962 | dbadmin
 45035996273704968 | public       | 45035996273724496 | comments   | 45035996273704962 | dbadmin
 45035996273730528 | s1           | 45035996273730548 | t1         | 45035996273730516 | User1
 45035996273704968 | public       | 45035996273795846 | t33        | 45035996273724576 | Alice
(5 rows)
```

Now the superuser changes table ownership back to Bob and queries the TABLES table again. Nothing changes but the owner_name row, from Alice to Bob.

```
=> ALTER TABLE t33 OWNER TO Bob;
ALTER TABLE
=> SELECT table_schema_id, table_schema, table_id, table_name, owner_id, owner_name FROM tables;
  table_schema_id  | table_schema |     table_id      | table_name |     owner_id      | owner_name
-------------------+--------------+-------------------+------------+-------------------+------------
 45035996273704968 | public       | 45035996273713634 | applog     | 45035996273704962 | dbadmin
 45035996273704968 | public       | 45035996273724496 | comments   | 45035996273704962 | dbadmin
 45035996273730528 | s1           | 45035996273730548 | t1         | 45035996273730516 | User1
 45035996273704968 | public       | 45035996273793876 | foo        | 45035996273724576 | Alice
 45035996273704968 | public       | 45035996273795846 | t33        | 45035996273714428 | Bob
(5 rows)
```

Sequences

Sequences can be used to set the default values of columns to sequential integer values. Sequences guarantee uniqueness, and help avoid constraint enforcement problems and overhead. Sequences are especially useful for primary key columns.

While sequence object values are guaranteed to be unique, they are not guaranteed to be contiguous. For example, two nodes can increment a sequence at different rates. The node with a heavier processing load increments the sequence, but the values are not contiguous with those being incremented on a node with less processing. For details, see Distributing sequences .

Vertica supports the following sequence types:

- *Named sequences* are database objects that generates unique numbers in sequential ascending or descending order. Named sequences are defined independently through CREATE SEQUENCE statements, and are managed independently of the tables that reference them. A table can set the default values of one or more columns to named sequences.
- *IDENTITY column sequences* increment or decrement column's value as new rows are added. Unlike named sequences, IDENTITY sequence types are defined in a table's DDL, so they do not persist independently of that table. A table can contain only one IDENTITY column.

In this section

- Sequence types compared
- Named sequences
- IDENTITY sequences
- Sequence caching

Sequence types compared

The following table lists the differences between the two sequence types:

Supported Behavior	Named Sequence	IDENTITY
Default cache value 250K	•	•
Set initial cache	•	•
Define start value	•	•
Specify increment unit	•	•
Exists as an independent object	•	
Exists only as part of table		•
Create as column constraint		•
Requires name	•	
Use in expressions	•	
Unique across tables	•	
Change parameters	•	
Move to different schema	•	
Set to increment or decrement	•	
Grant privileges to object	•	
Specify minimum value	•	
Specify maximum value	•	

Named sequences

Named sequences are sequences that are defined by CREATE SEQUENCE . Unlike IDENTITY sequences , which are defined in a table's DDL, you create a named sequence as an independent object, and then set it as the default value of a table column.

Named sequences are used most often when an application requires a unique identifier in a table or an expression. After a named sequence returns a value, it never returns the same value again in the same session.

In this section

- Creating and using named sequences
- Distributing sequences
- Altering sequences
- Dropping sequences

Creating and using named sequences

You create a named sequence with CREATE SEQUENCE . The statement requires only a sequence name; all other parameters are optional. To create a sequence, a user must have CREATE privileges on a schema that contains the sequence.

The following example creates an ascending named sequence, my_seq , starting at the value 100:

```
=> CREATE SEQUENCE my_seq START 100;
CREATE SEQUENCE
```

Incrementing and decrementing a sequence

When you create a named sequence object, you can also specify its increment or decrement value by setting its INCREMENT parameter. If you omit this parameter, as in the previous example, the default is set to 1.

You increment or decrement a sequence by calling the function NEXTVAL on it—either directly on the sequence itself, or indirectly by adding new rows to a table that references the sequence . When called for the first time on a new sequence, NEXTVAL initializes the sequence to its start value. Vertica also creates a cache for the sequence. Subsequent NEXTVAL calls on the sequence increment its value.

The following call to NEXTVAL initializes the new my_seq sequence to 100:

```
=> SELECT NEXTVAL('my_seq');
 nextval
---------
     100
(1 row)
```

Getting a sequence's current value

You can obtain the current value of a sequence by calling CURRVAL on it. For example:

```
=> SELECT CURRVAL('my_seq');
 CURRVAL
---------
     100
(1 row)
```

Note

CURRVAL returns an error if you call it on a new sequence that has not yet been initialized by NEXTVAL , or an existing sequence that has not yet been accessed in a new session. For example:

```
=> CREATE SEQUENCE seq2;
CREATE SEQUENCE
=> SELECT currval('seq2');
ERROR 4700:  Sequence seq2 has not been accessed in the session
```

Referencing sequences in tables

A table can set the default values of any column to a named sequence. The table creator must have the following privileges: SELECT on the sequence, and USAGE on its schema.

In the following example, column id gets its default values from named sequence my_seq :

```
=> CREATE TABLE customer(id INTEGER DEFAULT my_seq.NEXTVAL,
  lname VARCHAR(25),
  fname VARCHAR(25),
  membership_card INTEGER
);
```

For each row that you insert into table customer , the sequence invokes the NEXTVAL function to set the value of the id column. For example:

```
=> INSERT INTO customer VALUES (default, 'Carr', 'Mary', 87432);
=> INSERT INTO customer VALUES (default, 'Diem', 'Nga', 87433);
=> COMMIT;
```

For each row, the insert operation invokes NEXTVAL on the sequence my_seq , which increments the sequence to 101 and 102, and sets the id column to those values:

```
=> SELECT * FROM customer;
 id  | lname | fname | membership_card
-----+-------+-------+-----------------
 101 | Carr  | Mary  |           87432
 102 | Diem  | Nga   |           87433
(1 row)
```

Distributing sequences

When you create a sequence, its CACHE parameter determines the number of sequence values each node maintains during a session. The default cache value is 250K, so each node reserves 250,000 values per session for each sequence. The default cache size provides an efficient means for large insert or copy operations.

If sequence caching is set to a low number, nodes are liable to request a new set of cache values more frequently. While it supplies a new cache, Vertica must lock the catalog. Until Vertica releases the lock, other database activities such as table inserts are blocked, which can adversely affect overall performance.

When a new session starts, node caches are initially empty. By default, the initiator node requests and reserves cache for all nodes in a cluster. You can change this default so each node requests its own cache, by setting configuration parameter ClusterSequenceCacheMode to 0.

For information on how Vertica requests and distributes cache among all nodes in a cluster, refer to Sequence caching .

Effects of distributed sessions

Vertica distributes a session across all nodes. The first time a cluster node calls the function NEXTVAL on a sequence to increment (or decrement) its value, the node requests its own cache of sequence values. The node then maintains that cache for the current session. As other nodes call NEXTVAL, they too create and maintain their own cache of sequence values.

During a session, nodes call NEXTVAL independently and at different frequencies. Each node uses its own cache to populate the sequence. All sequence values are guaranteed to be unique, but can be out of order with a NEXTVAL statement executed on another node. As a result, sequence values are often non-contiguous.

In all cases, increments a sequence only once per row. Thus, if the same sequence is referenced by multiple columns, NEXTVAL sets all columns in that row to the same value. This applies to rows of joined tables.

Calculating sequences

Vertica calculates the current value of a sequence as follows:

- At the end of every statement, the state of all sequences used in the session is returned to the initiator node.
- The initiator node calculates the maximum CURRVAL of each sequence across all states on all nodes.
- This maximum value is used as CURRVAL in subsequent statements until another NEXTVAL is invoked.

Losing sequence values

Sequence values in cache can be lost in the following situations:

- If a statement fails after NEXTVAL is called (thereby consuming a sequence value from the cache), the value is lost.
- If a disconnect occurs (for example, dropped session), any remaining values in cache that have not been returned through NEXTVAL are lost.
- When the initiator node distributes a new block of cache to each node where one or more nodes has not used up its current cache allotment. For information on this scenario, refer to Sequence caching .

You can recover lost sequence values by using ALTER SEQUENCE...RESTART , which resets the sequence to the specified value in the next session.

> **Caution**
> Using ALTER SEQUENCE to set a sequence start value below its current value can result in duplicate keys.

Altering sequences

ALTER SEQUENCE can change sequences in two ways:

- Resets parameters that control sequence behavior—for example, its start value, and range of minimum and maximum values. These changes take effect only when you start a new database session.
- Resets sequence name, schema, or ownership. These changes take effect immediately.

> **Note**
> The same ALTER SEQUENCE statement cannot make both types of changes.

Changing sequence behavior

ALTER SEQUENCE can change one or more sequence attributes through the following parameters:

These parameters...	Control...
INCREMENT	How much to increment or decrement the sequence on each call to NEXTVAL .
MINVALUE / MAXVALUE	Range of valid integers.
RESTART	Sequence value on its next call to NEXTVAL.
CACHE / NO CACHE	How many sequence numbers are pre-allocated and stored in memory for faster access.
CYCLE / NO CYCLE	Whether the sequence wraps when its minimum or maximum values are reached.

These changes take effect only when you start a new database session. For example, if you create a named sequence my_sequence that starts at 10 and increments by 1 (the default), each sequence call to NEXTVAL increments its value by 1:

```
=> CREATE SEQUENCE my_sequence START 10;
=> SELECT NEXTVAL('my_sequence');
 nextval
---------
      10
(1 row)
=> SELECT NEXTVAL('my_sequence');
 nextval
---------
      11
(1 row)
```

The following ALTER SEQUENCE statement specifies to restart the sequence at 50:

```
=>ALTER SEQUENCE my_sequence RESTART WITH 50;
```

However, this change has no effect in the current session. The next call to NEXTVAL increments the sequence to 12:

```
=> SELECT NEXTVAL('my_sequence');
 NEXTVAL
---------
      12
(1 row)
```

The sequence restarts at 50 only after you start a new database session:

```
=> \q
$ vsql
Welcome to vsql, the Vertica Analytic Database interactive terminal.

=> SELECT NEXTVAL('my_sequence');
 NEXTVAL
---------
      50
(1 row)
```

Changing sequence name, schema, and ownership

You can use ALTER SEQUENCE to make the following changes to a sequence:

- Rename it (supported only for named sequences).
- Move it to another schema (supported only for named sequences).
- Reassign ownership.

Each of these changes requires separate ALTER SEQUENCE statements. These changes take effect immediately.

For example, the following statement renames a sequence from my_seq to serial :

```
=> ALTER SEQUENCE s1.my_seq RENAME TO s1.serial;
```

This statement moves sequence s1.serial to schema s2 :

```
=> ALTER SEQUENCE s1.my_seq SET SCHEMA TO s2;
```

The following statement reassigns ownership of s2.serial to another user:

```
=> ALTER SEQUENCE s2.serial OWNER TO bertie;
```

> **Note**
> Only a superuser or the sequence owner can change its ownership. Reassignment does not transfer grants from the original owner to the new owner. Grants made by the original owner are dropped.

Dropping sequences

Use DROP SEQUENCE to remove a named sequence. For example:

```
=> DROP SEQUENCE my_sequence;
```

You cannot drop a sequence if one of the following conditions is true:

- Other objects depend on the sequence. DROP SEQUENCE does not support cascade operations.
- A column's DEFAULT expression references the sequence. Before dropping the sequence, you must remove all column references to it.

IDENTITY sequences

IDENTITY (synonymous with AUTO_INCREMENT) columns are defined with a sequence that automatically increments column values as new rows are added. You define an IDENTITY column in a table as follows:

```
CREATE TABLE table-name...
 (column-name {IDENTITY | AUTO_INCREMENT}
    ( [ cache-size | start, increment [, cache-size ] ] )
```

Settings

start	First value to set for this column. **Default:** 1
increment	Positive or negative integer that specifies how much to increment or decrement the sequence on each new row insertion from the previous row value, by default set to 1. To decrement sequence values, specify a negative value. **Note** The actual amount by which column values are incremented or decremented might be larger than the *increment* setting, unless sequence caching is disabled. **Default:** 1

cache-size	How many unique numbers each node caches per session. A value of 0 or 1 disables sequence caching. For details, see Sequence caching . **Default:** 250,000

Managing settings

Like named sequences , you can manage an IDENTITY column with ALTER SEQUENCE —for example, reset its start integer. Two exceptions apply: because the sequence is defined as part of a table column, you cannot change the sequence name or schema. You can query the SEQUENCES system table for the name of an IDENTITY column's sequence. This name is automatically created when you define the table, and conforms to the following convention:

```
table-name_col-name_seq
```

For example, you can change the maximum value of an IDENTITY column that is defined in the testAutoId table:

```
=> SELECT * FROM sequences WHERE identity_table_name  = 'testAutoId';
-[ RECORD 1 ]-------+-------------------------
sequence_schema     | public
sequence_name       | testAutoId_autoIdCol_seq
owner_name          | dbadmin
identity_table_name | testAutoId
session_cache_count | 250000
allow_cycle         | f
output_ordered      | f
increment_by        | 1
minimum             | 1
maximum             | 1000
current_value       | 1
sequence_schema_id  | 45035996273704980
sequence_id         | 45035996274278950
owner_id            | 45035996273704962
identity_table_id   | 45035996274278948

=> ALTER SEQUENCE testAutoId_autoIdCol_seq maxvalue 10000;
ALTER SEQUENCE
```

This change, like other changes to a sequence, take effect only when you start a new database session. One exception applies: changes to the sequence owner take effect immediately.

You can obtain the last value generated for an IDENTITY column by calling LAST_INSERT_ID .

Restrictions

The following restrictions apply to IDENTITY columns:

- A table can contain only one IDENTITY column.
- IDENTITY column values automatically increment before the current transaction is committed; rolling back the transaction does not revert the change.
- You cannot change the value of an IDENTITY column.

Examples

The following example shows how to use the IDENTITY column-constraint to create a table with an ID column. The ID column has an initial value of 1. It is incremented by 1 every time a row is inserted.

1. Create table Premium_Customer :

```
=> CREATE TABLE Premium_Customer(
    ID IDENTITY(1,1),
    lname VARCHAR(25),
    fname VARCHAR(25),
    store_membership_card INTEGER
);
=> INSERT INTO Premium_Customer (lname, fname, store_membership_card )
    VALUES ('Gupta', 'Saleem', 475987);
```

The IDENTITY column has a seed of 1, which specifies the value for the first row loaded into the table, and an increment of 1, which specifies the value that is added to the IDENTITY value of the previous row.

2. Confirm the row you added and see the ID value:

```
=> SELECT * FROM Premium_Customer;
 ID | lname | fname  | store_membership_card
----+-------+--------+----------------------
  1 | Gupta | Saleem |                475987
(1 row)
```

3. Add another row:

```
=> INSERT INTO Premium_Customer (lname, fname, store_membership_card)
   VALUES ('Lee', 'Chen', 598742);
```

4. Call the Vertica function LAST_INSERT_ID . The function returns value 2 because you previously inserted a new customer (Chen Lee), and this value is incremented each time a row is inserted:

```
=> SELECT LAST_INSERT_ID();
 last_insert_id
----------------
              2
(1 row)
```

5. View all the ID values in the Premium_Customer table:

```
=> SELECT * FROM Premium_Customer;
 ID | lname | fname  | store_membership_card
----+-------+--------+----------------------
  1 | Gupta | Saleem |                475987
  2 | Lee   | Chen   |                598742
(2 rows)
```

The next three examples illustrate the three valid ways to use IDENTITY arguments.

The first example uses a cache of 100, and the defaults for start value (1) and increment value (1):

```
=> CREATE TABLE t1(x IDENTITY(100), y INT);
```

The next example specifies the start and increment values as 1, and defaults to a cache value of 250,000:

```
=> CREATE TABLE t2(y IDENTITY(1,1), x INT);
```

The third example specifies start and increment values of 1, and a cache value of 100:

```
=> CREATE TABLE t3(z IDENTITY(1,1,100), zx INT);
```

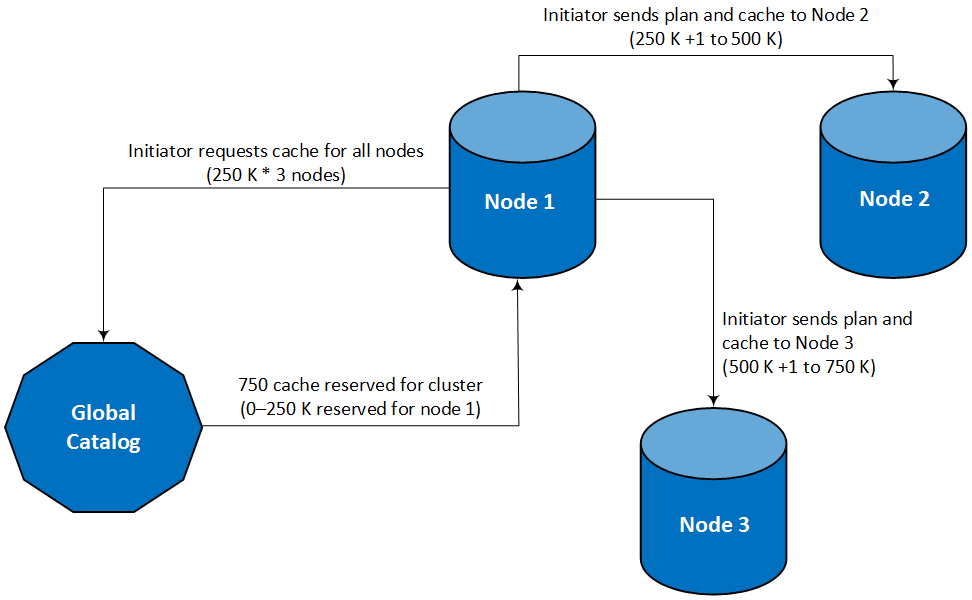
Sequence caching

Caching is similar for all sequence types: named sequences and IDENTITY column sequences. To allocate cache among the nodes in a cluster for a given sequence, Vertica uses the following process.

1. By default, when a session begins, the cluster initiator node requests cache for itself and other nodes in the cluster.
2. The initiator node distributes cache to other nodes when it distributes the execution plan.
3. Because the initiator node requests caching for all nodes, only the initiator locks the global catalog for the cache request.

This approach is optimal for handling large INSERT-SELECT and COPY operations. The following figure shows how the initiator request and distributes cache for a named sequence in a three-node cluster, where caching for that sequence is set to 250 K:

First Request for Cache in a Session



Nodes run out of cache at different times. While executing the same query, nodes individually request additional cache as needed.

For new queries in the same session, the initiator might have an empty cache if it used all of its cache to execute the previous query execution. In this case, the initiator requests cache for all nodes.

Configuring sequence caching

You can change how nodes obtain sequence caches by setting the configuration parameter ClusterSequenceCacheMode to 0 (disabled). When this parameter is set to 0, all nodes in the cluster request their own cache and catalog lock. However, for initial large INSERT-SELECT and COPY operations, when the cache is empty for all nodes, each node requests cache at the same time. These multiple requests result in simultaneous locks on the global catalog, which can adversely affect performance. For this reason, ClusterSequenceCacheMode should remain set to its default value of 1 (enabled).

The following example compares how different settings of ClusterSequenceCacheMode affect how Vertica manages sequence caching. The example assumes a three-node cluster, 250 K caches for each node (the default), and sequence ID values that increment by 1.

Workflow step	**ClusterSequenceCacheMode = 1**	**ClusterSequenceCacheMode = 0**
1	Cache is empty for all nodes. Initiator node requests 250 K cache for each node.	Cache is empty for all nodes. Each node, including initiator, requests its own 250 K cache.
2	Blocks of cache are distributed to each node as follows: • Node 1: 0–250 K • Node 2: 250 K + 1 to 500 K • Node 3: 500 K + 1 to 750 K Each node begins to use its cache as it processes sequence updates.	
3	Initiator node and node 3 run out of cache. Node 2 only uses 250 K +1 to 400 K, 100 K of cache remains from 400 K +1 to 500 K.	

4	Executing same statement: • As each node uses up its cache, it requests a new cache allocation. • If node 2 never uses its cache, the 100-K unused cache becomes a gap in sequence IDs. Executing a new statement in same session, if initiator node cache is empty: • It requests and distributes new cache blocks for all nodes. • Nodes receive a new cache before the old cache is used, creating a gap in ID sequencing.	Executing same or new statement: • As each node uses up its cache, it requests a new cache allocation. • If node 2 never uses its cache, the 100 K unused cache becomes a gap in sequence IDs.

Merging table data

MERGE statements can perform update and insert operations on a target table based on the results of a join with a source data set. The join can match a source row with only one target row; otherwise, Vertica returns an error.

MERGE has the following syntax:

```
MERGE INTO target-table USING source-dataset ON  join-condition
matching-clause[ matching-clause ]
```

Merge operations have at least three components:

- The target table on which to perform update and insert operations. MERGE takes an X (exclusive) lock on the target table until the merge operation is complete.
- Join to another data set , one of the following: a table, view, or subquery result set.
- One or both matching clauses: WHEN MATCHED THEN UPDATE SET and WHEN NOT MATCHED THEN INSERT .

In this section

- Basic MERGE example
- MERGE source options
- MERGE matching clauses
- Update and insert filters
- MERGE optimization
- MERGE restrictions

Basic MERGE example

In this example, a merge operation involves two tables:

- visits_daily logs daily restaurant traffic, and is updated with each customer visit. Data in this table is refreshed every 24 hours.
- visits_history stores the history of customer visits to various restaurants, accumulated over an indefinite time span.

Each night, you merge the daily visit count from visits_daily into visits_history . The merge operation modifies the target table in two ways:

- Updates existing customer data.
- Inserts new rows of data for first-time customers.

One MERGE statement executes both operations as a single (upsert) transaction.

Source and target tables

The source and target tables visits_daily and visits_history are defined as follows:

```
CREATE TABLE public.visits_daily
(
    customer_id int,
    location_name varchar(20),
    visit_time time(0) DEFAULT (now())::timetz(6)
);

CREATE TABLE public.visits_history
(
    customer_id int,
    location_name varchar(20),
    visit_count int
);
```

Table visits_history contains rows of three customers who between them visited two restaurants, Etoile and LaRosa:

```
=> SELECT * FROM visits_history ORDER BY customer_id, location_name;
 customer_id | location_name | visit_count
-------------+---------------+-------------
        1001 | Etoile        |           2
        1002 | La Rosa       |           4
        1004 | Etoile        |           1
(3 rows)
```

By close of business, table visits_daily contains three rows of restaurant visits:

```
=> SELECT * FROM visits_daily ORDER BY customer_id, location_name;
 customer_id | location_name | visit_time
-------------+---------------+------------
        1001 | Etoile        | 18:19:29
        1003 | Lux Cafe      | 08:07:00
        1004 | La Rosa       | 11:49:20
(3 rows)
```

Table data merge

The following MERGE statement merges visits_daily data into visits_history :

- For matching customers, MERGE updates the occurrence count.
- For non-matching customers, MERGE inserts new rows.

```
=> MERGE INTO visits_history h USING visits_daily d
    ON (h.customer_id=d.customer_id AND h.location_name=d.location_name)
    WHEN MATCHED THEN UPDATE SET visit_count = h.visit_count  + 1
    WHEN NOT MATCHED THEN INSERT (customer_id, location_name, visit_count)
    VALUES (d.customer_id, d.location_name, 1);
 OUTPUT
--------
      3
(1 row)
```

MERGE returns the number of rows updated and inserted. In this case, the returned value specifies three updates and inserts:

- Customer 1001 's third visit to Etoile
- New customer 1003 's first visit to new restaurant Lux Cafe
- Customer 1004 's first visit to La Rosa

If you now query table visits_history , the result set shows the merged (updated and inserted) data. Updated and new rows are highlighted:

```
=> SELECT * FROM visits_history ORDER BY customer_id, location_name
 customer_id | location_name | visit_count
-------------+---------------+-------------
        1001 | Etoile        |           3
        1002 | La Rosa       |           4
        1003 | Lux Cafe      |           1
        1004 | Etoile        |           1
        1004 | La Rosa       |           1
(5 rows)
```

MERGE source options

A MERGE operation joins the target table to one of the following data sources:

- Another table
- View
- Subquery result set

Merging from table and view data

You merge data from one table into another as follows:

```
MERGE INTO target-table USING { source-table | source-view } join-condition
  matching-clause[ matching-clause ]
```

If you specify a view, Vertica expands the view name to the query that it encapsulates, and uses the result set as the merge source data.

For example, the VMart table public.product_dimension contains current and discontinued products. You can move all discontinued products into a separate table public.product_dimension_discontinued , as follows:

```
=> CREATE TABLE public.product_dimension_discontinued (
    product_key int,
    product_version int,
    sku_number char(32),
    category_description char(32),
    product_description varchar(128));

=> MERGE INTO product_dimension_discontinued tgt
    USING product_dimension src ON tgt.product_key = src.product_key
                          AND tgt.product_version = src.product_version
    WHEN NOT MATCHED AND src.discontinued_flag='1' THEN INSERT VALUES
      (src.product_key,
       src.product_version,
       src.sku_number,
       src.category_description,
       src.product_description);
 OUTPUT
--------
   1186
(1 row)
```

Source table product_dimension uses two columns, product_key and product_version , to identify unique products. The MERGE statement joins the source and target tables on these columns in order to return single instances of non-matching rows. The WHEN NOT MATCHED clause includes a filter (src.discontinued_flag='1'), which reduces the result set to include only discontinued products. The remaining rows are inserted into target table product_dimension_discontinued .

Merging from a subquery result set

You can merge into a table the result set that is returned by a subquery, as follows:

```
MERGE INTO target-table USING (subquery) sq-alias join-condition
  matching-clause[ matching-clause ]
```

For example, the VMart table public.product_dimension is defined as follows (DDL truncated):

```
CREATE TABLE public.product_dimension
(
    product_key int NOT NULL,
    product_version int NOT NULL,
    product_description varchar(128),
    sku_number char(32),
    ...
)
ALTER TABLE public.product_dimension
    ADD CONSTRAINT C_PRIMARY PRIMARY KEY (product_key, product_version) DISABLED;
```

Columns product_key and product_version comprise the table's primary key. You can modify this table so it contains a single column that concatenates the values of these two columns. This column can be used to uniquely identify each product, while also maintaining the original values from product_key and product_version .

You populate the new column with a MERGE statement that queries the other two columns:

```
=> ALTER TABLE public.product_dimension ADD COLUMN product_ID numeric(8,2);
ALTER TABLE

=> MERGE INTO product_dimension tgt
     USING (SELECT (product_key||'.0'||product_version)::numeric(8,2) AS pid, sku_number
     FROM product_dimension) src
     ON tgt.product_key||'.0'||product_version::numeric=src.pid
     WHEN MATCHED THEN UPDATE SET product_ID = src.pid;
 OUTPUT
--------
  60000
(1 row)
```

The following query verifies that the new column values correspond to the values in product_key and product_version :

```
=> SELECT product_ID, product_key, product_version, product_description
   FROM product_dimension
   WHERE category_description = 'Medical'
     AND product_description ILIKE '%diabetes%'
     AND discontinued_flag = 1 ORDER BY product_ID;
 product_ID | product_key | product_version |          product_description
------------+-------------+-----------------+---------------------------------------
    5836.02 |        5836 |               2 | Brand #17487 diabetes blood testing kit
   14320.02 |       14320 |               2 | Brand #43046 diabetes blood testing kit
   18881.01 |       18881 |               1 | Brand #56743 diabetes blood testing kit
(3 rows)
```

MERGE matching clauses

MERGE supports one instance of the following matching clauses:

-
-

Each matching clause can specify an additional filter, as described in Update and insert filters .

WHEN MATCHED THEN UPDATE SET

Updates all target table rows that are joined to the source table, typically with data from the source table:

```
WHEN MATCHED [ AND update-filter ] THEN UPDATE
  SET { target-column = expression }[,...]
```

Vertica can execute the join only on unique values in the source table's join column. If the source table's join column contains more than one matching value, the MERGE statement returns with a run-time error.

WHEN NOT MATCHED THEN INSERT

WHEN NOT MATCHED THEN INSERT inserts into the target table a new row for each source table row that is excluded from the join:

```
WHEN NOT MATCHED [ AND insert-filter ] THEN INSERT
  [ ( column-list ) ] VALUES ( values-list )
```

column-list is a comma-delimited list of one or more target columns in the target table, listed in any order. MERGE maps *column-list* columns to *values-list* values in the same order, and each column-value pair must be compatible . If you omit *column-list* , Vertica maps *values-list* values to columns according to column order in the table definition.

For example, given the following source and target table definitions:

```
CREATE TABLE t1 (a int, b int, c int);
CREATE TABLE t2 (x int, y int, z int);
```

The following WHEN NOT MATCHED clause implicitly sets the values of the target table columns a , b , and c in the newly inserted rows:

```
MERGE INTO t1 USING t2 ON t1.a=t2.x
  WHEN NOT MATCHED THEN INSERT VALUES (t2.x, t2.y, t2.z);
```

In contrast, the following WHEN NOT MATCHED clause excludes columns t1.b and t2.y from the merge operation. The WHEN NOT MATCHED clause explicitly pairs two sets of columns from the target and source tables: t1.a to t2.x , and t1.c to t2.z . Vertica sets excluded column t1.b . to null:

```
MERGE INTO t1 USING t2 ON t1.a=t2.x
  WHEN NOT MATCHED THEN INSERT (a, c) VALUES (t2.x, t2.z);
```

Update and insert filters

Each WHEN MATCHED and WHEN NOT MATCHED clause in a MERGE statement can optionally specify an update filter and insert filter, respectively:

```
WHEN MATCHED AND update-filter THEN UPDATE ...
WHEN NOT MATCHED AND insert-filter THEN INSERT ...
```

Vertica also supports Oracle syntax for specifying update and insert filters:

```
WHEN MATCHED THEN UPDATE SET column-updates WHERE update-filter
WHEN NOT MATCHED THEN INSERT column-values WHERE insert-filter
```

Each filter can specify multiple conditions. Vertica handles the filters as follows:

- An update filter is applied to the set of matching rows in the target table that are returned by the MERGE join. For each row where the update filter evaluates to true, Vertica updates the specified columns.
- An insert filter is applied to the set of source table rows that are excluded from the MERGE join. For each row where the insert filter evaluates to true, Vertica adds a new row to the target table with the specified values.

For example, given the following data in tables t11 and t22 :

```
=> SELECT * from t11 ORDER BY pk;
 pk | col1 | col2 | SKIP_ME_FLAG
----+------+------+--------------
  1 |    2 |    3 | t
  2 |    3 |    4 | t
  3 |    4 |    5 | f
  4 |      |    6 | f
  5 |    6 |    7 | t
  6 |      |    8 | f
  7 |    8 |      | t
(7 rows)

=> SELECT * FROM t22 ORDER BY pk;
 pk | col1 | col2
----+------+------
  1 |    2 |    4
  2 |    4 |    8
  3 |    6 |
  4 |    8 |   16
(4 rows)
```

You can merge data from table t11 into table t22 with the following MERGE statement, which includes update and insert filters:

```
=> MERGE INTO t22 USING t11 ON ( t11.pk=t22.pk )
  WHEN MATCHED
     AND t11.SKIP_ME_FLAG=FALSE AND (
       COALESCE (t22.col1<>t11.col1, (t22.col1 is null)<>(t11.col1 is null))
     )
  THEN UPDATE SET col1=t11.col1, col2=t11.col2
  WHEN NOT MATCHED
    AND t11.SKIP_ME_FLAG=FALSE
  THEN INSERT (pk, col1, col2) VALUES (t11.pk, t11.col1, t11.col2);
 OUTPUT
--------
     3
(1 row)

=> SELECT * FROM t22 ORDER BY pk;
 pk | col1 | col2
----+------+------
  1 |    2 |    4
  2 |    4 |    8
  3 |    4 |    5
  4 |      |    6
  6 |      |    8
(5 rows)
```

Vertica uses the update and insert filters as follows:

- Evaluates all matching rows against the update filter conditions. Vertica updates each row where the following two conditions both evaluate to true:
  - Source column t11.SKIP_ME_FLAG is set to false.
  - The COALESCE function evaluates to true.
- Evaluates all non-matching rows in the source table against the insert filter. For each row where column t11.SKIP_ME_FLAG is set to false, Vertica inserts a new row in the target table.

MERGE optimization

You can improve MERGE performance in several ways:

- Design projections for optimal MERGE performance .
- Facilitate creation of optimized query plans .
- Use source tables that are smaller than target tables.

Projections for MERGE operations

The Vertica query optimizer automatically chooses the best projections to implement a merge operation. A good projection design strategy provides projections that help the query optimizer avoid extra sort and data transfer operations, and facilitate MERGE performance.

> **Tip**
> You can rely on Database Designer to generate projections that address merge requirements. You can then customize these projections as needed.

For example, the following MERGE statement fragment joins source and target tables tgt and src , respectively, on columns tgt.a and src.b :

```
=> MERGE INTO tgt USING src ON tgt.a = src.b ...
```

Vertica can use a local merge join if projections for tables tgt and src use one of the following projection designs, where inputs are presorted by projection ORDER BY clauses:

- Replicated projections are sorted on:
  - Column a for table tgt
  - Column b for table src
- Segmented projections are identically segmented on:

- Column a for table tgt
- Column b for table src
- Corresponding segmented columns

Optimizing MERGE query plans

Vertica prepares an optimized query plan if the following conditions are all true:

- The MERGE statement contains both matching clauses WHEN MATCHED THEN UPDATE SET and WHEN NOT MATCHED THEN INSERT . If the MERGE statement contains only one matching clause, it uses a non-optimized query plan.
- The MERGE statement excludes update and insert filters .
- The target table join column has a unique or primary key constraint. This requirement does not apply to the source table join column.
- Both matching clauses specify all columns in the target table.
- Both matching clauses specify identical source values.

For details on evaluating an EXPLAIN -generated query plan, see MERGE path .

The examples that follow use a simple schema to illustrate some of the conditions under which Vertica prepares or does not prepare an optimized query plan for MERGE :

```
CREATE TABLE target(a INT PRIMARY KEY, b INT, c INT) ORDER BY b,a;
CREATE TABLE source(a INT, b INT, c INT) ORDER BY b,a;
INSERT INTO target VALUES(1,2,3);
INSERT INTO target VALUES(2,4,7);
INSERT INTO source VALUES(3,4,5);
INSERT INTO source VALUES(4,6,9);
COMMIT;
```

Optimized MERGE statement

Vertica can prepare an optimized query plan for the following MERGE statement because:

- The target table's join column t.a has a primary key constraint.
- All columns in the target table (a,b,c) are included in the UPDATE and INSERT clauses.
- The UPDATE and INSERT clauses specify identical source values: s.a , s.b , and s.c .

```
MERGE INTO target t USING source s ON t.a = s.a
  WHEN MATCHED THEN UPDATE SET a=s.a, b=s.b, c=s.c
  WHEN NOT MATCHED THEN INSERT(a,b,c) VALUES(s.a, s.b, s.c);

 OUTPUT
--------
2
(1 row)
```

The output value of 2 indicates success and denotes the number of rows updated/inserted from the source into the target.

Non-optimized MERGE statement

In the next example, the MERGE statement runs without optimization because the source values in the UPDATE/INSERT clauses are not identical. Specifically, the UPDATE clause includes constants for columns s.a and s.c and the INSERT clause does not:

```
MERGE INTO target t USING source s ON t.a = s.a
  WHEN MATCHED THEN UPDATE SET a=s.a + 1, b=s.b, c=s.c - 1
  WHEN NOT MATCHED THEN INSERT(a,b,c) VALUES(s.a, s.b, s.c);
```

To make the previous MERGE statement eligible for optimization, rewrite the statement so that the source values in the UPDATE and INSERT clauses are identical:

```
MERGE INTO target t USING source s ON t.a = s.a
  WHEN MATCHED THEN UPDATE SET a=s.a + 1, b=s.b, c=s.c -1
  WHEN NOT MATCHED THEN INSERT(a,b,c) VALUES(s.a + 1, s.b, s.c - 1);
```

MERGE restrictions

The following restrictions apply to updating and inserting table data with MERGE .

Constraint enforcement

If primary key, unique key, or check constraints are enabled for automatic enforcement in the target table, Vertica enforces those constraints when you load new data. If a violation occurs, Vertica rolls back the operation and returns an error.

> **Caution**
> If you run MERGE multiple times using the same target and source table, each iteration is liable to introduce duplicate values into the target columns and return with an error.

Columns prohibited from merge

The following columns cannot be specified in a merge operation; attempts to do so return with an error:

- IDENTITY columns, or columns whose default value is set to a named sequence .
- Vmap columns such as __raw__ in flex tables.
- Columns of complex types ARRAY, SET, or ROW.

Removing table data

Vertica provides several ways to remove data from a table:

Delete operation	Description
Drop a table	Permanently remove a table and its definition, optionally remove associated views and projections .
Delete table rows	Mark rows with delete vectors and store them so data can be rolled back to a previous epoch. The data must be purged to reclaim disk space.
Truncate table data	Remove all storage and history associated with a table. The table structure is preserved for future use.
Purge data	Permanently remove historical data from physical storage and free disk space for reuse.
Drop partitions	Remove one more partitions from a table. Each partition contains a related subset of data in the table. Dropping partitioned data is efficient, and provides query performance benefits.

In this section

- Data removal operations compared
- Optimizing DELETE and UPDATE
- Purging deleted data
- Truncating tables

Data removal operations compared

The following table summarizes differences between various data removal operations.

Operations and options	Performance	Auto commits	Saves history
DELETE FROM ``table	Normal	No	Yes
DELETE FROM ``temp-table	High	No	No
DELETE FROM *table* *where-clause*	Normal	No	Yes
DELETE FROM *temp-table* *where-clause*	Normal	No	Yes

DELETE FROM *temp-table* *where-clause* ON COMMIT PRESERVE ROWS	Normal	No	Yes
DELETE FROM *temp-table* *where-clause* ON COMMIT DELETE ROWS	High	Yes	No
DROP *table*	High	Yes	No
TRUNCATE *table*	High	Yes	No
TRUNCATE *temp-table*	High	Yes	No
SELECT DROP_PARTITIONS (...)	High	Yes	No

Choosing the best operation

The following table can help you decide which operation is best for removing table data:

If you want to...	**Use...**
Delete both table data and definitions and start from scratch.	DROP TABLE
Quickly drop data while preserving table definitions, and reload data.	TRUNCATE TABLE
Regularly perform bulk delete operations on logical sets of data.	DROP_PARTITIONS
Occasionally perform small deletes with the option to roll back or review history.	DELETE

Optimizing DELETE and UPDATE

Vertica is optimized for query-intensive workloads, so DELETE and UPDATE queries might not achieve the same level of performance as other queries. A DELETE and UPDATE operation must update all projections, so the operation can only be as fast as the slowest projection.

To improve the performance of DELETE and UPDATE queries, consider the following issues:

- **Query performance after large DELETE operations** : Vertica's implementation of DELETE differs from traditional databases: it does not delete data from disk storage; rather, it marks rows as deleted so they are available for historical queries. Deletion of 10% or more of the total rows in a table can adversely affect queries on that table. In that case, consider purging those rows to improve performance.
- **Recovery performance** : Recovery is the action required for a cluster to restore K-safety after a crash. Large numbers of deleted records can degrade the performance of a recovery. To improve recovery performance, purge the deleted rows .
- **Concurrency** : DELETE and UPDATE take exclusive locks on the table. Only one DELETE or UPDATE transaction on a table can be in progress at a time and only when no load operations are in progress. Delete and update operations on different tables can run concurrently.

> **Tip**
> To optimize large bulk deletion, consider dropping partitions where possible.

Projection column requirements for optimized delete

A projection is optimized for delete and update operations if it contains all columns required by the query predicate. In general, DML operations are significantly faster when performed on optimized projections than on non-optimized projections.

For example, consider the following table and projections:

```
=> CREATE TABLE t (a INTEGER, b INTEGER, c INTEGER);
=> CREATE PROJECTION p1 (a, b, c) AS SELECT * FROM t ORDER BY a;
=> CREATE PROJECTION p2 (a, c) AS SELECT a, c FROM t ORDER BY c, a;
```

In the following query, both p1 and p2 are eligible for DELETE and UPDATE optimization because column a is available:

```
=> DELETE from t WHERE a = 1;
```

In the following example, only projection p1 is eligible for DELETE and UPDATE optimization because the b column is not available in p2 :

```
=> DELETE from t WHERE b = 1;
```

Optimized DELETE in subqueries

To be eligible for DELETE optimization, all target table columns referenced in a DELETE or UPDATE statement's WHERE clause must be in the projection definition.

For example, the following simple schema has two tables and three projections:

```
=> CREATE TABLE tb1 (a INT, b INT, c INT, d INT);
=> CREATE TABLE tb2 (g INT, h INT, i INT, j INT);
```

The first projection references all columns in tb1 and sorts on column a :

```
=> CREATE PROJECTION tb1_p AS SELECT a, b, c, d FROM tb1 ORDER BY a;
```

The buddy projection references and sorts on column a in tb1 :

```
=> CREATE PROJECTION tb1_p_2 AS SELECT a FROM tb1 ORDER BY a;
```

This projection references all columns in tb2 and sorts on column i :

```
=> CREATE PROJECTION tb2_p AS SELECT g, h, i, j FROM tb2 ORDER BY i;
```

Consider the following DML statement, which references tb1.a in its WHERE clause. Since both projections on tb1 contain column a , both are eligible for the optimized DELETE:

```
=> DELETE FROM tb1 WHERE tb1.a IN (SELECT tb2.i FROM tb2);
```

Restrictions

Optimized DELETE operations are not supported under the following conditions:

- With replicated projections if subqueries reference the target table. For example, the following syntax is not supported:
  ```
  => DELETE FROM tb1 WHERE tb1.a IN (SELECT e FROM tb2, tb2 WHERE tb2.e = tb1.e);
  ```
- With subqueries that do not return multiple rows. For example, the following syntax is not supported:
  ```
  => DELETE FROM tb1 WHERE tb1.a = (SELECT k from tb2);
  ```

Projection sort order for optimizing DELETE

Design your projections so that frequently-used DELETE or UPDATE predicate columns appear in the sort order of all projections for large DELETE and UPDATE operations.

For example, suppose most of the DELETE queries you perform on a projection look like the following:

```
=> DELETE from t where time_key < '1-1-2007'
```

To optimize the delete operations, make time_key appear in the ORDER BY clause of all projections. This schema design results in better performance of the delete operation.

In addition, add sort columns to the sort order such that each combination of the sort key values uniquely identifies a row or a small set of rows. For more information, see Choosing sort order: best practices . To analyze projections for sort order issues, use the EVALUATE_DELETE_PERFORMANCE function.

Purging deleted data

In Vertica, delete operations do not remove rows from physical storage. DELETE marks rows as deleted, as does UPDATE , which combines delete and insert operations. In both cases, Vertica retains discarded rows as historical data, which remains accessible to historical queries until it is purged.

The cost of retaining historical data is twofold:

- Disk space is allocated to deleted rows and delete markers.
- Typical (non-historical) queries must read and skip over deleted data, which can impact performance.

A purge operation permanently removes historical data from physical storage and frees disk space for reuse. Only historical data that precedes the Ancient History Mark (AHM) is eligible to be purged.

You can purge data in two ways:

- Set a purge policy .
- Manually purge data .

In both cases, Vertica purges all historical data up to and including the AHM epoch and resets the AHM. See Epochs for additional information about how Vertica uses epochs.

> **Caution**
> Large delete and purge operations can take a long time to complete, so use them sparingly. If your application requires deleting data on a regular basis, such as by month or year, consider designing tables that take advantage of table partitioning . If partitioning is not suitable, consider rebuilding the table .

In this section

- Setting a purge policy
- Manually purging data

Setting a purge policy

The preferred method for purging data is to establish a policy that determines which deleted data is eligible to be purged. Eligible data is automatically purged when the Tuple Mover performs mergeout operations.

Vertica provides two methods for determining when deleted data is eligible to be purged:

- Specifying the time for which delete data is saved
- Specifying the number of epochs that are saved

Specifying the time for which delete data is saved

Specifying the time for which delete data is saved is the preferred method for determining which deleted data can be purged. By default, Vertica saves historical data only when nodes are down.

To change the specified time for saving deleted data, use the HistoryRetentionTime configuration parameter :

```
=> ALTER DATABASE DEFAULT SET HistoryRetentionTime = {seconds | -1};
```

In the above syntax:

- *seconds* is the amount of time (in seconds) for which to save deleted data.
- *-1* indicates that you do not want to use the HistoryRetentionTime configuration parameter to determine which deleted data is eligible to be purged. Use this setting if you prefer to use the other method (HistoryRetentionEpochs) for determining which deleted data can be purged.

The following example sets the history epoch retention level to 240 seconds:

```
=> ALTER DATABASE DEFAULT SET HistoryRetentionTime = 240;
```

Specifying the number of epochs that are saved

Unless you have a reason to limit the number of epochs, Vertica recommends that you specify the time over which delete data is saved.

To specify the number of historical epoch to save through the HistoryRetentionEpochs configuration parameter:

1. Turn off the HistoryRetentionTime configuration parameter:
   ```
   => ALTER DATABASE DEFAULT SET HistoryRetentionTime = -1;
   ```
2. Set the history epoch retention level through the HistoryRetentionEpochs configuration parameter:
   ```
   => ALTER DATABASE DEFAULT SET HistoryRetentionEpochs = {num_epochs | -1};
   ```
   - *num_epochs* is the number of historical epochs to save.
   - *-1* indicates that you do not want to use the HistoryRetentionEpochs configuration parameter to trim historical epochs from the epoch map. By default, HistoryRetentionEpochs is set to -1.

The following example sets the number of historical epochs to save to 40:

```
=> ALTER DATABASE DEFAULT SET HistoryRetentionEpochs = 40;
```

Modifications are immediately implemented across all nodes within the database cluster. You do not need to restart the database.

> **Note**
> If both HistoryRetentionTime and HistoryRetentionEpochs are specified, HistoryRetentionTime takes precedence.

See Epoch management parameters for additional details. See Epochs for information about how Vertica uses epochs.

Disabling purge

If you want to preserve all historical data, set the value of both historical epoch retention parameters to -1, as follows:

```
=> ALTER DABABASE mydb SET HistoryRetentionTime = -1;
=> ALTER DATABASE DEFAULT SET HistoryRetentionEpochs = -1;
```

Manually purging data

You manually purge deleted data as follows:

1. Set the cut-off date for purging deleted data. First, call one of the following functions to verify the current ancient history mark (AHM):
   - GET_AHM_TIME returns a TIMESTAMP value of the AHM.
   - GET_AHM_EPOCH returns the number of the epoch in which the AHM is located.
2. Set the AHM to the desired cut-off date with one of the following functions:
   - SET_AHM_TIME sets the AHM to the epoch that includes the specified TIMESTAMP value on the initiator node .
   - SET_AHM_EPOCH sets the AHM to the specified epoch.
   - MAKE_AHM_NOW sets the AHM to the greatest allowable value. This lets you purge all deleted data.

   If you call SET_AHM_TIME , keep in mind that the timestamp you specify is mapped to an epoch, which by default has a three-minute granularity. Thus, if you specify an AHM time of 2008-01-01 00:00:00.00 , Vertica might purge data from the first three minutes of 2008, or retain data from last three minutes of 2007.

   > **Note**
   > You cannot advance the AHM beyond a point where Vertica is unable to recover data for a down node.

3. Purge deleted data from the desired projections with one of the following functions:
   - PURGE purges all projections in the physical schema.
   - PURGE_TABLE purges all projections anchored to the specified table.
   - PURGE_PROJECTION purges the specified projection.
   - PURGE_PARTITION purges a specified partition.

   The tuple mover performs a mergeout operation to purge the data. Vertica periodically invokes the tuple mover to perform mergeout operations, as configured by tuple mover parameters . You can manually invoke the tuple mover by calling the function DO_TM_TASK .

> **Caution**
> Manual purge operations can take a long time.

See Epochs for additional information about how Vertica uses epochs.

Truncating tables

TRUNCATE TABLE removes all storage associated with the target table and its projections. Vertica preserves the table and the projection definitions. If the truncated table has out-of-date projections, those projections are cleared and marked up-to-date when TRUNCATE TABLE returns.

TRUNCATE TABLE commits the entire transaction after statement execution, even if truncating the table fails. You cannot roll back a TRUNCATE TABLE statement.

Use TRUNCATE TABLE for testing purposes. You can use it to remove all data from a table and load it with fresh data, without recreating the table and its projections.

Table locking

TRUNCATE TABLE takes an O (owner) lock on the table until the truncation process completes. The savepoint is then released.

If the operation cannot obtain an O lock on the target table, Vertica tries to close any internal Tuple Mover sessions that are running on that table. If successful, the operation can proceed. Explicit Tuple Mover operations that are running in user sessions do not close. If an explicit Tuple Mover operation is running on the table, the operation proceeds only when the operation is complete.

Restrictions

You cannot truncate an external table.

Examples

```
=> INSERT INTO sample_table (a) VALUES (3);
=> SELECT * FROM sample_table;
a
---
3
(1 row)
=> TRUNCATE TABLE sample_table;
TRUNCATE TABLE
=> SELECT * FROM sample_table;
a
---
(0 rows)
```

Rebuilding tables

You can reclaim disk space on a large scale by rebuilding tables, as follows:

1. Create a table with the same (or similar) definition as the table to rebuild.
2. Create projections for the new table.
3. Copy data from the target table into the new one with INSERT...SELECT .
4. Drop the old table and its projections.

> **Note**
> Rather than dropping the old table, you can rename it and use it as a backup copy. Before doing so, verify that you have sufficient disk space for both the new and old tables.

5. Rename the new table with ALTER TABLE...RENAME , using the name of the old table.

> **Caution**
> When you rebuild a table, Vertica purges the table of all delete vectors that precede the AHM. This prevents historical queries on any older epoch.

Projection considerations

- You must have enough disk space to contain the old and new projections at the same time. If necessary, you can drop some of the old projections before loading the new table. You must, however, retain at least one superprojection of the old table (or two buddy superprojections to maintain K-safety) until the new table is loaded. (See Prepare disk storage locations for disk space requirements.)
- You can specify different names for the new projections or use ALTER TABLE...RENAME to change the names of the old projections.
- The relationship between tables and projections does not depend on object names. Instead, it depends on object identifiers that are not affected by rename operations. Thus, if you rename a table, its projections continue to work normally.

Dropping tables

DROP TABLE drops a table from the database catalog. If any projections are associated with the table, DROP TABLE returns an error message unless it also includes the CASCADE option. One exception applies: the table only has an auto-generated superprojection (auto-projection) associated with it.

Using CASCADE

In the following example, DROP TABLE tries to remove a table that has several projections associated with it. Because it omits the CASCADE option, Vertica returns an error:

```
=> DROP TABLE d1;
NOTICE: Constraint - depends on Table d1
NOTICE: Projection d1p1 depends on Table d1
NOTICE: Projection d1p2 depends on Table d1
NOTICE: Projection d1p3 depends on Table d1
NOTICE: Projection f1d1p1 depends on Table d1
NOTICE: Projection f1d1p2 depends on Table d1
NOTICE: Projection f1d1p3 depends on Table d1
ERROR: DROP failed due to dependencies: Cannot drop Table d1 because other objects depend on it
HINT: Use DROP ... CASCADE to drop the dependent objects too.
=> DROP TABLE d1 CASCADE;
DROP TABLE
=> CREATE TABLE mytable (a INT, b VARCHAR(256));
CREATE TABLE
=> DROP TABLE IF EXISTS mytable;
DROP TABLE
=> DROP TABLE IF EXISTS mytable; -- Doesn't exist
NOTICE:  Nothing was dropped
DROP TABLE
```

The next attempt includes the CASCADE option and succeeds:

```
=> DROP TABLE d1 CASCADE;
DROP TABLE
=> CREATE TABLE mytable (a INT, b VARCHAR(256));
CREATE TABLE
=> DROP TABLE IF EXISTS mytable;
DROP TABLE
=> DROP TABLE IF EXISTS mytable; -- Doesn't exist
NOTICE:  Nothing was dropped
DROP TABLE
```

Using IF EXISTS

In the following example, DROP TABLE includes the option IF EXISTS . This option specifies not to report an error if one or more of the tables to drop does not exist. This clause is useful in SQL scripts—for example, to ensure that a table is dropped before you try to recreate it:

```
=> DROP TABLE IF EXISTS mytable;
DROP TABLE
=> DROP TABLE IF EXISTS mytable; -- Table doesn't exist
NOTICE:  Nothing was dropped
DROP TABLE
```

Dropping and restoring view tables

Views that reference a table that is dropped and then replaced by another table with the same name continue to function and use the contents of the new table. The new table must have the same column definitions.

Managing client connections

Vertica provides several settings to control client connections:

- Limit the number of client connections a user can have open at the same time.
- Limit the time a client connection can be idle before being automatically disconnected.
- Use connection load balancing to spread the overhead of servicing client connections among nodes.
- Detect unresponsive clients with TCP keepalive.
- Drain a subcluster to reject any new client connections to that subcluster. For details, see Drain client connections .

Total client connections to a given node cannot exceed the limits set in MaxClientSessions.

Changes to a client's MAXCONNECTIONS property have no effect on current sessions; these changes apply only to new sessions. For example, if you change user's connection mode from DATABASE to NODE , current node connections are unaffected. This change applies only to new sessions, which are reserved on the invoking node.

When Vertica closes a client connection, the client's ongoing operations, if any, are canceled.

In this section

- Limiting the number and length of client connections
- Drain client connections
- Connection load balancing

Limiting the number and length of client connections

You can manage how many active sessions a user can open to the server, and the duration of those sessions. Doing so helps prevent overuse of available resources, and can improve overall throughput.

You can define connection limits at two levels:

- Set the MAXCONNECTIONS property on individual users. This property specifies how many sessions a user can open concurrently on individual nodes, or across the database cluster. For example, the following ALTER USER statement allows user Joe up to 10 concurrent sessions:

```
=> ALTER USER Joe MAXCONNECTIONS 10 ON DATABASE;
```

- Set the configuration parameter MaxClientSessions on the database or individual nodes. This parameter specifies the maximum number of client sessions that can run on nodes in the database cluster, by default set to 50. An extra five sessions are always reserved to dbadmin users. This enables them to log in when the total number of client sessions equals MaxClientSessions.

Total client connections to a given node cannot exceed the limits set in MaxClientSessions.

Changes to a client's MAXCONNECTIONS property have no effect on current sessions; these changes apply only to new sessions. For example, if you change user's connection mode from DATABASE to NODE, current node connections are unaffected. This change applies only to new sessions, which are reserved on the invoking node.

Managing TCP keepalive settings

Vertica uses kernel TCP keepalive parameters to detect unresponsive clients and determine when the connection should be closed.Vertica also supports a set of equivalent KeepAlive parameters that can override TCP keepalive parameter settings. By default, all Vertica KeepAlive parameters are set to 0, which signifies to use TCP keepalive settings. To override TCP keepalive settings, set the equivalent parameters at the database level with ALTER DATABASE , or for the current session with ALTER SESSION .

TCP keepalive Parameter	Vertica Parameter	Description
tcp_keepalive_time	KeepAliveIdleTime	Length (in seconds) of the idle period before the first TCP keepalive probe is sent to ensure that the client is still connected.
tcp_keepalive_probes	KeepAliveProbeCount	Number of consecutive keepalive probes that must go unacknowledged by the client before the client connection is considered lost and closed
tcp_keepalive_intvl	KeepAliveProbeInterval	Time interval (in seconds) between keepalive probes.

Examples

The following examples show how to use Vertica KeepAlive parameters to override TCP keepalive parameters as follows:

- After 600 seconds (ten minutes), the first keepalive probe is sent to the client.
- Consecutive keepalive probes are sent every 30 seconds.
- If the client fails to respond to 10 keepalive probes, the connection is considered lost and is closed.

To make this the default policy for client connections, use ALTER DATABASE :

```
=> ALTER DATABASE DEFAULT SET KeepAliveIdleTime = 600;
=> ALTER DATABASE DEFAULT SET KeepAliveProbeInterval = 30;
=> ALTER DATABASE DEFAULT SET KeepAliveProbeCount = 10;
```

To override database-level policies for the current session, use ALTER SESSION :

```
=> ALTER SESSION SET KeepAliveIdleTime = 400;
=> ALTER SESSION SET KeepAliveProbeInterval = 72;
=> ALTER SESSION SET KeepAliveProbeCount = 60;
```

Query system table CONFIGURATION_PARAMETERS to verify database and session settings of the three Vertica KeepAlive parameters:

```
=> SELECT parameter_name, database_value, current_value FROM configuration_parameters WHERE parameter_name ILIKE 'KeepAlive%';
    parameter_name     | database_value | current_value
-----------------------+----------------+---------------
 KeepAliveProbeCount   | 10             | 60
 KeepAliveIdleTime     | 600            | 400
 KeepAliveProbeInterval | 30            | 72
(3 rows)
```

Limiting idle session length

If a client continues to respond to TCP keepalive probes , but is not running any queries, the client's session is considered idle. Idle sessions eventually time out. The maximum time that sessions are allowed to idle can be set at three levels, in descending order of precedence:

- As dbadmin, set the IDLESESSIONTIMEOUT property for individual users. This property overrides all other session timeout settings.
- Users can limit the idle time of the current session with SET SESSION IDLESESSIONTIMEOUT . Non-superusers can only set their session idle time to a value equal to or lower than their own IDLESESSIONTIMEOUT setting. If no session idle time is explicitly set for a user, the session idle time for that user is inherited from the node or database settings.
- As dbadmin, set configuration parameter DEFAULTIDLESESSIONTIMEOUT . on the database or on individual nodes. This You can limit the default database cluster or individual nodes, with configuration parameter DEFAULTIDLESESSIONTIMEOUT . This parameter sets the default timeout value for all non-superusers.

All settings apply to sessions that are continuously idle—that is, sessions where no queries are running. If a client is slow or unresponsive during query execution, that time does not apply to timeouts. For example, the time that is required for a streaming batch insert is not counted towards timeout. The server identifies a session as idle starting from the moment it starts to wait for any type of message from that session.

Viewing session settings

The following sections demonstrate how you can query the database for details about the session and connection limits.

Session length limits

Use SHOW DATABASE to view the session length limit for the database:

```
=> SHOW DATABASE DEFAULT DEFAULTIDLESESSIONTIMEOUT;
          name           | setting
-------------------------+---------
 DefaultIdleSessionTimeout | 2 day
(1 row)
```

Use SHOW to view the length limit for the current session:

```
=> SHOW IDLESESSIONTIMEOUT;
       name        | setting
-------------------+---------
 idlesessiontimeout | 1
(1 row)
```

Connection limits

Use SHOW DATABASE to view the connection limits for the database:

```
=> SHOW DATABASE DEFAULT MaxClientSessions;
       name        | setting
-------------------+---------
 MaxClientSessions | 50
(1 row)
```

Query USERS to view the connection limits for users:

```
=> SELECT user_name, max_connections, connection_limit_mode FROM users
    WHERE user_name != 'dbadmin';
 user_name | max_connections | connection_limit_mode
-----------+-----------------+-----------------------
 SuzyX     | 3               | database
 Joe       | 10              | database
(2 rows)
```

Closing user sessions

To manually close a user session, use CLOSE_USER_SESSIONS :

```
=> SELECT CLOSE_USER_SESSIONS ('Joe');
                          close_user_sessions
-------------------------------------------------------------------------------
 Close all sessions for user Joe sent. Check v_monitor.sessions for progress.
(1 row)
```

Example

A user executes a query, and for some reason the query takes an unusually long time to finish (for example, because of server traffic or query complexity). In this case, the user might think the query failed, and opens another session to run the same query. Now, two sessions run the same query, using an extra connection.

To prevent this situation, you can limit how many sessions individual users can run, by modifying their MAXCONNECTIONS user property. This can help minimize the chances of running redundant queries. It also helps prevent users from consuming all available connections, as set by the database For example, the following setting on user SuzyQ limits her to two database sessions at any time:

```
=> CREATE USER SuzyQ MAXCONNECTIONS 2 ON DATABASE;
```

Limiting Another issue setting client connections prevents is when a user connects to the server many times. Too many user connections exhausts the number of allowable connections set by database configuration parameter MaxClientSessions .

> **Note**
> No user can have a MAXCONNECTIONS limit greater than the MaxClientSessions setting.

Cluster changes and connections

Behavior changes can occur with client connection limits when the following changes occur to a cluster:

- You add or remove a node.
- A node goes down or comes back up.

Changes in node availability between connection requests have little impact on connection limits.

In terms of honoring connection limits, no significant impact exists when nodes go down or come up in between connection requests. No special actions are needed to handle this. However, if a node goes down, its active session exits and other nodes in the cluster also drop their sessions. This frees up connections. The query may hang in which case the blocked sessions are reasonable and as expected.

Drain client connections

Eon Mode only

Draining client connections in a subclusters prepares the subcluster for shutdown by marking all nodes in the subcluster as draining. Work from existing user sessions continues on draining nodes, but the nodes refuse new client connections and are excluded from load-balancing operations. If clients attempt to connect to a draining node, they receive an error that informs them of the draining status. Load balancing operations exclude draining nodes, so clients that opt-in to connection load balancing should receive a connection error only if all nodes in the load balancing policy are draining. You do not need to change any connection load balancing configurations to use this feature. dbadmin can still connect to draining nodes.

To drain client connections before shutting down a subcluster, you can use the SHUTDOWN_WITH_DRAIN function. This function performs a Graceful Shutdown that marks a subcluster as draining until either the existing connections complete their work and close or a user-specified timeout is reached. When one of these conditions is met, the function proceeds to shutdown the subcluster. Vertica provides several meta-functions that allow you to independently perform each step of the SHUTDOWN_WITH_DRAIN process. You can use the START_DRAIN_SUBCLUSTER function to mark a subcluster as draining and then the SHUTDOWN_SUBCLUSTER function to shut down a subcluster once its connections have closed.

You can use the CANCEL_DRAIN_SUBCLUSTER function to mark all nodes in a subcluster as not draining. As soon as a node is both UP and not draining, the node accepts new client connections. If all nodes in a draining subcluster are down, the draining status of its nodes is automatically reset to not draining.

You can query the DRAINING_STATUS system table to monitor the draining status of each node as well as client connection information, such as the number of active user sessions on each node.

The following example drains a subcluster named analytics, then cancels the draining of the subcluster.

To mark the analytics subcluster as draining, call SHUTDOWN_WITH_DRAIN with a negative timeout value:

```
=> SELECT SHUTDOWN_WITH_DRAIN('analytics', -1);
NOTICE 0:  Draining has started on subcluster (analytics)
```

You can confirm that the subcluster is draining by querying the DRAINING_STATUS system table:

```
=> SELECT node_name, subcluster_name, is_draining FROM draining_status ORDER BY 1;
node_name          | subcluster_name    | is_draining
-------------------+--------------------+-------------
verticadb_node0001 | default_subcluster | f
verticadb_node0002 | default_subcluster | f
verticadb_node0003 | default_subcluster | f
verticadb_node0004 | analytics          | t
verticadb_node0005 | analytics          | t
verticadb_node0006 | analytics          | t
```

If a client attempts to connect directly to a node in the draining subcluster, they receive the following error message:

```
$ /opt/vertica/bin/vsql -h nodeIP --password password verticadb analyst
vsql: FATAL 10611:  New session rejected because subcluster to which this node belongs is draining connections
```

To cancel the graceful shutdown of the analytics subcluster, you can type Ctrl+C:

```
=> SELECT SHUTDOWN_WITH_DRAIN('analytics', -1);
NOTICE 0:  Draining has started on subcluster (analytics)
^CCancel request sent
ERROR 0:  Cancel received after draining started and before shutdown issued. Nodes will not be shut down. The subclusters are still in the draining state.
HINT:  Run cancel_drain_subcluster('') to restore all nodes to the 'not_draining' state
```

As mentioned in the above hint, you can run CANCEL_DRAIN_SUBCLUSTER to reset the status of the draining nodes in the subcluster to not draining:

```
=> SELECT CANCEL_DRAIN_SUBCLUSTER('analytics');
             CANCEL_DRAIN_SUBCLUSTER
-----------------------------------------------------------
Targeted subcluster: 'analytics'
Action: CANCEL DRAIN

(1 row)
```

To confirm that the subcluster is no longer draining, you can again query the DRAINING_STATUS system table:

```
=> SELECT node_name, subcluster_name, is_draining FROM draining_status ORDER BY 1;
node_name          | subcluster_name    | is_draining
-------------------+--------------------+-------------
verticadb_node0001 | default_subcluster | f
verticadb_node0002 | default_subcluster | f
verticadb_node0003 | default_subcluster | f
verticadb_node0004 | analytics          | f
verticadb_node0005 | analytics          | f
verticadb_node0006 | analytics          | f
(6 rows)
```

Connection load balancing

Each client connection to a host in the Vertica cluster requires a small overhead in memory and processor time. If many clients connect to a single host, this overhead can begin to affect the performance of the database. You can spread the overhead of client connections by dictating that certain clients connect to specific hosts in the cluster. However, this manual balancing becomes difficult as new clients and hosts are added to your environment.

Connection load balancing helps automatically spread the overhead of client connections across the cluster by having hosts redirect client connections to other hosts. By redirecting connections, the overhead from client connections is spread across the cluster without having to manually assign particular hosts to individual clients. Clients can connect to a small handful of hosts, and they are naturally redirected to other hosts in the cluster. Load balancing does not redirect connections to draining hosts. For more information see, Drain client connections .

Native connection load balancing overview

Native connection load balancing is a feature built into the Vertica Analytic Database server and client libraries as well as vsql . Both the server and the client need to enable load balancing for it to function. If connection load balancing is enabled, a host in the database cluster can redirect a client's attempt to connect to it to another currently-active host in the cluster. This redirection is based on a load balancing policy. This redirection only takes place once, so a client is not bounced from one host to another.

Because native connection load balancing is incorporated into the Vertica client libraries, any client application that connects to Vertica transparently takes advantage of it simply by setting a connection parameter.

How you choose to implement connection load balancing depends on your network environment. Since native load connection balancing is easier to implement, you should use it unless your network configuration requires that clients be separated from the hosts in the Vertica database by a firewall.

For more about native connection load balancing, see About Native Connection Load Balancing.

In this section

- About native connection load balancing
- Classic connection load balancing
- Connection load balancing policies

About native connection load balancing

Native connection load balancing is a feature built into the Vertica server and client libraries that helps spread the CPU and memory overhead caused by client connections across the hosts in the database. It can prevent unequal distribution of client connections among hosts in the cluster.

There are two types of native connection load balancing:

- Cluster-wide balancing—This method the legacy method of connection load balancing. It was the only type of load balancing prior to Vertica version 9.2. Using this method, you apply a single load balancing policy across the entire cluster. All connection to the cluster are handled the same way.
- Load balancing policies—This method lets you set different load balancing policies depending on the source of client connection. For example, you can have a policy that redirects connections from outside of your local network to one set of nodes in your cluster, and connections from within your local network to another set of nodes.

Classic connection load balancing

The classic connection load balancing feature applies a single policy for all client connections to your database. Both your database and the client must enable the load balancing option in order for connections to be load balanced. When both client and server enable load balancing, the following process takes place when the client attempts to open a connection to Vertica:

1. The client connects to a host in the database cluster, with a connection parameter indicating that it is requesting a load-balanced connection.
2. The host chooses a host from the list of currently up hosts in the cluster, according to the current load balancing scheme . Under all schemes, it is possible for a host to select itself.
3. The host tells the client which host it selected to handle the client's connection.
4. If the host chose another host in the database to handle the client connection, the client disconnects from the initial host. Otherwise, the client jumps to step 6.
5. The client establishes a connection to the host that will handle its connection. The client sets this second connection request so that the second host does not interpret the connection as a request for load balancing.
6. The client connection proceeds as usual, (negotiating encryption if the connection has SSL enabled, and proceeding to authenticating the user).

This process is transparent to the client application. The client driver automatically disconnects from the initial host and reconnects to the host selected for load balancing.

Requirements

- In mixed IPv4 and IPv6 environments, balancing only works for the address family for which you have configured native load balancing. For example, if you have configured load balancing using an IPv4 address, then IPv6 clients cannot use load balancing, however the IPv6 clients can still connect, but load balancing does not occur.
- The native load balancer returns an IP address for the client to use. This address must be one that the client can reach. If your nodes are on a private network, native load-balancing requires you to publish a public address in one of two ways:
  - Set the public address on each node. Vertica saves that address in the export_address field in the NODES system table.
  - Set the subnet on the database. Vertica saves that address in the export_subnet field in the DATABASES system table.

Load balancing schemes

The load balancing scheme controls how a host selects which host to handle a client connection. There are three available schemes:

- NONE (default): Disables native connection load balancing.
- ROUNDROBIN : Chooses the next host from a circular list of hosts in the cluster that are up—for example, in a three-node cluster, iterates over node1, node2, and node3, then wraps back to node1. Each host in the cluster maintains its own pointer to the next host in the circular list, rather than there being a single cluster-wide state.
- RANDOM : Randomly chooses a host from among all hosts in the cluster that are up.

You set the native connection load balancing scheme using the SET_LOAD_BALANCE_POLICY function. See Enabling and Disabling Native Connection Load Balancing for instructions.

Driver notes

- Native connection load balancing works with the ADO.NET driver's connection pooling. The connection the client makes to the initial host, and the final connection to the load-balanced host, use pooled connections if they are available.
- If a client application uses the JDBC and ODBC driver with third-party connection pooling solutions, the initial connection is not pooled because it is not a full client connection. The final connection is pooled because it is a standard client connection.

Connection failover

The client libraries include a failover feature that allow them to connect to backup hosts if the host specified in the connection properties is unreachable. When using native connection load balancing, this failover feature is only used for the initial connection to the database. If the host to which the client was redirected does not respond to the client's connection request, the client does not attempt to connect to a backup host and instead returns a connection error to the user.

Clients are redirected only to hosts that are known to be up. Thus, this sort of connection failure should only occur if the targeted host goes down at the same moment the client is redirected to it. For more information, see ADO.NET connection failover , JDBC connection failover , and Connection failover .

In this section

- Enabling and disabling classic connection load balancing
- Monitoring legacy connection load balancing

Enabling and disabling classic connection load balancing

Only a database superuser can enable or disable classic cluster-wide connection load balancing. To enable or disable load balancing, use the SET_LOAD_BALANCE_POLICY function to set the load balance policy. Setting the load balance policy to anything other than 'NONE' enables load balancing on the server. The following example enables native connection load balancing by setting the load balancing policy to ROUNDROBIN.

```
=> SELECT SET_LOAD_BALANCE_POLICY('ROUNDROBIN');
                SET_LOAD_BALANCE_POLICY
--------------------------------------------------------------------------------
Successfully changed the client initiator load balancing policy to: roundrobin
(1 row)
```

To disable native connection load balancing, use SET_LOAD_BALANCE_POLICY to set the policy to 'NONE':

```
=> SELECT SET_LOAD_BALANCE_POLICY('NONE');
SET_LOAD_BALANCE_POLICY
------------------------------------------------------------------------
Successfully changed the client initiator load balancing policy to: none
(1 row)
```

> **Note**
> When a client makes a connection, the native load-balancer chooses a node and returns the value from the export_address column in the NODES table. The client then uses the export_address to connect. The node_address specifies the address to use for inter-node and spread

communications. When a database is installed, the export_address and node_address are set to the same value. If you installed Vertica on a private address, then you must set the export_address to a *public* address for each node.

By default, client connections are not load balanced, even when connection load balancing is enabled on the server. Clients must set a connection parameter to indicates they are willing to have their connection request load balanced. See Load balancing in ADO.NET , Load balancing in JDBC , and Load balancing , for information on enabling load balancing on the client. For vsql, use the -C command-line option to enable load balancing.

Important
In mixed IPv4 and IPv6 environments, balancing only works for the address family for which you have configured load balancing. For example, if you have configured load balancing using an IPv4 address, then IPv6 clients cannot use load balancing, however the IPv6 clients can still connect, but load balancing does not occur.

Resetting the load balancing state

When the load balancing policy is ROUNDROBIN, each host in the Vertica cluster maintains its own state of which host it will select to handle the next client connection. You can reset this state to its initial value (usually, the host with the lowest-node id) using the RESET_LOAD_BALANCE_POLICY function:

```
=> SELECT RESET_LOAD_BALANCE_POLICY();
RESET_LOAD_BALANCE_POLICY
--------------------------------------------------------------------------------
Successfully reset stateful client load balance policies: "roundrobin".
(1 row)
```

See also

- Monitoring legacy connection load balancing
- Load balancing in JDBC
- Load balancing
- Load balancing in ADO.NET

Monitoring legacy connection load balancing

Query the LOAD_BALANCE_POLICY column of the V_CATALOG. DATABASES to determine the state of native connection load balancing on your server:

```
=> SELECT LOAD_BALANCE_POLICY FROM V_CATALOG.DATABASES;
LOAD_BALANCE_POLICY
---------------------
roundrobin
(1 row)
```

Determining to which node a client has connected

A client can determine the node to which is has connected by querying the NODE_NAME column of the V_MONITOR. CURRENT_SESSION table:

```
=> SELECT NODE_NAME FROM V_MONITOR.CURRENT_SESSION;
NODE_NAME
------------------
v_vmart_node0002
(1 row)
```

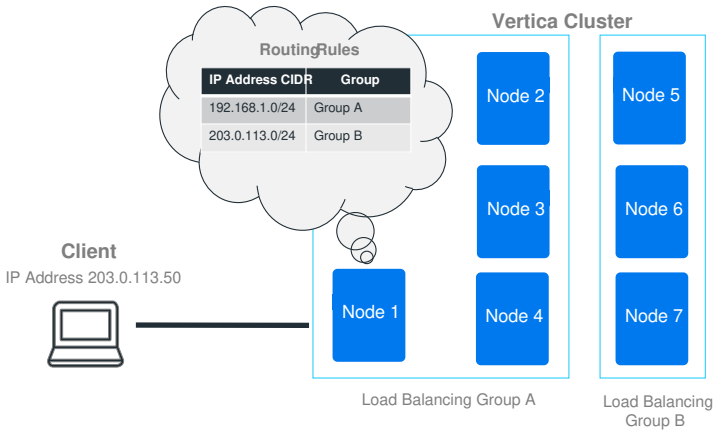
Connection load balancing policies

Connection load balancing policies help spread the load of servicing client connections by redirecting connections based on the connection's origin. These policies can also help prevent nodes reaching their client connection limits and rejecting new connections by spreading connections among nodes. See Limiting the number and length of client connections for more information about client connection limits.

A load balancing policy consists of:

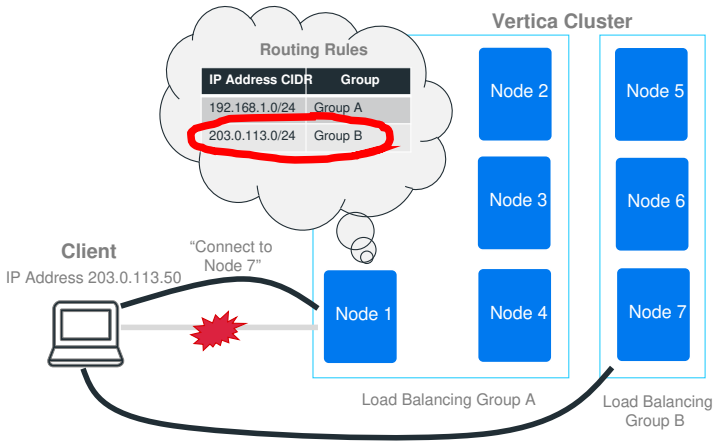
- **Network addresses** that identify particular IP address and port number combinations on a node.
- One or more **connection load balancing groups** that consists of network addresses that you want to handle client connections. You define load balancing groups using fault groups , subclusters , or a list of network addresses.
- One or more **routing rules** that map a range of client IP addresses to a connection load balancing group.

When a client connects to a node in the database with load balancing enabled, the node evaluates all of the routing rules based on the client's IP address to determine if any match. If more than one rule matches the IP address, the node applies the most specific rule (the one that affects the fewest IP addresses).



If the node finds a matching rule, it uses the rule to determine the pool of potential nodes to handle the client connection. When evaluating potential target nodes, it always ensures that the nodes are currently up. The initially-contacted node then chooses one of the nodes in the group based on the group's distribution scheme. This scheme can be either choosing a node at random, or choosing a node in a rotating "round-robin" order. For example, in a three-node cluster, the round robin order would be node 1, then node 2, then node 3, and then back to node 1 again.

After it processes the rules, if the node determines that another node should handle the client's connection, it tells the client which node it has chosen. The client disconnects from the initial node and connects to the chosen node to continue with the connection process (either negotiating encryption if the connection has TLS/SSL enabled, or authentication).



If the initial node chooses itself based on the routing rules, it tells the client to proceed to the next step of the connection process.

If no routing rule matches the incoming IP address, the node checks to see if classic connection load balancing is enabled by both Vertica and the client. If so, it handles the connection according to the classic load balancing policy. See Classic connection load balancing for more information.

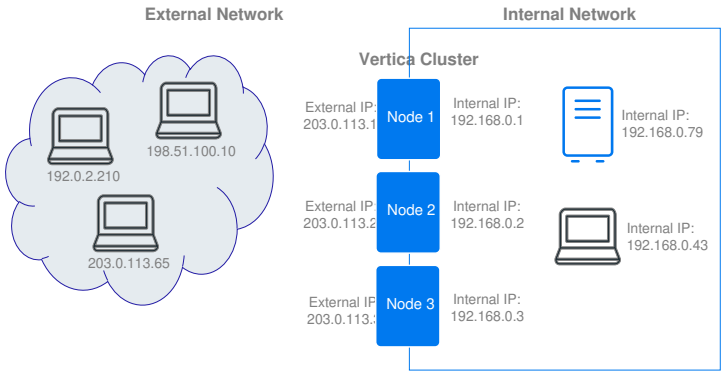
Finally, if the database is running in Eon Mode, the node tries to apply a default interior load balancing rule. See Default Subcluster Interior Load Balancing Policy below.

If no routing rule matches the incoming IP address and classic load balancing and the default subcluster interior load balancing rule did not apply, the node handles the connection itself. It also handles the connection itself if it cannot follow the load balancing rule. For example, if all nodes in the load balancing group targeted by the rule are down, then the initially-contacted node handles the client connection itself. In this case, the node does not attempt to apply any other less-restrictive load balancing rules that would apply to the incoming connection. It only attempts to apply a single load balancing rule.

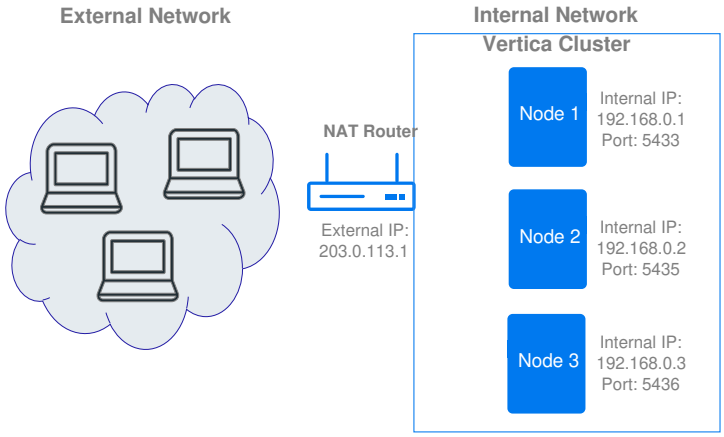
Use cases

Using load balancing policies you can:

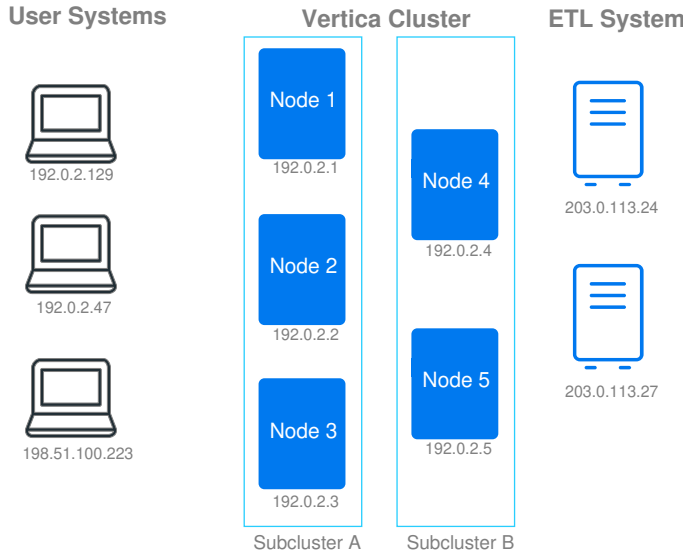
- Ensure connections originating from inside or outside of your internal network are directed to a valid IP address for the client. For example, suppose your Vertica nodes have two IP addresses: one for the external network and another for the internal network. These networks are mutually exclusive. You cannot reach the private network from the public, and you cannot reach the public network from the private. Your load balancing rules need to provide the client with an IP address they can actually reach.



- Enable access to multiple nodes of a Vertica cluster that are behind a NAT router. A NAT router is accessible from the outside network via a single IP address. Systems within the NAT router's private network can be accessed on this single IP address using different port numbers. You can create a load balancing policy that redirects a client connection to the NAT's IP address but with a different port number.

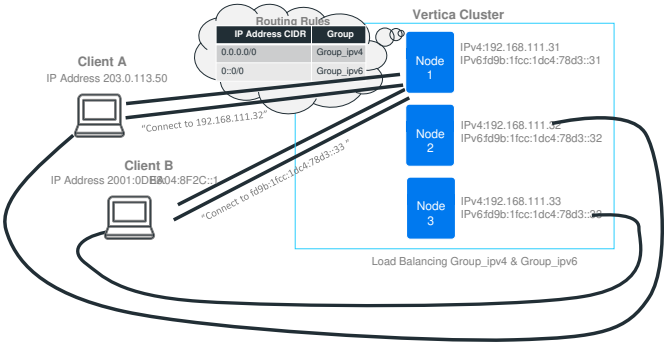


- Designate sets of nodes to service client connections from an IP address range. For example, if your ETL systems have a set range of IP addresses, you could limit their client connections to an arbitrary set of Vertica nodes, a subcluster, or a fault group. This technique lets you isolate the overhead of servicing client connections to a few nodes. It is useful when you are using subclusters in an Eon Mode database to isolate workloads (see Subclusters for more information).



Using connection load balancing policies with IPv4 and IPv6

Connection load balancing policies work with both IPv4 and IPv6. As far as the load balancing policies are concerned, the two address families represent separate networks. If you want your load balancing policy to handle both IPv4 and IPv6 addresses, you must create separate sets of network addresses, load balancing groups, and rules for each protocol. When a client opens a connection to a node in the cluster, the addressing protocol it uses determines which set of rules Vertica consults when deciding whether and how to balance the connection.



Default subcluster interior load balancing policy

Databases running in Eon Mode have a default connection load balancing policy that helps spread the load of handling client connections among the nodes in a subcluster. When a client connects to a node while opting into connection load balancing, the node checks for load balancing policies that apply to the client's IP address. If it does not find any applicable load balancing rule, and classic load balancing is not enabled, the node falls back to the default interior load balancing rule. This rule distributes connections among the nodes in the same subcluster as the initially-contacted node.

As with other connection load balancing policies, the nodes in the subcluster must have a network address defined for them to be eligible to handle the client connection. If no nodes in the subcluster have a network address, the node does not apply the default subcluster interior load balancing rule, and the connection is not load balanced.

This default rule is convenient when you are primarily interested in load balancing connections within each subcluster. You just create network addresses for the nodes in your subcluster. You do not need to create load balancing groups or rules. Clients that opt-in to load balancing are then automatically balanced among the nodes in the subcluster.

Interior load balancing policy with multiple network addresses

If your nodes have multiple network addresses, the default subcluster interior load balancing rule chooses the address that was created first as the target of load balancing rule. For example, suppose you create a network address on a node for the private IP address 192.168.1.10. Then you create another network address for the node for the public IP address 233.252.0.1. The default subcluster interior connection load balancing rule always selects 192.168.1.10 as the target of the rule.

If you want the default interior load balancing rule to choose a different network address as its target, drop the other network addresses on the node and then recreate them. Deleting and recreating other addresses makes the address you want the rule to select the oldest address. For example, suppose you want the rule to use a public address (233.252.0.1) that was created after a private address (192.168.1.10). In this case, you can drop the address for 192.168.1.10 and then recreate it. The rule then defaults to the older public 233.252.0.1 address.

If you intend to create multiple network addresses for the nodes in your subcluster, create the network addresses you want to use with the default subcluster interior load balancing first. For example, suppose you want to use the default interior load balancing subcluster rule to load balance most client connections. However, you also want to create a connection load balancing policy to manage connections coming in from a group of ETL systems. In this case, create the network addresses you want to use for the default interior load balancing rule first, then create the network addresses for the ETL systems.

Load balancing policies vs. classic load balancing

There are several differences between the classic load balancing feature and the load balancing policy feature:

- In classic connection load balancing, you just enable the load balancing option on both client and server, and load balancing is enabled. There are more steps to implement load balancing policies: you have to create addresses, groups, and rules and then enable load balancing on the client.
- Classic connection load balancing only supports a single, cluster-wide policy for redirecting connections. With connection load balancing policies, you get to choose which nodes handle client connections based on the connection's origin. This gives you more flexibility to handle complex situations. Examples include routing connections through a NAT-based router or having nodes that are accessible via multiple IP addresses on different networks.
- In classic connection load balancing, each node in the cluster can only be reached via a single IP address. This address is set in the EXPORT_ADDRESS column of the NODES system table. With connection load balancing policies, you can create a network address for each IP address associated with a node. Then you create rules that redirect to those addresses.

Steps to create a load balancing policy

There are three steps you must follow to create a load balancing policy:

1. Create one or more network addresses for each node that you want to participate in the connection load balancing policies.
2. Create one or more load balancing groups to be the target of the routing rules. Load balancing groups can target a collection of specific network addresses. Alternatively, you can create a group from a fault group or subcluster. You can limit the members of the load balance group to a subset of the fault group or subcluster using an IP address filter.
3. Create one or more routing rules.

While not absolutely necessary, it is always a good idea to idea to test your load balancing policy to ensure it works the way you expect it to.

After following these steps, Vertica will apply the load balancing policies to client connections that opt into connection load balancing. See Load balancing in ADO.NET , Load balancing in JDBC , and Load balancing , for information on enabling load balancing on the client. For vsql, use the -C command-line option to enable load balancing.

These steps are explained in the other topics in this section.

See also

In this section

- Creating network addresses
- Creating connection load balance groups
- Creating load balancing routing rules
- Testing connection load balancing policies
- Load balancing policy examples
- Viewing load balancing policy configurations
- Maintaining load balancing policies

Creating network addresses

Network addresses assign a name to an IP address and port number on a node. You use these addresses when you define load balancing groups. A node can have multiple network addresses associated with it. For example, suppose a node has one IP address that is only accessible from outside of the local network, and another that is accessible only from inside the network. In this case, you can define one network address using the external IP address, and another using the internal address. You can then create two different load balancing policies, one for external clients, and another for internal clients.

> **Note**
> You must create network addresses for your nodes, even if you intend to base your connection load balance groups on fault groups or subclusters. Load balancing rules can only select nodes that have a network address defined for them.

You create a network address using the CREATE NETWORK ADDRESS statement. This statement takes:

- The name to assign to the network address
- The name of the node
- The IP address of the node to associate with the network address
- The port number the node uses to accept client connections (optional)

> **Note**
> You can use hostnames instead of IP addresses when creating network addresses. However, doing so may lead to confusion if you are not sure which IP address a hostname resolves to. Using hostnames can also cause problems if your DNS server maps the hostname to multiple IP addresses.

The following example demonstrates creating three network addresses, one for each node in a three-node database.

```
=> SELECT node_name,node_address,node_address_family FROM v_catalog.nodes;
   node_name     | node_address | node_address_family
-----------------+--------------+---------------------
 v_vmart_node0001 | 10.20.110.21 | ipv4
 v_vmart_node0002 | 10.20.110.22 | ipv4
 v_vmart_node0003 | 10.20.110.23 | ipv4
(4 rows)


=> CREATE NETWORK ADDRESS node01 ON v_vmart_node0001 WITH '10.20.110.21';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node02 ON v_vmart_node0002 WITH '10.20.110.22';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node03 on v_vmart_node0003 WITH '10.20.110.23';
CREATE NETWORK ADDRESS
```

Creating network addresses for IPv6 addresses works the same way:

```
=> CREATE NETWORK ADDRESS node1_ipv6 ON v_vmart_node0001 WITH '2001:0DB8:7D5F:7433::';
CREATE NETWORK ADDRESS
```

Vertica does not perform any tests on the IP address you supply in the CREATE NETWORK ADDRESS statement. You must test the IP addresses you supply to this statement to confirm they correspond to the right node.

Vertica does not restrict the address you supply because it is often not aware of all the network addresses through which the node is accessible. For example, your node may be accessible from an external network via an IP address that Vertica is not configured to use. Or, your node can have both an IPv4 and an IPv6 address, only one of which Vertica is aware of.

For example, suppose v_vmart_node0003 from the previous example is **not** accessible via the IP address 192.168.1.5. You can still create a network address for it using that address:

```
=> CREATE NETWORK ADDRESS node04 ON v_vmart_node0003 WITH '192.168.1.5';
CREATE NETWORK ADDRESS
```

If you create a network group and routing rule that targets this address, client connections would either connect to the wrong node, or fail due to being connected to a host that's not part of a Vertica cluster.

Specifying a port number in a network address

By default, the CREATE NETWORK ADDRESS statement assumes the port number for the node's client connection is the default 5433. Sometimes, you may have a node listening for client connections on a different port. You can supply an alternate port number for the network address using the PORT keyword.

For example, suppose your nodes are behind a NAT router. In this case, you can have your nodes listen on different port numbers so the NAT router can route connections to them. When creating network addresses for these nodes, you supply the IP address of the NAT router and the port number the node is listening on. For example:

```
=> CREATE NETWORK ADDRESS node1_nat ON v_vmart_node0001 WITH '192.168.10.10' PORT 5433;
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node2_nat ON v_vmart_node0002 with '192.168.10.10' PORT 5434;
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node3_nat ON v_vmart_node0003 with '192.168.10.10' PORT 5435;
CREATE NETWORK ADDRESS
```

Creating connection load balance groups

After you have created network addresses for nodes, you create collections of them so you can target them with routing rules. These collections of network addresses are called load balancing groups. You have two ways to select the addresses to include in a load balancing group:

- A list of network addresses
- The name of one or more fault groups or subclusters , plus an IP address range in CIDR format. The address range selects which network addresses in the fault groups or subclusters Vertica adds to the load balancing group. Only the network addresses that are within the IP address range you supply are added to the load balance group. This filter lets you base your load balance group on a portion of the nodes that make up the fault group or subcluster.

> **Note**
> Load balance groups can only be based on fault groups or subclusters, or contain an arbitrary list of network addresses. You cannot mix these sources. For example, if you create a load balance group based on one or more fault groups, then you can only add additional fault groups to it. Vertica will return an error if you try to add a network address or subcluster to the load balance group.

You create a load balancing group using the CREATE LOAD BALANCE GROUP statement. When basing your group on a list of addresses, this statement takes the name for the group and the list of addresses. The following example demonstrates creating addresses for four nodes, and then creating two groups based on those nodes.

```
=> CREATE NETWORK ADDRESS addr01 ON v_vmart_node0001 WITH '10.20.110.21';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS addr02 ON v_vmart_node0002 WITH '10.20.110.22';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS addr03 on v_vmart_node0003 WITH '10.20.110.23';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS addr04 on v_vmart_node0004 WITH '10.20.110.24';
CREATE NETWORK ADDRESS
=> CREATE LOAD BALANCE GROUP group_1 WITH ADDRESS addr01, addr02;
CREATE LOAD BALANCE GROUP
=> CREATE LOAD BALANCE GROUP group_2 WITH ADDRESS addr03, addr04;
CREATE LOAD BALANCE GROUP

=> SELECT * FROM LOAD_BALANCE_GROUPS;
   name   |  policy  |    filter     |        type         | object_name
----------+----------+---------------+---------------------+-------------
 group_1  | ROUNDROBIN |             | Network Address Group | addr01
 group_1  | ROUNDROBIN |             | Network Address Group | addr02
 group_2  | ROUNDROBIN |             | Network Address Group | addr03
 group_2  | ROUNDROBIN |             | Network Address Group | addr04
(4 rows)
```

A network address can be a part of as many load balancing groups as you like. However, each group can only have a single network address per node. You cannot add two network addresses belonging to the same node to the same load balancing group.

Creating load balancing groups from fault groups

To create a load balancing group from one or more fault groups, you supply:

- The name for the load balancing group
- The name of one or more fault groups
- An IP address filter in CIDR format that filters the fault groups to be added to the load balancing group basd on their IP addresses. Vertica excludes any network addresses in the fault group that do not fall within this range. If you want all of the nodes in the fault groups to be added to the load balance group, specify the filter 0.0.0.0/0.

This example creates two load balancing groups from a fault group. The first includes all network addresses in the group by using the CIDR notation for all IP addresses. The second limits the fault group to three of the four nodes in the fault group by using the IP address filter.

```
=> CREATE FAULT GROUP fault_1;
CREATE FAULT GROUP
=> ALTER FAULT GROUP fault_1 ADD NODE  v_vmart_node0001;
ALTER FAULT GROUP
=> ALTER FAULT GROUP fault_1 ADD NODE  v_vmart_node0002;
ALTER FAULT GROUP
=> ALTER FAULT GROUP fault_1 ADD NODE  v_vmart_node0003;
ALTER FAULT GROUP
=> ALTER FAULT GROUP fault_1 ADD NODE  v_vmart_node0004;
ALTER FAULT GROUP
=> SELECT node_name,node_address,node_address_family,export_address
   FROM v_catalog.nodes;
    node_name     | node_address | node_address_family | export_address
------------------+--------------+---------------------+----------------
 v_vmart_node0001 | 10.20.110.21 | ipv4                | 10.20.110.21
 v_vmart_node0002 | 10.20.110.22 | ipv4                | 10.20.110.22
 v_vmart_node0003 | 10.20.110.23 | ipv4                | 10.20.110.23
 v_vmart_node0004 | 10.20.110.24 | ipv4                | 10.20.110.24
(4 rows)

=> CREATE LOAD BALANCE GROUP group_all WITH FAULT GROUP fault_1 FILTER
   '0.0.0.0/0';
CREATE LOAD BALANCE GROUP

=> CREATE LOAD BALANCE GROUP group_some WITH FAULT GROUP fault_1 FILTER
   '10.20.110.21/30';
CREATE LOAD BALANCE GROUP

=> SELECT * FROM LOAD_BALANCE_GROUPS;
      name      |   policy   |     filter      |         type          | object_name
----------------+------------+-----------------+-----------------------+-------------
 group_all      | ROUNDROBIN | 0.0.0.0/0       | Fault Group           | fault_1
 group_some     | ROUNDROBIN | 10.20.110.21/30 | Fault Group           | fault_1
(2 rows)
```

You can also supply multiple fault groups to the CREATE LOAD BALANCE GROUP statement:

```
=> CREATE LOAD BALANCE GROUP group_2_faults WITH FAULT GROUP
   fault_2, fault_3 FILTER '0.0.0.0/0';
CREATE LOAD BALANCE GROUP
```

> **Note**
> If you supply a filter range that does not match any network addresses of the nodes in the fault groups, Vertica creates an empty load balancing group. Any routing rules that direct connections to the empty load balance group will fail, because no nodes are set to handle connections for the group. In this case, the node that the client connected to initially handles the client connection itself.

Creating load balance groups from subclusters

Creating a load balance group from a subcluster is similar to creating a load balance group from a fault group. You just use WITH SUBCLUSTER instead of WITH FAULT GROUP in the CREATE LOAD BALANCE GROUP statement.

```
=> SELECT node_name,node_address,node_address_family,subcluster_name
   FROM v_catalog.nodes;
     node_name       | node_address | node_address_family |  subcluster_name
---------------------+--------------+---------------------+--------------------
 v_verticadb_node0001 | 10.11.12.10  | ipv4                | load_subcluster
 v_verticadb_node0002 | 10.11.12.20  | ipv4                | load_subcluster
 v_verticadb_node0003 | 10.11.12.30  | ipv4                | load_subcluster
 v_verticadb_node0004 | 10.11.12.40  | ipv4                | analytics_subcluster
 v_verticadb_node0005 | 10.11.12.50  | ipv4                | analytics_subcluster
 v_verticadb_node0006 | 10.11.12.60  | ipv4                | analytics_subcluster
(6 rows)

=> CREATE NETWORK ADDRESS node01 ON v_verticadb_node0001 WITH '10.11.12.10';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node02 ON v_verticadb_node0002 WITH '10.11.12.20';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node03 ON v_verticadb_node0003 WITH '10.11.12.30';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node04 ON v_verticadb_node0004 WITH '10.11.12.40';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node05 ON v_verticadb_node0005 WITH '10.11.12.50';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node06 ON v_verticadb_node0006 WITH '10.11.12.60';
CREATE NETWORK ADDRESS

=> CREATE LOAD BALANCE GROUP load_subcluster WITH SUBCLUSTER load_subcluster
   FILTER '0.0.0.0/0';
CREATE LOAD BALANCE GROUP
=> CREATE LOAD BALANCE GROUP analytics_subcluster WITH SUBCLUSTER
   analytics_subcluster FILTER '0.0.0.0/0';
CREATE LOAD BALANCE GROUP
```

Setting the group's distribution policy

A load balancing group has a policy setting that determines how the initially-contacted node chooses a target from the group. CREATE LOAD BALANCE GROUP supports three policies:

- ROUNDROBIN (default) rotates among the available members of the load balancing group. The initially-contacted node keeps track of which node it chose last time, and chooses the next one in the cluster.

  > **Note**
  > Each node in the cluster maintains its own round-robin pointer that indicates which node it should pick next for each load-balancing group. Therefore, if clients connect to different initial nodes, they may be redirected to the same node.

- RANDOM chooses an available node from the group randomly.
- NONE disables load balancing.

The following example demonstrates creating a load balancing group with a RANDOM distribution policy.

```
=> CREATE LOAD BALANCE GROUP group_random WITH ADDRESS node01, node02,
   node03, node04 POLICY 'RANDOM';
CREATE LOAD BALANCE GROUP
```

The next step

After creating the load balancing group, you must add a load balancing routing rule that tells Vertica how incoming connections should be redirected to the groups. See Creating load balancing routing rules .

Creating load balancing routing rules

Once you have created one or more connection load balancing groups, you are ready to create load balancing routing rules. These rules tell Vertica how to redirect client connections based on their IP addresses.

You create routing rules using the CREATE ROUTING RULE statement. You pass this statement:

- The name for the rule
- The source IP address range (either IPv4 or IPv6) in CIDR format the rule applies to
- The name of the load balancing group to handle the connection

The following example creates two rules. The first redirects connections coming from the IP address range 192.168.1.0 through 192.168.1.255 to a load balancing group named group_1. The second routes connections from the IP range 10.20.1.0 through 10.20.1.255 to the load balancing group named group_2.

```
=> CREATE ROUTING RULE internal_clients ROUTE '192.168.1.0/24' TO group_1;
CREATE ROUTING RULE

=> CREATE ROUTING RULE external_clients ROUTE '10.20.1.0/24' TO group_2;
CREATE ROUTING RULE
```

Creating a catch-all routing rule

Vertica applies routing rules in most specific to least specific order. This behavior lets you create a "catch-all" rule that handles all incoming connections. Then you can create rules to handle smaller IP address ranges for specific purposes. For example, suppose you wanted to create a catch-all rule that worked with the rules created in the previous example. Then you can create a new rule that routes 0.0.0.0/0 (the CIDR notation for all IP addresses) to a group that should handle connections that aren't handled by either of the previously-created rules. For example:

```
=> CREATE LOAD BALANCE GROUP group_all WITH ADDRESS node01, node02, node03, node04;
CREATE LOAD BALANCE GROUP

=> CREATE ROUTING RULE catch_all ROUTE '0.0.0.0/0' TO group_all;
CREATE ROUTING RULE
```

After running the above statements, any connection that does not originate from the IP address ranges 192.168.1.* or 10.20.1.* are routed to the group_all group.

Testing connection load balancing policies

After creating your routing rules, you should test them to verify that they perform the way you expect. The best way to test your rules is to call the DESCRIBE_LOAD_BALANCE_DECISION function with an IP address. This function evaluates the routing rules and reports back how Vertica would route a client connection from the IP address. It uses the same logic that Vertica uses when handling client connections, so the results reflect the actual connection load balancing result you will see from client connections. It also reflects the current state of the your Vertica cluster, so it will not redirect connections to down nodes.

The following example demonstrates testing a set of rules. One rule handles all connections from the range 192.168.1.0 to 192.168.1.255, while the other handles all connections originating from the 192 subnet. The third call demonstrates what happens when no rules apply to the IP address you supply.

```
=> SELECT describe_load_balance_decision('192.168.1.25');
                        describe_load_balance_decision
--------------------------------------------------------------------------------
 Describing load balance decision for address [192.168.1.25]
Load balance cache internal version id (node-local): [2]
Considered rule [etl_rule] source ip filter [10.20.100.0/24]... input address
does not match source ip filter for this rule.
Considered rule [internal_clients] source ip filter [192.168.1.0/24]... input
address matches this rule
Matched to load balance group [group_1] the group has policy [ROUNDROBIN]
number of addresses [2]
(0) LB Address: [10.20.100.247]:5433
(1) LB Address: [10.20.100.248]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [10.20.100.248] port [5433]

(1 row)

=> SELECT describe_load_balance_decision('192.168.2.25');
                        describe_load_balance_decision
--------------------------------------------------------------------------------
 Describing load balance decision for address [192.168.2.25]
Load balance cache internal version id (node-local): [2]
Considered rule [etl_rule] source ip filter [10.20.100.0/24]... input address
does not match source ip filter for this rule.
Considered rule [internal_clients] source ip filter [192.168.1.0/24]... input
address does not match source ip filter for this rule.
Considered rule [subnet_192] source ip filter [192.0.0.0/8]... input address
matches this rule
Matched to load balance group [group_all] the group has policy [ROUNDROBIN]
number of addresses [3]
(0) LB Address: [10.20.100.247]:5433
(1) LB Address: [10.20.100.248]:5433
(2) LB Address: [10.20.100.249]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [10.20.100.248] port [5433]

(1 row)

=> SELECT describe_load_balance_decision('1.2.3.4');
                         describe_load_balance_decision
--------------------------------------------------------------------------------
 Describing load balance decision for address [1.2.3.4]
Load balance cache internal version id (node-local): [2]
Considered rule [etl_rule] source ip filter [10.20.100.0/24]... input address
does not match source ip filter for this rule.
Considered rule [internal_clients] source ip filter [192.168.1.0/24]... input
address does not match source ip filter for this rule.
Considered rule [subnet_192] source ip filter [192.0.0.0/8]... input address
does not match source ip filter for this rule.
Routing table decision: No matching routing rules: input address does not match
any routing rule source filters. Details: [Tried some rules but no matching]
No rules matched. Falling back to classic load balancing.
Classic load balance decision: Classic load balancing considered, but either
the policy was NONE or no target was available. Details: [NONE or invalid]

(1 row)
```

The DESCRIBE_LOAD_BALANCE_DECISION function also takes into account the classic cluster-wide load balancing settings:

```
=>  SELECT SET_LOAD_BALANCE_POLICY('ROUNDROBIN');
                    SET_LOAD_BALANCE_POLICY
-------------------------------------------------------------------------------
 Successfully changed the client initiator load balancing policy to: roundrobin
(1 row)

=> SELECT DESCRIBE_LOAD_BALANCE_DECISION('1.2.3.4');
                    describe_load_balance_decision
-------------------------------------------------------------------------------
 Describing load balance decision for address [1.2.3.4]
Load balance cache internal version id (node-local): [2]
Considered rule [etl_rule] source ip filter [10.20.100.0/24]... input address
does not match source ip filter for this rule.
Considered rule [internal_clients] source ip filter [192.168.1.0/24]... input
address does not match source ip filter for this rule.
Considered rule [subnet_192] source ip filter [192.0.0.0/8]... input address
does not match source ip filter for this rule.
Routing table decision: No matching routing rules: input address does not
match any routing rule source filters. Details: [Tried some rules but no matching]
No rules matched. Falling back to classic load balancing.
Classic load balance decision: Success. Load balance redirect to: [10.20.100.247]
port [5433]

(1 row)
```

Note

The DESCRIBE_LOAD_BALANCE_DECISION function assumes the client connection has opted to be load balanced. In reality, clients may not enable load balancing. This setting prevents the load-balancing features from redirecting the connection.

The function can also help you debug connection issues you notice after going live with your load balancing policy. For example, if you notice that one node is handling a large number of client connections, you can test the client IP addresses against your policies to see why the connections are not being balanced.

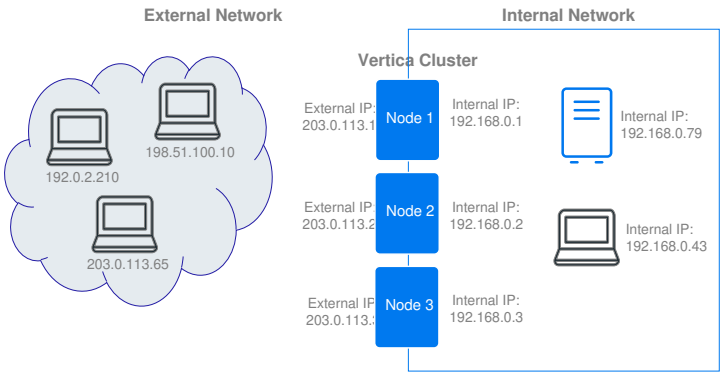
Load balancing policy examples

The following examples demonstrate some common use cases for connection load balancing policies.

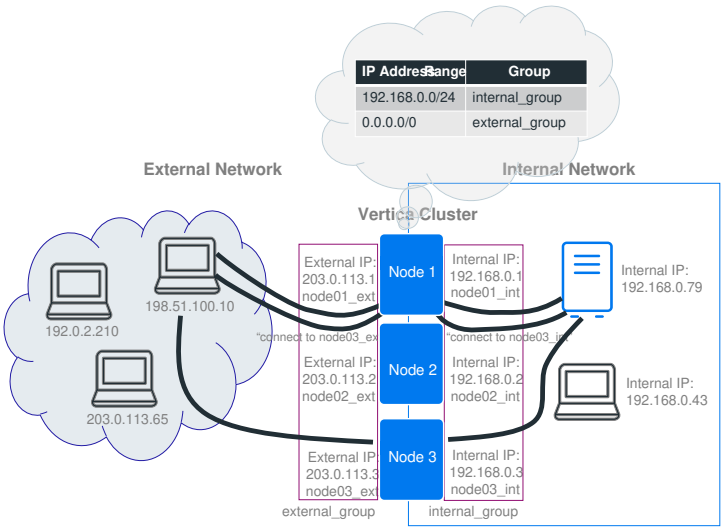
Enabling client connections from multiple networks

Suppose you have a Vertica cluster that is accessible from two (or more) different networks. Some examples of this situation are:

- You have an internal and an external network. In this configuration, your database nodes usually have two or more IP addresses, which each address only accessible from one of the networks. This configuration is common when running Vertica in a cloud environment. In many cases, you can create a catch-all rule that applies to all IP addresses, and then add additional routing rules for the internal subnets.
- You want clients to be load balanced whether they use IPv4 or IPv6 protocols. From the database's perspective, IPv4 and IPv6 connections are separate networks because each node has a separate IPv4 and IPv6 IP address.



When creating a load balancing policy for a database that is accessible from multiple networks, client connections must be directed to IP addresses on the network they can access. The best solution is to create load balancing groups for each set of IP addresses assigned to a node. Then create routing rules that redirect client connections to the IP addresses that are accessible from their network.



The following example:

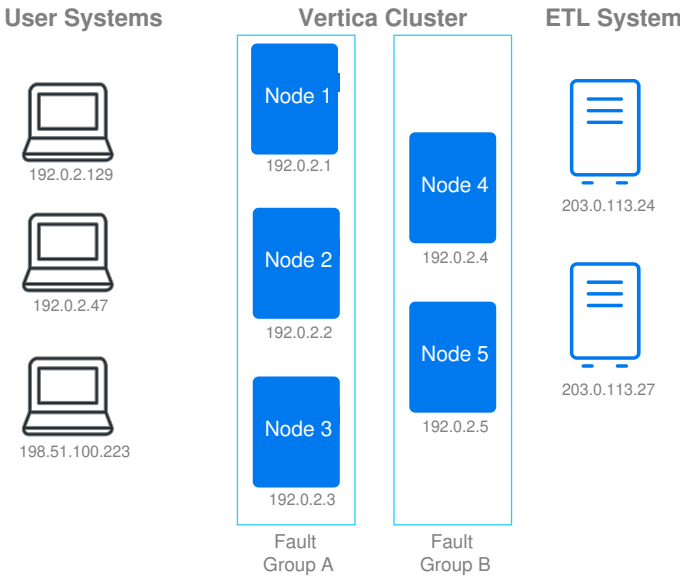
1. Creates two sets of network addresses: one for the internal network and another for the external network.
2. Creates two load balance groups: one for the internal network and one for the external.
3. Creates three routing rules: one for the internal network, and two for the external. The internal routing rule covers a subset of the network covered by one of the external rules.
4. Tests the routing rules using internal and external IP addresses.

```
=> CREATE NETWORK ADDRESS node01_int ON v_vmart_node0001 WITH '192.168.0.1';
CREATE NETWORK ADDRESS

=> CREATE NETWORK ADDRESS node01_ext ON v_vmart_node0001 WITH '203.0.113.1';
CREATE NETWORK ADDRESS

=> CREATE NETWORK ADDRESS node02_int ON v_vmart_node0002 WITH '192.168.0.2';
CREATE NETWORK ADDRESS

=> CREATE NETWORK ADDRESS node02_ext ON v_vmart_node0002 WITH '203.0.113.2';
CREATE NETWORK ADDRESS

=> CREATE NETWORK ADDRESS node03_int ON v_vmart_node0003 WITH '192.168.0.3';
CREATE NETWORK ADDRESS

=> CREATE NETWORK ADDRESS node03_ext ON v_vmart_node0003 WITH '203.0.113.3';
CREATE NETWORK ADDRESS

=> CREATE LOAD BALANCE GROUP internal_group WITH ADDRESS node01_int, node02_int, node03_int;
CREATE LOAD BALANCE GROUP

=> CREATE LOAD BALANCE GROUP external_group WITH ADDRESS node01_ext, node02_ext, node03_ext;
CREATE LOAD BALANCE GROUP

=> CREATE ROUTING RULE internal_rule ROUTE '192.168.0.0/24' TO internal_group;
CREATE ROUTING RULE

=> CREATE ROUTING RULE external_rule ROUTE '0.0.0.0/0' TO external_group;
CREATE ROUTING RULE

=> SELECT DESCRIBE_LOAD_BALANCE_DECISION('198.51.100.10');
                    DESCRIBE_LOAD_BALANCE_DECISION
-------------------------------------------------------------------------------
 Describing load balance decision for address [198.51.100.10]
```

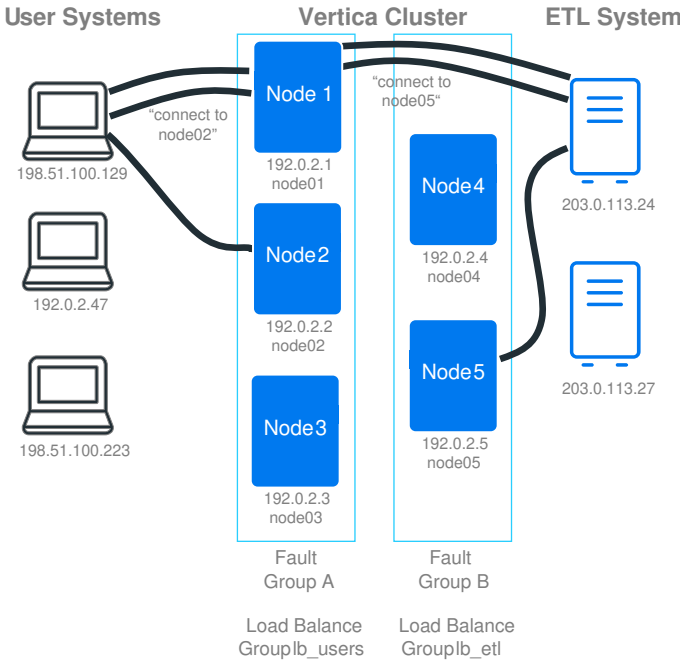
```
Describing load balance decision for address [198.51.100.10]
Load balance cache internal version id (node-local): [3]
Considered rule [internal_rule] source ip filter [192.168.0.0/24]... input
address does not match source ip filter for this rule.
Considered rule [external_rule] source ip filter [0.0.0.0/0]... input
address matches this rule
Matched to load balance group [external_group] the group has policy [ROUNDROBIN]
number of addresses [3]
(0) LB Address: [203.0.113.1]:5433
(1) LB Address: [203.0.113.2]:5433
(2) LB Address: [203.0.113.3]:5433
Chose address at position [2]
Routing table decision: Success. Load balance redirect to: [203.0.113.3] port [5433]

(1 row)

=> SELECT DESCRIBE_LOAD_BALANCE_DECISION('198.51.100.10');

                    DESCRIBE_LOAD_BALANCE_DECISION
-------------------------------------------------------------------------------
 Describing load balance decision for address [192.168.0.79]
Load balance cache internal version id (node-local): [3]
Considered rule [internal_rule] source ip filter [192.168.0.0/24]... input
address matches this rule
Matched to load balance group [internal_group] the group has policy [ROUNDROBIN]
number of addresses [3]
(0) LB Address: [192.168.0.1]:5433
(1) LB Address: [192.168.0.3]:5433
(2) LB Address: [192.168.0.2]:5433
Chose address at position [2]
Routing table decision: Success. Load balance redirect to: [192.168.0.2] port
[5433]

(1 row)
```

Isolating workloads

You may want to control which nodes in your cluster are used by specific types of clients. For example, you may want to limit clients that perform data-loading tasks to one set of nodes, and reserve the rest of the nodes for running queries. This separation of workloads is especially common for Eon Mode databases. See Controlling Where a Query Runs for an example of using load balancing policies in an Eon Mode database to control which subcluster a client connects to.



You can create client load balancing policies that support workload isolation if clients performing certain types of tasks always originate from a limited IP address range. For example, if the clients that load data into your system always fall into a specific subnet, you can create a policy that limits which nodes those clients can access.



In the following example:

- There are two fault groups (group_a and group_b) that separate workloads in an Eon Mode database. These groups are used as the basis of the load balancing groups.
- The ETL client connections all originate from the 203.0.113.0/24 subnet.
- User connections originate in the range of 192.0.0.0 to 199.255.255.255.

```
=> CREATE NETWORK ADDRESS node01 ON v_vmart_node0001 WITH '192.0.2.1';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node02 ON v_vmart_node0002 WITH '192.0.2.2';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node03 ON v_vmart_node0003 WITH '192.0.2.3';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node04 ON v_vmart_node0004 WITH '192.0.2.4';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node05 ON v_vmart_node0005 WITH '192.0.2.5';
CREATE NETWORK ADDRESS
                                    ^
=> CREATE LOAD BALANCE GROUP lb_users WITH FAULT GROUP group_a FILTER '192.0.2.0/24';
CREATE LOAD BALANCE GROUP
=> CREATE LOAD BALANCE GROUP lb_etl WITH FAULT GROUP group_b FILTER '192.0.2.0/24';
CREATE LOAD BALANCE GROUP
=> CREATE ROUTING RULE users_rule ROUTE '192.0.0.0/5' TO lb_users;
CREATE ROUTING RULE
=> CREATE ROUTING RULE etl_rule ROUTE '203.0.113.0/24' TO lb_etl;
CREATE ROUTING RULE

=> SELECT DESCRIBE_LOAD_BALANCE_DECISION('198.51.200.129');
                    DESCRIBE_LOAD_BALANCE_DECISION
--------------------------------------------------------------------------------
 Describing load balance decision for address [198.51.200.129]
Load balance cache internal version id (node-local): [6]
Considered rule [etl_rule] source ip filter [203.0.113.0/24]... input address
does not match source ip filter for this rule.
Considered rule [users_rule] source ip filter [192.0.0.0/5]... input address
matches this rule
Matched to load balance group [lb_users] the group has policy [ROUNDROBIN]
```

```
number of addresses [3]
(0) LB Address: [192.0.2.1]:5433
(1) LB Address: [192.0.2.2]:5433
(2) LB Address: [192.0.2.3]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [192.0.2.2] port
[5433]

(1 row)

=> SELECT DESCRIBE_LOAD_BALANCE_DECISION('203.0.113.24');
                        DESCRIBE_LOAD_BALANCE_DECISION
--------------------------------------------------------------------------------
 Describing load balance decision for address [203.0.113.24]
Load balance cache internal version id (node-local): [6]
Considered rule [etl_rule] source ip filter [203.0.113.0/24]... input address
matches this rule
Matched to load balance group [lb_etl] the group has policy [ROUNDROBIN] number
of addresses [2]
(0) LB Address: [192.0.2.4]:5433
(1) LB Address: [192.0.2.5]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [192.0.2.5] port
[5433]

(1 row)

=> SELECT DESCRIBE_LOAD_BALANCE_DECISION('10.20.100.25');
                      DESCRIBE_LOAD_BALANCE_DECISION
--------------------------------------------------------------------------------
 Describing load balance decision for address [10.20.100.25]
Load balance cache internal version id (node-local): [6]
Considered rule [etl_rule] source ip filter [203.0.113.0/24]... input address
does not match source ip filter for this rule.
Considered rule [users_rule] source ip filter [192.0.0.0/5]... input address
does not match source ip filter for this rule.
Routing table decision: No matching routing rules: input address does not match
any routing rule source filters. Details: [Tried some rules but no matching]
No rules matched. Falling back to classic load balancing.
Classic load balance decision: Classic load balancing considered, but either the
policy was NONE or no target was available. Details: [NONE or invalid]

(1 row)
```

Enabling the default subcluster interior load balancing policy

Vertica attempts to apply the default subcluster interior load balancing policy if no other load balancing policy applies to an incoming connection and classic load balancing is not enabled. See Default Subcluster Interior Load Balancing Policy for a description of this rule.

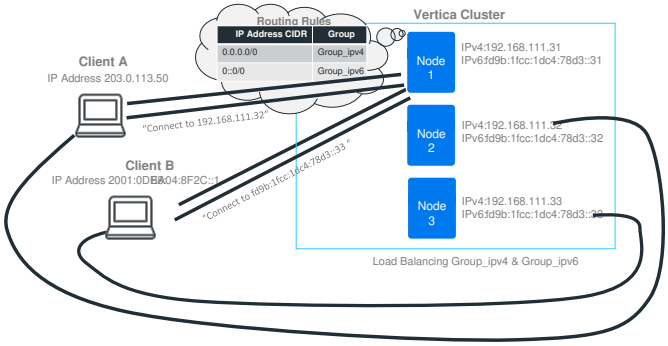
To enable default subcluster interior load balancing, you must create network addresses for the nodes in a subcluster. Once you create the addresses, Vertica attempts to apply this rule to load balance connections within a subcluster when no other rules apply.

The following example confirms the database has no load balancing groups or rules. Then it adds publicly-accessible network addresses to the nodes in the primary subcluster. When these addresses are added, Vertica applies the default subcluster interior load balancing policy.

```
=> SELECT * FROM LOAD_BALANCE_GROUPS;
 name | policy | filter | type | object_name
------+--------+--------+------+-------------
(0 rows)

=> SELECT * FROM ROUTING_RULES;
 name | source_address | destination_name
------+----------------+------------------
(0 rows)

=> CREATE NETWORK ADDRESS node01_ext ON v_verticadb_node0001 WITH '203.0.113.1';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node02_ext ON v_verticadb_node0002 WITH '203.0.113.2';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node03_ext ON v_verticadb_node0003 WITH '203.0.113.3';
CREATE NETWORK ADDRESS

=> SELECT describe_load_balance_decision('11.0.0.100');
                            describe_load_balance_decision
-----------------------------------------------------------------------------------------------
Describing load balance decision for address [11.0.0.100] on subcluster [default_subcluster]
Load balance cache internal version id (node-local): [2]
Considered rule [auto_rr_default_subcluster] subcluster interior filter  [default_subcluster]...
current subcluster matches this rule
Matched to load balance group [auto_lbg_sc_default_subcluster] the group has policy
[ROUNDROBIN] number of addresses [3]
(0) LB Address: [203.0.113.1]:5433
(1) LB Address: [203.0.113.2]:5433
(2) LB Address: [203.0.113.3]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [203.0.113.2] port [5433]

(1 row)
```

Load balance both IPv4 and IPv6 connections

Connection load balancing policies regard IPv4 and IPv6 connections as separate networks. To load balance both types of incoming client connections, create two sets of network addresses, at least two load balancing groups, and two load balancing , once for each network address family.



This example creates two load balancing policies for the default subcluster: one for the IPv4 network addresses (192.168.111.31 to 192.168.111.33) and one for the IPv6 network addresses (fd9b:1fcc:1dc4:78d3::31 to fd9b:1fcc:1dc4:78d3::33).

```
=> SELECT node_name,node_address,subcluster_name FROM NODES;
     node_name      |  node_address  |  subcluster_name
--------------------+----------------+--------------------
 v_verticadb_node0001 | 192.168.111.31 | default_subcluster
 v_verticadb_node0002 | 192.168.111.32 | default_subcluster
 v_verticadb_node0003 | 192.168.111.33 | default_subcluster

=> CREATE NETWORK ADDRESS node01 ON v_verticadb_node0001 WITH
   '192.168.111.31';
CREATE NETWORK ADDRESS
```

```
=> CREATE NETWORK ADDRESS node01_ipv6 ON v_verticadb_node0001 WITH
   'fd9b:1fcc:1dc4:78d3::31';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node02 ON v_verticadb_node0002 WITH
   '192.168.111.32';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node02_ipv6 ON v_verticadb_node0002 WITH
   'fd9b:1fcc:1dc4:78d3::32';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node03 ON v_verticadb_node0003 WITH
   '192.168.111.33';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node03_ipv6 ON v_verticadb_node0003 WITH
   'fd9b:1fcc:1dc4:78d3::33';
CREATE NETWORK ADDRESS

=> CREATE LOAD BALANCE GROUP group_ipv4 WITH SUBCLUSTER default_subcluster
   FILTER '192.168.111.0/24';
CREATE LOAD BALANCE GROUP
=> CREATE LOAD BALANCE GROUP group_ipv6 WITH SUBCLUSTER default_subcluster
   FILTER 'fd9b:1fcc:1dc4:78d3::0/64';
CREATE LOAD BALANCE GROUP

=> CREATE ROUTING RULE all_ipv4 route '0.0.0.0/0' TO group_ipv4;
CREATE ROUTING RULE
=> CREATE ROUTING RULE all_ipv6 route '0::0/0' TO group_ipv6;
CREATE ROUTING RULE

=> SELECT describe_load_balance_decision('203.0.113.50');

describe_load_balance_decision
--------------------------------------------------------------------------------------------
Describing load balance decision for address [203.0.113.50] on subcluster [default_subcluster]
Load balance cache internal version id (node-local): [3]
Considered rule [all_ipv4] source ip filter [0.0.0.0/0]... input address matches this rule
Matched to load balance group [ group_ipv4] the group has policy [ROUNDROBIN] number of addresses [3]
(0) LB Address: [192.168.111.31]:5433
(1) LB Address: [192.168.111.32]:5433
(2) LB Address: [192.168.111.33]:5433
Chose address at position [2]
Routing table decision: Success. Load balance redirect to: [192.168.111.33] port [5433]

(1 row)

=> SELECT describe_load_balance_decision('2001:0DB8:EA04:8F2C::1');

describe_load_balance_decision
-----------------------------------------------------------------------------------------------------
Describing load balance decision for address [2001:0DB8:EA04:8F2C::1] on subcluster [default_subcluster]
Load balance cache internal version id (node-local): [3]
Considered rule [all_ipv4] source ip filter [0.0.0.0/0]... input address does not match source ip filter for this rule.
Considered rule [all_ipv6] source ip filter [0::0/0]... input address matches this rule
Matched to load balance group [ group_ipv6] the group has policy [ROUNDROBIN] number of addresses [3]
(0) LB Address: [fd9b:1fcc:1dc4:78d3::31]:5433
(1) LB Address: [fd9b:1fcc:1dc4:78d3::32]:5433
(2) LB Address: [fd9b:1fcc:1dc4:78d3::33]:5433
Chose address at position [2]
Routing table decision: Success. Load balance redirect to: [fd9b:1fcc:1dc4:78d3::33] port [5433]

(1 row)
```

Other examples

For other examples of using connection load balancing, see the following topics:

- Using connection load balancing with the Kafka scheduler
- Distributing Clients Among the Throughput Subclusters

Viewing load balancing policy configurations

Query the following system tables in the V_CATALOG schema to see the load balance policies defined in your database:

- NETWORK_ADDRESSES lists all of the network addresses defined in your database.
- LOAD_BALANCE_GROUPS lists the contents of your load balance groups.

> **Note**
> This table does not directly lists all of your database's load balance groups. Instead, it lists the contents of the load balance groups. It is possible for your database to have load balancing groups that are not in this table because they do not contain any network addresses or fault groups.

- ROUTING_RULES lists all of the routing rules defined in your database.

This example demonstrates querying each of the load balancing policy system tables.

```
=> \x
Expanded display is on.
=> SELECT * FROM V_CATALOG.NETWORK_ADDRESSES;
-[ RECORD 1 ]----+----------------
name             | node01
node             | v_vmart_node0001
address          | 10.20.100.247
port             | 5433
address_family   | ipv4
is_enabled       | t
is_auto_detected | f
-[ RECORD 2 ]----+----------------
name             | node02
node             | v_vmart_node0002
address          | 10.20.100.248
port             | 5433
address_family   | ipv4
is_enabled       | t
is_auto_detected | f
-[ RECORD 3 ]----+----------------
name             | node03
node             | v_vmart_node0003
address          | 10.20.100.249
port             | 5433
address_family   | ipv4
is_enabled       | t
is_auto_detected | f
-[ RECORD 4 ]----+----------------
name             | alt_node1
node             | v_vmart_node0001
address          | 192.168.1.200
port             | 8080
address_family   | ipv4
is_enabled       | t
is_auto_detected | f
-[ RECORD 5 ]----+----------------
name             | test_addr
node             | v_vmart_node0001
address          | 192.168.1.100
```

port | 4000
address_family | ipv4
is_enabled | t
is_auto_detected | f

=> SELECT * FROM LOAD_BALANCE_GROUPS;
-[RECORD 1]----------
name | group_all
policy | ROUNDROBIN
filter |
type | Network Address Group
object_name | node01
-[RECORD 2]----------
name | group_all
policy | ROUNDROBIN
filter |
type | Network Address Group
object_name | node02
-[RECORD 3]----------
name | group_all
policy | ROUNDROBIN
filter |
type | Network Address Group
object_name | node03
-[RECORD 4]----------
name | group_1
policy | ROUNDROBIN
filter |
type | Network Address Group
object_name | node01
-[RECORD 5]----------
name | group_1
policy | ROUNDROBIN
filter |
type | Network Address Group
object_name | node02
-[RECORD 6]----------
name | group_2
policy | ROUNDROBIN
filter |
type | Network Address Group
object_name | node01
-[RECORD 7]----------
name | group_2
policy | ROUNDROBIN
filter |
type | Network Address Group
object_name | node02
-[RECORD 8]----------
name | group_2
policy | ROUNDROBIN
filter |
type | Network Address Group
object_name | node03
-[RECORD 9]----------
name | etl_group
policy | ROUNDROBIN
filter |
type | Network Address Group
object_name | node01

```
=> SELECT * FROM ROUTING_RULES;
-[ RECORD 1 ]----+-----------------
name             | internal_clients
source_address   | 192.168.1.0/24
destination_name | group_1
-[ RECORD 2 ]----+-----------------
name             | etl_rule
source_address   | 10.20.100.0/24
destination_name | etl_group
-[ RECORD 3 ]----+-----------------
name             | subnet_192
source_address   | 192.0.0.0/8
destination_name | group_all
```

Maintaining load balancing policies

Once you have created load balancing policies, you maintain them using the following statements:

- ALTER NETWORK ADDRESS letsyou: rename, change the IP address, and enable or disable a network address.
- ALTER LOAD BALANCE GROUP letsyou rename, add or remove network addresses or fault groups, change the fault group IP address filter, or change the policy of a load balance group.
- ALTER ROUTING RULE letsyou rename, change the source IP address, and the target load balance group of a rule.

See the refence pages for these statements for examples.

Deleting load balancing policy objects

You can also delete existing load balance policy objects using the following statements:

- DROP NETWORK ADDRESS
- DROP LOAD BALANCE GROUP
- DROP ROUTING RULE

Projections

Unlike traditional databases that store data in tables, Vertica physically stores table data in projections , which are collections of table columns.

Projections store data in a format that optimizes query execution. Similar to materialized views, they store result sets on disk rather than compute them each time they are used in a query. Vertica automatically refreshes these result sets with updated or new data.

Projections provide the following benefits:

- Compress and encode data to reduce storage space. Vertica also operates on the encoded data representation whenever possible to avoid the cost of decoding. This combination of compression and encoding optimizes disk space while maximizing query performance.
- Facilitate distribution across the database cluster. Depending on their size, projections can be segmented or replicated across cluster nodes. For instance, projections for large tables can be segmented and distributed across all nodes. Unsegmented projections for small tables can be replicated across all nodes.
- Transparent to end-users. The Vertica query optimizer automatically picks the best projection to execute a given query.
- Provide high availability and recovery. Vertica duplicates table columns on at least K+1 nodes in the cluster. If one machine fails in a K-Safe environment, the database continues to operate using replicated data on the remaining nodes. When the node resumes normal operation, it automatically queries other nodes to recover data and lost objects. For more information, see High availability with fault groups and High availability with projections .

In this section

- Projection types
- Creating projections
- Projection naming
- Auto-projections
- Unsegmented projections
- Segmented projections
- K-safe database projections
- Partition range projections
- Refreshing projections
- Dropping projections

Projection types

A Vertica table typically has multiple projections, each defined to contain different content. Content for the projections of a given table can differ in scope and organization. These differences can generally be divided into the following projection types:

- Superprojections
- Query-specific projections
- Aggregate projections

Superprojections

For each table in the database, Vertica requires at least one superprojection that contains all columns in the table. In the absence of a query-specific projection, Vertica uses the table's superprojection, which can support any query and DML operation.

Under certain conditions, Vertica automatically creates a table's superprojection immediately on table creation. Vertica also creates a superprojection when you first load data into that table, if none already exists. CREATE PROJECTION can create a superprojection if it specifies to include all table columns. A table can have multiple superprojections.

While superprojections can support all queries on a table, they do not facilitate optimal execution of specific queries.

Query-specific projections

A query-specific projection is a projection that contains only the subset of table columns needed to process a given query. Query-specific projections significantly improve performance of queries for which they are optimized.

Aggregate projections

Queries that include expressions or aggregate functions, such as SUM and COUNT , can perform more efficiently when using projections that already contain the aggregated data. This is especially true for queries on large quantities of data.

Vertica provides several types of projections for storing data that is returned from aggregate functions or expressions:

- Live aggregate projection: Projection that contains columns with values that are aggregated from columns in its anchor table. You can also define live aggregate projections that include user-defined transform functions .
- Top-K projection : Type of live aggregate projection that returns the top *k* rows from a partition of selected rows. Create a Top-K projection that satisfies the criteria for a Top-K query.
- Projection that pre-aggregates UDTF results : Live aggregate projection that invokes user-defined transform functions (UDTFs). To minimize overhead when you query those projections of this type, Vertica processes the UDTF functions in the background and stores their results on disk.
- Projection that contains expressions : Projection with columns whose values are calculated from anchor table columns.

For more information, see Pre-aggregating data in projections .

Creating projections

Vertica supports two methods for creating projections: Database Designer and the CREATE PROJECTION statement.

Creating projections with Database Designer

Vertica recommends that you use Database Designer to design your physical schema, by running it on a representative sample of your data. Database Designer generates SQL for creating projections as follows:

1. Analyzes your logical schema , sample data, and sample queries (optional).
2. Designs a physical schema in the form of a SQL script that you can deploy automatically or manually.

For more information, see Creating a database design .

Manually creating projections

CREATE PROJECTION defines a projection, as in the following example:

```
=> CREATE PROJECTION retail_sales_fact_p (
   store_key ENCODING RLE,
     pos_transaction_number ENCODING RLE,
   sales_dollar_amount,
   cost_dollar_amount )
AS SELECT
   store_key,
   pos_transaction_number,
   sales_dollar_amount,
   cost_dollar_amount
FROM store.store_sales_fact
ORDER BY store_key
SEGMENTED BY HASH(pos_transaction_number) ALL NODES;
```

A projection definition includes the following components:

- Column List and Encoding
- Base Query
- Sort Order
- Segmentation

Column list and encoding

This portion of the SQL statement lists every column in the projection and defines the encoding for each column. Vertica supports encoded data, which helps query execution to incur less disk I/O.

```
CREATE PROJECTION retail_sales_fact_P (
   store_key ENCODING RLE,
   pos_transaction_number ENCODING RLE,
   sales_dollar_amount,
     cost_dollar_amount )
```

Base query

A projection's base query clause identifies which columns to include in the projection.

```

AS SELECT
   store_key,
   pos_transaction_number,
   sales_dollar_amount,
   cost_dollar_amount
```

Sort order

A projection's ORDER BY clause determines how to sort projection data. The sort order localizes logically grouped values so a disk read can identify many results at once. For maximum performance, do not sort projections on LONG VARBINARY and LONG VARCHAR columns. For more information see ORDER BY clause .

```
ORDER BY store_key
```

Segmentation

A projection's segmentation clause specifies how to distribute projection data across all nodes in the database. Even load distribution helps maximize access to projection data. For large tables, distribute projection data in segments with SEGMENTED BY HASH . For example:

```
SEGMENTED BY HASH(pos_transaction_number) ALL NODES;
```

For small tables, use the UNSEGMENTED keyword to replicate table data. Vertica creates identical copies of an unsegmented projection on all cluster nodes. Replication ensures high availability and recovery.

For maximum performance, do not segment projections on LONG VARBINARY and LONG VARCHAR columns.

For more design considerations, see Creating custom designs .

Projection naming

Vertica identifies projections according to the following conventions, where *proj-basename* is the name assigned to this projection by CREATE PROJECTION .

Unsegmented projections

Unsegmented projections conform to the following naming conventions:

proj-basename _super	The auto projection that Vertica creates when data is loaded for the first time into a new unsegmented table. Vertica uses the anchor table name to create the projection base name *proj-basename* and appends the string _super . The auto projection is always a superprojection.
proj-basename _unseg	An unsegmented projection, where *proj-basename* and the anchor table name are identical. If no other projection was previously created with this base name (including an auto projection), Vertica appends the string _unseg to the projection name. If the projection is copied on all nodes, this projection name maps to all instances.

Segmented projections

Enterprise Mode

In Enterprise Mode, segmented projections use the following naming convention:

```
proj-basename_boffset
```

This name identifies buddy projections for a segmented projection, where *offset* is the projection's node location relative to all other buddy projections. All buddy projections share the same project base name. For example:

```
=> SELECT projection_basename, projection_name FROM projections WHERE anchor_table_name = 'store_orders';
 projection_basename | projection_name
---------------------+-----------------
 store_orders        | store_orders_b0
 store_orders        | store_orders_b1
(2 rows)
```

One exception applies: Vertica uses the following convention to name live aggregate projections: *proj-basename* , *proj-basename* _b1 , and so on.

Eon Mode

In Eon Mode, segmented projections use the following naming convention:

```
proj-basename
```

> **Note**
> Eon Mode uses shards in communal storage to segment table data, which are functionally equivalent to Enterprise Mode buddy projections. For details, see Shards and subscriptions .

Projections of renamed and copied tables

Vertica uses the same logic to rename existing projections in two cases:

- You rename a table with ALTER TABLE...RENAME TABLE .
- You create a table from an existing one with CREATE TABLE LIKE...INCLUDING PROJECTIONS .

In both cases, Vertica uses the following algorithm to rename projections:

1. Iterate over all projections anchored on the renamed or new table, and check whether their names are prefixed by the original table name:
   - No: Retain projection name
   - Yes: Rename projection
2. If yes, compare the original table name and projection base name:
   - If the new base name is the same as the original table name, then replace the base name with the new table name in the table and projection names.
   - If the new base name is prefixed by the original table name, then replace the prefix with the new table name, remove any version strings that were appended to the old base name (such as *old-basename* _v1), and generate projection names with the new base name.
3. Check whether the new projection names already exist. If not, save them. Otherwise, resolve name conflicts by appending version numbers as needed to the new base name— *new-basename* _v1 , *new-basename* _v2 , and so on.

Examples

An auto projection is always a superprojection:

```
=> CREATE TABLE store.store_dimension
    store_key int NOT NULL,
    store_name varchar(64),
    ...
) UNSEGMENTED ALL NODES;
CREATE TABLE
=> COPY store.store_dim FROM '/home/dbadmin/store_dimension_data.txt';
50
=> SELECT anchor_table_name, projection_basename, projection_name FROM projections WHERE anchor_table_name = 'store_dimension';
 anchor_table_name | projection_basename |    projection_name
-------------------+---------------------+-----------------------
 store_dimension   | store_dimension     | store_dimension_super
 store_dimension   | store_dimension     | store_dimension_super
 store_dimension   | store_dimension     | store_dimension_super
(3 rows)
```

An unsegmented projection name has the _unseg suffix on all nodes:

```
=> CREATE TABLE store.store_dimension(
    store_key int NOT NULL,
    store_name varchar(64),
    ...
);
CREATE TABLE
=> CREATE PROJECTION store_dimension AS SELECT * FROM store.store_dimension UNSEGMENTED ALL NODES;
WARNING 6922:  Projection name was changed to store_dimension_unseg because it conflicts with the basename of the table store_dimension
CREATE PROJECTION
=> SELECT anchor_table_name, projection_basename, projection_name FROM projections WHERE anchor_table_name = 'store_dimension';
 anchor_table_name | projection_basename |    projection_name
-------------------+---------------------+-----------------------
 store_dimension   | store_dimension     | store_dimension_unseg
 store_dimension   | store_dimension     | store_dimension_unseg
 store_dimension   | store_dimension     | store_dimension_unseg
(3 rows)
```

The following example creates the segmented table testRenameSeg and populates it with data:

```
=> CREATE TABLE testRenameSeg (a int, b int);
CREATE TABLE
dbadmin=> INSERT INTO testRenameSeg VALUES (1,2);
 OUTPUT
--------
      1
(1 row)

dbadmin=> COMMIT;
COMMIT
```

Vertica automatically creates two buddy superprojections for this table:

```
=> \dj testRename*
                        List of projections
 Schema |         Name          |  Owner  |       Node       | Comment
--------+-----------------------+---------+------------------+---------
 public | testRenameSeg_b0      | dbadmin |                  |
 public | testRenameSeg_b1      | dbadmin |                  |
```

The following CREATE PROJECTION statements explicitly create additional projections for the table:

```
=> CREATE PROJECTION nameTestRenameSeg_p AS SELECT * FROM testRenameSeg;
=> CREATE PROJECTION testRenameSeg_p AS SELECT * FROM testRenameSeg;
=> CREATE PROJECTION testRenameSeg_pLap AS SELECT b, MAX(a) a FROM testRenameSeg GROUP BY b;
=> CREATE PROJECTION newTestRenameSeg AS SELECT * FROM testRenameSeg;
=> \dj *testRenameSeg*
                List of projections
 Schema |          Name          |  Owner  | Node | Comment
--------+------------------------+---------+------+---------
 public | nameTestRenameSeg_p_b0 | dbadmin |      |
 public | nameTestRenameSeg_p_b1 | dbadmin |      |
 public | newTestRenameSeg_b0    | dbadmin |      |
 public | newTestRenameSeg_b1    | dbadmin |      |
 public | testRenameSeg_b0       | dbadmin |      |
 public | testRenameSeg_b1       | dbadmin |      |
 public | testRenameSeg_pLap     | dbadmin |      |
 public | testRenameSeg_pLap_b1  | dbadmin |      |
 public | testRenameSeg_p_b0     | dbadmin |      |
 public | testRenameSeg_p_b1     | dbadmin |      |
(10 rows)
```

If you rename the anchor table, Vertica also renames its projections:

```
=> ALTER TABLE testRenameSeg RENAME TO newTestRenameSeg;
ALTER TABLEn=> \dj *testRenameSeg*
                 List of projections
 Schema |          Name           |  Owner  | Node | Comment
--------+-------------------------+---------+------+---------
 public | nameTestRenameSeg_p_b0  | dbadmin |      |
 public | nameTestRenameSeg_p_b1  | dbadmin |      |
 public | newTestRenameSeg_b0     | dbadmin |      |
 public | newTestRenameSeg_b1     | dbadmin |      |
 public | newTestRenameSeg_pLap_b0 | dbadmin |      |
 public | newTestRenameSeg_pLap_b1 | dbadmin |      |
 public | newTestRenameSeg_p_b0   | dbadmin |      |
 public | newTestRenameSeg_p_b1   | dbadmin |      |
 public | newTestRenameSeg_v1_b0  | dbadmin |      |
 public | newTestRenameSeg_v1_b1  | dbadmin |      |
(10 rows)
```

Two sets of buddy projections are not renamed, as their names are not prefixed by the original table name:

- nameTestRenameSeg_p_b0
- nameTestRenameSeg_p_b1
- newTestRenameSeg_b0
- newTestRenameSeg_b1

When renaming the other projections, Vertica identified a potential conflict between the table's superprojection—originally testRenameSeg —and existing projection newTestRenameSeg . It resolved this conflict by appending version numbers _v1 and _v2 to the superprojection's new name:

- newTestRenameSeg_v1_b0
- newTestRenameSeg_v1_b1

Auto-projections

Auto-projections are superprojections that Vertica automatically generates for tables, both temporary and persistent. In general, if no projections have been defined for a table, Vertica automatically creates projections for that table when you first load data into it. The following rules apply to all auto-projections:

- Vertica creates the auto-projection in the same schema as the table.
- Auto-projections conform to encoding, sort order, segmentation, and K-safety as specified in the table's creation statement.
- If the table creation statement contains an AS SELECT clause, Vertica uses some properties of the projection definition's underlying query.

Auto-projection triggers

The conditions for creating auto-projections differ, depending on whether the table is temporary or persistent:

Table type	**Auto-projection trigger**
Temporary	CREATE TEMPORARY TABLE statement, unless it includes NO PROJECTION .
Persistent	CREATE TABLE statement contains one of these clauses: • AS SELECT • ENCODED BY • ORDER BY • SEGMENTED BY / UNSEGMENTED • KSAFE If none of these conditions are true, Vertica automatically creates a superprojection (if one does not already exist) when you first load data into the table with INSERT or COPY .

Default segmentation and sort order

If CREATE TABLE or CREATE TEMPORARY TABLE omits a segmentation (SEGMENTED BY or UNSEGMENTED) or ORDER BY clause, Vertica segments and sorts auto-projections as follows:

1. If the table creation statement omits a segmentation (SEGMENTED BY or UNSEGMENTED) clause, Vertica checks configuration parameter SegmentAutoProjection to determine whether to create an auto projection that is segmented or unsegmented. By default, this parameter is set to 1 (enable).
2. If SegmentAutoProjection is enabled and a table's creation statement also omits an ORDER BY clause, Vertica segments and sorts the table's auto-projection according to the table's manner of creation:
   - If CREATE [TEMPORARY] TABLE contains an AS SELECT clause and the query output is segmented, the auto-projection uses the same segmentation. If the result set is already sorted, the projection uses the same sort order.
   - In all other cases, Vertica evaluates table column constraints to determine how to sort and segment the projection, as shown below:

Constraints	**Sorted by:**	**Segmented by:**
Primary key	Primary key	Primary key
Primary and foreign keys	1. Foreign keys 2. Primary key	Primary key
Foreign keys only	1. Foreign keys 2. Remaining columns excluding LONG data types , up to the limit set in configuration parameter MaxAutoSortColumns (by default 8).	All columns excluding LONG data types , up to the limit set in configuration parameter MaxAutoSegColumns (by default 8). Vertica orders segmentation as follows: 1. Small (≤ 8 byte) data type columns: Columns are specified in the same order as they are defined in the table CREATE statement. 1. Large (>8 byte) data type columns: Columns are ordered by ascending size.
None	All columns excluding LONG data types , in the order specified by CREATE TABLE .	

For example, the following table is defined with no primary or foreign keys:

```
=> CREATE TABLE testAutoProj(c10 char (10), v1 varchar(140) DEFAULT v2||v3, i int, c5 char(5), v3 varchar (80), d timestamp, v2 varchar(60), c1 char(1));
CREATE TABLE
=> INSERT INTO testAutoProj VALUES
  ('1234567890',
  DEFAULT,
  1,
  'abcde',
  'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor ',
  current_timestamp,
  'incididunt ut labore et dolore magna aliqua. Eu scelerisque',
  'a');
 OUTPUT
--------
  1
(1 row)
=> COMMIT;
COMMIT
```

Before the INSERT statement loads data into this table for the first time, Vertica automatically creates a superprojection for the table:

```
=> SELECT export_objects('', 'testAutoProj_b0');
------------------------------------------------------------

CREATE PROJECTION public.testAutoProj_b0 /*+basename(testAutoProj),createtype(L)*/
( c10, v1, i, c5, v3, d, v2, c1 )
AS
 SELECT testAutoProj.c10,
    testAutoProj.v1,
    testAutoProj.i,
    testAutoProj.c5,
    testAutoProj.v3,
    testAutoProj.d,
    testAutoProj.v2,
    testAutoProj.c1
 FROM public.testAutoProj
 ORDER BY testAutoProj.c10,
      testAutoProj.v1,
      testAutoProj.i,
      testAutoProj.c5,
      testAutoProj.v3,
      testAutoProj.d,
      testAutoProj.v2,
      testAutoProj.c1
SEGMENTED BY hash(testAutoProj.i, testAutoProj.c5, testAutoProj.d, testAutoProj.c1, testAutoProj.c10, testAutoProj.v2, testAutoProj.v3, testAutoProj.v1)
ALL NODES OFFSET 0;

SELECT MARK_DESIGN_KSAFE(1);

(1 row)
```

Unsegmented projections

In many cases, dimension tables are relatively small, so you do not need to segment them. Accordingly, you should design a K-safe database so projections for its dimension tables are replicated without segmentation on all cluster nodes. You create unsegmented projections with a CREATE PROJECTION statement that includes the clause UNSEGMENTED ALL NODES . This clause specifies to create identical instances of the projection on all cluster nodes.

The following example shows how to create an unsegmented projection for the table store.store_dimension :

```
=> CREATE PROJECTION store.store_dimension_proj (storekey, name, city, state)
        AS SELECT store_key, store_name, store_city, store_state
        FROM store.store_dimension
        UNSEGMENTED ALL NODES;
CREATE PROJECTION
```

Vertica uses the same name to identify all instances of the unsegmented projection—in this example, store.store_dimension_proj . The keyword ALL NODES specifies to replicate the projection on all nodes:

```
=> \dj store.store_dimension_proj
                    List of projections
 Schema |        Name        |  Owner  |       Node       | Comment
--------+----------------------+---------+------------------+---------
 store  | store_dimension_proj | dbadmin | v_vmart_node0001 |
 store  | store_dimension_proj | dbadmin | v_vmart_node0002 |
 store  | store_dimension_proj | dbadmin | v_vmart_node0003 |
(3 rows)
```

For more information about projection name conventions, see Projection naming .

Segmented projections

You typically create segmented projections for large fact tables. Vertica splits segmented projections into chunks (segments) of similar size and distributes these segments evenly across the cluster. System K-safety determines how many duplicates (*buddies*) of each segment are created and maintained on different nodes.

Projection segmentation achieves the following goals:

- Ensures high availability and recovery.
- Spreads the query execution workload across multiple nodes.
- Allows each node to be optimized for different query workloads.

Hash Segmentation

Vertica uses hash segmentation to segment large projections. Hash segmentation allows you to segment a projection based on a built-in hash function that provides even distribution of data across multiple nodes, resulting in optimal query execution. In a projection, the data to be hashed consists of one or more column values, each having a large number of unique values and an acceptable amount of skew in the value distribution. Primary key columns typically meet these criteria, so they are often used as hash function arguments.

You create segmented projections with a CREATE PROJECTION statement that includes a SEGMENTED BY clause .

The following CREATE PROJECTION statement creates projection public.employee_dimension_super . It specifies to include all columns in table public.employee_dimension . The hash segmentation clause invokes the Vertica HASH function to segment projection data on the column employee_key ; it also includes the ALL NODES clause, which specifies to distribute projection data evenly across all nodes in the cluster:

```
=> CREATE PROJECTION public.employee_dimension_super
   AS SELECT * FROM public.employee_dimension
   ORDER BY employee_key
   SEGMENTED BY hash(employee_key) ALL NODES;
```

If the database is K-safe, Vertica creates multiple buddies for this projection and distributes them on different nodes across the cluster. In this case, database K-safety is set to 1, so Vertica creates two buddies for this projection. It uses the projection name employee_dimension_super as the basename for the two buddy identifiers it creates—in this example, employee_dimension_super_b0 and employee_dimension_super_b1 :

```
=> SELECT projection_name FROM projections WHERE projection_basename='employee_dimension_super';
     projection_name
----------------------------
 employee_dimension_super_b0
 employee_dimension_super_b1
(2 rows)
```

K-safe database projections

K-safety is implemented differently for segmented and unsegmented projections, as described below. Examples assume database K-safety is set to 1 in a 3-node database, and uses projections for two tables:

- store.store_orders_fact is a large fact table. The projection for this table should be segmented. Vertica distributes projection segments uniformly across the cluster.
- store.store_dimension is a smaller dimension table. The projection for this table should be unsegmented. Vertica copies a complete instance of this projection on each cluster node.

Segmented projections

In a K-safe database, the database requires K+1 instances, or buddies, of each projection segment. For example, if database K-safety is set to 1, the database requires two instances, or buddies, of each projection segment.

You can set K-safety on individual segmented projections through the CREATE PROJECTION option KSAFE . Projection K-safety must be equal to or greater than database K-safety. If you omit setting KSAFE , the projection obtains K-safety from the database.

The following CREATE PROJECTION defines a segmented projection for the fact table store.store_orders_fact :

```
=> CREATE PROJECTION store.store_orders_fact
      (prodkey, ordernum, storekey, total)
      AS SELECT product_key, order_number, store_key, quantity_ordered*unit_price
      FROM store.store_orders_fact
      SEGMENTED BY HASH(product_key, order_number) ALL NODES KSAFE 1;
CREATE PROJECTION
```

The following keywords in the CREATE PROJECTION statement pertain to setting projection K-safety:

SEGMENTED BY	Specifies how to segment projection data for distribution across the cluster. In this example, the segmentation expression specifies Vertica's built-in HASH function.
ALL NODES	Specifies to distribute projection segments across all cluster nodes.
K-SAFE 1	Sets K-safety to 1. Vertica creates two projection buddies with these identifiers: • store.store_orders_fact_b0 • store.store_orders_fact_b1

Unsegmented projections

In a K-safe database, unsegmented projections must be replicated on all nodes. Thus, the CREATE PROJECTION statement for an unsegmented projection must include the segmentation clause UNSEGMENTED ALL NODES . This instructs Vertica to create identical instances (buddies) of the projection on all cluster nodes. If you create an unsegmented projection on a single node, Vertica regards it unsafe and does not use it.

The following example shows how to create an unsegmented projection for the table store.store_dimension :

```
=> CREATE PROJECTION store.store_dimension_proj (storekey, name, city, state)
      AS SELECT store_key, store_name, store_city, store_state
      FROM store.store_dimension
      UNSEGMENTED ALL NODES;
CREATE PROJECTION
```

Vertica uses the same name to identify all instances of the unsegmented projection—in this example, store.store_dimension_proj . The keyword ALL NODES specifies to replicate the projection on all nodes:

```
=> \dj store.store_dimension_proj
                    List of projections
 Schema |        Name         | Owner |       Node       | Comment
--------+---------------------+-------+------------------+---------
 store  | store_dimension_proj | dbadmin | v_vmart_node0001 |
 store  | store_dimension_proj | dbadmin | v_vmart_node0002 |
 store  | store_dimension_proj | dbadmin | v_vmart_node0003 |
(3 rows)
```

For more information about projection name conventions, see Projection naming .

Partition range projections

Vertica supports projections that specify a range of partition keys. By default, projections store all rows of partitioned table data. Over time, this requirement can incur increasing overhead:

- As data accumulates, increasing amounts of storage are required for large amounts of data that are queried infrequently, if at all.
- Large projections can deter optimizations such as better encodings, or changes to the projection sort order or segmentation. Changes to the projection's DDL like these require you to refresh the entire projection. Depending on the projection size, this refresh operation might span hours or even days.

You can minimize these problems by creating projections for partitioned tables that specify a relatively narrow range of partition keys. For example, the table store_orders is partitioned on order_date , as follows:

```
=> CREATE TABLE public.store_orders(order_no int, order_date timestamp NOT NULL, shipper varchar(20), ship_date date);
CREATE TABLE
=> ALTER TABLE store_orders PARTITION BY order_date::DATE GROUP BY date_trunc('month', (order_date)::DATE);
ALTER TABLE
```

If desired, you can create a projection of store_orders that specifies a contiguous range of the table's partition keys. In the following example, the projection ytd_orders specifies to include only orders that were placed since the first day of the year:

```
=> CREATE PROJECTION ytd_orders AS SELECT * FROM store_orders ORDER BY order_date
   ON PARTITION RANGE BETWEEN date_trunc('year',now())::date AND NULL;
WARNING 4468:  Projection <public.ytd_orders_b0> is not available for query processing. Execute the select start_refresh() function to copy data into this projection.
        The projection must have a sufficient number of buddy projections and all nodes must be up before starting a refresh
WARNING 4468:  Projection <public.ytd_orders_b1> is not available for query processing. Execute the select start_refresh() function to copy data into this projection.
        The projection must have a sufficient number of buddy projections and all nodes must be up before starting a refresh
CREATE PROJECTION
=> SELECT refresh();
                                refresh
-------------------------------------------------------------------------------------------
 Refresh completed with the following outcomes:
Projection Name: [Anchor Table] [Status] [Refresh Method] [Error Count] [Duration (sec)]
-------------------------------------------------------------------------------------------
"public"."ytd_orders_b1": [store_orders] [refreshed] [scratch] [0] [0]
"public"."ytd_orders_b0": [store_orders] [refreshed] [scratch] [0] [0]

(1 row)
```

Each ytd_orders buddy projection requires only 7 ROS containers per node, versus the 77 containers required by the anchor table's superprojection:

```
=> SELECT COUNT (DISTINCT ros_id) NumROS, projection_name, node_name FROM PARTITIONS WHERE projection_name ilike 'store_orders_b%'
GROUP BY node_name, projection_name ORDER BY node_name;
 NumROS | projection_name |    node_name
--------+-----------------+-----------------
     77 | store_orders_b0 | v_vmart_node0001
     77 | store_orders_b1 | v_vmart_node0001
     77 | store_orders_b0 | v_vmart_node0002
     77 | store_orders_b1 | v_vmart_node0002
     77 | store_orders_b0 | v_vmart_node0003
     77 | store_orders_b1 | v_vmart_node0003
(6 rows)

=> SELECT COUNT (DISTINCT ros_id) NumROS, projection_name, node_name FROM PARTITIONS WHERE projection_name ilike 'ytd_orders%' GROUP
BY node_name, projection_name ORDER BY node_name;
 NumROS | projection_name |    node_name
--------+-----------------+-----------------
      7 | ytd_orders_b0   | v_vmart_node0001
      7 | ytd_orders_b1   | v_vmart_node0001
      7 | ytd_orders_b0   | v_vmart_node0002
      7 | ytd_orders_b1   | v_vmart_node0002
      7 | ytd_orders_b0   | v_vmart_node0003
      7 | ytd_orders_b1   | v_vmart_node0003
(6 rows)
```

Partition range requirements

Partition range expressions must conform with requirements that apply to table-level partitioning —for example, partition key format and data type validation.

The following requirements and constraints specifically apply to partition range projections:

- The anchor table must already be partitioned.
- Partition range expressions must be compatible with the table's partition expression.
- The first range expression must resolve to a partition key that is smaller or equal to the second expression.
- If the projection is unsegmented, at least one superprojection of the anchor table must also be unsegmented. If not, Vertica adds the projection to the database catalog, but throws a warning that this projection cannot be used to process queries until you create an unsegmented superprojection.
- Partition range expressions do not support subqueries.

Anchor table dependencies

As noted earlier, a partition range projection depends on the anchor table being partitioned on the same expression. If you remove table partitioning from the projection's anchor table, Vertica drops the dependent projection. Similarly, if you modify the anchor table's partition clause, Vertica drops the projection.

The following exception applies: if the anchor table's new partition clause leaves the partition expression unchanged, the dependent projection is not dropped and remains available for queries. For example, the table store_orders and its projection ytd_orders were originally partitioned as follows:

```
=> ALTER TABLE store_orders PARTITION BY order_date::DATE GROUP BY DATE_TRUNC('month', (order_date)::DATE);
 ...
=> CREATE PROJECTION ytd_orders AS SELECT * FROM store_orders ORDER BY order_date
   ON PARTITION RANGE BETWEEN date_trunc('year',now())::date AND NULL;
```

If you now modify store_orders to use hierarchical partitioning, Vertica repartitions the table data as well as its partition range projection:

```
=> ALTER TABLE store_orders PARTITION BY order_date::DATE GROUP BY CALENDAR_HIERARCHY_DAY(order_date::DATE, 2, 2) REORGANIZE;
NOTICE 4785:  Started background repartition table task
ALTER TABLE
```

Because both store_orders and the ytd_orders projection remain partitioned on the order_date column, the ytd_orders projection remains valid. Also, the scope of projection data remains unchanged, so the projection requires no refresh. However, in the background, the Tuple Mover silently reorganizes the projection ROS containers as per the new hierarchical partitioning of its anchor table:

```
=> SELECT COUNT (DISTINCT ros_id) NumROS, projection_name, node_name FROM PARTITIONS WHERE projection_name ilike 'ytd_orders%' GROUP
BY node_name, projection_name ORDER BY node_name;
 NumROS | projection_name |    node_name
--------+-----------------+------------------
     38 | ytd_orders_b0   | v_vmart_node0001
     38 | ytd_orders_b1   | v_vmart_node0001
     38 | ytd_orders_b0   | v_vmart_node0002
     38 | ytd_orders_b1   | v_vmart_node0002
     38 | ytd_orders_b0   | v_vmart_node0003
     38 | ytd_orders_b1   | v_vmart_node0003
(6 rows)
```

Modifying existing projections

You can modify the partition range of a projection with ALTER PROJECTION . No refresh is required if the new range is within the previous range. Otherwise, a refresh is required before the projection reflects the modified partition range. Prior to refreshing, the projection continues to return data within the unmodified range.

For example, projection ytd_orders previously specified a partition range that starts on the first day of the current year. The following ALTER PROJECTION statement changes the range to start on October 1 of last year. The new range precedes the previous one, so Vertica issues a warning to refresh the specified projection ytd_orders_b0 and its buddy projection ytd_orders_b1 :

```
=> ALTER PROJECTION ytd_orders_b0 ON PARTITION RANGE BETWEEN
    add_months(date_trunc('year',now())::date, -3) AND NULL;
WARNING 10001:  Projection "public.ytd_orders_b0" changed to out-of-date state as new partition range is not covered by existing partition range
HINT:  Call refresh() or start_refresh() to refresh the projections
WARNING 10001:  Projection "public.ytd_orders_b1" changed to out-of-date state as new partition range is not covered by existing partition range
HINT:  Call refresh() or start_refresh() to refresh the projections
ALTER PROJECTION
```

You can change a regular projection to a partition range projection provided no history or data loss will occur, such as by doing the following:

```
=> ALTER PROJECTION foo ON PARTITION RANGE BETWEEN '22' AND NULL;
```

Dynamic partition ranges

A projection's partition range can be static, set by expressions that always resolve to the same value. For example, the following projection specifies a static range between 06/01/21 and 06/30/21:

```
=> CREATE PROJECTION last_month_orders AS SELECT * FROM store_orders ORDER BY order_date ON PARTITION RANGE BETWEEN
    '2021-06-01' AND '2021-06-30';
 ...
CREATE PROJECTION
```

More typically, partition range expressions use stable date functions such as ADD_MONTHS , DATE_TRUNC and NOW to specify a dynamic range. In the following example, the partition range is set from the first day of the previous month. As the calendar date advances to the next month, the partition range advances with it:

```
=> ALTER PROJECTION last_month_orders_b0 ON PARTITION RANGE BETWEEN
    add_months(date_trunc('month', now())::date, -1) AND NULL;
ALTER PROJECTION
```

As a best practice, always leave the maximum range open-ended by setting it to NULL, and rely on queries to determine the maximum amount of data to fetch. For example, a query that fetches all store orders placed last month might look like this:

```
=> SELECT * from store_orders WHERE order_date BETWEEN
    add_months(date_trunc('month', now())::date, -1) AND
    add_months(date_trunc('month', now())::date + dayofmonth(now()), -1);
```

The query plan generated to execute this query shows that it uses the partition range projection last_month_orders :

```
=> EXPLAIN SELECT * from store_orders WHERE order_date BETWEEN
    add_months(date_trunc('month', now())::date, -1) AND
    add_months(date_trunc('month', now())::date + dayofmonth(now()), -1);

 Access Path:
 +-STORAGE ACCESS for store_orders [Cost: 34, Rows: 763 (NO STATISTICS)] (PATH ID: 1)
 |  Projection: public.last_month_orders_b0
 |  Materialize: store_orders.order_date, store_orders.order_no, store_orders.shipper, store_orders.ship_date
 |  Filter: ((store_orders.order_date >= '2021-06-01 00:00:00'::timestamp(0)) AND (store_orders.order_date <= '2021-06-3
0 00:00:00'::timestamp(0)))
 |  Execute on: All Nodes
```

Dynamic partition range maintenance

The Projection Maintainer is a background service that checks projections with projection range expressions hourly. If the value of either expression in a projection changes, the Projection Maintainer compares the new and old values in PARTITION_RANGE_MIN and PARTITION_RANGE_MAX to determine whether the partition range contracted or expanded:

- If the partition range contracted in either direction—that is, PARTITION_RANGE_MIN is greater, or PARTITION_RANGE_MAX is smaller than its previous value—then the Projection Maintainer acts as follows:
  - Updates the system table PROJECTIONS with new values in columns PARTITION_RANGE_MIN and PARTITION_RANGE_MAX .
  - Queues a MERGEOUT request to purge unused data from this range. The projection remains available to execute queries within the updated range.
- If the partition range expanded in either direction—that is, PARTITION_RANGE_MIN is smaller, or PARTITION_RANGE_MAX is greater than its previous value—then the Projection Maintainer leaves the projection and the PROJECTIONS table unchanged. Because the partition range remains unchanged, Vertica regards the existing projection data as up to date, so it also can never be refreshed.

For example, the following projection creates a partition range that includes all orders in the current month:

```
=> CREATE PROJECTION mtd_orders AS SELECT * FROM store_orders ON PARTITION RANGE BETWEEN
   date_trunc('month', now())::date AND NULL;
```

If you create this partition in July of 2021, the minimum partition range expression— date_trunc('month', now())::date —initially resolves to the first day of the month: 2021-07-01. At the start of the following month, sometime between 2021-08-01 00:00 and 2021-08-01 01:00 , the Projection Maintainer compares the minimum range expression against system time. It then acts as follows:

1. Updates the PROJECTIONS table and sets PARTITION_RANGE_MIN for projection mtd_orders to 2021-08-01 .
2. Queues a MERGEOUT request to purge from this projection's partition range all rows with keys that predate 2021-08-01 .

Important

Given the example shown above, you might consider setting the projection's maximum partition range expression as follows:

```
add_months(date_trunc('month', now()), 1) - 1
```

This expression would always resolve to the last day of the current month. With each new month, the maximum partition range would be one month greater than its earlier value. As noted earlier, the Projection Maintainer ignores any expansion of a partition range, so it would leave the minimum and maximum partition range values for mtd_orders unchanged. To avoid issues like this, always set the maximum partition expression to NULL.

Refreshing projections

When you create a projection for a table that already contains data, Vertica does not automatically load that data into the new projection. Instead, you must explicitly refresh that projection. Until you do so, the projection cannot participate in executing queries on its anchor table.

You can refresh a projection with one of the following functions:

- START_REFRESH refreshes projections in the current schema with the latest data of their respective anchor tables . START_REFRESH runs asynchronously in the background.
- REFRESH synchronously refreshes one or more table projections in the foreground.

Both functions update system tables that maintain information about a projection's refresh status: PROJECTION_REFRESHES , PROJECTIONS , and PROJECTION_CHECKPOINT_EPOCHS .

If a refresh would violate a table or schema disk quota, the operation fails. For more information, see Disk quotas .

Getting projection refresh information

You can query the PROJECTION_REFRESHES and PROJECTIONS system tables to view the progress of the refresh operation. You can also call the GET_PROJECTIONS function to view the final status of projection refreshes for a given table:

```
=> SELECT GET_PROJECTIONS('customer_dimension');
                                GET_PROJECTIONS
-------------------------------------------------------------------------------------------------------
 Current system K is 1.
# of Nodes: 3.
Table public.customer_dimension has 2 projections.

Projection Name: [Segmented] [Seg Cols] [# of Buddies] [Buddy Projections] [Safe] [UptoDate] [Stats]
-------------------------------------------------------------------------------------------------------
public.customer_dimension_b1 [Segmented: Yes] [Seg Cols: "public.customer_dimension.customer_key"] [K: 1]
      [public.customer_dimension_b0] [Safe: Yes] [UptoDate: Yes] [Stats: RowCounts]
public.customer_dimension_b0 [Segmented: Yes] [Seg Cols: "public.customer_dimension.customer_key"] [K: 1]
      [public.customer_dimension_b1] [Safe: Yes] [UptoDate: Yes] [Stats: RowCounts]

(1 row)
```

Refresh methods

Vertica can refresh a projection from one of its buddies, if one is available. In this case, the target projection gets the source buddy's historical data. Otherwise, the projection is refreshed from scratch with data of the latest epoch at the time of the refresh operation. In this case, the projection cannot participate in historical queries on any epoch that precedes the refresh operation.

To determine the method used to refresh a given projection, query the REFRESH_METHOD column from the PROJECTION_REFRESHES system table.

Dropping projections

Projections can be dropped explicitly through the DROP PROJECTION statement. They are also implicitly dropped when you drop their anchor table.

Partitioning tables

Data partitioning is defined as a table property , and is implemented on all projections of that table. On all load, refresh, and recovery operations, the Vertica Tuple Mover automatically partitions data into separate ROS containers. Each ROS container contains data for a single partition or partition group ; depending on space requirements, a partition or partition group can span multiple ROS containers.

For example, it is common to partition data by time slices. If a table contains decades of data, you can partition it by year. If the table contains only one year of data, you can partition it by month.

Logical divisions of data can significantly improve query execution. For example, if you query a table on a column that is in the table's partition clause, the query optimizer can quickly isolate the relevant ROS containers (see Partition pruning).

Partitions can also facilitate DML operations. For example, given a table that is partitioned by months, you might drop all data for the oldest month when a new month begins. In this case, Vertica can easily identify the ROS containers that store the partition data to drop. For details, see Managing partitions .

In this section

- Defining partitions
- Hierarchical partitioning
- Partitioning and segmentation
- Managing partitions
- Active and inactive partitions
- Partition pruning

Defining partitions

You can specify partitioning for new and existing tables:

- Define partitioning for a table with CREATE TABLE .
- Specify partitioning for an existing table by modifying its definition with ALTER TABLE .
- Create partition groups to consolidate partitions into logical subsets , minimizing the use of ROS storage.

In this section

- Partitioning a new table

- Partitioning existing table data
- Partition grouping

Partitioning a new table

Use CREATE TABLE to partition a new table, as specified by the PARTITION BY clause :

```
CREATE TABLE table-name... PARTITION BY partition-expression [ GROUP BY group-expression ] [ REORGANIZE ];
```

The following statements create the store_orders table and load data into it. The CREATE TABLE statement includes a simple partition clause that specifies to partition data by year:

```
=> CREATE TABLE public.store_orders
(
    order_no int,
    order_date timestamp NOT NULL,
    shipper varchar(20),
    ship_date date
)
UNSEGMENTED ALL NODES
PARTITION BY YEAR(order_date);
CREATE TABLE
=> COPY store_orders FROM '/home/dbadmin/export_store_orders_data.txt';
41834
```

As COPY loads the new table data into ROS storage, the Tuple Mover executes the table's partition clause by dividing orders for each year into separate partitions, and consolidating these partitions in ROS containers.

In this case, the Tuple Mover creates four partition keys for the loaded data—2017, 2016, 2015, and 2014—and divides the data into separate ROS containers accordingly:

```
=> SELECT dump_table_partition_keys('store_orders');
... Partition keys on node v_vmart_node0001
  Projection 'store_orders_super'
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: 2017
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: 2016
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: 2015
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: 2014

 Partition keys on node v_vmart_node0002
  Projection 'store_orders_super'
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: 2017
...

(1 row)
```

As new data is loaded into store_orders , the Tuple Mover merges it into the appropriate partitions, creating partition keys as needed for new years.

Partitioning existing table data

Use ALTER TABLE to partition or repartition an existing table, as specified by the PARTITION BY clause:

```
ALTER TABLE table-name PARTITION BY partition-expression [ GROUP BY group-expression ] [ REORGANIZE ];
```

For example, you might repartition the store_orders table, defined earlier . The following ALTER TABLE divides all store_orders data into monthly partitions for each year, each partition key identifying the order date year and month:

```
=> ALTER TABLE store_orders
    PARTITION BY EXTRACT(YEAR FROM order_date)*100 + EXTRACT(MONTH FROM order_date)
       GROUP BY EXTRACT(YEAR from order_date)*100 + EXTRACT(MONTH FROM order_date);
NOTICE 8364:  The new partitioning scheme will produce partitions in 42 physical storage containers per projection
WARNING 6100:  Using PARTITION expression that returns a Numeric value
HINT:  This PARTITION expression may cause too many data partitions.  Use of an expression that returns a more accurate value, such as a regular
VARCHAR or INT, is encouraged
WARNING 4493:  Queries using table "store_orders" may not perform optimally since the data may not be repartitioned in accordance with the new partition
expression
HINT:  Use "ALTER TABLE public.store_orders REORGANIZE;" to repartition the data
```

After executing this statement, Vertica drops existing partition keys. However, the partition clause omits REORGANIZE , so existing data remains stored according to the previous partition clause. This can put table partitioning in an inconsistent state and adversely affect query performance, DROP_PARTITIONS , and node recovery. In this case, you must explicitly request Vertica to reorganize existing data into new partitions, in one of the following ways:

- Issue ALTER TABLE...REORGANIZE:
  ```
  ALTER TABLE table-name REORGANIZE;
  ```
- Call the Vertica meta-function PARTITION_TABLE .

For example:

```
=> ALTER TABLE store_orders REORGANIZE;
NOTICE 4785:  Started background repartition table task
ALTER TABLE
```

ALTER TABLE...REORGANIZE and PARTITION_TABLE operate identically: both split any ROS containers where partition keys do not conform with the new partition clause. On executing its next mergeout, the Tuple Mover merges partitions into the appropriate ROS containers.

Partition grouping

Partition groups consolidate partitions into logical subsets that minimize use of ROS storage. Reducing the number of ROS containers to store partitioned data helps facilitate DML operations such as DELETE and UPDATE , and avoid ROS pushback. For example, you can group date partitions by year. By doing so, the Tuple Mover allocates ROS containers for each year group, and merges individual partitions into these ROS containers accordingly.

Creating partition groups

You create partition groups by qualifying the PARTITION BY clause with a GROUP BY clause:

```
ALTER TABLE table-name PARTITION BY partition-expression [ GROUP BY group-expression ]
```

The GROUP BY clause specifies how to consolidate partition keys into groups, where each group is identified by a unique partition group key. For example, the following ALTER TABLE statement specifies to repartition the store_orders table (shown in Partitioning a new table) by order dates, grouping partition keys by year. The group expression— DATE_TRUNC('year', (order_date)::DATE) —uses the partition expression order_date::DATE to generate partition group keys:

```
=> ALTER TABLE store_orders
    PARTITION BY order_date::DATE GROUP BY DATE_TRUNC('year', (order_date)::DATE) REORGANIZE;
NOTICE 8364:  The new partitioning scheme will produce partitions in 4 physical storage containers per projection
NOTICE 4785:  Started background repartition table task
```

In this case, the order_date column dates span four years. The Tuple Mover creates four partition group keys, and merges store_orders partitions into group-specific ROS storage containers accordingly:

```
=> SELECT DUMP_TABLE_PARTITION_KEYS('store_orders');
 ...
 Partition keys on node v_vmart_node0001
  Projection 'store_orders_super'
   Storage [ROS container]
     No of partition keys: 173
     Partition keys: 2017-01-02 2017-01-03 2017-01-04 ... 2017-09-25 2017-09-26 2017-09-27
   Storage [ROS container]
     No of partition keys: 212
     Partition keys: 2016-01-01 2016-01-04 2016-01-05 ... 2016-11-23 2016-11-24 2016-11-25
   Storage [ROS container]
     No of partition keys: 213
     Partition keys: 2015-01-01 2015-01-02 2015-01-05 ... 2015-11-23 2015-11-24 2015-11-25
2015-11-26 2015-11-27
   Storage [ROS container]
     No of partition keys: 211
     Partition keys: 2014-01-01 2014-01-02 2014-01-03 ... 2014-11-25 2014-11-26 2014-11-27
  Projection 'store_orders_super'
   Storage [ROS container]
     No of partition keys: 173
...
```

Caution

This example demonstrates how partition grouping can facilitate more efficient use of ROS storage. However, grouping all partitions into several large and static ROS containers can adversely affect performance, especially for a table that is subject to frequent DML operations. Frequent load operations in particular can incur considerable merge overhead, which, in turn, reduces performance.

Vertica recommends that you use CALENDAR_HIERARCHY_DAY , as a partition clause's group expression. This function automatically groups DATE partition keys into a dynamic hierarchy of years, months, and days. Doing so helps minimize merge-related issues. For details, see Hierarchical partitioning .

Managing partitions within groups

You can use various partition management functions , such as DROP_PARTITIONS or MOVE_PARTITIONS_TO_TABLE , to target a range of order dates within a given partition group, or across multiple partition groups. In the previous example, each group contains partition keys of different dates within a given year. You can use DROP_PARTITIONS to drop order dates that span two years, 2014 and 2015:

```
=> SELECT DROP_PARTITIONS('store_orders', '2014-05-30', '2015-01-15', 'true');
```

Important

The drop operation requires Vertica to split the ROS containers that store partition groups for these two years. To do so, the function's force_split parameter must be set to true.

Hierarchical partitioning

The meta-function CALENDAR_HIERARCHY_DAY leverages partition grouping . You specify this function as the partitioning GROUP BY expression. CALENDAR_HIERARCHY_DAY organizes a table's date partitions into a hierarchy of groups: the oldest date partitions are grouped by year, more recent partitions are grouped by month, and the most recent date partitions remain un-grouped. Grouping is dynamic : as recent data ages, the Tuple Mover merges their partitions into month groups, and eventually into year groups.

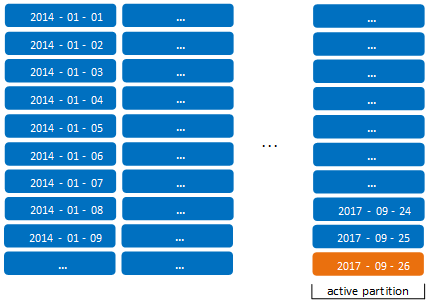
Managing timestamped data

Partition consolidation strategies are especially important for managing timestamped data, where the number of partitions can quickly escalate and risk ROS pushback. For example, the following statements create the store_orders table and load data into it. The CREATE TABLE statement includes a simple partition clause that specifies to partition data by date:

```
=> DROP TABLE IF EXISTS public.store_orders CASCADE;
=> CREATE TABLE public.store_orders
(
    order_no int,
    order_date timestamp NOT NULL,
    shipper varchar(20),
    ship_date date
)
UNSEGMENTED ALL NODES PARTITION BY order_date::DATE;
CREATE TABLE
=> COPY store_orders FROM '/home/dbadmin/export_store_orders_data.txt';
41834
(1 row)
```

As COPY loads the new table data into ROS storage, it executes this table's partition clause by dividing daily orders into separate partitions—in this case, 809 partitions, where each partition requires its own ROS container:

```
=> SELECT COUNT (DISTINCT ros_id) NumROS, node_name FROM PARTITIONS
   WHERE projection_name ilike '%store_orders_super%' GROUP BY node_name ORDER BY node_name;
 NumROS |    node_name
--------+------------------
    809 | v_vmart_node0001
    809 | v_vmart_node0002
    809 | v_vmart_node0003
(3 rows)
```



This is far above the recommended maximum of 50 partitions per projection. This number is also close to the default system limit of 1024 ROS containers per projection, risking ROS pushback in the near future.

You can approach this problem in several ways:

- Consider consolidating table data into larger partitions—for example, partition by month instead of day. However, partitioning data at this level might limit effective use of partition management functions .
- Regularly archive older partitions , and thereby minimize the number of accumulated partitions. However, this requires an extra layer of data management, and also inhibits access to historical data.

Alternatively, you can use CALENDAR_HIERARCHY_DAY to automatically merge partitions into a date-based hierarchy of partition groups. Each partition group is stored in its own set of ROS containers, apart from other groups. You specify this function in the table partition clause as follows:

```
PARTITION BY partition-expression
 GROUP BY CALENDAR_HIERARCHY_DAY( partition-expression [, active-months[, active-years] ] )
```

Important
Two requirements apply to using CALENDAR_HIERARCHY_DAY in a partition clause:

- *partition-expression* must be a DATE .
- The partition expressions specified by the PARTITION BY clause and CALENDAR_HIERARCHY_DAY must be identical.

For example, given the previous table, you can repartition it as follows:

```
=> ALTER TABLE public.store_orders
    PARTITION BY order_date::DATE
    GROUP BY CALENDAR_HIERARCHY_DAY(order_date::DATE, 2, 2) REORGANIZE;
```

Grouping DATE data hierarchically

CALENDAR_HIERARCHY_DAY creates hierarchies of partition groups, and merges partitions into the appropriate groups. It does so by evaluating the partition expression of each table row with the following algorithm, to determine its partition group key:

```
GROUP BY (
CASE WHEN DATEDIFF('YEAR', partition-expression, NOW()::TIMESTAMPTZ(6)) >= active-years
     THEN DATE_TRUNC('YEAR', partition-expression::DATE)
    WHEN DATEDIFF('MONTH', partition-expression, NOW()::TIMESTAMPTZ(6)) >= active-months
     THEN DATE_TRUNC('MONTH', partition-expression::DATE)
    ELSE DATE_TRUNC('DAY', partition-expression::DATE) END);
```

In this example, the algorithm compares order_date in each store_orders row to the current date as follows:

1. Determines whether order_date is in an inactive year.
   If order_date is in an inactive year, the row's partition group key resolves to that year. The row is merged into a ROS container for that year.
2. If order_date is an active year, CALENDAR_HIERARCHY_DAY evaluates order_date to determine whether it is in an inactive month.
   If order_date is in an inactive month, the row's partition group key resolves to that month. The row is merged into a ROS container for that month.
3. If order_date is in an active month, the row's partition group key resolves to the order_date day. This row is merged into a ROS container for that day. Any rows where order_date is a future date is treated in the same way.

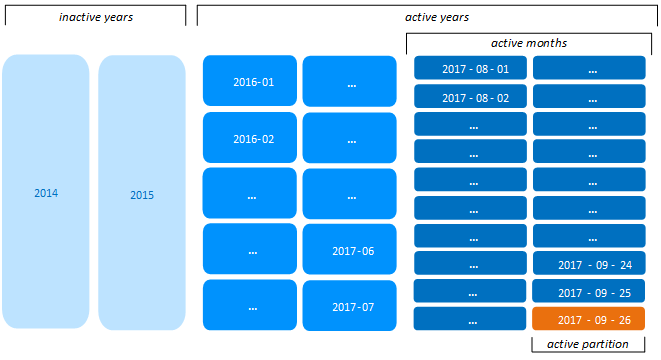
Important

The CALENDAR_HIERARCHY_DAY algorithm assumes that most table activity is focused on recent dates. Setting *active-years* and *active-months* to a low number ≥ 2 serves to isolate most merge activity to date-specific containers, and incurs minimal overhead. Vertica recommends that you use the default setting of 2 for *active-years* and *active-months* . For most users, these settings achieve an optimal balance between ROS storage and performance.

As a best practice, never set *active-years* and *active-months* to 0.

For example, if the current date is 2017-09-26, CALENDAR_HIERARCHY_DAY resolves *active-years* and *active-months* to the following date spans:

- *active-years* : 2016-01-01 to 2017-12-31. Partitions in active years are grouped into monthly ROS containers or are merged into daily ROS containers. Partitions from earlier years are regarded as inactive and merged into yearly ROS containers.
- *active-months* : 2017-08-01 to 2017-09-30. Partitions in active months are merged into daily ROS containers.



Now, the total number of ROS containers is reduced to 40 per projection:

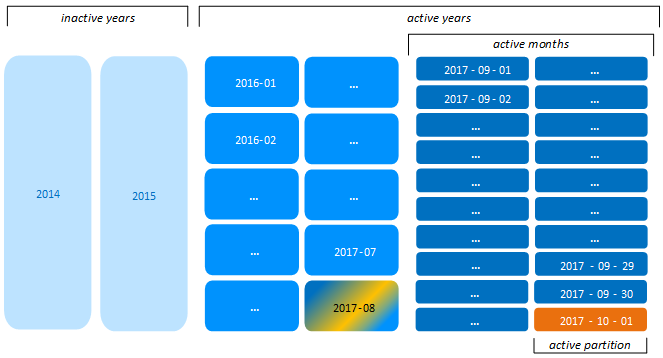
```
=> SELECT COUNT (DISTINCT ros_id) NumROS, node_name FROM PARTITIONS
   WHERE projection_name ilike '%store_orders_super%' GROUP BY node_name ORDER BY node_name;
 NumROS |    node_name
--------+------------------
    40 | v_vmart_node0001
    40 | v_vmart_node0002
    40 | v_vmart_node0003
(3 rows)
```

Note
Regardless of how the Tuple Mover groups and merges partitions, it always identifies one or more partitions or partition groups as active. For details, see Active and inactive partitions .

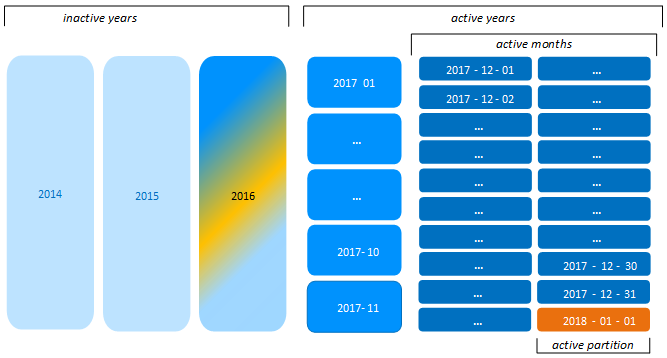
Dynamic regrouping

As shown earlier, CALENDAR_HIERARCHY_DAY references the current date when it creates partition group keys and merges partitions. As the calendar advances, the Tuple Mover reevaluates the partition group keys of tables that are partitioned with this function, and moves partitions as needed to different ROS containers.

Thus, given the previous example, on 2017-10-01 the Tuple Mover creates a monthly ROS container for August partitions. All partition keys between 2017-08-01 and 2017-08-31 are merged into the new ROS container 2017-08:



Likewise, on 2018-01-01, the Tuple Mover creates a ROS container for 2016 partitions. All partition keys between 2016-01-01 and 2016-12-31 that were previously grouped by month are merged into the new yearly ROS container:



Caution
After older partitions are grouped into months and years, any partition operation that acts on a subset of older partition groups is liable to split ROS containers into smaller ROS containers for each partition—for example, MOVE_PARTITIONS_TO_TABLE , where *force-split* is set to true. These operations can lead to ROS pushback. If you anticipate frequent partition operations on hierarchically grouped partitions, consider modifying the partition expression so partitions are grouped no higher than months.

Customizing partition group hierarchies

Vertica provides a single function, CALENDAR_HIERARCHY_DAY, to facilitate hierarchical partitioning. Vertica stores the GROUP BY clause as a CASE statement that you can edit to suit your own requirements.

For example, Vertica stores the store_orders partition clause as follows:

```
=> ALTER TABLE public.store_orders
     PARTITION BY order_date::DATE
     GROUP BY CALENDAR_HIERARCHY_DAY(order_date::DATE, 2, 2);
=> select export_tables('','store_orders');
...
CREATE TABLE public.store_orders ( ... )

PARTITION BY ((store_orders.order_date)::date)
GROUP BY (
CASE WHEN ("datediff"('year', (store_orders.order_date)::date, ((now())::timestamptz(6))::date) >= 2)
      THEN (date_trunc('year', (store_orders.order_date)::date))::date
    WHEN ("datediff"('month', (store_orders.order_date)::date, ((now())::timestamptz(6))::date) >= 2)
      THEN (date_trunc('month', (store_orders.order_date)::date))::date
    ELSE (store_orders.order_date)::date END);
```

You can modify the CASE statement to customize the hierarchy of partition groups. For example, the following CASE statement creates a hierarchy of months, days, and hours:

```
=> ALTER TABLE store_orders
PARTITION BY (store_orders.order_date)
GROUP BY (
CASE WHEN DATEDIFF('MONTH', store_orders.order_date, NOW()::TIMESTAMPTZ(6)) >= 2
      THEN DATE_TRUNC('MONTH', store_orders.order_date::DATE)
    WHEN DATEDIFF('DAY', store_orders.order_date, NOW()::TIMESTAMPTZ(6)) >= 2
      THEN DATE_TRUNC('DAY', store_orders.order_date::DATE)
    ELSE DATE_TRUNC('hour', store_orders.order_date::DATE) END);
```

Partitioning and segmentation

In Vertica, partitioning and segmentation are separate concepts and achieve different goals to localize data:

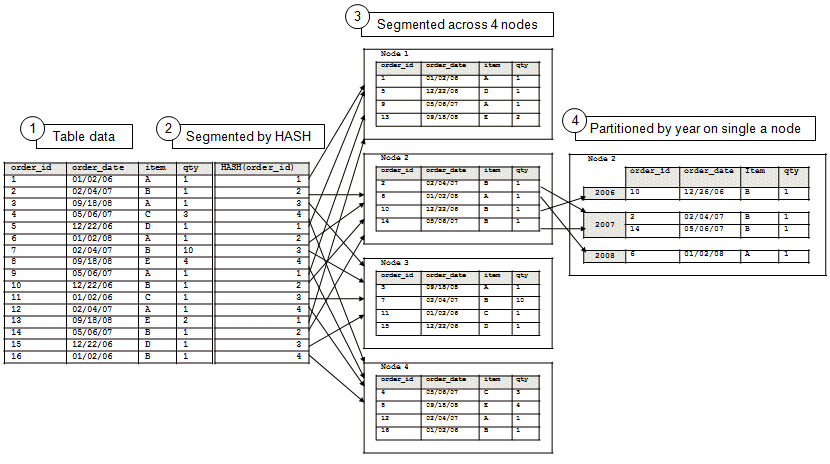
- **Segmentation** refers to organizing and distributing data across cluster nodes for fast data purges and query performance. Segmentation aims to distribute data evenly across multiple database nodes so all nodes participate in query execution. You specify segmentation with the CREATE PROJECTION statement's hash segmentation clause .
- **Partitioning** specifies how to organize data within individual nodes for distributed computing. Node partitions let you easily identify data you wish to drop and help reclaim disk space. You specify partitioning with the CREATE TABLE statement's PARTITION BY clause.

For example: partitioning data by year makes sense for retaining and dropping annual data. However, segmenting the same data by year would be inefficient, because the node holding data for the current year would likely answer far more queries than the other nodes.

The following diagram illustrates the flow of segmentation and partitioning on a four-node database cluster:

1. Example table data
2. Data segmented by HASH(order_id)
3. Data segmented by hash across four nodes
4. Data partitioned by year on a single node

While partitioning occurs on all four nodes, the illustration shows partitioned data on one node for simplicity.



See also

- Reclaiming disk space from deleted table data
- Identical segmentation
- Segmented projections
- Unsegmented projections
- CREATE PROJECTION
- CREATE TABLE

Managing partitions

You can manage partitions with the following operations:

- Drop partitions
- Archive partitions
- Swap partitions
- Minimize partitions
- View partition storage

In this section

- Dropping partitions
- Archiving partitions
- Swapping partitions
- Minimizing partitions
- Viewing partition storage data

Dropping partitions

Use the DROP_PARTITIONS function to drop one or more partition keys for a given table. You can specify a single partition key or a range of partition keys.

For example, the table shown in Partitioning a new table is partitioned by column order_date :

```
=> CREATE TABLE public.store_orders
(
    order_no int,
    order_date timestamp NOT NULL,
    shipper varchar(20),
    ship_date date
)
PARTITION BY YEAR(order_date);
```

Given this table definition, Vertica creates a partition key for each unique order_date year—in this case, 2017, 2016, 2015, and 2014—and divides the data into separate ROS containers accordingly.

The following DROP_PARTITIONS statement drops from table store_orders all order records associated with partition key 2014:

```
=> SELECT DROP_PARTITIONS ('store_orders', 2014, 2014);
Partition dropped
```

Splitting partition groups

If a table partition clause includes a GROUP BY clause, partitions are consolidated in the ROS by their partition group keys. DROP_PARTITIONS can then specify a range of partition keys within a given partition group, or across multiple partition groups. In either case, the drop operation requires Vertica to split the ROS containers that store these partitions. To do so, the function's force_split parameter must be set to true.

For example, the store_orders table shown above can be repartitioned with a GROUP BY clause as follows:

```
=> ALTER TABLE store_orders
    PARTITION BY order_date::DATE GROUP BY DATE_TRUNC('year', (order_date)::DATE) REORGANIZE;
```

With all 2014 order records having been dropped earlier, order_date values now span three years—2017, 2016, and 2015. Accordingly, the Tuple Mover creates three partition group keys for each year, and designates one or more ROS containers for each group. It then merges store_orders partitions into the appropriate groups.

The following DROP_PARTITIONS statement specifies to drop order dates that span two years, 2014 and 2015:

```
=> SELECT DROP_PARTITIONS('store_orders', '2015-05-30', '2016-01-16', 'true');
Partition dropped
```

The drop operation requires Vertica to drop partitions from two partition groups—2015 and 2016. These groups span at least two ROS containers, which must be split in order to remove the target partitions. Accordingly, the function's force_split parameter is set to true.

Scheduling partition drops

If your hardware has fixed disk space, you might need to configure a regular process to roll out old data by dropping partitions.

For example, if you have only enough space to store data for a fixed number of days, configure Vertica to drop the oldest partition keys. To do so, create a time-based job scheduler such as cron to schedule dropping the partition keys during low-load periods.

If the ingest rate for data has peaks and valleys, you can use two techniques to manage how you drop partition keys:

- Set up a process to check the disk space on a regular (daily) basis. If the percentage of used disk space exceeds a certain threshold—for example, 80%—drop the oldest partition keys.
- Add an artificial column in a partition that increments based on a metric like row count. For example, that column might increment each time the row count increases by 100 rows. Set up a process that queries this column on a regular (daily) basis. If the value in the new column exceeds a certain threshold—for example, 100—drop the oldest partition keys, and set the column value back to 0.

Table locking

DROP_PARTITIONS requires an exclusive D lock on the target table. This lock is only compatible with I-lock operations, so only table load operations such as INSERT and COPY are allowed during drop partition operations.

Archiving partitions

You can move partitions from one table to another with the Vertica function MOVE_PARTITIONS_TO_TABLE . This function is useful for archiving old partitions, as part of the following procedure:

1. Identify the partitions to archive, and move them to a temporary staging table with MOVE_PARTITIONS_TO_TABLE .
2. Back up the staging table .
3. Drop the staging table .

You restore archived partitions at any time.

Move partitions to staging tables

You archive historical data by identifying the partitions you wish to remove from a table. You then move each partition (or group of partitions) to a temporary staging table.

Before calling MOVE_PARTITIONS_TO_TABLE :

- Refresh all out-of-date projections.

The following recommendations apply to staging tables:

- To facilitate the backup process, create a unique schema for the staging table of each archiving operation.
- Specify new names for staging tables. This ensures that they do not contain partitions from previous move operations.

  If the table does not exist, Vertica creates a table from the source table's definition, by calling CREATE TABLE with LIKE and INCLUDING PROJECTIONS clause. The new table inherits ownership from the source table. For details, see Replicating a table .
- Use staging names that enable other users to easily identify partition contents. For example, if a table is partitioned by dates, use a name that specifies a date or date range.

In the following example, MOVE_PARTITIONS_TO_TABLE specifies to move a single partition to the staging table partn_backup.tradfes_200801 .

```
=> SELECT MOVE_PARTITIONS_TO_TABLE (
        'prod_trades',
        '200801',
        '200801',
        'partn_backup.trades_200801');
MOVE_PARTITIONS_TO_TABLE
-------------------------------------------------
 1 distinct partition values moved at epoch 15.
(1 row)
```

Back up the staging table

After you create a staging table, you archive it through an object-level backup using a vbr configuration file. For detailed information, see Backing up and restoring the database .

Important

Vertica recommends performing a full database backup before the object-level backup, as a precaution against data loss. You can only restore object-level backups to the original database.

Drop the staging tables

After the backup is complete, you can drop the staging table as described in Dropping tables .

Restoring archived partitions

You can restore partitions that you previously moved to an intermediate table, archived as an object-level backup, and then dropped.

> **Note**
> Restoring an archived partition requires that the original table definition is unchanged since the partition was archived and dropped. If the table definition changed, you can restore an archived partition with INSERT...SELECT statements, which are not described here.

These are the steps to restoring archived partitions:

1. Restore the backup of the intermediate table you saved when you moved one or more partitions to archive (see Archiving partitions).
2. Move the restored partitions from the intermediate table to the original table.
3. Drop the intermediate table.

Swapping partitions

SWAP_PARTITIONS_BETWEEN_TABLES combines the operations of DROP_PARTITIONS and MOVE_PARTITIONS_TO_TABLE as a single transaction. SWAP_PARTITIONS_BETWEEN_TABLES is useful if you regularly load partitioned data from one table into another and need to refresh partitions in the second table.

For example, you might have a table of revenue that is partitioned by date, and you routinely move data into it from a staging table. Occasionally, the staging table contains data for dates that are already in the target table. In this case, you must first remove partitions from the target table for those dates, then replace them with the corresponding partitions from the staging table. You can accomplish both tasks with a single call to SWAP_PARTITIONS_BETWEEN_TABLES.

By wrapping the drop and move operations within a single transaction, SWAP_PARTITIONS_BETWEEN_TABLES maintains integrity of the swapped data. If any task in the swap operation fails, the entire operation fails and is rolled back.

Example

The following example creates two partitioned tables and then swaps certain partitions between them.

Both tables have the same definition and have partitions for various year values. You swap the partitions where year = 2008 and year = 2009. Both tables have at least two rows to swap.

1. Create the customer_info table:

```
=> CREATE TABLE customer_info (
    customer_id INT NOT NULL,
    first_name VARCHAR(25),
    last_name VARCHAR(35),
    city VARCHAR(25),
    year INT NOT NULL)
    ORDER BY last_name
    PARTITION BY year;
```

2. Insert data into the customer_info table:

```
INSERT INTO customer_info VALUES
 (1,'Joe','Smith','Denver',2008),
 (2,'Bob','Jones','Boston',2008),
 (3,'Silke','Muller','Frankfurt',2007),
 (4,'Simone','Bernard','Paris',2014),
 (5,'Vijay','Kumar','New Delhi',2010);
 OUTPUT
--------
     5
(1 row)

=> COMMIT;
```

3. View the table data:

```
=> SELECT * FROM customer_info ORDER BY year DESC;
 customer_id | first_name | last_name |   city    | year
-------------+------------+-----------+-----------+------
           4 | Simone     | Bernard   | Paris     | 2014
           5 | Vijay      | Kumar     | New Delhi | 2010
           2 | Bob        | Jones     | Boston    | 2008
           1 | Joe        | Smith     | Denver    | 2008
           3 | Silke      | Muller    | Frankfurt | 2007
(5 rows)
```

4. Create a second table, member_info , that has the same definition as customer_info :

```
=> CREATE TABLE member_info LIKE customer_info INCLUDING PROJECTIONS;
CREATE TABLE
```

5. Insert data into the member_info table:

```
=> INSERT INTO member_info VALUES
(1,'Jane','Doe','Miami',2001),
(2,'Mike','Brown','Chicago',2014),
(3,'Patrick','OMalley','Dublin',2008),
(4,'Ana','Lopez','Madrid',2009),
(5,'Mike','Green','New York',2008);
 OUTPUT
--------
     5
(1 row)
=> COMMIT;
COMMIT
```

6. View the data in the member_info table:

```
=> SELECT * FROM member_info ORDER BY year DESC;
 customer_id | first_name | last_name |   city   | year
-------------+------------+-----------+----------+------
           2 | Mike       | Brown     | Chicago  | 2014
           4 | Ana        | Lopez     | Madrid   | 2009
           5 | Mike       | Green     | New York | 2008
           3 | Patrick    | OMalley   | Dublin   | 2008
           1 | Jane       | Doe       | Miami    | 2001
(5 rows)
```

7. To swap the partitions, run the SWAP_PARTITIONS_BETWEEN_TABLES function:

```
=> SELECT SWAP_PARTITIONS_BETWEEN_TABLES('customer_info', 2008, 2009, 'member_info');
                              SWAP_PARTITIONS_BETWEEN_TABLES
-----------------------------------------------------------------------------------------------
 1 partition values from table customer_info and 2 partition values from table member_info are swapped at epoch 1045.

(1 row)
```

8. Query both tables to confirm that they swapped their respective 2008 and 2009 records:

```
=> SELECT * FROM customer_info ORDER BY year DESC;
 customer_id | first_name | last_name |   city    | year
-------------+------------+-----------+-----------+------
           4 | Simone     | Bernard   | Paris     | 2014
           5 | Vijay      | Kumar     | New Delhi | 2010
           4 | Ana        | Lopez     | Madrid    | 2009
           3 | Patrick    | OMalley   | Dublin    | 2008
           5 | Mike       | Green     | New York  | 2008
           3 | Silke      | Muller    | Frankfurt | 2007
(6 rows)
=> SELECT * FROM member_info ORDER BY year DESC;
 customer_id | first_name | last_name |  city   | year
-------------+------------+-----------+---------+------
           2 | Mike       | Brown     | Chicago | 2014
           2 | Bob        | Jones     | Boston  | 2008
           1 | Joe        | Smith     | Denver  | 2008
           1 | Jane       | Doe       | Miami   | 2001
(4 rows)
```

Minimizing partitions

By default, Vertica supports up to 1024 ROS containers to store partitions for a given projection (see Projection parameters). A ROS container contains data that share the same partition key, or the same partition group key. Depending on the amount of data per partition, a partition or partition group can span multiple ROS containers.

Given this limit, it is inadvisable to partition a table on highly granular data—for example, on a TIMESTAMP column. Doing so can generate a very high number of partitions. If the number of partitions requires more than 1024 ROS containers, Vertica issues a ROS pushback warning and refuses to load more table data. A large number of ROS containers also can adversely affect DML operations such as DELETE , which requires Vertica to open all ROS containers.

In practice, it is unlikely you will approach this maximum. For optimal performance, Vertica recommends that the number of ungrouped partitions range between 10 and 20, and not exceed 50. This range is typically compatible with most business requirements.

You can also reduce the number of ROS containers by grouping partitions. For more information, see Partition grouping and Hierarchical partitioning .

Viewing partition storage data

Vertica provides various ways to view how your table partitions are organized and stored:

- Query the PARTITIONS system table.
- Dump partition keys.

Querying PARTITIONS table

The following table and projection definitions partition store_order data on order dates, and groups together partitions of the same year:

```
=> CREATE TABLE public.store_orders
  (order_no int, order_date timestamp NOT NULL, shipper varchar(20), ship_date date)
  PARTITION BY ((order_date)::date) GROUP BY (date_trunc('year', (order_date)::date));

=> CREATE PROJECTION public.store_orders_super
   AS SELECT order_no, order_date, shipper, ship_date FROM store_orders
   ORDER BY order_no, order_date, shipper, ship_date UNSEGMENTED ALL NODES;

=> COPY store_orders FROM '/home/dbadmin/export_store_orders_data.txt';
```

After loading data into this table, you can query the PARTITIONS table to determine how many ROS containers store the grouped partitions for projection store_orders_unseg , across all nodes. Each node has eight ROS containers, each container storing partitions of one partition group:

```
=> SELECT COUNT (partition_key) NumPartitions, ros_id, node_name FROM PARTITIONS
     WHERE projection_name ilike 'store_orders%' GROUP BY ros_id, node_name ORDER BY node_name, NumPartitions;
 NumPartitions |      ros_id       |    node_name
---------------+-------------------+------------------
           173 | 45035996274562779 | v_vmart_node0001
           211 | 45035996274562791 | v_vmart_node0001
           212 | 45035996274562783 | v_vmart_node0001
           213 | 45035996274562787 | v_vmart_node0001
           173 | 49539595901916471 | v_vmart_node0002
           211 | 49539595901916483 | v_vmart_node0002
           212 | 49539595901916475 | v_vmart_node0002
           213 | 49539595901916479 | v_vmart_node0002
           173 | 54043195529286985 | v_vmart_node0003
           211 | 54043195529286997 | v_vmart_node0003
           212 | 54043195529286989 | v_vmart_node0003
           213 | 54043195529286993 | v_vmart_node0003
(12 rows)
```

Dumping partition keys

Vertica provides several functions that let you inspect how individual partitions are stored on the cluster, at several levels:

- DUMP_PARTITION_KEYS dumps partition keys of all projections in the system.
- DUMP_TABLE_PARTITION_KEYS dumps partition keys of all projections for the specified table.
- DUMP_PROJECTION_PARTITION_KEYS dumps partition keys of the specified projection.

Given the previous table and projection, DUMP_PROJECTION_PARTITION_KEYS shows the contents of four ROS containers on each node:

```
=> SELECT DUMP_PROJECTION_PARTITION_KEYS('store_orders_super');
 ...
 Partition keys on node v_vmart_node0001
  Projection 'store_orders_super'
   Storage [ROS container]
     No of partition keys: 173
     Partition keys: 2017-01-02 2017-01-03 2017-01-04 2017-01-05 2017-01-06 2017-01-09 2017-01-10
2017-01-11 2017-01-12 2017-01-13 2017-01-16 2017-01-17 2017-01-18 2017-01-19 2017-01-20 2017-01-23
2017-01-24 2017-01-25 2017-01-26 2017-01-27 2017-02-01 2017-02-02 2017-02-03 2017-02-06 2017-02-07
2017-02-08 2017-02-09 2017-02-10 2017-02-13 2017-02-14 2017-02-15 2017-02-16 2017-02-17 2017-02-20
...
2017-09-01 2017-09-04 2017-09-05 2017-09-06 2017-09-07 2017-09-08 2017-09-11 2017-09-12 2017-09-13
2017-09-14 2017-09-15 2017-09-18 2017-09-19 2017-09-20 2017-09-21 2017-09-22 2017-09-25 2017-09-26 2017-09-27
   Storage [ROS container]
     No of partition keys: 212
     Partition keys: 2016-01-01 2016-01-04 2016-01-05 2016-01-06 2016-01-07 2016-01-08 2016-01-11
2016-01-12 2016-01-13 2016-01-14 2016-01-15 2016-01-18 2016-01-19 2016-01-20 2016-01-21 2016-01-22
2016-01-25 2016-01-26 2016-01-27 2016-02-01 2016-02-02 2016-02-03 2016-02-04 2016-02-05 2016-02-08
2016-02-09 2016-02-10 2016-02-11 2016-02-12 2016-02-15 2016-02-16 2016-02-17 2016-02-18 2016-02-19
...
2016-11-01 2016-11-02 2016-11-03 2016-11-04 2016-11-07 2016-11-08 2016-11-09 2016-11-10 2016-11-11
2016-11-14 2016-11-15 2016-11-16 2016-11-17 2016-11-18 2016-11-21 2016-11-22 2016-11-23 2016-11-24 2016-11-25
   Storage [ROS container]
     No of partition keys: 213
     Partition keys: 2015-01-01 2015-01-02 2015-01-05 2015-01-06 2015-01-07 2015-01-08 2015-01-09
2015-01-12 2015-01-13 2015-01-14 2015-01-15 2015-01-16 2015-01-19 2015-01-20 2015-01-21 2015-01-22
2015-01-23 2015-01-26 2015-01-27 2015-02-02 2015-02-03 2015-02-04 2015-02-05 2015-02-06 2015-02-09
2015-02-10 2015-02-11 2015-02-12 2015-02-13 2015-02-16 2015-02-17 2015-02-18 2015-02-19 2015-02-20
...
2015-11-02 2015-11-03 2015-11-04 2015-11-05 2015-11-06 2015-11-09 2015-11-10 2015-11-11 2015-11-12
2015-11-13 2015-11-16 2015-11-17 2015-11-18 2015-11-19 2015-11-20 2015-11-23 2015-11-24 2015-11-25
2015-11-26 2015-11-27
   Storage [ROS container]
     No of partition keys: 211
     Partition keys: 2014-01-01 2014-01-02 2014-01-03 2014-01-06 2014-01-07 2014-01-08 2014-01-09
2014-01-10 2014-01-13 2014-01-14 2014-01-15 2014-01-16 2014-01-17 2014-01-20 2014-01-21 2014-01-22
2014-01-23 2014-01-24 2014-01-27 2014-02-03 2014-02-04 2014-02-05 2014-02-06 2014-02-07 2014-02-10
2014-02-11 2014-02-12 2014-02-13 2014-02-14 2014-02-17 2014-02-18 2014-02-19 2014-02-20 2014-02-21
...
2014-11-04 2014-11-05 2014-11-06 2014-11-07 2014-11-10 2014-11-11 2014-11-12 2014-11-13 2014-11-14
2014-11-17 2014-11-18 2014-11-19 2014-11-20 2014-11-21 2014-11-24 2014-11-25 2014-11-26 2014-11-27
   Storage [ROS container]
     No of partition keys: 173
...
```

Active and inactive partitions

The Tuple Mover assumes that all loads and updates to a partitioned table are targeted to one or more partitions that it identifies as *active* . In general, the partitions with the largest partition keys—typically, the most recently created partitions—are regarded as active. As the partition ages, its workload typically shrinks and becomes mostly read-only.

Setting active partition count

You can specify how many partitions are active for partitioned tables at two levels, in ascending order of precedence:

- Configuration parameter ActivePartitionCount determines how many partitions are active for partitioned tables in the database. By default, ActivePartitionCount is set to 1. The Tuple Mover applies this setting to all tables that do not set their own active partition count.
- Individual tables can supersede ActivePartitionCount by setting their own active partition count with CREATE TABLE and ALTER TABLE .

Partitioned tables in the same database can be subject to different distributions of update and load activity. When these differences are significant, it might make sense for some tables to set their own active partition counts.

For example, table store_orders is partitioned by month and gets its active partition count from configuration parameter ActivePartitionCount . If the parameter is set to 1, the Tuple Mover identifes the latest month—typically, the current one—as the table's active partition. If store_orders is subject to frequent activity on data for the current month and the one before it, you might want the table to supersede the configuration parameter, and set its active partition count to 2:

```
ALTER TABLE public.store_orders SET ACTIVEPARTITIONCOUNT 2;
```

> **Note**
> For tables partitioned by non-temporal attributes, set its active partition count to reflect the number of partitions that are subject to a high level of activity—for example, frequent loads or queries.

Identifying the active partition

The Tuple Mover typically identifies the active partition as the one most recently created. Vertica uses the following algorithm to determine which partitions are older than others:

1. If partition X was created before partition Y, partition X is older.
2. If partitions X and Y were created at the same time, but partition X was last updated before partition Y, partition X is older.
3. If partitions X and Y were created and last updated at the same time, the partition with the smaller key is older.

You can obtain the active partitions for a table by joining system tables PARTITIONS and STRATA and querying on its projections. For example, the following query gets the active partition for projection store_orders_super :

```
=> SELECT p.node_name, p.partition_key, p.ros_id, p.ros_size_bytes, p.ros_row_count, ROS_container_count
    FROM partitions p JOIN strata s ON p.partition_key = s.stratum_key AND p.node_name=s.node_name
    WHERE p.projection_name = 'store_orders_super' ORDER BY p.node_name, p.partition_key;
   node_name     | partition_key |      ros_id       | ros_size_bytes | ros_row_count | ROS_container_count
-----------------+---------------+-------------------+----------------+---------------+---------------------
 v_vmart_node0001 | 2017-09-01    | 45035996279322851 |           6905 |           960 |                   1
 v_vmart_node0002 | 2017-09-01    | 49539595906590663 |           6905 |           960 |                   1
 v_vmart_node0003 | 2017-09-01    | 54043195533961159 |           6905 |           960 |                   1
(3 rows)
```

Active partition groups

If a table's partition clause includes a GROUP BY expression, Vertica applies the table's active partition count to its largest partition group key, and regards all the partitions in that group as active. If you group partitions with Vertica meta-function CALENDAR_HIERARCHY_DAY , the most recent date partitions are also grouped by day. Thus, the largest partition group key and largest partition key are identical. In effect, this means that only the most recent partitions are active.

For more information about partition grouping, see Partition grouping and Hierarchical partitioning .

Partition pruning

If a query predicate specifies a partitioning expression, the query optimizer evaluates the predicate against the ROS containers of the partitioned data. Each ROS container maintains the minimum and maximum values of its partition key data. The query optimizer uses this metadata to determine which ROS containers it needs to execute the query, and omits, or *prunes* , the remaining containers from the query plan. By minimizing the number of ROS containers that it must scan, the query optimizer enables faster execution of the query.

For example, a table might be partitioned by year as follows:

```
=> CREATE TABLE ... PARTITION BY EXTRACT(year FROM date);
```

Given this table definition, its projection data is partitioned into ROS containers according to year, one for each year—in this case, 2007, 2008, 2009.

The following query specifies the partition expression date :

```
=> SELECT ... WHERE date = '12-2-2009';
```

Given this query, the ROS containers that contain data for 2007 and 2008 fall outside the boundaries of the requested year (2009). The query optimizer prunes these containers from the query plan before the query executes:

Examples

Assume a table that is partitioned by time and will use queries that restrict data on time.

```
=> CREATE TABLE time ( tdate DATE NOT NULL, tnum INTEGER)
    PARTITION BY EXTRACT(year FROM tdate);
=> CREATE PROJECTION time_p (tdate, tnum) AS
=> SELECT * FROM time ORDER BY tdate, tnum UNSEGMENTED ALL NODES;
```

> **Note**
> Projection sort order has no effect on partition pruning.

```
=> INSERT INTO time VALUES ('03/15/04' , 1);
=> INSERT INTO time VALUES ('03/15/05' , 2);
=> INSERT INTO time VALUES ('03/15/06' , 3);
=> INSERT INTO time VALUES ('03/15/06' , 4);
```

The data inserted in the previous series of commands are loaded into three ROS containers, one per year, as that is how the data is partitioned:

```
=> SELECT * FROM time ORDER BY tnum;
  tdate    | tnum
------------+------
 2004-03-15 |    1  --ROS1 (min 03/01/04, max 03/15/04)
 2005-03-15 |    2  --ROS2 (min 03/15/05, max 03/15/05)
 2006-03-15 |    3  --ROS3 (min 03/15/06, max 03/15/06)
 2006-03-15 |    4  --ROS3 (min 03/15/06, max 03/15/06)
(4 rows)
```

Here's what happens when you query the time table:

- In this query, Vertica can omit container ROS2 because it is only looking for year 2004:
  ```
  => SELECT COUNT(*) FROM time WHERE tdate = '05/07/2004';
  ```
- In the next query, Vertica can omit two containers, ROS1 and ROS3:
  ```
  => SELECT COUNT(*) FROM time WHERE tdate = '10/07/2005';
  ```
- The following query has an additional predicate on the tnum column for which no minimum/maximum values are maintained. In addition, the use of logical operator OR is not supported, so no ROS elimination occurs:
  ```
  => SELECT COUNT(*) FROM time WHERE tdate = '05/07/2004' OR tnum = 7;
  ```

Constraints

Constraints set rules on what data is allowed in table columns. Using constraints can help maintain data integrity. For example, you can constrain a column to allow only unique values, or to disallow NULL values. Constraints such as primary keys also help the optimizer generate query plans that facilitate faster data access, particularly for joins.

You set constraints on a new table and an existing one with CREATE TABLE and ALTER TABLE...ADD CONSTRAINT, respectively.

To view current constraints, see Column-constraint.

In this section

- Supported constraints
- Setting constraints
- Dropping constraints
- Naming constraints
- Detecting constraint violations
- Constraint enforcement

Supported constraints

Vertica supports standard SQL constraints, as described in this section.

In this section

- Primary key constraints

- Foreign key constraints
- Unique constraints
- Check constraints
- NOT NULL constraints

Primary key constraints

A primary key comprises one or multiple columns of primitive types, whose values can uniquely identify table rows. A table can specify only one primary key. You identify a table's primary key when you create the table, or in an existing table with ALTER TABLE . You cannot designate a column with a collection type as a key.

For example, the following CREATE TABLE statement defines the order_no column as the primary key of the store_orders table:

```
=> CREATE TABLE public.store_orders(
    order_no int PRIMARY KEY,
    order_date timestamp NOT NULL,
    shipper varchar(20),
    ship_date date,
    product_key int,
    product_version int
)
PARTITION BY ((date_part('year', order_date))::int);
CREATE TABLE
```

Multi-column primary keys

A primary key can comprise multiple columns. In this case, the CREATE TABLE statement must specify the constraint after all columns are defined, as follows:

```
=> CREATE TABLE public.product_dimension(
    product_key int,
    product_version int,
    product_description varchar(128),
    sku_number char(32) UNIQUE,
    category_description char(32),
    CONSTRAINT pk PRIMARY KEY (product_key, product_version) ENABLED
);
CREATE TABLE
```

Alternatively, you can specify the table's primary key with a separate ALTER TABLE...ADD CONSTRAINT statement, as follows:

```
=> ALTER TABLE product_dimension ADD CONSTRAINT pk PRIMARY KEY (product_key, product_version) ENABLED;
ALTER TABLE
```

Enforcing primary keys

You can prevent loading duplicate values into primary keys by enforcing the primary key constraint. Doing so allows you to join tables on their primary and foreign keys . When a query joins a dimension table to a fact table, each primary key in the dimension table must uniquely match each foreign key value in the fact table. Otherwise, attempts to join these tables return a key enforcement error.

You enforce primary key constraints globally with configuration parameter EnableNewPrimaryKeysByDefault . You can also enforce primary key constraints for specific tables by qualifying the constraint with the keyword ENABLED . In both cases, Vertica checks key values as they are loaded into tables, and returns errors on any constraint violations. Alternatively, use ANALYZE_CONSTRAINTS to validate primary keys after updating table contents. For details, see Constraint enforcement .

> **Tip**
> Consider using sequences for primary key columns to guarantee uniqueness, and avoid the resource overhead that primary key constraints can incur.

Setting NOT NULL on primary keys

When you define a primary key , Vertica automatically sets the primary key columns to NOT NULL . For example, when you create the table product_dimension as shown earlier , Vertica sets primary key columns product_key and product_version to NOT NULL , and stores them in the catalog accordingly:

```
> SELECT EXPORT_TABLES('','product_dimension');
...
CREATE TABLE public.product_dimension
(
    product_key int NOT NULL,
    product_version int NOT NULL,
    product_description varchar(128),
    sku_number char(32),
    category_description char(32),
    CONSTRAINT C_UNIQUE UNIQUE (sku_number) DISABLED,
    CONSTRAINT pk PRIMARY KEY (product_key, product_version) ENABLED
);

(1 row)
```

If you specify a primary key for an existing table with ALTER TABLE , Vertica notifies you that it set the primary key columns to NOT NULL :

```
WARNING 2623: Column "column-name" definition changed to NOT NULL
```

> **Note**
> If you drop a primary key constraint, the columns that comprised it remain set to NOT NULL . This constraint can only be removed explicitly, through ALTER TABLE...ALTER COLUMN .

Foreign key constraints

A foreign key joins a table to another table by referencing its primary key. A foreign key constraint specifies that the key can only contain values that are in the referenced primary key, and thus ensures the referential integrity of data that is joined on the two keys.

You can identify a table's foreign key when you create the table, or in an existing table with ALTER TABLE . For example, the following CREATE TABLE statement defines two foreign key constraints: fk_store_orders_store and fk_store_orders_vendor :

```
=> CREATE TABLE store.store_orders_fact(
    product_key int NOT NULL,
    product_version int NOT NULL,
    store_key int NOT NULL CONSTRAINT fk_store_orders_store REFERENCES store.store_dimension (store_key),
    vendor_key int NOT NULL CONSTRAINT fk_store_orders_vendor REFERENCES public.vendor_dimension (vendor_key),
    employee_key int NOT NULL,
    order_number int NOT NULL,
    date_ordered date,
    date_shipped date,
    expected_delivery_date date,
    date_delivered date,
    quantity_ordered int,
    quantity_delivered int,
    shipper_name varchar(32),
    unit_price int,
    shipping_cost int,
    total_order_cost int,
    quantity_in_stock int,
    reorder_level int,
    overstock_ceiling int
);
```

The following ALTER TABLE statement adds foreign key constraint fk_store_orders_employee to the same table:

```
=> ALTER TABLE store.store_orders_fact ADD CONSTRAINT fk_store_orders_employee
     FOREIGN KEY (employee_key) REFERENCES public.employee_dimension (employee_key);
```

The REFERENCES clause can omit the name of the referenced column if it is the same as the foreign key column name. For example, the following ALTER TABLE statement is equivalent to the one above:

```
=> ALTER TABLE store.store_orders_fact ADD CONSTRAINT fk_store_orders_employee
     FOREIGN KEY (employee_key) REFERENCES public.employee_dimension;
```

Multi-column foreign keys

If a foreign key refrences a primary key that contains multiple columns, the foreign key must contain the same number of columns. For example, the primary key for table public.product_dimension contains two columns, product_key and product_version . In this case, CREATE TABLE can define a foreign key constraint that references this primary key as follows:

```
=> CREATE TABLE store.store_orders_fact3(
    product_key int NOT NULL,
    product_version int NOT NULL,
    ...
   CONSTRAINT fk_store_orders_product
     FOREIGN KEY (product_key, product_version) REFERENCES public.product_dimension (product_key, product_version)
);

CREATE TABLE
```

CREATE TABLE can specify multi-column foreign keys only after all table columns are defined. You can also specify the table's foreign key with a separate ALTER TABLE...ADD CONSTRAINT statement:

```
=> ALTER TABLE store.store_orders_fact ADD CONSTRAINT fk_store_orders_product
    FOREIGN KEY (product_key, product_version) REFERENCES public.product_dimension (product_key, product_version);
```

In both examples, the constraint specifies the columns in the referenced table. If the referenced column names are the same as the foreign key column names, the REFERENCES clause can omit them. For example, the following ALTER TABLE statement is equivalent to the previous one:

```
=> ALTER TABLE store.store_orders_fact ADD CONSTRAINT fk_store_orders_product
    FOREIGN KEY (product_key, product_version) REFERENCES public.product_dimension;
```

NULL values in foreign key

A foreign key that whose columns omit NOT NULL can contain NULL values, even if the primary key contains no NULL values. Thus, you can insert rows into the table even if their foreign key is not yet known.

Unique constraints

You can specify a unique constraint on a column so each value in that column is unique among all other values. You can define a unique constraint when you create a table, or you can add a unique constraint to an existing table with ALTER TABLE . You cannot use a uniqueness constraint on a column with a collection type.

For example, the following ALTER TABLE statement defines the sku_number column in the product_dimensions table as unique:

```
=> ALTER TABLE public.product_dimension ADD UNIQUE(sku_number);
WARNING 4887:  Table product_dimension has data. Queries using this table may give wrong results
if the data does not satisfy this constraint
HINT:  Use analyze_constraints() to check constraint violation on data
```

Enforcing unique constraints

You enforce unique constraints globally with configuration parameter EnableNewUniqueKeysByDefault . You can also enforce unique constraints for specific tables by qualifying their unique constraints with the keyword ENABLED . In both cases, Vertica checks values as they are loaded into unique columns, and returns with errors on any constraint violations. Alternatively, you can use ANALYZE_CONSTRAINTS to validate unique constraints after updating table contents. For details, see Constraint enforcement .

For example, the previous example does not enforce the unique constraint in column sku_number . The following statement enables this constraint:

```
=> ALTER TABLE public.product_dimension ALTER CONSTRAINT C_UNIQUE ENABLED;
ALTER TABLE
```

Multi-column unique constraints

You can define a unique constraint that is comprised of multiple columns. The following CREATE TABLE statement specifies that the combined values of columns c1 and c2 in each row must be unique among all other rows:

```
CREATE TABLE dim1 (c1 INTEGER,
   c2 INTEGER,
   c3 INTEGER,
  UNIQUE (c1, c2) ENABLED
);
```

Check constraints

A *check constraint* specifies a Boolean expression that evaluates a column's value on each row. If the expression resolves to false for a given row, the column value is regarded as violating the constraint.

For example, the following table specifies two named check constraints:

- IsYear2018 specifies to allow only 2018 dates in column order_date .
- Ship5dAfterOrder specifies to check whether each ship_date value is no more than 5 days after order_date .

```
CREATE TABLE public.store_orders_2018 (
   order_no int CONSTRAINT pk PRIMARY KEY,
   product_key int,
   product_version int,
   order_date timestamp NOT NULL,
   shipper varchar(20),
   ship_date date,
   CONSTRAINT IsYear2018 CHECK (DATE_PART('year', order_date)::int = 2018),
   CONSTRAINT Ship5dAfterOrder CHECK (DAYOFYEAR(ship_date) - DAYOFYEAR(order_date) <=5)
);
```

When Vertica checks the data in store_orders_2018 for constraint violations, it evaluates the values of order_date and ship_date on each row, to determine whether they comply with their respective check constraints.

Check expressions

A check expression can only reference the current table row; it cannot access data stored in other tables or database objects, such as sequences. It also cannot access data in other table rows.

A check constraint expression can include:

- Arithmetic and concatenated string operators
- Logical operators such as AND , OR , NOT
- WHERE prepositions such as CASE , IN , LIKE , BETWEEN , IS [NOT] NULL
- Calls to the following function types:
  - Immutable built-in SQL functions such as LENGTH
  - Immutable SQL macros (see Check Constraints and SQL Macros below)
  - User-defined scalar functions that are marked as immutable in the component factory (see Check Constraints and UDSFs below)

Example check expressions

The following check expressions assume that the table contains all referenced columns and that they have the appropriate data types:

- CONSTRAINT chk_pos_quant CHECK (quantity > 0)
- CONSTRAINT chk_pqe CHECK (price*quantity = extended_price)
- CONSTRAINT size_sml CHECK (size in ('s', 'm', 'l', 'xl'))
- CHECK (regexp_like(dept_name, '^[a-z]+$', 'i') OR (dept_name = 'inside sales'))

Check expression restrictions

A check expression must evaluate to a Boolean value. However, Vertica does not support implicit conversions to Boolean values. For example, the following check expressions are invalid:

- CHECK (1)
- CHECK ('hello')

A check expression cannot include the following elements:

- Subqueries—for example, CHECK (dept_id in (SELECT id FROM dept))
- Aggregates—for example, CHECK (quantity < sum(quantity)/2)
- Window functions—for example, CHECK (RANK() over () < 3)
- SQL meta-functions—for example, CHECK (START_REFRESH('') = 0)
- References to the epoch column
- References to other tables or objects (for example, sequences), or system context
- Invocation of functions that are not immutable in time and space

Enforcing check constraints

You can enforce check constraints globally with configuration parameter EnableNewCheckConstraintsByDefault . You an also enforce check constraints for specific tables by qualifying unique constraints with the keyword ENABLED . In both cases, Vertica evaluates check constraints as new values are loaded into the table, and returns with errors on any constraint violations. Alternatively, you can use ANALYZE_CONSTRAINTS to validate check constraints after updating the table contents. For details, see Constraint enforcement .

For example, you can enable the constraints shown earlier with ALTER TABLE...ALTER CONSTRAINT :

```
=> ALTER TABLE store_orders_2018 ALTER CONSTRAINT IsYear2018 ENABLED;
ALTER TABLE
=> ALTER TABLE store_orders_2018 ALTER CONSTRAINT Ship5dAfterOrder ENABLED;
ALTER TABLE
```

Check constraints and nulls

If a check expression evaluates to unknown for a given row because a column within the expression contains a null, the row passes the constraint condition. Vertica evaluates the expression and considers it satisfied if it resolves to either true or unknown. For example, check (quantity > 0) passes validation if quantity is null. This result differs from how a WHERE clause works. With a WHERE clause, the row would not be included in the result set.

You can prohibit nulls in a check constraint by explicitly including a null check in the check constraint expression. For example: CHECK (quantity IS NOT NULL AND (quantity > 0))

> **Tip**
> Alternatively, set a NOT NULL constraint on the same column.

Check constraints and SQL macros

A check constraint can call a SQL macro (a function written in SQL) if the macro is immutable. An immutable macro always returns the same value for a given set of arguments.

When a DDL statement specifies a macro in a check expression, Vertica determines if it is immutable. If it is not, Vertica rolls back the statement.

The following example creates the macro mycompute and then uses it in a check expression:

```
=> CREATE OR REPLACE FUNCTION mycompute(j int, name1 varchar)
RETURN int AS BEGIN RETURN (j + length(name1)); END;
=> ALTER TABLE sampletable
ADD CONSTRAINT chk_compute
CHECK(mycompute(weekly_hours, name1))<50);
```

Check constraints and UDSFs

A check constraint can call user-defined scalar functions (UDSFs). The following requirements apply:

- The UDSF must be marked as immutable in the UDx factory.
- The constraint handles null values properly.

> **Caution**
> Vertica evaluates an enabled check constraint on every row that is loaded or updated. Invoking a computationally expensive check constraint on a large table is liable to incur considerable system overhead.

For a usage example, see C++ example: calling a UDSF from a check constraint .

NOT NULL constraints

A NOT NULL constraint specifies that a column cannot contain a null value. All table updates must specify values in columns with this constraint. You can set a NOT NULL constraint on columns when you create a table, or set the constraint on an existing table with ALTER TABLE .

The following CREATE TABLE statement defines three columns as NOT NULL. You cannot store any NULL values in those columns.

```
=> CREATE TABLE inventory ( date_key INTEGER NOT NULL,
                            product_key INTEGER NOT NULL,
                            warehouse_key INTEGER NOT NULL, ... );
```

The following ALTER TABLE statement defines column sku_number in table product_dimensions as NOT NULL:

```
=> ALTER TABLE public.product_dimension ALTER COLUMN sku_number SET NOT NULL;
ALTER TABLE
```

Enforcing NOT NULL constraints

You cannot enable enforcement of a NOT NULL constraint. You must use ANALYZE_CONSTRAINTS to determine whether column data contains null values, and then manually fix any constraint violations that the function finds.

NOT NULL and primary keys

When you define a primary key, Vertica automatically sets the primary key columns to NOT NULL. If you drop a primary key constraint, the columns that comprised it remain set to NOT NULL. This constraint can only be removed explicitly, through ALTER TABLE...ALTER COLUMN .

Setting constraints

You can set constraints on a new table and an existing one with CREATE TABLE and ALTER TABLE...ADD CONSTRAINT , respectively.

Setting constraints on a new table

CREATE TABLE can specify a constraint in two ways: as part of the column definition, or following all column definitions.

For example, the following CREATE TABLE statement sets two constraints on column sku_number , NOT NULL and UNIQUE . After all columns are defined, the statement also sets a primary key that is composed of two columns, product_key and product_version :

```
=> CREATE TABLE public.prod_dimension(
   product_key int,
   product_version int,
   product_description varchar(128),
   sku_number char(32) NOT NULL UNIQUE,
   category_description char(32),
   CONSTRAINT pk PRIMARY KEY (product_key, product_version) ENABLED
);
CREATE TABLE
```

Setting constraints on an existing table

ALTER TABLE...ADD CONSTRAINT adds a constraint to an existing table. For example, the following statement specifies unique values for column product_version :

```
=> ALTER TABLE prod_dimension ADD CONSTRAINT u_product_versions UNIQUE (product_version) ENABLED;
ALTER TABLE
```

Validating existing data

When you add a constraint on a column that already contains data, Vertica immediately validates column values if the following conditions are both true:

- The constraint is a primary key , unique , or check constraint.
- The constraint is enforced .

If either of these conditions is not true, Vertica does not validate the column values. In this case, you must call ANALYZE_CONSTRAINTS to find constraint violations. Otherwise, queries are liable to return unexpected results. For details, see Detecting constraint violations .

Exporting table constraints

Whether you specify constraints in the column definition or on the table, Vertica stores the table DDL as part of the CREATE statement and exports them as such. One exception applies: foreign keys are stored and exported as ALTER TABLE statements.

For example:

```
=> SELECT EXPORT_TABLES('','prod_dimension');
...
CREATE TABLE public.prod_dimension
(
    product_key int NOT NULL,
    product_version int NOT NULL,
    product_description varchar(128),
    sku_number char(32) NOT NULL,
    category_description char(32),
    CONSTRAINT C_UNIQUE UNIQUE (sku_number) DISABLED,
    CONSTRAINT pk PRIMARY KEY (product_key, product_version) ENABLED,
    CONSTRAINT u_product_versions UNIQUE (product_version) ENABLED
);
(1 row)
```

Dropping constraints

ALTER TABLE drops constraints from tables in two ways:

- ALTER TABLE...DROP CONSTRAINT removes a named table constraint.
- ALTER TABLE...ALTER COLUMN removes a column's NOT NULL constraint.

For example, table store_orders_2018 specifies the following constraints:

- Named constraint pk identifies column order_no as a primary key.
- Named constraint IsYear2018 specifies a check constraint that allows only 2018 dates in column order_date .
- Named constraint Ship5dAfterOrder specifies a check constraint that disallows any ship_date value that is more than 5 days after order_date .
- Columns order_no and order_date are set to NOT NULL .

```
CREATE TABLE public.store_orders_2018 (
    order_no int NOT NULL CONSTRAINT pk PRIMARY KEY,
    product_key int,
    product_version int,
    order_date timestamp NOT NULL,
    shipper varchar(20),
    ship_date date,
    CONSTRAINT IsYear2018 CHECK (DATE_PART('year', order_date)::int = 2018),
    CONSTRAINT Ship5dAfterOrder CHECK (DAYOFYEAR(ship_date) - DAYOFYEAR(order_date) <=5)
);
```

Dropping named constraints

You remove primary, foreign key, check, and unique constraints with ALTER TABLE...DROP CONSTRAINT , which requires you to supply their names. For example, you remove the primary key constraint in table store_orders_2018 as follows:

```
=> ALTER TABLE store_orders_2018 DROP CONSTRAINT pk;
ALTER TABLE
=> SELECT export_tables('','store_orders_2018');
                                                export_tables
-------------------------------------------------------------------------------------------------------------------------------------------

CREATE TABLE public.store_orders_2018
(
    order_no int NOT NULL,
    product_key int,
    product_version int,
    order_date timestamp NOT NULL,
    shipper varchar(20),
    ship_date date,
    CONSTRAINT IsYear2018 CHECK (((date_part('year', store_orders_2018.order_date))::int = 2018)) ENABLED,
    CONSTRAINT Ship5dAfterOrder CHECK (((dayofyear(store_orders_2018.ship_date) - dayofyear(store_orders_2018.order_date)) <= 5)) ENABLED
);
```

Important
If you do not explicitly name a constraint, Vertica assigns its own name . You can obtain all constraint names from the Vertica catalog with EXPORT_TABLES , or by querying the following system tables:

- CONSTRAINT_COLUMNS
- TABLE_CONSTRAINTS
- PRIMARY_KEYS

Dropping NOT NULL constraints

You drop a column's NOT NULL constraint with ALTER TABLE...ALTER COLUMN , as in the following example:

```
=> ALTER TABLE store_orders_2018 ALTER COLUMN order_date DROP NOT NULL;
ALTER TABLE
```

Dropping primary keys

You cannot drop a primary key constraint if another table has a foreign key constraint that references the primary key. To drop the primary key, you must first drop all foreign keys that reference it.

Dropping constraint-referenced columns

If you try to drop a column that is referenced by a constraint in the same table, the drop operation returns with an error. For example, check constraint Ship5dAfterOrder references two columns, order_date and ship_date . If you try to drop either column, Vertica returns the following error message:

```
=> ALTER TABLE public.store_orders_2018 DROP COLUMN ship_date;
ROLLBACK 3128:  DROP failed due to dependencies
DETAIL:
Constraint Ship5dAfterOrder references column ship_date
HINT:  Use DROP .. CASCADE to drop or modify the dependent objects
```

In this case, you must qualify the DROP COLUMN clause with the CASCADE option, which specifies to drop the column and its dependent objects—in this case, constraint Ship5dAfterOrder :

```
=> ALTER TABLE public.store_orders_2018 DROP COLUMN ship_date CASCADE;
ALTER TABLE
```

A call to Vertica function EXPORT_TABLES confirms that the column and the constraint were both removed:

```
=> ALTER TABLE public.store_orders_2018 DROP COLUMN ship_date CASCADE;
ALTER TABLE
dbadmin=> SELECT export_tables('','store_orders_2018');
                                        export_tables
-----------------------------------------------------------------------------------------------------------

CREATE TABLE public.store_orders_2018
(
    order_no int NOT NULL,
    product_key int,
    product_version int,
    order_date timestamp,
    shipper varchar(20),
    CONSTRAINT IsYear2018 CHECK (((date_part('year', store_orders_2018.order_date))::int = 2018)) ENABLED
);

(1 row)
```

Naming constraints

The following constraints must be named.

- PRIMARY KEY
- REFERENCES (foreign key)
- CHECK
- UNIQUE

You name these constraints when you define them. If you omit assigning a name, Vertica automatically assigns one.

User-assigned constraint names

You assign names to constraints when you define them with CREATE TABLE or ALTER TABLE...ADD CONSTRAINT . For example, the following CREATE TABLE statement names primary key and check constraints pk and date_c , respectively:

```
=> CREATE TABLE public.store_orders_2016
(
    order_no int CONSTRAINT pk PRIMARY KEY,
    product_key int,
    product_version int,
    order_date timestamp NOT NULL,
    shipper varchar(20),
    ship_date date,
    CONSTRAINT date_c CHECK (date_part('year', order_date)::int = 2016)
)
PARTITION BY ((date_part('year', order_date))::int);
CREATE TABLE
```

The following ALTER TABLE statement adds foreign key constraint fk :

```
=> ALTER TABLE public.store_orders_2016 ADD CONSTRAINT fk
    FOREIGN KEY (product_key, product_version)
    REFERENCES public.product_dimension (product_key, product_version);
```

Auto-assigned constraint names

Naming a constraint is optional. If you omit assigning a name to a constraint, Vertica assigns its own name using the following convention:

```
C_constraint-type[_integer]
```

For example, the following table defines two columns a and b and constrains them to contain unique values:

```
=> CREATE TABLE t1 (a int UNIQUE, b int UNIQUE );
CREATE TABLE
```

When you export the table's DDL with EXPORT_TABLES , the function output shows that Vertica assigned constraint names C_UNIQUE and C_UNIQUE_1 to columns a and b , respectively:

```
=> SELECT EXPORT_TABLES('','t1');
CREATE TABLE public.t1
(
    a int,
    b int,
    CONSTRAINT C_UNIQUE UNIQUE (a) DISABLED,
    CONSTRAINT C_UNIQUE_1 UNIQUE (b) DISABLED
);

(1 row)
```

Viewing constraint names

You can view the names of table constraints by exporting the table's DDL with EXPORT_TABLES , as shown earlier. You can also query the following system tables:

- CONSTRAINT_COLUMNS
- TABLE_CONSTRAINTS
- PRIMARY_KEYS

For example, the following query gets the names of all primary and foreign key constraints in schema online_sales :

```
=> SELECT table_name, constraint_name, column_name, constraint_type FROM constraint_columns
    WHERE constraint_type in ('p','f') AND table_schema='online_sales'
    ORDER BY table_name, constraint_type, constraint_name;
     table_name       |      constraint_name      |   column_name   | constraint_type
----------------------+---------------------------+-----------------+-----------------
 call_center_dimension | C_PRIMARY                 | call_center_key | p
 online_page_dimension | C_PRIMARY                 | online_page_key | p
 online_sales_fact     | fk_online_sales_cc        | call_center_key | f
 online_sales_fact     | fk_online_sales_customer  | customer_key    | f
 online_sales_fact     | fk_online_sales_op        | online_page_key | f
 online_sales_fact     | fk_online_sales_product   | product_version | f
 online_sales_fact     | fk_online_sales_product   | product_key     | f
 online_sales_fact     | fk_online_sales_promotion | promotion_key   | f
 online_sales_fact     | fk_online_sales_saledate  | sale_date_key   | f
 online_sales_fact     | fk_online_sales_shipdate  | ship_date_key   | f
 online_sales_fact     | fk_online_sales_shipping  | shipping_key    | f
 online_sales_fact     | fk_online_sales_warehouse | warehouse_key   | f
(12 rows)
```

Using constraint names

You must reference a constraint name in order to perform the following tasks:

- Enable or disable constraint enforcement.
- Drop a constraint.

For example, the following ALTER TABLE statement enables enforcement of constraint pk in table store_orders_2016 :

```
=> ALTER TABLE public.store_orders_2016 ALTER CONSTRAINT pk ENABLED;
ALTER TABLE
```

The following statement drops another constraint in the same table:

```
=> ALTER TABLE public.store_orders_2016 DROP CONSTRAINT date_c;
ALTER TABLE
```

Detecting constraint violations

ANALYZE_CONSTRAINTS analyzes and reports on table constraint violations within a given schema. You can use ANALYZE_CONSTRAINTS to analyze an individual table, specific columns within a table, or all tables within a schema. You typically use this function on tables where primary key, unique, or check constraints are not enforced. You can also use ANALYZE_CONSTRAINTS to check the referential integrity of foreign keys.

In the simplest use case, ANALYZE_CONSTRAINTS is a two-step process:

1. Run ANALYZE_CONSTRAINTS on the desired table. ANALYZE_CONSTRAINTS reports all rows that violate constraints.
2. Use the report to fix violations.

You can also use ANALYZE_CONSTRAINTS in the following cases:

- Analyze tables with enforced constraints.
- Detect constraint violations introduced by a COPY operation and address them before the copy transaction is committed.

Analyzing tables with enforced constraints

If constraints are enforced on a table and a DML operation returns constraint violations, Vertica reports on a limited number of constraint violations before it rolls back the operation. This can be problematic when you try to load a large amount of data that includes many constraint violations—for example, duplicate key values. In this case, use ANALYZE_CONSTRAINTS as follows:

1. Temporarily disable enforcement of all constraints on the target table.
2. Run the DML operation.
3. After the operation returns, run ANALYZE_CONSTRAINTS on the table. ANALYZE_CONSTRAINTS reports all rows that violate constraints.
4. Use the report to fix the violations.
5. Re-enable constraint enforcement on the table.

Using ANALYZE_CONSTRAINTS in a COPY transaction

Use ANALYZE_CONSTRAINTS to detect and address constraint violations introduced by a COPY operation as follows:

1. Copy the source data into the target table with COPY...NO COMMIT .
2. Call ANALYZE_CONSTRAINTS to check the target table with its uncommitted updates.
3. If ANALYZE_CONSTRAINTS reports constraint violations, roll back the copy transaction.
4. Use the report to fix the violations, and then re-execute the copy operation.

For details about using COPY...NO COMMIT , see Using transactions to stage a load .

Distributing constraint analysis

ANALYZE_CONSTRAINTS runs as an atomic operation—that is, it does not return until it evaluates all constraints within the specified scope. For example, if you run ANALYZE_CONSTRAINTS against a table, the function returns only after it evaluates all column constraints against column data. If the table has a large number of columns with constraints, and contains a very large data set, ANALYZE_CONSTRAINTS is liable to exhaust all available memory and return with an out-of-memory error. This risk is increased by running ANALYZE_CONSTRAINTS against multiple tables simultaneously, or against the entire database.

You can minimize the risk of out-of-memory errors by setting configuration parameter MaxConstraintChecksPerQuery (by default set to -1) to a positive integer. For example, if this parameter is set to 20, and you run ANALYZE_CONSTRAINTS on a table that contains 38 column constraints, the function divides its work into two separate queries. ANALYZE_CONSTRAINTS creates a temporary table for loading and compiling results from the two queries, and then returns the composite result set.

MaxConstraintChecksPerQuery can only be set at the database level, and can incur a certain amount of overhead. When set, commits to the temporary table created by ANALYZE_CONSTRAINTS cause all pending database transactions to auto-commit. Setting this parameter to a reasonable number such as 20 should minimize its performance impact.

Constraint enforcement

You can enforce the following constraints:

- PRIMARY KEY
- UNIQUE
- CHECK

When you enable constraint enforcement on a table, Vertica applies that constraint immediately to the table's current content, and to all content that is added or updated later.

Operations that invoke constraint enforcement

The following DDL and DML operations invoke constraint enforcement:

- ALTER TABLE...ADD CONSTRAINT and ALTER TABLE...ALTER CONSTRAINT
- INSERT and INSERT...SELECT
- COPY
- UPDATE
- MERGE
- Partitioning functions:
  - COPY_PARTITIONS_TO_TABLE
  - MOVE_PARTITIONS_TO_TABLE
  - SWAP_PARTITIONS_BETWEEN_TABLES

Benefits and costs

Enabling constraint enforcement can help minimize post-load maintenance tasks, such as validating data separately with ANALYZE_CONSTRAINTS, and then dealing with the constraint violations that it returns.

Enforcing key constraints, particularly on primary keys, can help the optimizer produce faster query plans, particularly for joins. When a primary key constraint is enforced on a table, the optimizer assumes that no rows in that table contain duplicate key values.

Under certain circumstances, widespread constraint enforcement, especially in large fact tables, can incur significant system overhead. For details, see Constraint enforcement and performance.

In this section

- Levels of constraint enforcement
- Verifying constraint enforcement
- Reporting constraint violations
- Constraint enforcement and locking
- Constraint enforcement and performance
- Projections for enforced constraints
- Constraint enforcement limitations

Levels of constraint enforcement

Constraints can be enforced at two levels:

- Database configuration parameters
- Table constraints

Constraint enforcement parameters

Vertica supports three Boolean parameters to enforce constraints:

Enforcement parameter	Default setting
EnableNewPrimaryKeysByDefault	0 (false/disabled)
EnableNewUniqueKeysByDefault	0 (false/disabled)
EnableNewCheckConstraintsByDefault	1 (true/enabled)

Table constraint enforcement

You set constraint enforcement on tables through CREATE TABLE and ALTER TABLE, by qualifying the constraints with the keywords ENABLED or DISABLED. The following CREATE TABLE statement enables enforcement of a check constraint in its definition of column order_qty:

```
=> CREATE TABLE new_orders (
   cust_id int,
   order_date timestamp DEFAULT CURRENT_TIMESTAMP,
   product_id varchar(12),
   order_qty int CHECK(order_qty > 0) ENABLED,
   PRIMARY KEY(cust_id, order_date) ENABLED
);
CREATE TABLE
```

ALTER TABLE can enable enforcement on existing constraints. The following statement modifies table customer_dimension by enabling enforcement on named constraint C_UNIQUE :

```
=> ALTER TABLE public.customer_dimension ALTER CONSTRAINT C_UNIQUE ENABLED;
ALTER TABLE
```

Enforcement level precedence

Table and column enforcement settings have precedence over enforcement parameter settings. If a table or column constraint omits ENABLED or DISABLED , Vertica uses the current settings of the pertinent configuration parameters.

Important

Changing constraint enforcement parameters has no effect on existing table constraints that omit ENABLED or DISABLED . These table constraints retain the enforcement settings that they previously acquired. You can change the enforcement settings on these constraints only with ALTER TABLE...ALTER CONSTRAINT .

The following CREATE TABLE statement creates table new_sales with columns order_id and order_qty , which are defined with constraints PRIMARY KEY and CHECK , respectively:

```
=> CREATE TABLE new_sales ( order_id int PRIMARY KEY, order_qty int CHECK (order_qty > 0) );
```

Neither constraint is explicitly enabled or disabled, so Vertica uses configuration parameters EnableNewPrimaryKeysByDefault and EnableNewCheckConstraintsByDefault to set enforcement in the table definition:

```
=> SHOW CURRENT EnableNewPrimaryKeysByDefault, EnableNewCheckConstraintsByDefault;
  level  |                name                | setting
---------+------------------------------------+---------
 DEFAULT | EnableNewPrimaryKeysByDefault      | 0
 DEFAULT | EnableNewCheckConstraintsByDefault | 1
(2 rows)

=> SELECT EXPORT_TABLES('','new_sales');
...

CREATE TABLE public.new_sales
(
    order_id int NOT NULL,
    order_qty int,
    CONSTRAINT C_PRIMARY PRIMARY KEY (order_id) DISABLED,
    CONSTRAINT C_CHECK CHECK ((new_sales.order_qty > 0)) ENABLED
);

(1 row)
```

In this case, changing EnableNewPrimaryKeysByDefault to 1 (enabled) has no effect on the C_PRIMARY constraint in table new_sales . You can enforce this constraint with ALTER TABLE...ALTER CONSTRAINT :

```
=> ALTER TABLE public.new_sales ALTER CONSTRAINT C_PRIMARY ENABLED;
ALTER TABLE
```

Verifying constraint enforcement

SHOW CURRENT can return the settings of constraint enforcement parameters:

```
=> SHOW CURRENT EnableNewCheckConstraintsByDefault, EnableNewUniqueKeysByDefault, EnableNewPrimaryKeysByDefault;
  level   |                name                | setting
----------+------------------------------------+---------
 DEFAULT  | EnableNewCheckConstraintsByDefault | 1
 DEFAULT  | EnableNewUniqueKeysByDefault       | 0
 DATABASE | EnableNewPrimaryKeysByDefault      | 1
(3 rows)
```

You can also query the following system tables to check table enforcement settings:

- CONSTRAINT_COLUMNS
- TABLE_CONSTRAINTS
- PRIMARY_KEYS

For example, the following statement queries TABLE_CONSTRAINTS and returns all constraints in database tables. Column is_enabled is set to true or false for all constraints that can be enabled or disabled— PRIMARY KEY , UNIQUE , and CHECK :

```
=> SELECT constraint_name, table_name, constraint_type, is_enabled FROM table_constraints ORDER BY is_enabled, table_name;
     constraint_name      |      table_name       | constraint_type | is_enabled
--------------------------+-----------------------+-----------------+------------
 C_PRIMARY                | call_center_dimension | p               | f
 C_PRIMARY                | date_dimension        | p               | f
 C_PRIMARY                | employee_dimension    | p               | f
 C_PRIMARY                | online_page_dimension | p               | f
 C_PRIMARY                | product_dimension     | p               | f
 C_PRIMARY                | promotion_dimension   | p               | f
 C_PRIMARY                | shipping_dimension    | p               | f
 C_PRIMARY                | store_dimension       | p               | f
 C_UNIQUE_1               | tabletemp             | u               | f
 C_PRIMARY                | vendor_dimension      | p               | f
 C_PRIMARY                | warehouse_dimension   | p               | f
 C_PRIMARY                | customer_dimension    | p               | t
 C_PRIMARY                | new_sales             | p               | t
 C_CHECK                  | new_sales             | c               | t
 fk_inventory_date        | inventory_fact        | f               |
 fk_inventory_product     | inventory_fact        | f               |
 fk_inventory_warehouse   | inventory_fact        | f               |
 ...
```

The following query returns all tables that have primary key, unique, and check constraints, and shows whether the constraints are enabled:

```
=> SELECT table_name, constraint_name, constraint_type, is_enabled FROM constraint_columns
  WHERE constraint_type in ('p', 'u', 'c')
  ORDER BY table_name, constraint_type;
```

```
=> SELECT table_name, constraint_name, constraint_type, is_enabled FROM constraint_columns WHERE constraint_type in ('p', 'u', 'c') ORDER BY
table_name, constraint_type;
     table_name       | constraint_name | constraint_type | is_enabled
----------------------+-----------------+-----------------+------------
 call_center_dimension | C_PRIMARY       | p               | f
 customer_dimension    | C_PRIMARY       | p               | t
 customer_dimension2   | C_PRIMARY       | p               | t
 customer_dimension2   | C_PRIMARY       | p               | t
 date_dimension        | C_PRIMARY       | p               | f
 employee_dimension    | C_PRIMARY       | p               | f
 new_sales             | C_CHECK         | c               | t
 new_sales             | C_PRIMARY       | p               | t
 ...
```

Reporting constraint violations

Vertica reports constraint violations in two cases:

- ALTER TABLE tries to enable constraint enforcement on a table that already contains data, and the data does not comply with the constraint.
- A DML operation tries to add or update data on a table with enforced constraints, and the new data does not comply with one or more constraints.

DDL constraint violations

When you enable constraint enforcement on an existing table with ALTER TABLE...ADD CONSTRAINT or ALTER TABLE...ALTER CONSTRAINT , Vertica applies that constraint immediately to the table's current content. If Vertica detects constraint violations, Vertica returns with an error that reports on violations and then rolls back the ALTER TABLE statement.

For example:

```
=> ALTER TABLE public.customer_dimension ADD CONSTRAINT unique_cust_types UNIQUE (customer_type) ENABLED;
ERROR 6745:  Duplicate key values: 'customer_type=Company'
-- violates constraint 'public.customer_dimension.unique_cust_types'
DETAIL:  Additional violations:
Constraint 'public.customer_dimension.unique_cust_types':
duplicate key values: 'customer_type=Individual'
```

DML constraint violations

When you invoke DML operations that add or update data on a table with enforced constraints, Vertica checks that the new data complies with these constraints. If Vertica detects constraint violations, the operation returns with an error that reports on violations, and then rolls back.

For example, table store_orders and store_orders_2015 are defined with the same primary key and check constraints. Both tables enable enforcement of the primary key constraint; only store_orders_2015 enforces the check constraint:

```
CREATE TABLE public.store_orders
(
    order_no int NOT NULL,
    order_date timestamp NOT NULL,
    shipper varchar(20),
    ship_date date
)
PARTITION BY ((date_part('year', store_orders.order_date))::int);

ALTER TABLE public.store_orders ADD CONSTRAINT C_PRIMARY PRIMARY KEY (order_no) ENABLED;
ALTER TABLE public.store_orders ADD CONSTRAINT C_CHECK CHECK (((date_part('year', store_orders.order_date))::int = 2014)) DISABLED;

CREATE TABLE public.store_orders_2015
(
    order_no int NOT NULL,
    order_date timestamp NOT NULL,
    shipper varchar(20),
    ship_date date
)
PARTITION BY ((date_part('year', store_orders_2015.order_date))::int);

ALTER TABLE public.store_orders_2015 ADD CONSTRAINT C_PRIMARY PRIMARY KEY (order_no) ENABLED;
ALTER TABLE public.store_orders_2015 ADD CONSTRAINT C_CHECK CHECK (((date_part('year', store_orders_2015.order_date))::int = 2015))
ENABLED;
```

If you try to insert data with duplicate key values into store_orders , the insert operation returns with an error message. The message contains detailed information about the first violation. It also returns abbreviated information about subsequent violations, up to the first 30. If necessary , the error message also includes a note that more than 30 violations occurred:

```
=> INSERT INTO store_orders SELECT order_number, date_ordered, shipper_name, date_shipped FROM store.store_orders_fact;
ERROR 6745:  Duplicate key values: 'order_no=10' -- violates constraint 'public.store_orders.C_PRIMARY'
DETAIL:  Additional violations:
Constraint 'public.store_orders.C_PRIMARY':
duplicate key values:
'order_no=11'; 'order_no=12'; 'order_no=13'; 'order_no=14'; 'order_no=15'; 'order_no=17';
'order_no=21'; 'order_no=23'; 'order_no=26'; 'order_no=27'; 'order_no=29'; 'order_no=33';
'order_no=35'; 'order_no=38'; 'order_no=39'; 'order_no=4'; 'order_no=41'; 'order_no=46';
'order_no=49'; 'order_no=6'; 'order_no=62'; 'order_no=67'; 'order_no=68'; 'order_no=70';
'order_no=72'; 'order_no=75'; 'order_no=76'; 'order_no=77'; 'order_no=79';
Note: there were additional errors
```

Similarly, the following attempt to copy data from store_orders into store_orders_2015 violates the table's check constraint. It returns with an error message like the one shown earlier:

```
=> SELECT COPY_TABLE('store_orders', 'store_orders_2015');
NOTICE 7636:  Validating enabled constraints on table 'public.store_orders_2015'...
ERROR 7231:  Check constraint 'public.store_orders_2015.C_CHECK' ((date_part('year', store_orders_2015.order_date))::int = 2015)
violation in table 'public.store_orders_2015': 'order_no=101,order_date=2007-05-02 00:00:00'
DETAIL:  Additional violations:
Check constraint 'public.store_orders_2015.C_CHECK':violations:
'order_no=106,order_date=2016-07-01 00:00:00'; 'order_no=119,order_date=2016-01-04 00:00:00';
'order_no=14,order_date=2016-07-01 00:00:00'; 'order_no=154,order_date=2016-11-06 00:00:00';
'order_no=156,order_date=2016-04-10 00:00:00'; 'order_no=171,order_date=2016-10-08 00:00:00';
'order_no=203,order_date=2016-03-01 00:00:00'; 'order_no=204,order_date=2016-06-09 00:00:00';
'order_no=209,order_date=2016-09-07 00:00:00'; 'order_no=214,order_date=2016-11-02 00:00:00';
'order_no=223,order_date=2016-12-08 00:00:00'; 'order_no=227,order_date=2016-08-02 00:00:00';
'order_no=240,order_date=2016-03-09 00:00:00'; 'order_no=262,order_date=2016-02-09 00:00:00';
'order_no=280,order_date=2016-10-10 00:00:00';
Note: there were additional errors
```

Partition management functions that add or update table content must also respect enforced constraints in the target table. For example, the following MOVE_PARTITIONS_TO_TABLE operation attempts to move a partition from store_orders into store_orders_2015 . However, the source partition includes data that violates the target table's check constraint. Thus, the function returns with results that indicate it failed to move any data:

```
=> SELECT MOVE_PARTITIONS_TO_TABLE ('store_orders','2014','2014','store_orders_2015');
NOTICE 7636:  Validating enabled constraints on table 'public.store_orders_2015'...
            MOVE_PARTITIONS_TO_TABLE
-----------------------------------------------------
 0 distinct partition values moved at epoch 204.
```

Constraint enforcement and locking

Vertica uses an insert/validate (IV) lock for DML operations that require validation for enabled primary key and unique constraints.

When you run these operations on tables that enforce primary or unique key constraints, Vertica sets locks on the tables as follows:

1. Sets an I (insert) lock in order to load data. Multiple sessions can acquire an I lock on the same table simultaneously, and load data concurrently.
2. Sets an IV lock on the table to validate the loaded data against table primary and unique constraints. Only one session at a time can acquire an IV lock on a given table. Other sessions that need to access this table are blocked until the IV lock is released. A session retains its IV lock until one of two events occur:
   - Validation is complete and the DML operation is committed.
   - A constraint violation is detected and the operation is rolled back.

   In either case, Vertica releases the IV lock.

IV lock blocking

While Vertica validates a table's primary or unique key constraints, it temporarily blocks other DML operations on the table. These delays can be especially noticeable when multiple sessions concurrently try to perform extensive changes to data on the same table.

For example, each of three concurrent sessions attempts to load data into table t1 , as follows:

1. All three session acquire an I lock on t1 and begin to load data into the table.
2. Session 2 acquires an exclusive IV lock on t1 to validate table constraints on the data that it loaded. Only one session at a time can acquire an IV lock on a table, so sessions 1 and 3 must wait for session 2 to complete validation before they can begin their own validation.
3. Session 2 successfully validates all data that it loaded into t1 . On committing its load transaction, it releases its IV lock on the table.
4. Session 1 acquires an IV lock on t1 and begins to validate the data that it loaded. In this case, Vertica detects a constraint violation and rolls back the load transaction. Session 1 releases its IV lock on t1 .
5. Session 3 now acquires an IV lock on t1 and begins to validate the data that it loaded. On completing validation, session 3 commits its load transaction and releases the IV lock on t1 . The table is now available for other DML operations.

See also

For information on lock modes and compatibility and conversion matrices, see Lock modes . See also LOCKS and LOCK_USAGE .

Constraint enforcement and performance

In some cases, constraint enforcement can significantly affect overall system performance. This is especially true when constraints are enforced on large fact tables that are subject to frequent and concurrent bulk updates. Every update operation that invokes constraint enforcement requires

Vertica to check each table row for all constraint violations. Thus, enforcing multiple constraints on a table with a large amount of data can cause noticeable delays.

To minimize the overhead incurred by enforcing constraints, omit constraint enforcement on large, often updated tables. You can evaluate these tables for constraint violations by running ANALYZE_CONSTRAINTS during off-peak hours.

Several aspects of constraint enforcement have specific impact on system performance. These include:

- Table locking
- Projections for enforced constraints
- Constraint-triggered rollback within transactions

Table locking

If a table enforces constraints, Vertica sets an insert/validate (IV) lock on that table during a DML operation while it undergoes validation. Only one session at a time can acquire an IV lock on that table. As long as the session retains this lock, no other session can access the table. Lengthy loads is liable to cause performance bottlenecks, especially if multiple sessions try to load the same table simultaneously. For details, see Constraint enforcement and locking .

Enforced constraint projections

To enforce primary key and unique constraints, Vertica creates special projections that it uses to validate data. Depending on the amount of data in the anchor table, creating the projection might incur significant system overhead.

Rollback in transactions

Vertica validates enforced constraints for each SQL statement, and rolls back each statement that encounters a constraint violation. You cannot defer enforcement until the transaction commits. Thus, if multiple DML statements comprise a single transaction, Vertica validates each statement separately for constraint compliance, and rolls back any statement that fails validation. It commits the transaction only after all statements in it return.

For example, you might issue ten INSERT statements as a single transaction on a table that enforces UNIQUE on one of its columns. If the sixth statement attempts to insert a duplicate value in that column, that statement is rolled back. However, the other statements can commit.

Projections for enforced constraints

To enforce primary key and unique constraints, Vertica creates special constraint enforcement projections that it uses to validate new and updated data. If you add a constraint on an empty table, Vertica creates a constraint enforcement projection for that table only when data is added to it. If you add a primary key or unique constraint to a populated table and enable enforcement, Vertica chooses an existing projection to enforce the constraint, if one exists. Otherwise, Vertica creates a projection for that constraint. If a constraint violation occurs, Vertica rolls back the statement and any projection it created for the constraint.

If you drop an enforced primary key or unique constraint, Vertica automatically drops the projection associated with that constraint. You can also explicitly drop constraint projections with DROP PROJECTION . If the statement omits CASCADE , Vertica issues a warning about dropping this projection for an enabled constraint; otherwise, it silently drops the projection. In either case, the next time Vertica needs to enforce this constraint, it recreates the projection. Depending on the amount of data in the anchor table, creating the projection can incur significant overhead.

You can query system table PROJECTIONS on Boolean column IS_KEY_CONSTRAINT_PROJECTION to obtain constraint-specific projections.

> **Note**
> Constraint enforcement projections in a table can significantly facilitate its analysis by ANALYZE_CONSTRAINTS .

Constraint enforcement limitations

Vertica does not support constraint enforcement for foreign keys or external tables. Restrictions also apply to temporary tables.

Foreign keys

Vertica does not support enforcement of foreign keys and referential integrity. Thus, it is possible to load data that can return errors in the following cases:

- An inner join query is processed.
- An outer join is treated as an inner join due to the presence of foreign keys.

To validate foreign key constraints, use ANALYZE_CONSTRAINTS .

External tables

Vertica does not support automatic enforcement of constraints on external tables.

Local temporary tables

ALTER TABLE can set enforcement on a primary key or unique constraint in a local temporary table only if the table contains no data. If you try to enforce a constraint in a table that contains data, ALTER TABLE returns an error.

Global temporary tables

In a global temporary table, you set enforcement on a primary key or unique constraint only with CREATE TEMPORARY TABLE . ALTER TABLE returns an error if you try to set enforcement on a primary key or unique constraint in an existing table, whether populated or empty.

> **Note**
> You can always use ALTER TABLE...DROP CONSTRAINT to disable primary and unique key constraints in local and global temporary tables.

Managing queries

This section covers the following topics:

- Query plans : Describes how Vertica creates and uses query plans, which optimize access to information in the Vertica database.
- Directed queries : Shows how to save query plan information.

In this section

- Query plans
- Directed queries

Query plans

When you submit a query, the query optimizer quickly chooses the projections to use, optimizes and plans the query execution, and logs the SQL statement to its log. This planning results in an *query plan* , which maps out the steps the query performs.

A query plan is a sequence of step-like paths that the Vertica cost-based query optimizer uses to execute queries. Vertica can produce different query plans for a given query. For each query plan, the query optimizer evaluates the data to be queried: number of rows, column statistics such as number of distinct values (cardinality), distribution of data across nodes. It also evaluates available resources such as CPUs and network topology, and other environment factors. The query optimizer uses this information to develop several potential plans. It then compares plans and chooses one, generally the plan with the lowest cost .

The optimizer breaks down the query plan into smaller local plans and distributes them to executor nodes. The executor nodes process the smaller plans in parallel. Tasks associated with a query are recorded in the executor's log files.

In the final stages of query plan execution, the initiator node performs the following tasks:

- Combines results in a grouping operation.
- Merges multiple sorted partial result sets from all the executors.
- Formats the results to return to the client.

Before executing a query, you can view its plan in by embedding the query in an EXPLAIN statement; you can also view it in the Management Console.

In this section

- Viewing query plans
- Query plan cost estimation
- Query plan information and structure

Viewing query plans

You can obtain query plans in two ways:

- The EXPLAIN statement outputs query plans in various text formats (see below).
- Management Console provides a graphical interface for viewing query plans. For detailed information, see Working with query plans in MC .

You can also observe the real-time flow of data through a query plan by querying the system table QUERY_PLAN_PROFILES . For more information, see Profiling query plans .

EXPLAIN output options

By default, EXPLAIN output represents the query plan as a hierarchy, where each level, or path , represents a single database operation that the optimizer uses to execute a query. EXPLAIN output also appends DOT language source so you can display this output graphically with open source Graphviz tools.

EXPLAIN supports options for producing verbose and JSON output. You can also show the local query plans that are assigned to each node, which together comprise the total (global) query plan.

EXPLAIN also supports an ANNOTATED option. EXPLAIN ANNOTATED returns a query with embedded optimizer hints, which encapsulate the query plan for this query. For an example of usage, see Using optimizer-generated and custom directed queries together .

In this section

- EXPLAIN-Generated query plans
- JSON-Formatted query plans
- Verbose query plans
- Local query plans

EXPLAIN-Generated query plans

EXPLAIN returns the optimizer's query plan for executing a specified query. For example:

```
QUERY PLAN DESCRIPTION:
------------------------------

 EXPLAIN SELECT customer_name, customer_state FROM customer_dimension WHERE customer_state IN ('MA','NH') AND customer_gender='Male'
ORDER BY customer_name LIMIT 10;

 Access Path:
+-SELECT  LIMIT 10 [Cost: 365, Rows: 10] (PATH ID: 0)
|  Output Only: 10 tuples
|  Execute on: Query Initiator
| +---> SORT [TOPK] [Cost: 365, Rows: 544] (PATH ID: 1)
| |      Order: customer_dimension.customer_name ASC
| |      Output Only: 10 tuples
| |      Execute on: Query Initiator
| | +---> STORAGE ACCESS for customer_dimension [Cost: 326, Rows: 544] (PATH ID: 2)
| | |      Projection: public.customer_dimension_DBD_1_rep_VMartDesign_node0001
| | |      Materialize: customer_dimension.customer_state, customer_dimension.customer_name
| | |      Filter: (customer_dimension.customer_gender = 'Male')
| | |      Filter: (customer_dimension.customer_state = ANY (ARRAY['MA', 'NH']))
| | |      Execute on: Query Initiator
| | |      Runtime Filter: (SIP1(TopK): customer_dimension.customer_name)
```

You can use EXPLAIN to evaluate choices that the optimizer makes with respect to a given query. If you think query performance is less than optimal, run it through the Database Designer. For more information, see Incremental Design and Reducing query run time .

JSON-Formatted query plans

EXPLAIN JSON returns a query plan in JSON format. For example:

```
=> EXPLAIN JSON SELECT customer_name, customer_state FROM customer_dimension
    WHERE customer_state IN ('MA','NH') AND customer_gender='Male' ORDER BY customer_name LIMIT 10;
 ------------------------------
{
    "PARAMETERS" : {
       "QUERY_STRING" : "EXPLAIN JSON SELECT customer_name, customer_state FROM customer_dimension \n
       WHERE customer_state IN ('MA','NH') AND customer_gender='Male' ORDER BY customer_name LIMIT 10;"
    },
    "PLAN" : {
       "PATH_ID" : 0,
       "PATH_NAME" : "SELECT",
       "EXTRA" : " LIMIT 10",
       "COST" : 2114.000000,
       "ROWS" : 10.000000,
       "COST_STATUS" : "NO_STATISTICS",
       "TUPLE_LIMIT" : 10,
       "EXECUTE_NODE" : "Query Initiator",
       "INPUT" : {
          "PATH_ID" : 1,
          "PATH_NAME" : "SORT",
          "EXTRA" : "[TOPK]",
          "COST" : 2114.000000,
          "ROWS" : 49998.000000,
          "COST_STATUS" : "NO_STATISTICS",
          "ORDER" : ["customer_dimension.customer_name", "customer_dimension.customer_state"],
          "TUPLE_LIMIT" : 10,
          "EXECUTE_NODE" : "All Nodes",
          "INPUT" : {
             "PATH_ID" : 2,
             "PATH_NAME" : "STORAGE ACCESS",
             "EXTRA" : "for customer_dimension",
             "COST" : 252.000000,
             "ROWS" : 49998.000000,
             "COST_STATUS" : "NO_STATISTICS",
             "TABLE" : "public.customer_dimension",
             "PROJECTION" : "public.customer_dimension_b0",
             "MATERIALIZE" : ["customer_dimension.customer_name", "customer_dimension.customer_state"],
             "FILTER" : ["(customer_dimension.customer_state = ANY (ARRAY['MA', 'NH']))", "(customer_dimension.customer_gender = 'Male')"],
             "EXECUTE_NODE" : "All Nodes"
       "SIP" : "Runtime Filter: (SIP1(TopK): customer_dimension.customer_name)"

          }
       }
    }
}
(40 rows)
```

Verbose query plans

You can qualify EXPLAIN with the VERBOSE option. This option, valid for default and JSON output, increases the amount of detail in the rendered query plan

For example, the following EXPLAIN statement specifies to produce verbose output. Added information is set off in bold:

```
=> EXPLAIN VERBOSE SELECT customer_name, customer_state FROM customer_dimension
    WHERE customer_state IN ('MA','NH') AND customer_gender='Male' ORDER BY customer_name LIMIT 10;

QUERY PLAN DESCRIPTION:
 ------------------------------

 Opt Vertica Options
 --------------------
 PLAN_OUTPUT_SUPER_VERBOSE

 EXPLAIN VERBOSE SELECT customer_name, customer_state FROM customer_dimension
 WHERE customer_state IN ('MA','NH') AND customer_gender='Male'
 ORDER BY customer_name LIMIT 10;

Access Path:
+-SELECT  LIMIT 10 [Cost: 756.000000, Rows: 10.000000 Disk(B): 0.000000 CPU(B): 0.000000 Memory(B): 0.000000 Netwrk(B): 0.000000 Parallelism:
1.000000] [OutRowSz (B): 274](PATH ID: 0)
|  Output Only: 10 tuples
|  Execute on: Query Initiator
|  Sort Key: (customer_dimension.customer_name)
|  LDISTRIB_UNSEGMENTED
| +---> SORT [TOPK] [Cost: 756.000000, Rows: 9998.000000 Disk(B): 0.000000 CPU(B): 34274697.123457 Memory(B): 2739452.000000 Netwrk(B):
0.000000 Parallelism: 4.000000 (NO STATISTICS)] [OutRowSz (B): 274] (PATH ID: 1)
| |      Order: customer_dimension.customer_name ASC
| |      Output Only: 10 tuples
| |      Execute on: Query Initiator
| |      Sort Key: (customer_dimension.customer_name)
| |      LDISTRIB_UNSEGMENTED
| | +---> STORAGE ACCESS for customer_dimension [Cost: 513.000000, Rows: 9998.000000 Disk(B): 0.000000 CPU(B): 0.000000 Memory(B): 0.000000
Netwrk(B): 0.000000 Parallelism: 4.000000 (NO STATISTICS)] [OutRowSz (B): 274] (PATH ID: 2)
| | |      Column Cost Aspects: [ Disk(B): 7371817.156569 CPU(B): 4914708.578284 Memory(B): 2659466.004399 Netwrk(B): 0.000000 Parallelism:
4.000000 ]
| | |      Projection: public.customer_dimension_P1
| | |      Materialize: customer_dimension.customer_state, customer_dimension.customer_name
| | |      Filter: (customer_dimension.customer_gender = 'Male')/* sel=0.999800 ndv= 500 */
| | |      Filter: (customer_dimension.customer_state = ANY (ARRAY['MA', 'NH']))/* sel=0.999800 ndv= 500 */
| | |      Execute on: All Nodes
| | |      Runtime Filter: (SIP1(TopK): customer_dimension.customer_name)
| | |      Sort Key: (customer_dimension.household_id, customer_dimension.customer_key, customer_dimension.store_membership_card,
customer_dimension.customer_type, customer_dimension.customer_region, customer_dimension.title,
customer_dimension.number_of_children)
| | |      LDISTRIB_SEGMENTED
```

Local query plans

EXPLAIN LOCAL (on a multi-node database) shows the local query plans assigned to each node, which together comprise the total (global) query plan. If you omit this option, Vertica shows only the global query plan. Local query plans are shown only in DOT language source, which can be rendered in Graphviz .

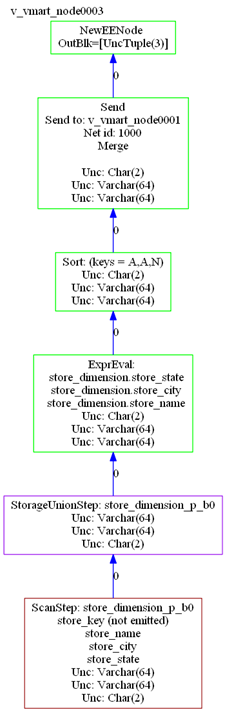
For example, the following EXPLAIN statement includes the LOCAL option:

```
=> EXPLAIN LOCAL SELECT store_name, store_city, store_state
   FROM store.store_dimension ORDER BY store_state ASC, store_city ASC;
```

The output includes GraphViz source, which describes the local query plans assigned to each node. For example, output for this statement on a three-node database includes a GraphViz description of the following query plan for one node (v_vmart_node0003):

```
------------------------------------------------
PLAN: v_vmart_node0003 (GraphViz Format)
------------------------------------------------
digraph G {
graph [rankdir=BT, label = "v_vmart_node0003\n", labelloc=t, labeljust=l ordering=out]
0[label = "NewEENode \nOutBlk=[UncTuple(3)]", color = "green", shape = "box"];
1[label = "Send\nSend to: v_vmart_node0001\nNet id: 1000\nMerge\n\nUnc: Char(2)\nUnc: Varchar(64)\nUnc: Varchar(64)", color = "green", shape = "box"];
2[label = "Sort: (keys = A,A,N)\nUnc: Char(2)\nUnc: Varchar(64)\nUnc: Varchar(64)", color = "green", shape = "box"];
3[label = "ExprEval: \n  store_dimension.store_state\n  store_dimension.store_city\n  store_dimension.store_name\nUnc: Char(2)\nUnc: Varchar(64)\nUnc: Varchar(64)
", color = "green", shape = "box"];
4[label = "StorageUnionStep: store_dimension_p_b0\nUnc: Varchar(64)\nUnc: Varchar(64)\nUnc: Char(2)", color = "purple", shape = "box"];
5[label = "ScanStep: store_dimension_p_b0\nstore_key (not emitted)\nstore_name\nstore_city\nstore_state\nUnc: Varchar(64)\nUnc: Varchar(64)\nUnc: Char(2)", color
= "brown", shape = "box"];
1->0 [label = "0",color = "blue"];
2->1 [label = "0",color = "blue"];
3->2 [label = "0",color = "blue"];
4->3 [label = "0",color = "blue"];
5->4 [label = "0",color = "blue"];
}
```

GraphViz renders this output as follows:



Query plan cost estimation

The query optimizer chooses a query plan based on cost estimates. The query optimizer uses information from a number of sources to develop potential plans and determine their relative costs. These include:

- Number of table rows
- Column statistics, including: number of distinct values (cardinality), minimum/maximum values, distribution of values, and disk space usage
- Access path that is likely to require fewest I/O operations, and lowest CPU, memory, and network usage
- Available eligible projections
- Join options: join types (merge versus hash joins), join order
- Query predicates
- Data segmentation across cluster nodes

Many important optimizer decisions rely on statistics, which the query optimizer uses to determine the final plan to execute a query. Therefore, it is important that statistics be up to date. Without reasonably accurate statistics, the optimizer could choose a suboptimal plan, which might affect query performance.

Vertica provides hints about statistics in the query plan. See Query plan statistics .

Cost versus execution runtime

Although costs correlate to query runtime, they do not provide an *estimate* of actual runtime. For example, if the optimizer determines that Plan A costs twice as much as Plan B, it is likely that Plan A will require more time to run. However, this cost estimate does not necessarily indicate that Plan A will run twice as long as Plan B.

Also, plan costs for different queries are not directly comparable. For example, if the estimated cost of Plan X for query1 is greater than the cost of Plan Y for query2, it is not necessarily true that Plan X's runtime is greater than Plan Y's runtime.

Query plan information and structure

Depending on the query and database schema, EXPLAIN output includes the following information:

- Tables referenced by the statement
- Estimated costs
- Estimated row cardinality
- Path ID, an integer that links to error messages and profiling counters so you troubleshoot performance issues more easily. For more information, see Profiling query plans .
- Data operations such as SORT , FILTER , LIMIT , and GROUP BY
- Projections used
- Information about statistics—for example, whether they are current or out of range
- Algorithms chosen for operations into the query, such as HASH / MERGE or GROUPBY HASH / GROUPBY PIPELINED
- Data redistribution (broadcast, segmentation) across cluster nodes

Example

In the EXPLAIN output that follows, the optimizer processes a query in three steps, where each step identified by a unique path ID:

- 0: Limit
- 1: Sort
- 2: Storage access and filter

```
QUERY PLAN DESCRIPTION:
------------------------------

EXPLAIN SELECT customer_name, customer_state FROM customer_dimension
WHERE customer_state IN ('MA','NH') AND customer_gender = 'Male'
ORDER BY customer_name LIMIT 10;

Access Path:
+-SELECT  LIMIT 10 [Cost: 365, Rows: 10] (PATH ID: 0) <-- Limit
|  Output Only: 10 tuples
|  Execute on: Query Initiator
| +---> SORT [TOPK] [Cost: 365, Rows: 544] (PATH ID: 1) <-- Sort
| |      Order: customer_dimension.customer_name ASC
| |      Output Only: 10 tuples
| |      Execute on: Query Initiator
| | +---> STORAGE ACCESS for customer_dimension [Cost: 326, Rows: 544] (PATH ID: 2)
| | |      Projection: public.customer_dimension_DBD_1_rep_VMartDesign_node0001
| | |      Materialize: customer_dimension.customer_state, customer_dimension.customer_name
| | |      Filter: (customer_dimension.customer_gender = 'Male')
| | |      Filter: (customer_dimension.customer_state = ANY (ARRAY['MA', 'NH']))
| | |      Execute on: Query Initiator
```

Storage accesss and filter

> **Note**
> A storage access operation can scan more than the columns in the SELECT list— for example, columns referenced in WHERE clause.

In this section

- Query plan statistics
- Cost and rows path
- Projection path
- Join path
- Path ID
- Filter path
- GROUP BY paths
- Sort path
- Limit path

- Data redistribution path
- Analytic function path
- Node down information
- MERGE path

Query plan statistics

If you query a table whose statistics are unavailable or out-of-date, the optimizer might choose a sub-optimal query plan.

You can resolve many issues related to table statistics by calling ANALYZE_STATISTICS . This function let you update statistics at various scopes: one or more table columns, a single table, or all database tables.

If you update statistics and find that the query still performs sub-optimally, run your query through Database Designer and choose incremental design as the design type.

For detailed information about updating database statistics, see Collecting database statistics .

Statistics hints in query plans

Query plans can contain information about table statistics through two hints: NO STATISTICS and STALE STATISTICS . For example, the following query plan fragment includes NO STATISTICS to indicate that histograms are unavailable:

```
| | +-- Outer -> STORAGE ACCESS for fact [Cost: 604, Rows: 10K (NO STATISTICS)]
```

The following query plan fragment includes STALE STATISTICS to indicate that the predicate has fallen outside the histogram range:

```
| | +-- Outer -> STORAGE ACCESS for fact [Cost: 35, Rows: 1 (STALE STATISTICS)]
```

Cost and rows path

The following EXPLAIN output shows the Cost operator:

```
Access Path: +-SELECT  LIMIT 10 [Cost: 370, Rows: 10] (PATH ID: 0)
|  Output Only: 10 tuples
|  Execute on: Query Initiator
| +---> SORT [Cost: 370, Rows: 544] (PATH ID: 1)
| |      Order: customer_dimension.customer_name ASC
| |      Output Only: 10 tuples
| |      Execute on: Query Initiator
| | +---> STORAGE ACCESS for customer_dimension [Cost: 331, Rows: 544] (PATH ID: 2)
| | |      Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
| | |      Materialize: customer_dimension.customer_state, customer_dimension.customer_name
| | |      Filter: (customer_dimension.customer_gender = 'Male')
| | |      Filter: (customer_dimension.customer_state = ANY (ARRAY['MA', 'NH']))
| | |      Execute on: Query Initiator
```

The Row operator is the number of rows the optimizer estimates the query will return. Letters after numbers refer to the units of measure (K=thousand, M=million, B=billion, T=trillion), so the output for the following query indicates that the number of rows to return is 50 *thousand* .

```
=> EXPLAIN SELECT customer_gender FROM customer_dimension;
 Access Path:
+-STORAGE ACCESS for customer_dimension [Cost: 17, Rows: 50K (3 RLE)] (PATH ID: 1)
|  Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
|  Materialize: customer_dimension.customer_gender
|  Execute on: Query Initiator
```

The reference to (3 RLE) in the STORAGE ACCESS path means that the optimizer estimates that the storage access operator returns 50K rows. Because the column is run-length encoded (RLE), the real number of RLE rows returned is only three rows:

- 1 row for female
- 1 row for male
- 1 row that represents unknown (NULL) gender

> **Note**
> See Query plans for more information about how the optimizer estimates cost.

Projection path

You can see which projections the optimizer chose for the query plan by looking at the Projection path in the textual output:

```
EXPLAIN SELECT
  customer_name,
  customer_state
 FROM customer_dimension
 WHERE customer_state in ('MA','NH')
 AND customer_gender = 'Male'
 ORDER BY customer_name
 LIMIT 10;
 Access Path:
 +-SELECT  LIMIT 10 [Cost: 370, Rows: 10] (PATH ID: 0)
 |  Output Only: 10 tuples
 |  Execute on: Query Initiator
 | +---> SORT [Cost: 370, Rows: 544] (PATH ID: 1)
 | |      Order: customer_dimension.customer_name ASC
 | |      Output Only: 10 tuples
 | |      Execute on: Query Initiator
 | | +---> STORAGE ACCESS for customer_dimension [Cost: 331, Rows: 544] (PATH ID: 2)
 | | |      Projection: public.customer_dimension_DBD_1_rep_vmart_vmart_node0001
 | | |      Materialize: customer_dimension.customer_state, customer_dimension.customer_name
 | | |      Filter: (customer_dimension.customer_gender = 'Male')
 | | |      Filter: (customer_dimension.customer_state = ANY (ARRAY['MA', 'NH']))
 | | |      Execute on: Query Initiator
```

The query optimizer automatically picks the best projections, but without reasonably accurate statistics, the optimizer could choose a suboptimal projection or join order for a query. For details, see Collecting Statistics .

Vertica considers which projection to choose for a plan by considering the following aspects:

- How columns are joined in the query
- How the projections are grouped or sorted
- Whether SQL analytic operations applied
- Any column information from a projection's storage on disk

As Vertica scans the possibilities for each plan, projections with the higher initial costs could end up in the final plan because they make joins cheaper. For example, a query can be answered with many possible plans, which the optimizer considers before choosing one of them. For efficiency, the optimizer uses sophisticated algorithms to prune intermediate partial plan fragments with higher cost. The optimizer knows that intermediate plan fragments might initially look bad (due to high storage access cost) but which produce excellent final plans due to other optimizations that it allows.

If your statistics are up to date but the query still performs poorly, run the query through the Database Designer. For details, see Incremental Design .

Tips

- To test different segmented projections, refer to the projection by name in the query.
- For optimal performance, write queries so the columns are sorted the same way that the projection columns are sorted.

See also

- Reducing query run time
- Creating custom designs
- Projections

Join path

Just like a join query, which references two or more tables, the Join step in a query plan has two input branches:

- The left input, which is the outer table of the join
- The right input, which is the inner table of the join

In the following query, the T1 table is the left input because it is on the left side of the JOIN keyword, and the T2 table is the right input, because it is on the right side of the JOIN keyword:

```
SELECT * FROM T1 JOIN T2 ON T1.x = T2.x;
```

Outer versus inner join

Query performance is better if the small table is used as the inner input to the join. The query optimizer automatically reorders the inputs to joins to ensure that this is the case unless the join in question is an outer join.

> **Note**
> If the configuration parameter EnableForceOuter is set to 1, you can control join inputs for specific tables through ALTER TABLE..FORCE OUTER . For details, see Controlling join inputs .

The following example shows a query and its plan for a left outer join:

```
=> EXPLAIN SELECT CD.annual_income,OSI.sale_date_key
-> FROM online_sales.online_sales_fact OSI
-> LEFT OUTER JOIN customer_dimension CD ON CD.customer_key = OSI.customer_key;
 Access Path:
 +-JOIN HASH [LeftOuter] [Cost: 4K, Rows: 5M] (PATH ID: 1)
 |  Join Cond: (CD.customer_key = OSI.customer_key)
 |  Materialize at Output: OSI.sale_date_key
 |  Execute on: All Nodes
 | +-- Outer -> STORAGE ACCESS for OSI [Cost: 3K, Rows: 5M] (PATH ID: 2)
 | |      Projection: online_sales.online_sales_fact_DBD_12_seg_vmartdb_design_vmartdb_design
 | |      Materialize: OSI.customer_key
 | |      Execute on: All Nodes
 | +-- Inner -> STORAGE ACCESS for CD [Cost: 264, Rows: 50K] (PATH ID: 3)
 | |      Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
 | |      Materialize: CD.annual_income, CD.customer_key
 | |      Execute on: All Nodes
```

The following example shows a query and its plan for a full outer join:

```
=> EXPLAIN SELECT CD.annual_income,OSI.sale_date_key
-> FROM online_sales.online_sales_fact OSI
-> FULL OUTER JOIN customer_dimension CD ON CD.customer_key = OSI.customer_key;
 Access Path:
 +-JOIN HASH [FullOuter] [Cost: 18K, Rows: 5M] (PATH ID: 1) Outer (RESEGMENT) Inner (FILTER)
 |  Join Cond: (CD.customer_key = OSI.customer_key)
 |  Execute on: All Nodes
 | +-- Outer -> STORAGE ACCESS for OSI [Cost: 3K, Rows: 5M] (PATH ID: 2)
 | |      Projection: online_sales.online_sales_fact_DBD_12_seg_vmartdb_design_vmartdb_design
 | |      Materialize: OSI.sale_date_key, OSI.customer_key
 | |      Execute on: All Nodes
 | +-- Inner -> STORAGE ACCESS for CD [Cost: 264, Rows: 50K] (PATH ID: 3)
 | |      Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
 | |      Materialize: CD.annual_income, CD.customer_key
 | |      Execute on: All Nodes
```

Hash and merge joins

Vertica has two join algorithms to choose from: merge join and hash join. The optimizer automatically chooses the most appropriate algorithm, given the query and projections in a system.

For the following query, the optimizer chooses a hash join.

```
=> EXPLAIN SELECT CD.annual_income,OSI.sale_date_key
-> FROM online_sales.online_sales_fact OSI
-> INNER JOIN customer_dimension CD ON CD.customer_key = OSI.customer_key;
 Access Path:
 +-JOIN HASH [Cost: 4K, Rows: 5M] (PATH ID: 1)
 |  Join Cond: (CD.customer_key = OSI.customer_key)
 |  Materialize at Output: OSI.sale_date_key
 |  Execute on: All Nodes
 | +-- Outer -> STORAGE ACCESS for OSI [Cost: 3K, Rows: 5M] (PATH ID: 2)
 | |      Projection: online_sales.online_sales_fact_DBD_12_seg_vmartdb_design_vmartdb_design
 | |      Materialize: OSI.customer_key
 | |      Execute on: All Nodes
 | +-- Inner -> STORAGE ACCESS for CD [Cost: 264, Rows: 50K] (PATH ID: 3)
 | |      Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
 | |      Materialize: CD.annual_income, CD.customer_key
 | |      Execute on: All Nodes
```

Tip

If you get a hash join when you are expecting a merge join, it means that at least one of the projections is not sorted on the join column (for example, customer_key in the preceding query). To facilitate a merge join, you might need to create different projections that are sorted on the join columns.

In the next example, the optimizer chooses a merge join. The optimizer's first pass performs a merge join because the inputs are presorted, and then it performs a hash join.

```
=> EXPLAIN SELECT count(*) FROM online_sales.online_sales_fact OSI
-> INNER JOIN customer_dimension CD ON CD.customer_key = OSI.customer_key
-> INNER JOIN product_dimension PD ON PD.product_key = OSI.product_key;
 Access Path:
 +-GROUPBY NOTHING [Cost: 8K, Rows: 1] (PATH ID: 1)
 |  Aggregates: count(*)
 |  Execute on: All Nodes
 | +---> JOIN HASH [Cost: 7K, Rows: 5M] (PATH ID: 2)
 | |      Join Cond: (PD.product_key = OSI.product_key)
 | |      Materialize at Input: OSI.product_key
 | |      Execute on: All Nodes
 | | +-- Outer -> JOIN MERGEJOIN(inputs presorted) [Cost: 4K, Rows: 5M] (PATH ID: 3)
 | | |      Join Cond: (CD.customer_key = OSI.customer_key)
 | | |      Execute on: All Nodes
 | | | +-- Outer -> STORAGE ACCESS for OSI [Cost: 3K, Rows: 5M] (PATH ID: 4)
 | | | |      Projection: online_sales.online_sales_fact_DBD_12_seg_vmartdb_design_vmartdb_design
 | | | |      Materialize: OSI.customer_key
 | | | |      Execute on: All Nodes
 | | | +-- Inner -> STORAGE ACCESS for CD [Cost: 132, Rows: 50K] (PATH ID: 5)
 | | | |      Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
 | | | |      Materialize: CD.customer_key
 | | | |      Execute on: All Nodes
 | | +-- Inner -> STORAGE ACCESS for PD [Cost: 152, Rows: 60K] (PATH ID: 6)
 | | |      Projection: public.product_dimension_DBD_2_rep_vmartdb_design_vmartdb_design_node0001
 | | |      Materialize: PD.product_key
 | | |      Execute on: All Nodes
```

Inequality joins

Vertica processes joins with equality predicates very efficiently. The query plan shows equality join predicates as join condition (Join Cond).

```
=> EXPLAIN SELECT CD.annual_income, OSI.sale_date_key
-> FROM online_sales.online_sales_fact OSI
-> INNER JOIN customer_dimension CD
-> ON CD.customer_key = OSI.customer_key;
 Access Path:
 +-JOIN HASH [Cost: 4K, Rows: 5M] (PATH ID: 1)
 |  Join Cond: (CD.customer_key = OSI.customer_key)
 |  Materialize at Output: OSI.sale_date_key
 |  Execute on: All Nodes
 | +-- Outer -> STORAGE ACCESS for OSI [Cost: 3K, Rows: 5M] (PATH ID: 2)
 | |      Projection: online_sales.online_sales_fact_DBD_12_seg_vmartdb_design_vmartdb_design
 | |      Materialize: OSI.customer_key
 | |      Execute on: All Nodes
 | +-- Inner -> STORAGE ACCESS for CD [Cost: 264, Rows: 50K] (PATH ID: 3)
 | |      Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
 | |      Materialize: CD.annual_income, CD.customer_key
 | |      Execute on: All Nodes
```

However, inequality joins are treated like cross joins and can run less efficiently, which you can see by the change in cost between the two queries:

```
=> EXPLAIN SELECT CD.annual_income, OSI.sale_date_key
-> FROM online_sales.online_sales_fact OSI
-> INNER JOIN customer_dimension CD
-> ON CD.customer_key < OSI.customer_key;
 Access Path:
 +-JOIN HASH [Cost: 98M, Rows: 5M] (PATH ID: 1)
 |  Join Filter: (CD.customer_key < OSI.customer_key)
 |  Materialize at Output: CD.annual_income
 |  Execute on: All Nodes
 | +-- Outer -> STORAGE ACCESS for CD [Cost: 132, Rows: 50K] (PATH ID: 2)
 | |      Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
 | |      Materialize: CD.customer_key
 | |      Execute on: All Nodes
 | +-- Inner -> STORAGE ACCESS for OSI [Cost: 3K, Rows: 5M] (PATH ID: 3)
 | |      Projection: online_sales.online_sales_fact_DBD_12_seg_vmartdb_design_vmartdb_design
 | |      Materialize: OSI.sale_date_key, OSI.customer_key
 | |      Execute on: All Nodes
```

Event series joins

Event series joins are denoted by the INTERPOLATED path.

```
=> EXPLAIN SELECT * FROM hTicks h FULL OUTER JOIN aTicks a -> ON (h.time INTERPOLATE PREVIOUS
 Access Path:
 +-JOIN  (INTERPOLATED) [FullOuter] [Cost: 31, Rows: 4 (NO STATISTICS)] (PATH ID: 1)
   Outer (SORT ON JOIN KEY) Inner (SORT ON JOIN KEY)
 |  Join Cond: (h."time" = a."time")
 |  Execute on: Query Initiator
 | +-- Outer -> STORAGE ACCESS for h [Cost: 15, Rows: 4 (NO STATISTICS)] (PATH ID: 2)
 | |      Projection: public.hTicks_node0004
 | |      Materialize: h.stock, h."time", h.price
 | |      Execute on: Query Initiator
 | +-- Inner -> STORAGE ACCESS for a [Cost: 15, Rows: 4 (NO STATISTICS)] (PATH ID: 3)
 | |      Projection: public.aTicks_node0004
 | |      Materialize: a.stock, a."time", a.price
 | |      Execute on: Query Initiator
```

Path ID

The PATH ID is a unique identifier that Vertica assigns to each operation (path) within a query plan. The same identifier is shared by:

- Query plans
- Join error messages

- System tables EXECUTION_ENGINE_PROFILES and QUERY_PLAN_PROFILES

Path IDs can help you trace issues to their root cause. For example, if a query returns a join error, preface the query with EXPLAIN and look for PATH ID *n* in the query plan to see which join in the query had the problem.

For example, the following EXPLAIN output shows the path ID for each path in the optimizer's query plan:

```
=> EXPLAIN SELECT * FROM fact JOIN dim ON x=y JOIN ext on y=z;
 Access Path:
 +-JOIN MERGEJOIN(inputs presorted) [Cost: 815, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
 | Join Cond: (dim.y = ext.z)
 | Materialize at Output: fact.x
 | Execute on: All Nodes
 | +-- Outer -> JOIN MERGEJOIN(inputs presorted) [Cost: 408, Rows: 10K (NO STATISTICS)] (PATH ID: 2)
 | | Join Cond: (fact.x = dim.y)
 | | Execute on: All Nodes
 | | +-- Outer -> STORAGE ACCESS for fact [Cost: 202, Rows: 10K (NO STATISTICS)] (PATH ID: 3)
 | | | Projection: public.fact_super
 | | | Materialize: fact.x
 | | | Execute on: All Nodes
 | | +-- Inner -> STORAGE ACCESS for dim [Cost: 202, Rows: 10K (NO STATISTICS)] (PATH ID: 4)
 | | | Projection: public.dim_super
 | | | Materialize: dim.y
 | | | Execute on: All Nodes
 | +-- Inner -> STORAGE ACCESS for ext [Cost: 202, Rows: 10K (NO STATISTICS)] (PATH ID: 5)
 | | Projection: public.ext_super
 | | Materialize: ext.z
 | | Execute on: All Nodes
```

Filter path

The Filter step evaluates predicates on a single table. It accepts a set of rows, eliminates some of them (based on the criteria you provide in your query), and returns the rest. For example, the optimizer can filter local data of a join input that will be joined with another re-segmented join input.

The following statement queries the customer_dimension table and uses the WHERE clause to filter the results only for male customers in Massachusetts and New Hampshire.

```
EXPLAIN SELECT
  CD.customer_name,
  CD.customer_state,
  AVG(CD.customer_age) AS avg_age,
  COUNT(*) AS count
FROM customer_dimension CD
WHERE CD.customer_state in ('MA','NH') AND CD.customer_gender = 'Male'
GROUP BY CD.customer_state, CD.customer_name;
```

The query plan output is as follows:

```
 Access Path:
 +-GROUPBY HASH [Cost: 378, Rows: 544] (PATH ID: 1)
 |  Aggregates: sum_float(CD.customer_age), count(CD.customer_age), count(*)
 |  Group By: CD.customer_state, CD.customer_name
 |  Execute on: Query Initiator
 | +---> STORAGE ACCESS for CD [Cost: 372, Rows: 544] (PATH ID: 2)
 | |       Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
 | |       Materialize: CD.customer_state, CD.customer_name, CD.customer_age
 | |       Filter: (CD.customer_gender = 'Male')
 | |       Filter: (CD.customer_state = ANY (ARRAY['MA', 'NH']))
 | |       Execute on: Query Initiator
```

GROUP BY paths

A GROUP BY operation has two algorithms:

- GROUPBY HASH input is not sorted by the group columns, so Vertica builds a hash table on those group columns in order to process the aggregates and group by expressions.
- GROUPBY PIPELINED requires that inputs be presorted on the columns specified in the group, which means that Vertica need only retain data in the current group in memory. GROUPBY PIPELINED operations are preferred because they are generally faster and require less memory than GROUPBY HASH. GROUPBY PIPELINED is especially useful for queries that process large numbers of high-cardinality group by columns or DISTINCT aggregates.

If possible, the query optimizer chooses the faster algorithm GROUPBY PIPELINED over GROUPBY HASH.

> **Note**
> For details, see GROUP BY implementation options .

In this section

- GROUPBY HASH query plan
- GROUPBY PIPELINED query plan

GROUPBY HASH query plan

Here's an example of how GROUPBY HASH operations look in EXPLAIN output.

```
=> EXPLAIN SELECT COUNT(DISTINCT annual_income)
   FROM customer_dimension
   WHERE customer_region='NorthWest';
```

The output shows that the optimizer chose the less efficient GROUPBY HASH path, which means the projection was not presorted on the annual_income column. If such a projection is available, the optimizer would choose the GROUPBY PIPELINED algorithm.

```
Access Path:
+-GROUPBY NOTHING [Cost: 256, Rows: 1 (NO STATISTICS)] (PATH ID: 1)
|  Aggregates: count(DISTINCT customer_dimension.annual_income)
| +---> GROUPBY HASH (LOCAL RESEGMENT GROUPS) [Cost: 253, Rows: 10K (NO STATISTICS)] (PATH ID: 2)
| |     Group By: customer_dimension.annual_income
| | +---> STORAGE ACCESS for customer_dimension [Cost: 227, Rows: 50K (NO STATISTICS)] (PATH ID: 3)
| | |     Projection: public.customer_dimension_super
| | |     Materialize: customer_dimension.annual_income
| | |     Filter: (customer_dimension.customer_region = 'NorthWest'
...
```

GROUPBY PIPELINED query plan

If you have a projection that is already sorted on the customer_gender column, the optimizer chooses the faster GROUPBY PIPELINED operation:

```
 => EXPLAIN SELECT COUNT(distinct customer_gender) from customer_dimension;
Access Path:
 +-GROUPBY NOTHING [Cost: 22, Rows: 1] (PATH ID: 1)
 |  Aggregates: count(DISTINCT customer_dimension.customer_gender)
 |  Execute on: Query Initiator
 | +---> GROUPBY PIPELINED [Cost: 20, Rows: 10K] (PATH ID: 2)
 | |     Group By: customer_dimension.customer_gender
 | |     Execute on: Query Initiator
 | | +---> STORAGE ACCESS for customer_dimension [Cost: 17, Rows: 50K (3 RLE)] (PATH ID: 3)
 | | |     Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
 | | |     Materialize: customer_dimension.customer_gender
 | | |     Execute on: Query Initiator
```

Similarly, the use of an equality predicate, such as in the following query, preserves GROUPBY PIPELINED :

```
=> EXPLAIN SELECT COUNT(DISTINCT annual_income)    FROM customer_dimension
   WHERE customer_gender = 'Female';

 Access Path: +-GROUPBY NOTHING [Cost: 161, Rows: 1] (PATH ID: 1)
 |  Aggregates: count(DISTINCT customer_dimension.annual_income)
 | +---> GROUPBY PIPELINED [Cost: 158, Rows: 10K] (PATH ID: 2)
 | |      Group By: customer_dimension.annual_income
 | | +---> STORAGE ACCESS for customer_dimension [Cost: 144, Rows: 47K] (PATH ID: 3)
 | | |      Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
 | | |      Materialize: customer_dimension.annual_income
 | | |      Filter: (customer_dimension.customer_gender = 'Female')
```

> **Tip**
> If EXPLAIN reports GROUPBY HASH , modify the projection design to force it to use GROUPBY PIPELINED .

Sort path

The SORT operator sorts the data according to a specified list of columns. The EXPLAIN output indicates the sort expressions and if the sort order is ascending (ASC) or descending (DESC).

For example, the following query plan shows the column list nature of the SORT operator:

```
EXPLAIN SELECT
  CD.customer_name,
  CD.customer_state,
  AVG(CD.customer_age) AS avg_age,
  COUNT(*) AS count
FROM customer_dimension CD
WHERE CD.customer_state in ('MA','NH')
  AND CD.customer_gender = 'Male'
GROUP BY CD.customer_state, CD.customer_name
ORDER BY avg_age, customer_name;
 Access Path:
 +-SORT [Cost: 422, Rows: 544] (PATH ID: 1)
 |  Order: (<SVAR> / float8(<SVAR>)) ASC, CD.customer_name ASC
 |  Execute on: Query Initiator
 | +---> GROUPBY HASH [Cost: 378, Rows: 544] (PATH ID: 2)
 | |      Aggregates: sum_float(CD.customer_age), count(CD.customer_age), count(*)
 | |      Group By: CD.customer_state, CD.customer_name
 | |      Execute on: Query Initiator
 | | +---> STORAGE ACCESS for CD [Cost: 372, Rows: 544] (PATH ID: 3)
 | | |      Projection: public.customer_dimension_DBD_1_rep_vmart_vmart_node0001
 | | |      Materialize: CD.customer_state, CD.customer_name, CD.customer_age
 | | |      Filter: (CD.customer_gender = 'Male')
 | | |      Filter: (CD.customer_state = ANY (ARRAY['MA', 'NH']))
 | | |      Execute on: Query Initiator
```

If you change the sort order to descending, the change appears in the query plan:

```
EXPLAIN SELECT
  CD.customer_name,
  CD.customer_state,
  AVG(CD.customer_age) AS avg_age,
  COUNT(*) AS count
FROM customer_dimension CD
WHERE CD.customer_state in ('MA','NH')
  AND CD.customer_gender = 'Male'
GROUP BY CD.customer_state, CD.customer_name
ORDER BY avg_age DESC, customer_name;
 Access Path:
 +-SORT [Cost: 422, Rows: 544] (PATH ID: 1)
 |  Order: (<SVAR> / float8(<SVAR>)) DESC, CD.customer_name ASC
 |  Execute on: Query Initiator
 | +---> GROUPBY HASH [Cost: 378, Rows: 544] (PATH ID: 2)
 | |      Aggregates: sum_float(CD.customer_age), count(CD.customer_age), count(*)
 | |      Group By: CD.customer_state, CD.customer_name
 | |      Execute on: Query Initiator
 | | +---> STORAGE ACCESS for CD [Cost: 372, Rows: 544] (PATH ID: 3)
 | | |      Projection: public.customer_dimension_DBD_1_rep_vmart_vmart_node0001
 | | |      Materialize: CD.customer_state, CD.customer_name, CD.customer_age
 | | |      Filter: (CD.customer_gender = 'Male')
 | | |      Filter: (CD.customer_state = ANY (ARRAY['MA', 'NH']))
 | | |      Execute on: Query Initiator
```

Limit path

The LIMIT path restricts the number of result rows based on the LIMIT clause in the query. Using the LIMIT clause in queries with thousands of rows might increase query performance.

The optimizer pushes the LIMIT operation as far down as possible in queries. A single LIMIT clause in the query can generate multiple Output Only plan annotations.

```
 => EXPLAIN SELECT COUNT(DISTINCT annual_income) FROM customer_dimension LIMIT 10;
Access Path:
 +-SELECT  LIMIT 10 [Cost: 161, Rows: 10] (PATH ID: 0)
 |  Output Only: 10 tuples
 | +---> GROUPBY NOTHING [Cost: 161, Rows: 1] (PATH ID: 1)
 | |      Aggregates: count(DISTINCT customer_dimension.annual_income)
 | |      Output Only: 10 tuples
 | | +---> GROUPBY HASH (SORT OUTPUT) [Cost: 158, Rows: 10K] (PATH ID: 2)
 | | |      Group By: customer_dimension.annual_income
 | | | +---> STORAGE ACCESS for customer_dimension [Cost: 132, Rows: 50K] (PATH ID: 3)
 | | | |      Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
 | | | |      Materialize: customer_dimension.annual_income
```

Data redistribution path

The optimizer can redistribute join data in two ways:

- Broadcasting
- Resegmentation

Broadcasting

Broadcasting sends a complete copy of an intermediate result to all nodes in the cluster. Broadcast is used for joins in the following cases:

- One table is very small (usually the inner table) compared to the other.
- Vertica can avoid other large upstream resegmentation operations.
- Outer join or subquery semantics require one side of the join to be replicated.

For example:

```
=> EXPLAIN SELECT * FROM T1 LEFT JOIN T2 ON T1.a > T2.y;
 Access Path:
+-JOIN HASH [LeftOuter] [Cost: 40K, Rows: 10K (NO STATISTICS)] (PATH ID: 1) Inner (BROADCAST)
|  Join Filter: (T1.a > T2.y)
|  Materialize at Output: T1.b
|  Execute on: All Nodes
| +-- Outer -> STORAGE ACCESS for T1 [Cost: 151, Rows: 10K (NO STATISTICS)] (PATH ID: 2)
| | |    Projection: public.T1_b0
| | |    Materialize: T1.a
| | |    Execute on: All Nodes
| +-- Inner -> STORAGE ACCESS for T2 [Cost: 302, Rows: 10K (NO STATISTICS)] (PATH ID: 3)
| | |    Projection: public.T2_b0
| | |    Materialize: T2.x, T2.y
| | |    Execute on: All Nodes
```

Resegmentation

Resegmentation takes an existing projection or intermediate relation and resegments the data evenly across all cluster nodes. At the end of the resegmentation operation, every row from the input relation is on exactly one node. Resegmentation is the operation used most often for distributed joins in Vertica if the data is not already segmented for local joins. For more detail, see Identical segmentation .

For example:

```
=> CREATE TABLE T1 (a INT, b INT) SEGMENTED BY HASH(a) ALL NODES;
=> CREATE TABLE T2 (x INT, y INT) SEGMENTED BY HASH(x) ALL NODES;
=> EXPLAIN SELECT * FROM T1 JOIN T2 ON T1.a = T2.y;

 ------------------------------ QUERY PLAN DESCRIPTION: ------------------------------
 Access Path:
+-JOIN HASH [Cost: 639, Rows: 10K (NO STATISTICS)] (PATH ID: 1) Inner (RESEGMENT)
|  Join Cond: (T1.a = T2.y)
|  Materialize at Output: T1.b
|  Execute on: All Nodes
| +-- Outer -> STORAGE ACCESS for T1 [Cost: 151, Rows: 10K (NO STATISTICS)] (PATH ID: 2)
| | |    Projection: public.T1_b0
| | |    Materialize: T1.a
| | |    Execute on: All Nodes
| +-- Inner -> STORAGE ACCESS for T2 [Cost: 302, Rows: 10K (NO STATISTICS)] (PATH ID: 3)
| | |    Projection: public.T2_b0
| | |    Materialize: T2.x, T2.y
| | |    Execute on: All Nodes
```

Analytic function path

Vertica attempts to optimize multiple SQL-99 Analytic functions from the same query by grouping them together in Analytic Group areas.

For each analytical group, Vertica performs a distributed sort and resegment of the data, if necessary.

You can tell how many sorts and resegments are required based on the query plan.

For example, the following query plan shows that the FIRST_VALUE and LAST_VALUE functions are in the same analytic group because their OVER clause is the same. In contrast, ROW_NUMBER() has a different ORDER BY clause, so it is in a different analytic group. Because both groups share the same PARTITION BY deal_stage clause, the data does not need to be resegmented between groups :

```
EXPLAIN SELECT
 first_value(deal_size) OVER (PARTITION BY deal_stage
 ORDER BY deal_size),
 last_value(deal_size) OVER (PARTITION BY deal_stage
 ORDER BY deal_size),
 row_number() OVER (PARTITION BY deal_stage
 ORDER BY largest_bill_amount)
 FROM customer_dimension;

 Access Path:
+-ANALYTICAL [Cost: 1K, Rows: 50K] (PATH ID: 1)
|  Analytic Group
|   Functions: row_number()
|   Group Sort: customer_dimension.deal_stage ASC, customer_dimension.largest_bill_amount ASC NULLS LAST
|  Analytic Group
|   Functions: first_value(), last_value()
|   Group Filter: customer_dimension.deal_stage
|   Group Sort: customer_dimension.deal_stage ASC, customer_dimension.deal_size ASC NULL LAST
|  Execute on: All Nodes
| +---> STORAGE ACCESS for customer_dimension [Cost: 263, Rows: 50K]
        (PATH ID: 2)
| |      Projection: public.customer_dimension_DBD_1_rep_vmart_vmart_node0001
| |      Materialize: customer_dimension.largest_bill_amount,
         customer_dimension.deal_stage, customer_dimension.deal_size
| |      Execute on: All Nodes
```

See also

Invoking analytic functions

Node down information

Vertica provides performance optimization when cluster nodes fail by distributing the work of the down nodes uniformly among available nodes throughout the cluster.

When a node in your cluster is down, the query plan identifies which node the query will execute on. To help you quickly identify down nodes on large clusters, EXPLAIN output lists up to six nodes, if the number of running nodes is less than or equal to six, and lists only down nodes if the number of running nodes is more than six.

> **Note**
> The node that executes down node queries is not always the same one.

The following table provides more detail:

Node state	EXPLAIN output
If all nodes are up, EXPLAIN output indicates All Nodes .	Execute on: All Nodes
If fewer than 6 nodes are up, EXPLAIN lists up to six running nodes.	Execute on: [node_list].
If more than 6 nodes are up, EXPLAIN lists only non-running nodes.	Execute on: All Nodes Except [node_list]
If the node list contains non-ephemeral nodes, the EXPLAIN output indicates All Permanent Nodes .	Execute on: All Permanent Nodes
If the path is being run on the query initiator, the EXPLAIN output indicates Query Initiator .	Execute on: Query Initiator

Examples

In the following example, the down node is v_vmart_node0005 , and the node v_vmart_node0006 will execute this run of the query.

```
=> EXPLAIN SELECT * FROM test;
QUERY PLAN
------------------------------------------------------------------------------
QUERY PLAN DESCRIPTION:
------------------------------
EXPLAIN SELECT * FROM my1table;
Access Path:
+-STORAGE ACCESS for my1table [Cost: 10, Rows: 2] (PATH ID: 1)
| Projection: public.my1table_b0
| Materialize: my1table.c1, my1table.c2
| Execute on: All Except v_vmart_node0005
+-STORAGE ACCESS for my1table (REPLACEMENT FOR DOWN NODE) [Cost: 66, Rows: 2]
| Projection: public.my1table_b1
| Materialize: my1table.c1, my1table.c2
| Execute on: v_vmart_node0006
```

The All Permanent Nodes output in the following example fragment denotes that the node list is for permanent (non-ephemeral) nodes only:

```
=> EXPLAIN SELECT * FROM my2table;
Access Path:
+-STORAGE ACCESS for my2table [Cost: 18, Rows:6 (NO STATISTICS)] (PATH ID: 1)
|  Projection: public.my2tablee_b0
|  Materialize: my2table.x, my2table.y, my2table.z
|  Execute on: All Permanent Nodes
```

MERGE path

Vertica prepares an optimized query plan for a MERGE statement if the statement and its tables meet the criteria described in MERGE optimization .

Use the EXPLAIN keyword to determine whether Vertica can produce an optimized query plan for a given MERGE statement. If optimization is possible, the EXPLAIN -generated output contains a [Semi] path, as shown in the following sample fragment:

```
...
Access Path:
+-DML DELETE [Cost: 0, Rows: 0]
|  Target Projection: public.A_b1 (DELETE ON CONTAINER)
|  Target Prep:
|  Execute on: All Nodes
| +---> JOIN MERGEJOIN(inputs presorted) [Semi] [Cost: 6, Rows: 1 (NO STATISTICS)] (PATH ID: 1)
        Inner (RESEGMENT)
| |      Join Cond: (A.a1 = VAL(2))
| |      Execute on: All Nodes
| | +-- Outer -> STORAGE ACCESS for A [Cost: 2, Rows: 2 (NO STATISTICS)] (PATH ID: 2)
...
```

Conversely, if Vertica cannot create an optimized plan, EXPLAIN -generated output contains RightOuter path:

```
...
Access Path: +-DML MERGE
|  Target Projection: public.locations_b1
|  Target Projection: public.locations_b0
|  Target Prep:
|  Execute on: All Nodes
| +---> JOIN MERGEJOIN(inputs presorted) [RightOuter] [Cost: 28, Rows: 3 (NO STATISTICS)] (PATH ID: 1) Outer (RESEGMENT) Inner (RESEGMENT)
| |      Join Cond: (locations.user_id = VAL(2)) AND (locations.location_x = VAL(2)) AND (locations.location_y = VAL(2))
| |      Execute on: All Nodes
| | +-- Outer -> STORAGE ACCESS for <No Alias> [Cost: 15, Rows: 2 (NO STATISTICS)] (PATH ID: 2)
...
```

Directed queries

Directed queries encapsulate information that the optimizer can use to create a query plan. Directed queries can serve the following goals:

- Preserve current query plans before a scheduled upgrade. In most instances, queries perform more efficiently after a Vertica upgrade. In the few cases where this is not so, you can use directed queries that you created before upgrading, to recreate query plans from the earlier version.
- Enable you to create query plans that improve optimizer performance. Occasionally, you might want to influence the optimizer to make better choices in executing a given query. For example, you can choose a different projection, or force a different join order. In this case, you can use a directed query to create a query plan that preempts any plan that the optimizer might otherwise create.
- Redirect an input query to a query that uses different semantics—for example, map a join query to a SELECT statement that queries a flattened table .

Directed query components

A directed query pairs two components:

- **Input query** : A query that triggers use of this directed query when it is active.
- **Annotated query** : A SQL statement with embedded optimizer hints, which instruct the optimizer how to create a query plan for the specified input query. These hints specify important query plan elements, such as join order and projection choices.

> **Tip**
> You can also use most optimizer hints directly in vsql. For details, see Hints .

Vertica provides two methods for creating directed queries:

- The optimizer can generate an annotated query from a given input query and pair the two as a directed query.
- You can write your own annotated query and pair it with an input query.

For a description of both methods, see Creating directed queries .

In this section

- Creating directed queries
- Setting hints in annotated queries
- Ignoring constants in directed queries
- Rewriting queries
- Managing directed queries
- Batch query plan export
- Half join and cross join semantics
- Directed query restrictions

Creating directed queries

CREATE DIRECTED QUERY associates an input query with a query annotated with optimizer hints. It stores the association under a unique identifier.

CREATE DIRECTED QUERY has two variants:

- CREATE DIRECTED QUERY OPTIMIZER directs the query optimizer to generate annotated SQL from the specified input query. The annotated query contains hints that the optimizer can use to recreate its current query plan for that input query.
- CREATE DIRECTED QUERY CUSTOM specifies an annotated query supplied by the user. Vertica associates the annotated query with the input query specified by the last SAVE QUERY statement.

In both cases, Vertica associates the annotated query and input query, and registers their association in the system table DIRECTED_QUERIES under query_name .

The two approaches can be used together : you can use the annotated SQL that the optimizer creates as the basis for creating your own (custom) directed queries.

In this section

- Optimizer-generated directed queries
- Custom directed queries
- Using optimizer-generated and custom directed queries together

Optimizer-generated directed queries

CREATE DIRECTED QUERY OPTIMIZER passes an input query to the optimizer, which generates an annotated query from its own query plan. It then pairs the input and annotated queries and saves them as a directed query. This directed query can be used to handle other queries that are identical

except for the predicate strings on which query results are filtered.

You can use optimizer-generated directed queries to capture query plans before you upgrade. Doing so can be especially useful if you detect diminished performance of a given query after the upgrade. In this case, you can use the corresponding directed query to recreate an earlier query plan, and compare its performance to the plan generated by the current optimizer.

You can also create multiple optimizer-generated directed queries from the most frequently executed queries, by invoking the meta-function SAVE_PLANS . For details, see Bulk-Creation of Directed Queries .

Example

The following SQL statements create and activate the directed query findEmployeesCityJobTitle_OPT :

```
=> CREATE DIRECTED QUERY OPTIMIZER 'findEmployeesCityJobTitle_OPT'
    SELECT employee_first_name, employee_last_name FROM public.employee_dimension
    WHERE employee_city='Boston' and job_title='Cashier' ORDER BY employee_last_name, employee_first_name;
CREATE DIRECTED QUERY

=> ACTIVATE DIRECTED QUERY findEmployeesCityJobTitle_OPT;
ACTIVATE DIRECTED QUERY
```

After this directed query plan is activated, the optimizer uses it to generate a query plan for all subsequent invocations of this input query, and others like it. You can view the optimizer-generated annotated query by calling GET DIRECTED QUERY or querying system table DIRECTED_QUERIES :

```
=> SELECT input_query, annotated_query FROM V_CATALOG.DIRECTED_QUERIES
     WHERE query_name = 'findEmployeesCityJobTitle_OPT';
-[ RECORD 1 ]---+-------------------------------------------------------------------------------
input_query     | SELECT employee_dimension.employee_first_name, employee_dimension.employee_last_name FROM public.employee_dimension
WHERE ((employee_dimension.employee_city = 'Boston'::varchar(6) /*+:v(1)*/) AND (employee_dimension.job_title = 'Cashier'::varchar(7) /*+:v(2)*/))
ORDER BY employee_dimension.employee_last_name, employee_dimension.employee_first_name
annotated_query | SELECT /*+verbatim*/ employee_dimension.employee_first_name AS employee_first_name, employee_dimension.employee_last_name
AS employee_last_name FROM public.employee_dimension AS employee_dimension/*+projs('public.employee_dimension')*/
WHERE (employee_dimension.employee_city = 'Boston'::varchar(6) /*+:v(1)*/) AND (employee_dimension.job_title = 'Cashier'::varchar(7) /*+:v(2)*/)
ORDER BY 2 ASC, 1 ASC
```

In this case, the annotated query includes the following hints:

- /*+verbatim*/ specifies to execute the annotated query exactly as written and produce a query plan accordingly.
- /*+projs('public.Emp_Dimension')*/ directs the optimizer to create a query plan that uses the projection public.Emp_Dimension .
- /*+:v(*n*)*/ (alias of /*+IGNORECONST(*n*)*/) is included several times in the annotated and input queries. These hints qualify two constants in the query predicates: Boston and Cashier . Each :v hint has an integer argument *n* that pairs corresponding constants in the input and annotated query queries: *+:v(1)*/ for Boston , and /*+:v(2)*/ for Cashier . The hints tell the optimizer to disregard these constants when it decides whether to apply this directed query to other input queries that are similar. Thus, ignore constant hints can let you use the same directed query for different input queries.

The following query uses different values for the columns employee_city and job_title , but is otherwise identical to the original input query of directed query EmployeesCityJobTitle_OPT :

```
=> SELECT employee_first_name, employee_last_name FROM public.employee_dimension
   WHERE employee_city = 'San Francisco' and job_title = 'Branch Manager' ORDER BY employee_last_name, employee_first_name;
```

If the directed query EmployeesCityJobTitle_OPT is active, the optimizer can use it for this query:

```
=> EXPLAIN SELECT employee_first_name, employee_last_name FROM employee_dimension
    WHERE employee_city='San Francisco' AND job_title='Branch Manager' ORDER BY employee_last_name, employee_first_name;
...
------------------------------
QUERY PLAN DESCRIPTION:
------------------------------
EXPLAIN SELECT employee_first_name, employee_last_name FROM employee_dimension WHERE employee_city='San Francisco' AND job_title='Branch Manager' ORDER BY employee_last_name, employee_first_name;

The following active directed query(query name: findEmployeesCityJobTitle_OPT) is being executed:
SELECT /*+verbatim*/  employee_dimension.employee_first_name, employee_dimension.employee_last_name
FROM public.employee_dimension employee_dimension/*+projs('public.employee_dimension')*/
WHERE ((employee_dimension.employee_city = 'San Francisco'::varchar(13)) AND (employee_dimension.job_title = 'Branch Manager'::varchar(14)))
ORDER BY employee_dimension.employee_last_name, employee_dimension.employee_first_name

Access Path:
+-SORT [Cost: 222, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
|  Order: employee_dimension.employee_last_name ASC, employee_dimension.employee_first_name ASC
|  Execute on: All Nodes
| +---> STORAGE ACCESS for employee_dimension [Cost: 60, Rows: 10K (NO STATISTICS)] (PATH ID: 2)
| |      Projection: public.employee_dimension_super
| |      Materialize: employee_dimension.employee_first_name, employee_dimension.employee_last_name
| |      Filter: (employee_dimension.employee_city = 'San Francisco')
| |      Filter: (employee_dimension.job_title = 'Branch Manager')
| |      Execute on: All Nodes
...
```

Bulk-creation of directed queries

The meta-function SAVE_PLANS lets you create multiple optimizer-generated directed queries from the most frequently executed queries. SAVE_PLANS works as follows:

1. Iterates over all queries in the data collector table dc_requests_issued and selects the most-frequently requested queries, up to the maximum specified by its *query-budget* argument. If the meta-function's *since-date* argument is also set, then SAVE_PLANS iterates only over queries that were issued on or after the specified date.
   As SAVE_PLANS iterates over dc_requests_issued , it tests queries against various restrictions. In general, directed queries support only SELECT statements as input. Within this broad requirement, input queries are subject to other restrictions .
2. Calls CREATE DIRECTED QUERY OPTIMIZER on all qualifying input queries, which creates a directed query for each unique input query as described above .
3. Saves metadata on the new set of directed queries to system table DIRECTED_QUERIES , where all directed queries of that set share the same SAVE_PLANS_VERSION integer. This integer is computed from the highest SAVE_PLANS_VERSION + 1.

You can later use SAVE_PLANS_VERSION identifiers to bulk activate , deactivate , and drop directed queries. For example:

```
=> SELECT save_plans (40);
                                        save_plans
-----------------------------------------------------------------------------------------------------------
 9 directed query supported queries out of 40 most frequently run queries were saved under the save_plans_version 3.

To view the saved queries, run:

       SELECT * FROM directed_queries WHERE save_plans_version = '3';

To drop the saved queries, run:

       DROP DIRECTED QUERY WHERE save_plans_version = '3';
(1 row)

=> SELECT input_query::VARCHAR(60) FROM directed_queries WHERE save_plans_version = 3 AND input_query ILIKE '%line_search%';
                    input_query
-----------------------------------------------------------
 SELECT public.line_search_logistic2(udtf1.deviance, udtf1.G
 SELECT public.line_search_logistic2(udtf1.deviance, udtf1.G
(2 rows)

=> ACTIVATE DIRECTED QUERY WHERE save_plans_version = 3 AND input_query ILIKE '%line_search%';
ACTIVATE DIRECTED QUERY
=> SELECT query_name, input_query::VARCHAR(60), is_active FROM directed_queries WHERE save_plans_version = 3 AND input_query ILIKE
'%line_search%';
      query_name       |                    input_query                     | is_active
-----------------------+----------------------------------------------------+-----------
 save_plans_nolabel_3_3 | SELECT public.line_search_logistic2(udtf1.deviance, udtf1.G  | t
 save_plans_nolabel_6_3 | SELECT public.line_search_logistic2(udtf1.deviance, udtf1.G  | t
(2 rows)
```

Note

query_name values are concatenated from the following strings:

```
save_plans_query-label_query-number_save-plans-version
```

where:

- *query-label* is a LABEL hint embedded in the input query associated with this directed query. If theinput query contains no label, then this string is set to nolabel .
- *query-number* is an integer in a continuous sequence between 0 and *budget-query* , which uniquely identifies this directed query from others in the same SAVE_PLANS-generated set.
- *[save-plans-version](/sql-reference/system-tables/v-catalog-schema/directed-queries.html#SAVE_PLANS_VERSION)* identifies the set of directed queries to which this directed query belongs.

Custom directed queries

CREATE DIRECTED QUERY CUSTOM specifies an annotated query and pairs it to an input query previously saved by SAVE QUERY . You must issue both statements in the same user session.

For example, you might want a query to use a specific projection:

1. Specify the query with SAVE QUERY:

```
=> SAVE QUERY SELECT employee_first_name, employee_last_name FROM employee_dimension
   WHERE employee_city='Boston' AND job_title='Cashier';
SAVE QUERY
```

Note

The input query that you supply to SAVE QUERY only supports the :v hint.

2. Create a custom directed query with CREATE DIRECTED QUERY CUSTOM, which specifies an annotated query and associates it with the saved query. The annotated query includes a /*+projs*/ hint, which instructs the optimizer to use the projection public.emp_dimension_unseg when users call the saved query:

```
=> CREATE DIRECTED QUERY CUSTOM 'findBostonCashiers_CUSTOM'
   SELECT employee_first_name, employee_last_name
   FROM employee_dimension /*+Projs('public.emp_dimension_unseg')*/
   WHERE employee_city='Boston' AND job_title='Cashier';
CREATE DIRECTED QUERY
```

Caution

Vertica associates a saved query and annotated query without checking whether the input and annotated queries are compatible. Be careful to sequence SAVE QUERY and CREATE DIRECTED QUERY CUSTOM statements so the saved and directed queries are correctly matched.

3. Activate the directed query:

```
=> ACTIVATE DIRECTED QUERY findBostonCashiers_CUSTOM;
ACTIVATE DIRECTED QUERY
```

4. After activation, the optimizer uses this directed query to generate a query plan for all subsequent invocations of its input query. The following EXPLAIN output verifies the optimizer's use of this directed query and the projection it specifies:

```
=> EXPLAIN SELECT employee_first_name, employee_last_name FROM employee_dimension
   WHERE employee_city='Boston' AND job_title='Cashier';

QUERY PLAN
------------------------------
QUERY PLAN DESCRIPTION:
------------------------------
EXPLAIN SELECT employee_first_name, employee_last_name FROM employee_dimension where employee_city='Boston' AND job_title='Cashier';

 The following active directed query(query name: findBostonCashiers_CUSTOM) is being executed:
 SELECT employee_dimension.employee_first_name, employee_dimension.employee_last_name
  FROM public.employee_dimension/*+Projs('public.emp_dimension_unseg')*/
  WHERE ((employee_dimension.employee_city = 'Boston'::varchar(6)) AND (employee_dimension.job_title = 'Cashier'::varchar(7)))

 Access Path:
 +-STORAGE ACCESS for employee_dimension [Cost: 158, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
 |  Projection: public.emp_dimension_unseg
 |  Materialize: employee_dimension.employee_first_name, employee_dimension.employee_last_name
 |  Filter: (employee_dimension.employee_city = 'Boston')
 |  Filter: (employee_dimension.job_title = 'Cashier')
 |  Execute on: Query Initiator
```

See also

Rewriting Join Queries

Using optimizer-generated and custom directed queries together

You can use the annotated SQL that the optimizer creates as the basis for creating your own custom directed queries. This approach can be especially useful in evaluating the plan that the optimizer creates to handle a given query, and testing plan modifications.

For example, you might want to modify how the optimizer implements the following query:

```
=> SELECT COUNT(customer_name) Total, customer_region Region
   FROM (store_sales s JOIN customer_dimension c ON c.customer_key = s.customer_key)
   JOIN product_dimension p ON s.product_key = p.product_key
   WHERE p.category_description ilike '%Medical%'
     AND p.product_description ilike '%antibiotics%'
     AND c.customer_age <= 30 AND YEAR(s.sales_date)=2017
   GROUP BY customer_region;
```

When you run EXPLAIN on this query, you discover that the optimizer uses projection customers_proj_age for the customer_dimension table. This projection is sorted on column customer_age . Consequently, the optimizer hash-joins the tables store_sales and customer_dimension on customer_key .

After analyzing customer_dimension table data, you observe that most customers are under 30, so it makes more sense to use projection customer_proj_id for the customer_dimension table, which is sorted on customer_key :

You can create a directed query that encapsulates this change as follows:

1. Obtain optimizer-generated annotations on the query with EXPLAIN ANNOTATED:

```
=> \o annotatedQuery
=> EXPLAIN ANNOTATED SELECT COUNT(customer_name) Total, customer_region Region
     FROM (store_sales s JOIN customer_dimension c ON c.customer_key = s.customer_key)
     JOIN product_dimension p ON s.product_key = p.product_key
     WHERE p.category_description ilike '%Medical%'
       AND p.product_description ilike '%antibiotics%'
       AND c.customer_age <= 30 AND YEAR(s.sales_date)=2017
     GROUP BY customer_region;
=> \o
=> \! cat annotatedQuery
...
SELECT /*+syntactic_join,verbatim*/ count(c.customer_name) AS Total, c.customer_region AS Region
 FROM ((public.store_sales AS s/*+projs('public.store_sales_super')*/
    JOIN /*+Distrib(L,B),JType(H)*/ public.customer_dimension AS c/*+projs('public.customers_proj_age')*/
      ON (c.customer_key = s.customer_key))
    JOIN /*+Distrib(L,B),JType(M)*/ public.product_dimension AS p/*+projs('public.product_dimension')*/
      ON (s.product_key = p.product_key))
 WHERE ((date_part('year'::varchar(4), (s.sales_date)::timestamp(0)))::int = 2017)
     AND (c.customer_age <= 30)
     AND ((p.category_description)::varchar(32) ~~* '%Medical%'::varchar(9))
     AND (p.product_description ~~* '%antibiotics%'::varchar(13))
 GROUP BY  /*+GByType(Hash)*/ 2
(4 rows)
```

2. Modify the annotated query:

```
SELECT /*+syntactic_join,verbatim*/ count(c.customer_name) AS Total, c.customer_region AS Region
 FROM ((public.store_sales AS s/*+projs('public.store_sales_super')*/
    JOIN /*+Distrib(L,B),JType(H)*/ public.customer_dimension AS c/*+projs('public.customer_proj_id')*/
      ON (c.customer_key = s.customer_key))
    JOIN /*+Distrib(L,B),JType(H)*/ public.product_dimension AS p/*+projs('public.product_dimension')*/
      ON (s.product_key = p.product_key))
 WHERE ((date_part('year'::varchar(4), (s.sales_date)::timestamp(0)))::int = 2017)
     AND (c.customer_age <= 30)
     AND ((p.category_description)::varchar(32) ~~* '%Medical%'::varchar(9))
     AND (p.product_description ~~* '%antibiotics%'::varchar(13))
 GROUP BY  /*+GByType(Hash)*/ 2
```

3. Use the modified annotated query to create the desired directed query:
   - Save the desired input query with SAVE QUERY :

```
=> SAVE QUERY SELECT COUNT(customer_name) Total, customer_region Region
   FROM (store_sales s JOIN customer_dimension c ON c.customer_key = s.customer_key)
   JOIN product_dimension p ON s.product_key = p.product_key
   WHERE p.category_description ilike '%Medical%'
     AND p.product_description ilike '%antibiotics%'
     AND c.customer_age <= 30 AND YEAR(s.sales_date)=2017
   GROUP BY customer_region;
```

- Create a custom directed query that associates the saved input query with the modified annotated query:

```
=> CREATE DIRECTED QUERY CUSTOM 'getCustomersUnder31'
  SELECT /*+syntactic_join,verbatim*/ count(c.customer_name) AS Total, c.customer_region AS Region
FROM ((public.store_sales AS s/*+projs('public.store_sales_super')*/
   JOIN /*+Distrib(L,B),JType(H)*/ public.customer_dimension AS c/*+projs('public.customer_proj_id')*/
     ON (c.customer_key = s.customer_key))
   JOIN /*+Distrib(L,B),JType(H)*/ public.product_dimension AS p/*+projs('public.product_dimension')*/
     ON (s.product_key = p.product_key))
WHERE ((date_part('year'::varchar(4), (s.sales_date)::timestamp(0)))::int = 2017)
    AND (c.customer_age <= 30)
    AND ((p.category_description)::varchar(32) ~~* '%Medical%'::varchar(9))
    AND (p.product_description ~~* '%antibiotics%'::varchar(13))
GROUP BY  /*+GByType(Hash)*/ 2;
CREATE DIRECTED QUERY
```

4. Activate this directed query:

```
=> ACTIVATE DIRECTED QUERY getCustomersUnder31;
ACTIVATE DIRECTED QUERY
```

When the optimizer processes a query that matches this directed query's input query, it uses the directed query's annotated query to generate a query plan:

```
=> EXPLAIN SELECT COUNT(customer_name) Total, customer_region Region
   FROM (store_sales s JOIN customer_dimension c ON c.customer_key = s.customer_key)
   JOIN product_dimension p ON s.product_key = p.product_key
   WHERE p.category_description ilike '%Medical%'
     AND p.product_description ilike '%antibiotics%'
     AND c.customer_age <= 30 AND YEAR(s.sales_date)=2017
   GROUP BY customer_region;

The following active directed query(query name: getCustomersUnder31) is being executed:
...
```

Setting hints in annotated queries

The hints in a directed query's annotated query provide the optimizer with instructions how to execute an input query. Annotated queries support the following hints:

- *Join hints* specify join order, join type, and join data distribution: SYNTACTIC_JOIN , DISTRIB , JTYPE , UTYPE .
- *Table hints* specify which projections to include and exclude in the query plan: PROJS , SKIP_PROJS .
- :v and its alias IGNORECONSTANT marks predicate string constants that you want the optimizer to ignore when it decides whether to use a directed query for a given input query. For details, see Ignoring constants in directed queries .
- :c hints mark predicate constants that must not be ignored .
- VERBATIM enforces execution of an annotated query exactly as written.

Other hints in annotated queries such as DIRECT or LABEL have no effect.

You can use hints in a vsql query the same as in an annotated query, with two exceptions: :v (IGNORECONSTANT) and VERBATIM.

Ignoring constants in directed queries

Optimizer-generated directed queries generally include one or more :v (alias of IGNORECONSTANT) hints, which mark predicate string constants that you want the optimizer to ignore when it decides whether to use a directed query for a given input query. :v hints enable multiple queries to use the same directed query, provided the queries are identical in all respects except their predicate strings.

For example, the following two queries are identical , except for the string constants Boston|San Francisco and Cashier|Branch Manager , which are specified for columns employee_city and job_title , respectively:

```
=> SELECT employee_first_name, employee_last_name FROM public.employee_dimension
    WHERE employee_city='Boston' and job_title ='Cashier' ORDER BY employee_last_name, employee_first_name;

=> SELECT employee_first_name, employee_last_name FROM public.employee_dimension
    WHERE employee_city = 'San Francisco' and job_title = 'Branch Manager' ORDER BY employee_last_name, employee_first_name;
```

In this case, an optimizer-generated directed query that you create from one query can be used for both:

```
=> CREATE DIRECTED QUERY OPTIMIZER 'findEmployeesCityJobTitle_OPT'
    SELECT employee_first_name, employee_last_name FROM public.employee_dimension
    WHERE employee_city='Boston' and job_title='Cashier' ORDER BY employee_last_name, employee_first_name;
CREATE DIRECTED QUERY

=> ACTIVATE DIRECTED QUERY findEmployeesCityJobTitle_OPT;
ACTIVATE DIRECTED QUERY
```

The directed query's input and annotated queries both include :v hints:

```
=> SELECT input_query, annotated_query FROM V_CATALOG.DIRECTED_QUERIES
     WHERE query_name = 'findEmployeesCityJobTitle_OPT';
-[ RECORD 1 ]---+-------------------------------------------------------------------------------
input_query     | SELECT employee_dimension.employee_first_name, employee_dimension.employee_last_name FROM public.employee_dimension
WHERE ((employee_dimension.employee_city = 'Boston'::varchar(6) /*+:v(1)*/) AND (employee_dimension.job_title = 'Cashier'::varchar(7) /*+:v(2)*/))
ORDER BY employee_dimension.employee_last_name, employee_dimension.employee_first_name
annotated_query | SELECT /*+verbatim*/ employee_dimension.employee_first_name AS employee_first_name, employee_dimension.employee_last_name
AS employee_last_name FROM public.employee_dimension AS employee_dimension/*+projs('public.employee_dimension')*/
WHERE (employee_dimension.employee_city = 'Boston'::varchar(6) /*+:v(1)*/) AND (employee_dimension.job_title = 'Cashier'::varchar(7) /*+:v(2)*/)
ORDER BY 2 ASC, 1 ASC
```

The hint arguments in the input and annotated queries pair two predicate constants:

- `/*+:v(1)*/` pairs input and annotated query settings for employee_city .
- /*+:v(2)*/ pairs input and annotated query settings for job_title .

The :v hints tell the optimizer to ignore values for these two columns when it decides whether it can use this directed query for a given input query.

For example, the following query uses different values for employee_city and job_title , but is otherwise identical to the query used to create the directed query EmployeesCityJobTitle_OPT :

```
=> SELECT employee_first_name, employee_last_name FROM public.employee_dimension
    WHERE employee_city = 'San Francisco' and job_title = 'Branch Manager' ORDER BY employee_last_name, employee_first_name;
```

If the directed query EmployeesCityJobTitle_OPT is active, the optimizer can use it in its query plan for this query:

```
=> EXPLAIN SELECT employee_first_name, employee_last_name FROM employee_dimension
    WHERE employee_city='San Francisco' AND job_title='Branch Manager' ORDER BY employee_last_name, employee_first_name;
...
------------------------------
QUERY PLAN DESCRIPTION:
------------------------------
EXPLAIN SELECT employee_first_name, employee_last_name FROM employee_dimension WHERE employee_city='San Francisco' AND job_title='Branch Manager' ORDER BY employee_last_name, employee_first_name;

The following active directed query(query name: findEmployeesCityJobTitle_OPT) is being executed:
SELECT /*+verbatim*/  employee_dimension.employee_first_name, employee_dimension.employee_last_name
FROM public.employee_dimension employee_dimension/*+projs('public.employee_dimension')*/
WHERE ((employee_dimension.employee_city = 'San Francisco'::varchar(13)) AND (employee_dimension.job_title = 'Branch Manager'::varchar(14)))
ORDER BY employee_dimension.employee_last_name, employee_dimension.employee_first_name

Access Path:
+-SORT [Cost: 222, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
|  Order: employee_dimension.employee_last_name ASC, employee_dimension.employee_first_name ASC
|  Execute on: All Nodes
| +---> STORAGE ACCESS for employee_dimension [Cost: 60, Rows: 10K (NO STATISTICS)] (PATH ID: 2)
| |      Projection: public.employee_dimension_super
| |      Materialize: employee_dimension.employee_first_name, employee_dimension.employee_last_name
| |      Filter: (employee_dimension.employee_city = 'San Francisco')
| |      Filter: (employee_dimension.job_title = 'Branch Manager')
| |      Execute on: All Nodes
...
```

Mapping one-to-many :v hints

The examples shown so far demonstrate one-to-one pairings of :v hints. You can also use :v hints to map one input constant to multiple constants in an annotated query. This approach can be especially useful when you want to provide the optimizer with explicit instructions how to execute a query that joins tables.

For example, the following query joins two tables:

```
SELECT * FROM S JOIN T ON S.a = T.b WHERE S.a = 8;
```

In this case, the optimizer can infer that S.a and T.b have the same value and implements the join accordingly:

```
<a name="simpleJoinExample"></a>=> CREATE DIRECTED QUERY OPTIMIZER simpleJoin SELECT * FROM S JOIN T ON S.a = T.b WHERE S.a = 8;
CREATE DIRECTED QUERY
=> SELECT input_query, annotated_query FROM directed_queries WHERE query_name = 'simpleJoin';
-[ RECORD 1 ]---+-----------------------------------------------------------------------------------------------------------------------------
input_query     | SELECT S.a, T.b FROM (public.S JOIN public.T ON ((S.a = T.b))) WHERE (S.a = 8 /*+:v(1)*/)
annotated_query | SELECT /*+syntactic_join,verbatim*/ S.a AS a, T.b AS b
FROM (public.S AS S/*+projs('public.S')*/ JOIN /*+Distrib(L,L),JType(M)*/ public.T AS T/*+projs('public.T')*/  ON (S.a = T.b))
WHERE (S.a = 8 /*+:v(1)*/) AND (T.b = 8 /*+:v(1)*/)
(1 row)

=> ACTIVATE DIRECTED QUERY simpleJoin;
ACTIVATED DIRECTED QUERY
```

Now, given the following input query:

```
SELECT * FROM S JOIN T ON S.a = T.b WHERE S.a = 3;
```

the optimizer disregards the join predicate constants and uses the directed query simpleJoin in its query plan:

```
------------------------------
QUERY PLAN DESCRIPTION:
------------------------------

EXPLAIN SELECT * FROM S JOIN T ON S.a = T.b WHERE S.a = 3;

The following active directed query(query name: simpleJoin) is being executed:
SELECT /*+syntactic_join,verbatim*/  S.a, T.b FROM (public.S S/*+projs('public.S')*/ JOIN /*+Distrib('L', 'L'), JType('
M')*/public.T T/*+projs('public.T')*/ ON ((S.a = T.b))) WHERE ((S.a = 3) AND (T.b = 3))

Access Path:
+-JOIN MERGEJOIN(inputs presorted) [Cost: 21, Rows: 4 (NO STATISTICS)] (PATH ID: 1)
|  Join Cond: (S.a = T.b)
|  Execute on: Query Initiator
| +-- Outer -> STORAGE ACCESS for S [Cost: 12, Rows: 4 (NO STATISTICS)] (PATH ID: 2)
| |      Projection: public.S_b0
| |      Materialize: S.a
| |      Filter: (S.a = 3)
| |      Execute on: Query Initiator
| |      Runtime Filter: (SIP1(MergeJoin): S.a)
| +-- Inner -> STORAGE ACCESS for T [Cost: 8, Rows: 3 (NO STATISTICS)] (PATH ID: 3)
| |      Projection: public.T_b0
| |      Materialize: T.b
| |      Filter: (T.b = 3)
| |      Execute on: Query Initiator
...
```

Conserving predicate constants in directed queries

By default, optimizer-generated directed queries set :v hints on predicate constants. You can override this behavior by marking predicate constants that must not be ignored with :c hints. For example, the following statement creates a directed query that can be used only for input queries where the join predicate constant is the same as in the original input query— 8 :

```
=> CREATE DIRECTED QUERY OPTIMIZER simpleJoin_KeepPredicateConstant SELECT * FROM S JOIN T ON S.a = T.b WHERE S.a = 8 /*+:c*/;
CREATE DIRECTED QUERY
=> ACTIVATE DIRECTED QUERY simpleJoin_KeepPredicateConstant;
```

The following query on system table DIRECTED_QUERIES compares directed queries simpleJoin (created in an earlier example) and simpleJoin_KeepPredicateConstant . Unlike simpleJoin , the join predicate of the input and annotated queries for simpleJoin_KeepPredicateConstant omit :v hints:

```
=> SELECT query_name, input_query, annotated_query FROM directed_queries WHERE query_name ILIKE'%simpleJoin%';
-[ RECORD 1 ]---+
query_name      | simpleJoin
input_query     | SELECT S.a, T.b FROM (public.S JOIN public.T ON ((S.a = T.b))) WHERE (S.a = 8 /*+:v(1)*/)
annotated_query | SELECT /*+syntactic_join,verbatim*/ S.a AS a, T.b AS b
FROM (public.S AS S/*+projs('public.S')*/ JOIN /*+Distrib(L,L),JType(M)*/ public.T AS T/*+projs('public.T')*/  ON (S.a = T.b))
WHERE (S.a = 8 /*+:v(1)*/) AND (T.b = 8 /*+:v(1)*/)
-[ RECORD 2 ]---+
query_name      | simpleJoin_KeepPredicateConstant
input_query     | SELECT S.a, T.b FROM (public.S JOIN public.T ON ((S.a = T.b))) WHERE (S.a = 8)
annotated_query | SELECT /*+syntactic_join,verbatim*/ S.a AS a, T.b AS b
FROM (public.S AS S/*+projs('public.S')*/ JOIN /*+Distrib(L,L),JType(M)*/ public.T AS T/*+projs('public.T')*/  ON (S.a = T.b))
WHERE (S.a = 8) AND (T.b = 8)
```

If you deactivate directed query simpleJoin (which would otherwise have precedence over simpleJoin_KeepPredicateConstant because it was created earlier), Vertica only applies simpleJoin_KeepPredicateConstant on input queries where the join predicate constant is the same as in the original input query. For example, compare the following two query plans:

```
=> EXPLAIN SELECT * FROM S JOIN T ON S.a = T.b WHERE S.a = 8;
 ...
 ------------------------------
 QUERY PLAN DESCRIPTION:
 ------------------------------

 EXPLAIN SELECT * FROM S JOIN T ON S.a = T.b WHERE S.a = 8;

 The following active directed query(query name: simpleJoin_KeepPredicateConstant) is being executed:
 SELECT /*+syntactic_join,verbatim*/  S.a, T.b FROM (public.S S/*+projs('public.S')*/ JOIN /*+Distrib('L', 'L'), JType('
M')*/public.T T/*+projs('public.T')*/ ON ((S.a = T.b))) WHERE ((S.a = 8) AND (T.b = 8))

 Access Path:
 +-JOIN MERGEJOIN(inputs presorted) [Cost: 21, Rows: 4 (NO STATISTICS)] (PATH ID: 1)
 |  Join Cond: (S.a = T.b)
 ...
```

```
=> EXPLAIN SELECT * FROM S JOIN T ON S.a = T.b WHERE S.a = 3
 ...
 ------------------------------
 QUERY PLAN DESCRIPTION:
 ------------------------------

 EXPLAIN SELECT * FROM S JOIN T ON S.a = T.b WHERE S.a = 3;

 Access Path:
 +-JOIN MERGEJOIN(inputs presorted) [Cost: 21, Rows: 4 (NO STATISTICS)] (PATH ID: 1)
 |  Join Cond: (S.a = T.b)
 ...
```

Rewriting queries

You can use directed queries to change the semantics of a given query—that is, substitute one query for another. This can be especially important when you have little or no control over the content and format of input queries that your Vertica database processes. You can map these queries to directed queries that rewrite the original input for optimal execution.

The following sections describe two use cases:

- Rewriting Join Queries
- Creating Query Templates

Rewriting join queries

Many of your input queries join multiple tables. You've determined that in many cases, it would be more efficient to denormalize much of your data in several flattened tables and query those tables directly. You cannot revise the input queries themselves. However, you can use directed queries to redirect join queries to the flattened table data.

For example, the following query aggregates regional sales of wine products by joining three tables in the VMart database:

```
=> SELECT SD.store_region AS Region, SD.store_city AS City, SUM(SF.gross_profit_dollar_amount) Total
    FROM store.store_sales_fact SF
    JOIN store.store_dimension SD ON SF.store_key=SD.store_key
    JOIN product_dimension P ON SF.product_key||SF.product_version=P.product_key||P.product_version
  WHERE P.product_description ILIKE '%wine%'
  GROUP BY ROLLUP (SD.store_region, SD.store_city)
  ORDER BY Region,Total DESC;
```

You can consolidate the joined table data in a single flattened table, and query this table instead. By doing so, you can access the same data faster. You can create the flattened table with the following DDL statements:

```
=> CREATE TABLE store.store_sales_wide AS SELECT * FROM store.store_sales_fact;
=> ALTER TABLE store.store_sales_wide ADD COLUMN store_name VARCHAR(64)
    SET USING (SELECT store_name FROM store.store_dimension WHERE store.store_sales_wide.store_key=store.store_dimension.store_key);
=> ALTER TABLE store.store_sales_wide ADD COLUMN store_city varchar(64)
   SET USING (SELECT store_city FROM store.store_dimension  WHERE store.store_sales_wide.store_key=store.store_dimension.store_key);
=> ALTER TABLE store.store_sales_wide ADD COLUMN store_state char(2)
   SET USING (SELECT store_state char FROM store.store_dimension WHERE store.store_sales_wide.store_key=store.store_dimension.store_key)
=> ALTER TABLE store.store_sales_wide ADD COLUMN store_region varchar(64)
   SET USING (SELECT store_region FROM store.store_dimension  WHERE store.store_sales_wide.store_key=store.store_dimension.store_key);
=> ALTER TABLE store.store_sales_wide ADD column product_description VARCHAR(128)
   SET USING (SELECT product_description FROM public.product_dimension
   WHERE store_sales_wide.product_key||store_sales_wide.product_version = product_dimension.product_key||product_dimension.product_version);
=> ALTER TABLE store.store_sales_wide ADD COLUMN sku_number char(32)
   SET USING (SELECT sku_number char FROM product_dimension
   WHERE store_sales_wide.product_key||store_sales_wide.product_version = product_dimension.product_key||product_dimension.product_version);

=> SELECT REFRESH_COLUMNS ('store.store_sales_wide','', 'rebuild');
```

After creating this table and refreshing its SET USING columns, you can rewrite the earlier query as follows:

```
=> SELECT SD.store_region AS Region, SD.store_city AS City, SUM(SF.gross_profit_dollar_amount) Total
   FROM store.store_sales_fact SF
   JOIN store.store_dimension SD ON SF.store_key=SD.store_key
   JOIN product_dimension P ON SF.product_key||SF.product_version=P.product_key||P.product_version
  WHERE P.product_description ILIKE '%wine%'
  GROUP BY ROLLUP (SD.store_region, SD.store_city)
  ORDER BY Region,Total DESC;
  Region   |       City       |  Total
-----------+------------------+---------
 East      |                  | 1679788
 East      | Boston           |  138494
 East      | Elizabeth        |  138071
 East      | Sterling Heights |  137719
 East      | Allentown        |  137122
 East      | New Haven        |  117751
 East      | Lowell           |  102670
 East      | Washington       |   84595
 East      | Charlotte        |   83255
 East      | Waterbury        |   81516
 East      | Erie             |   80784
 East      | Stamford         |   59935
 East      | Hartford         |   59843
 East      | Baltimore        |   55873
 East      | Clarksville      |   54117
 East      | Nashville        |   53757
 East      | Manchester       |   53290
 East      | Columbia         |   52799
 East      | Memphis          |   52648
 East      | Philadelphia     |   29711
 East      | Portsmouth       |   29316
 East      | New York         |   27033
 East      | Cambridge        |   26111
 East      | Alexandria       |   23378
 MidWest   |                  | 1073224
 MidWest   | Lansing          |  145616
 MidWest   | Livonia          |  129349
--More--
```

Querying the flattened table is more efficient; however, you still must account for input queries that continue to use the earlier join syntax. You can do so by creating a custom directed query , which redirects these input queries to syntax that targets the flattened table:

1. Save the input query :

```
=> SAVE QUERY SELECT SD.store_region AS Region, SD.store_city AS City, SUM(SF.gross_profit_dollar_amount) Total
   FROM store.store_sales_fact SF
   JOIN store.store_dimension SD ON SF.store_key=SD.store_key
   JOIN product_dimension P ON SF.product_key||SF.product_version=P.product_key||P.product_version
  WHERE P.product_description ILIKE '%wine%'
  GROUP BY ROLLUP (SD.store_region, SD.store_city)
  ORDER BY Region,Total DESC;
SAVE QUERY
```

2. Map the saved query to a directed query with the desired syntax, and activate the directed query:

```
=> CREATE DIRECTED QUERY CUSTOM 'RegionalSalesWine'
   SELECT store_region AS Region,
     store_city AS City,
     SUM(gross_profit_dollar_amount) AS Total
   FROM store.store_sales_wide
   WHERE product_description ILIKE '%wine%'
   GROUP BY ROLLUP (region, city)
   ORDER BY Region,Total DESC;
CREATE DIRECTED QUERY

=> ACTIVATE DIRECTED QUERY RegionalSalesWine;
ACTIVATE DIRECTED QUERY
```

When directed query RegionalSalesWine is active, the query optimizer maps all queries that match the original input format to the directed query, as shown in the following query plan:

```
=> EXPLAIN SELECT SD.store_region AS Region, SD.store_city AS City, SUM(SF.gross_profit_dollar_amount) Total
    FROM store.store_sales_fact SF
    JOIN store.store_dimension SD ON SF.store_key=SD.store_key
    JOIN product_dimension P ON SF.product_key||SF.product_version=P.product_key||P.product_version
  WHERE P.product_description ILIKE '%wine%'
  GROUP BY ROLLUP (SD.store_region, SD.store_city)
  ORDER BY Region,Total DESC;
...
The following active directed query(query name: RegionalSalesWine) is being executed:
SELECT store_sales_wide.store_region AS Region, store_sales_wide.store_city AS City, sum(store_sales_wide.gross_profit_dollar_amount) AS Total
 FROM store.store_sales_wide WHERE (store_sales_wide.product_description ~~* '%wine%'::varchar(6))
 GROUP BY GROUPING SETS((store_sales_wide.store_region, store_sales_wide.store_city), (store_sales_wide.store_region),())
 ORDER BY store_sales_wide.store_region, sum(store_sales_wide.gross_profit_dollar_amount) DESC

Access Path:
+-SORT [Cost: 2K, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
|  Order: store_sales_wide.store_region ASC, sum(store_sales_wide.gross_profit_dollar_amount) DESC
|  Execute on: All Nodes
| +---> GROUPBY HASH (GLOBAL RESEGMENT GROUPS) (LOCAL RESEGMENT GROUPS) [Cost: 2K, Rows: 10K (NO STATISTICS)] (PATH ID: 2)
| |      Aggregates: sum(store_sales_wide.gross_profit_dollar_amount)
| |      Group By: store_sales_wide.store_region, store_sales_wide.store_city
| |      Grouping Sets: (store_sales_wide.store_region, store_sales_wide.store_city, <SVAR>), (store_sales_wide.store_region, <SVAR>), (<SVAR>)
| |      Execute on: All Nodes
| | +---> STORAGE ACCESS for store_sales_wide [Cost: 864, Rows: 10K (NO STATISTICS)] (PATH ID: 3)
| | |      Projection: store.store_sales_wide_b0
| | |      Materialize: store_sales_wide.gross_profit_dollar_amount, store_sales_wide.store_city, store_sales_wide.store_region
| | |      Filter: (store_sales_wide.product_description ~~* '%wine%')
| | |      Execute on: All Nodes
```

To compare the costs of executing the directed query and executing the original input query, deactivate the directed query and use EXPLAIN on the original input query. The optimizer reverts to creating a plan for the input query that incurs significantly greater cost—188K versus 2K:

```
=> DEACTIVATE DIRECTED QUERY RegionalSalesWine;
DEACTIVATE DIRECTED QUERY
=> EXPLAIN SELECT SD.store_region AS Region, SD.store_city AS City, SUM(SF.gross_profit_dollar_amount) Total
    FROM store.store_sales_fact SF
    JOIN store.store_dimension SD ON SF.store_key=SD.store_key
    JOIN product_dimension P ON SF.product_key||SF.product_version=P.product_key||P.product_version
  WHERE P.product_description ILIKE '%wine%'
  GROUP BY ROLLUP (SD.store_region, SD.store_city)
  ORDER BY Region,Total DESC;
...
 Access Path:
+-SORT [Cost: 188K, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
|  Order: SD.store_region ASC, sum(SF.gross_profit_dollar_amount) DESC
|  Execute on: All Nodes
| +---> GROUPBY HASH (GLOBAL RESEGMENT GROUPS) (LOCAL RESEGMENT GROUPS) [Cost: 188K, Rows: 10K (NO STATISTICS)] (PATH ID: 2)
| |      Aggregates: sum(SF.gross_profit_dollar_amount)
| |      Group By: SD.store_region, SD.store_city
| |      Grouping Sets: (SD.store_region, SD.store_city, <SVAR>), (SD.store_region, <SVAR>), (<SVAR>)
| |      Execute on: All Nodes
| | +---> JOIN HASH [Cost: 12K, Rows: 5M (NO STATISTICS)] (PATH ID: 3) Inner (BROADCAST)
| | |      Join Cond: (concat((SF.product_key)::varchar, (SF.product_version)::varchar) = concat((P.product_key)::varchar, (P.product_version)::varchar))
| | |      Materialize at Input: SF.product_key, SF.product_version
| | |      Materialize at Output: SF.gross_profit_dollar_amount
| | |      Execute on: All Nodes
| | | +-- Outer -> JOIN HASH [Cost: 2K, Rows: 5M (NO STATISTICS)] (PATH ID: 4) Inner (BROADCAST)
| | | |      Join Cond: (SF.store_key = SD.store_key)
| | | |      Execute on: All Nodes
| | | | +-- Outer -> STORAGE ACCESS for SF [Cost: 1K, Rows: 5M (NO STATISTICS)] (PATH ID: 5)
| | | | |      Projection: store.store_sales_fact_super
| | | | |      Materialize: SF.store_key
| | | | |      Execute on: All Nodes
| | | | |      Runtime Filters: (SIP2(HashJoin): SF.store_key), (SIP1(HashJoin): concat((SF.product_key)::varchar, (SF.product_version)::varchar))
| | | | +-- Inner -> STORAGE ACCESS for SD [Cost: 13, Rows: 250 (NO STATISTICS)] (PATH ID: 6)
| | | | |      Projection: store.store_dimension_super
| | | | |      Materialize: SD.store_key, SD.store_city, SD.store_region
| | | | |      Execute on: All Nodes
| | | +-- Inner -> STORAGE ACCESS for P [Cost: 201, Rows: 60K (NO STATISTICS)] (PATH ID: 7)
| | | |      Projection: public.product_dimension_super
| | | |      Materialize: P.product_key, P.product_version
| | | |      Filter: (P.product_description ~~* '%wine%')
| | | |      Execute on: All Nodes
```

Creating query templates

You can use directed queries to implement multiple queries that are identical except for the predicate strings on which query results are filtered. For example, directed query RegionalSalesWine only handles input queries that filter on product_description values that contain the string wine . You can create a modified version of this directed query that matches the syntax of multiple input queries, which differ only in their input values—for example, tuna .

Create this query template in the following steps:

1. Create two optimizer-generated directed queries :
   - From the original query on the joined tables:

```
=> CREATE DIRECTED QUERY OPTIMIZER RegionalSalesProducts_JoinTables
    SELECT SD.store_region AS Region, SD.store_city AS City, SUM(SF.gross_profit_dollar_amount) Total
      FROM store.store_sales_fact SF
      JOIN store.store_dimension SD ON SF.store_key=SD.store_key
      JOIN product_dimension P ON SF.product_key||SF.product_version=P.product_key||P.product_version
      WHERE P.product_description ILIKE '%wine%'
    GROUP BY ROLLUP (SD.store_region, SD.store_city)
    ORDER BY Region,Total DESC;
CREATE DIRECTED QUERY
```

- From the query on the flattened table:

```
=> CREATE DIRECTED QUERY OPTIMIZER RegionalSalesProduct
   SELECT store_region AS Region, store_city AS City, SUM(gross_profit_dollar_amount) AS Total
   FROM store.store_sales_wide
   WHERE product_description ILIKE '%wine%'
   GROUP BY ROLLUP (region, city)
   ORDER BY Region,Total DESC;
CREATE DIRECTED QUERY
```

2. Query system table DIRECTED_QUERIES and copy the input query for directed query RegionalSalesProducts_JoinTables :

```
SELECT input_query FROM directed_queries WHERE query_name = 'RegionalSalesProducts_JoinTables';
```

3. Use the copied input query with SAVE QUERY:

```
SAVE QUERY SELECT SD.store_region AS Region, SD.store_city AS City, sum(SF.gross_profit_dollar_amount) AS Total
 FROM ((store.store_sales_fact SF
 JOIN store.store_dimension SD ON ((SF.store_key = SD.store_key)))
 JOIN public.product_dimension P ON ((concat((SF.product_key)::varchar, (SF.product_version)::varchar) = concat((P.product_key)::varchar,
(P.product_version)::varchar))))
 WHERE (P.product_description ~~* '%wine%'::varchar(6) /*+:v(1)*/)
 GROUP BY GROUPING SETS((SD.store_region, SD.store_city), (SD.store_region), ())
 ORDER BY SD.store_region, sum(SF.gross_profit_dollar_amount) DESC
(1 row)
```

4. Query system table DIRECTED_QUERIES and copy the annotated query for directed query RegionalSalesProducts_FlatTables :

```
SELECT input_query FROM directed_queries WHERE query_name = 'RegionalSalesProducts_JoinTables';
```

5. Use the copied annotated query to create a custom directed query :

```
=> CREATE DIRECTED QUERY CUSTOM RegionalSalesProduct  SELECT /*+verbatim*/ store_sales_wide.store_region AS Region,
store_sales_wide.store_city AS City, sum(store_sales_wide.gross_profit_dollar_amount) AS Total
    FROM store.store_sales_wide AS store_sales_wide/*+projs('store.store_sales_wide')*/
    WHERE (store_sales_wide.product_description ~~* '%wine%'::varchar(6) /*+:v(1)*/)
    GROUP BY  /*+GByType(Hash)*/ GROUPING SETS((1, 2), (1), ())
    ORDER BY 1 ASC, 3 DESC;
CREATE DIRECTED QUERY
```

6. Activate the directed query:

```
ACTIVATE DIRECTED QUERY RegionalSalesProduct;
```

After activating this directed query, Vertica can use it for input queries that match the template, differing only in the predicate value for product_description :

```
=> EXPLAIN SELECT SD.store_region AS Region, SD.store_city AS City, SUM(SF.gross_profit_dollar_amount) Total
        FROM store.store_sales_fact SF
        JOIN store.store_dimension SD ON SF.store_key=SD.store_key
        JOIN product_dimension P ON SF.product_key||SF.product_version=P.product_key||P.product_version
       WHERE P.product_description ILIKE '%tuna%'
       GROUP BY ROLLUP (SD.store_region, SD.store_city)
       ORDER BY Region,Total DESC;
...
 The following active directed query(query name: RegionalSalesProduct) is being executed:
 SELECT /*+verbatim*/  store_sales_wide.store_region AS Region, store_sales_wide.store_city AS City, sum(store_sales_wide.gross_profit_dollar_amount)
AS Total
  FROM store.store_sales_wide store_sales_wide/*+projs('store.store_sales_wide')*/
  WHERE (store_sales_wide.product_description ~~* '%tuna%'::varchar(6))
  GROUP BY /*+GByType(Hash)*/  GROUPING SETS((store_sales_wide.store_region, store_sales_wide.store_city), (store_sales_wide.store_region), ())
  ORDER BY store_sales_wide.store_region, sum(store_sales_wide.gross_profit_dollar_amount) DESC

 Access Path:
+-SORT [Cost: 2K, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
|  Order: store_sales_wide.store_region ASC, sum(store_sales_wide.gross_profit_dollar_amount) DESC
|  Execute on: All Nodes
| +---> GROUPBY HASH (GLOBAL RESEGMENT GROUPS) (LOCAL RESEGMENT GROUPS) [Cost: 2K, Rows: 10K (NO STATISTICS)] (PATH ID: 2)
| | |      Aggregates: sum(store_sales_wide.gross_profit_dollar_amount)
| | |      Group By: store_sales_wide.store_region, store_sales_wide.store_city
| | |      Grouping Sets: (store_sales_wide.store_region, store_sales_wide.store_city, <SVAR>), (store_sales_wide.store_region, <SVAR>), (<SVAR>)
| | |      Execute on: All Nodes
| | +---> STORAGE ACCESS for store_sales_wide [Cost: 864, Rows: 10K (NO STATISTICS)] (PATH ID: 3)
| | | |      Projection: store.store_sales_wide_b0
| | | |      Materialize: store_sales_wide.gross_profit_dollar_amount, store_sales_wide.store_city, store_sales_wide.store_region
| | | |      Filter: (store_sales_wide.product_description ~~* '%tuna%')
| | | |      Execute on: All Nodes
```

When you execute this query, it returns with the following results:

```
 Region  |      City       | Total
---------+-----------------+---------
East     |                 | 1564582
East     | Elizabeth       |  131328
East     | Allentown       |  129704
East     | Boston          |  128188
East     | Sterling Heights |  125877
East     | Lowell          |  112133
East     | New Haven       |  101161
East     | Waterbury       |   85401
East     | Washington      |   76127
East     | Erie            |   73002
East     | Charlotte       |   67850
East     | Memphis         |   53650
East     | Clarksville     |   53416
East     | Hartford        |   52583
East     | Columbia        |   51950
East     | Nashville       |   50031
East     | Manchester      |   48607
East     | Baltimore       |   48108
East     | Stamford        |   47302
East     | New York        |   30840
East     | Portsmouth      |   26485
East     | Alexandria      |   26391
East     | Philadelphia    |   23092
East     | Cambridge       |   21356
MidWest  |                 |  980209
MidWest  | Lansing         |  130044
MidWest  | Livonia         |  118740
--More--
```

Managing directed queries

Vertica provides a number of ways to manage directed queries:

- Get directed queries
- Identify directed queries that are active
- Activate and deactivate directed queries
- Export directed queries from the catalog
- Drop directed queries

In this section

- Getting directed queries
- Identifying active directed queries
- Activating and deactivating directed queries
- Exporting directed queries from the catalog
- Dropping directed queries

Getting directed queries

You can obtain catalog information about directed queries in two ways:

- Use GET DIRECTED QUERY .
- Query system table DIRECTED_QUERIES .

GET DIRECTED QUERY

GET DIRECTED QUERY queries the system table DIRECTED_QUERIES on the specified input query. It returns details of all directed queries that map to the input query.

The following GET DIRECTED QUERY statement returns the directed query that maps to the following input query, findEmployeesCityJobTitle_OPT :

```
=> GET DIRECTED QUERY SELECT employee_first_name, employee_last_name from employee_dimension where employee_city='Boston' AND
job_title='Cashier' order by employee_last_name, employee_first_name;
-[ RECORD 1 ]---+
query_name      | findEmployeesCityJobTitle_OPT
is_active       | t
vertica_version | Vertica Analytic Database v11.0.1-20210815
comment         | Optimizer-generated directed query
creation_date   | 2021-08-20 14:53:42.323963
annotated_query | SELECT /*+verbatim*/  employee_dimension.employee_first_name, employee_dimension.employee_last_name FROM
public.employee_dimension employee_dimension/*+projs('public.employee_dimension')*/ WHERE ((employee_dimension.employee_city =
'Boston'::varchar(6) /*+:v(1)*/) AND (employee_dimension.job_title = 'Cashier'::varchar(7) /*+:v(2)*/)) ORDER BY employee_dimension.employee_last_name,
employee_dimension.employee_first_name
```

DIRECTED_QUERIES

You can query the system table DIRECTED_QUERIES directly. For example:

```
=> SELECT query_name, is_active FROM V_CATALOG.DIRECTED_QUERIES WHERE query_name ILIKE '%findEmployeesCityJobTitle%';
          query_name           | is_active
-------------------------------+-----------
 findEmployeesCityJobTitle_OPT | t
(1 row)
```

Caution

Query results for the fields INPUT_QUERY and ANNOTATED_QUERY are truncated after ~32K characters. You can get the full content of both fields in two ways:

- Use the statement GET DIRECTED QUERY .
- Use EXPORT_CATALOG to export directed queries.

Identifying active directed queries

Multiple directed queries can map to the same input query. The is_active column in system table DIRECTED_QUERIES identifies directed queries that are active. If multiple directed queries are active for the same input query, the optimizer uses the first one to be created. In that case, you can use EXPLAIN to identify which directed query is active.

Tip

It is good practice to activate only one directed query at a time for a given input query.

The following query on DIRECTED_QUERIES finds all active directed queries where the input query contains the name of the queried table employee_dimension :

```
=> SELECT * FROM directed_queries WHERE input_query ILIKE ('%employee_dimension%') AND is_active='t';
-[ RECORD 1 ]------+
query_name         | findEmployeesCityJobTitle_OPT
is_active          | t
vertica_version    | Vertica Analytic Database v11.0.1-20210815
comment            | Optimizer-generated directed query
save_plans_version | 0
username           | dbadmin
creation_date      | 2021-08-20 14:53:42.323963
since_date         |
input_query        | SELECT employee_dimension.employee_first_name, employee_dimension.employee_last_name FROM public.employee_dimension
WHERE ((employee_dimension.employee_city = 'Boston'::varchar(6) /*+:v(1)*/) AND (employee_dimension.job_title = 'Cashier'::varchar(7) /*+:v(2)*/))
ORDER BY employee_dimension.employee_last_name, employee_dimension.employee_first_name
annotated_query    | SELECT /*+verbatim*/  employee_dimension.employee_first_name, employee_dimension.employee_last_name FROM
public.employee_dimension employee_dimension/*+projs('public.employee_dimension')*/ WHERE ((employee_dimension.employee_city =
'Boston'::varchar(6) /*+:v(1)*/) AND (employee_dimension.job_title = 'Cashier'::varchar(7) /*+:v(2)*/)) ORDER BY employee_dimension.employee_last_name,
employee_dimension.employee_first_name
```

If you run EXPLAIN on the input query, it returns with a query plan that confirms use of findEmployeesCityJobTitle_OPT as the active directed query:

```
=> EXPLAIN SELECT employee_first_name, employee_last_name FROM employee_dimension WHERE employee_city='Boston' AND job_title ='Cashier'
ORDER BY employee_last_name, employee_first_name;

 The following active directed query(query name: findEmployeesCityJobTitle_OPT) is being executed:
 SELECT /*+verbatim*/  employee_dimension.employee_first_name, employee_dimension.employee_last_name FROM public.employee_dimension
employee_dimension/*+projs('public.employee_dimension')*/ WHERE ((employee_dimension.employee_city = 'Boston'::varchar(6)) AND
(employee_dimension.job_title = 'Cashier'::varchar(7))) ORDER BY employee_dimension.employee_last_name, employee_dimension.employee_first_name
```

Activating and deactivating directed queries

The optimizer uses only directed queries that are active. If multiple directed queries share the same input query, the optimizer uses the first one to be created.

You activate and deactivate directed queries with ACTIVATE DIRECTED QUERY and DEACTIVATE DIRECTED QUERY , respectively. For example, the following ACTIVATE DIRECTED QUERY statement deactivates RegionalSalesProducts_JoinTables and activates RegionalSalesProduct :

```
=> DEACTIVATE DIRECTED QUERY RegionalSalesProducts_JoinTables;
DEACTIVATE DIRECTED QUERY;
=> ACTIVATE DIRECTED QUERY RegionalSalesProduct;
ACTIVATE DIRECTED QUERY;
```

ACTIVATE DIRECTED QUERY and DEACTIVATE DIRECTED QUERY can also activate and deactivate multiple directed queries that are filtered from system table DIRECTED_QUERIES . In the following example, DEACTIVATE DIRECTED QUERY deactivates all directed queries with the same save_plans_version identifier:

```
=> DEACTIVATE DIRECTED QUERY WHERE save_plans_version = 21;
```

Vertica uses the active directed query for a given query across all sessions until it is explicitly deactivated by DEACTIVATE DIRECTED QUERY or removed from storage by DROP DIRECTED QUERY . If a directed query is active at the time of database shutdown, Vertica automatically reactivates it when you restart the database.

After a direct query is deactivated, the query optimizer handles subsequent invocations of the input query by using another directed query, if one is available. Otherwise, it generates its own query plan.

Exporting directed queries from the catalog

> **Tip**
> You can also export query plans as directed queries to an external SQL file. See Batch query plan export .

Before upgrading to a new version of Vertica, you can export directed queries for those queries whose optimal performance is critical to your system:

1. Use EXPORT_CATALOG with the argument DIRECTED_QUERIES to export from the database catalog all current directed queries and their current activation status:

```
=> SELECT EXPORT_CATALOG('../../export_directedqueries', 'DIRECTED_QUERIES');
EXPORT_CATALOG
---------------------------------------
Catalog data exported successfully
```

2. EXPORT_CATALOG creates a script to recreate the directed queries, as in the following example:

```
SAVE QUERY SELECT employee_dimension.employee_first_name, employee_dimension.employee_last_name FROM public.employee_dimension
WHERE ((employee_dimension.employee_city = 'Boston'::varchar(6) /*+:v(1)*/) AND (employee_dimension.job_title = 'Cashier'::varchar(7) /*+:v(2)*/))
ORDER BY employee_dimension.employee_last_name, employee_dimension.employee_first_name;
CREATE DIRECTED QUERY CUSTOM findEmployeesCityJobTitle_OPT COMMENT 'Optimizer-generated directed query' OPTVER 'Vertica Analytic
Database v11.0.1-20210815' PSDATE '2021-08-20 14:53:42.323963' SELECT /*+verbatim*/  employee_dimension.employee_first_name,
employee_dimension.employee_last_name
FROM public.employee_dimension employee_dimension/*+projs('public.employee_dimension')*/
WHERE ((employee_dimension.employee_city = 'Boston'::varchar(6) /*+:v(1)*/) AND (employee_dimension.job_title = 'Cashier'::varchar(7) /*+:v(2)*/))
ORDER BY employee_dimension.employee_last_name, employee_dimension.employee_first_name;
ACTIVATE DIRECTED QUERY findEmployeesCityJobTitle_OPT;
```

> **Note**
> The script that EXPORT_CATALOG creates specifies to recreate all directed queries with CREATE DIRECTED QUERY CUSTOM, regardless of how they were created originally.

3. After the upgrade is complete, remove each directed query from the database catalog with DROP DIRECTED QUERY . Alternatively, edit the export script and insert a DROP DIRECTED QUERY statement before each CREATE DIRECTED QUERY statement. For example, you might modify the script generated earlier with the changes shown in bold:

```
SAVE QUERY SELECT employee_dimension.employee_first_name, ... ;
DROP DIRECTED QUERY findEmployeesCityJobTitle_OPT;
CREATE DIRECTED QUERY CUSTOM findEmployeesCityJobTitle_OPT COMMENT 'Optimizer-generated ... ;
ACTIVATE DIRECTED QUERY findEmployeesCityJobTitle_OPT;
```

4. When you run this script, Vertica recreates the directed queries and restores their activation status:

```
=> \i /home/dbadmin/export_directedqueries
SAVE QUERY
DROP DIRECTED QUERY
CREATE DIRECTED QUERY
ACTIVATE DIRECTED QUERY
```

Dropping directed queries

DROP DIRECTED QUERY removes the specified directed query from the database catalog. If the directed query is active, Vertica deactivates it before removal.

For example:

```
=> DROP DIRECTED QUERY findBostonCashiers_CUSTOM;
DROP DIRECTED QUERY
```

DROP DIRECTED QUERY can also drop multiple directed queries that are filtered from system table DIRECTED_QUERIES . In the following example, DROP DIRECTED QUERY drops all directed queries with the same save_plans_version identifier:

```
=> DROP DIRECTED QUERY WHERE save_plans_version = 21;
```

Batch query plan export

Before upgrading to a new Vertica version, you might want to use directed queries to save query plans for possible reuse in the new database. You cannot predict which query plans are likely candidates for reuse, so you probably want to save query plans for many, or all, database queries. However, you run hundreds of queries each day. Saving query plans for each one to the database catalog through repetitive calls to CREATE DIRECTED QUERY is impractical. Moreover, doing so can significantly increase catalog size and possibly impact performance.

In this case, you can bypass the database catalog and batch export query plans as directed queries to an external SQL file. By offloading query plan storage, you can save any number of query plans from the current database without impacting catalog size and performance. After the upgrade, you can decide which query plans you wish to retain in the new database, and selectively import the corresponding directed queries.

Vertica provides a set of meta-functions that support this approach:

- EXPORT_DIRECTED_QUERIES generates query plans from a set of input queries, and writes SQL for creating directed queries that encapsulate those plans.
- IMPORT_DIRECTED_QUERIES imports directed queries to the database catalog from a SQL file that was generated by EXPORT_DIRECTED_QUERIES.

In this section

- Exporting directed queries
- Importing directed queries

Exporting directed queries

You can batch export any number of query plans as directed queries to an external SQL file, as follows:

1. Create a SQL file that contains the input queries whose query plans you wish to save. See Input Format below.
2. Call the meta-function EXPORT_DIRECTED_QUERIES on that SQL file. The meta-function takes two arguments:
   - Input queries file .
   - Optionally, the name of an output file . EXPORT_DIRECTED_QUERIES writes SQL for creating directed queries to this file. If you supply an empty string, Vertica writes the SQL to standard output.

For example, the following EXPORT_DIRECTED_QUERIES statement specifies input file inputQueries and output file outputQueries :

```
=> SELECT EXPORT_DIRECTED_QUERIES('/home/dbadmin/inputQueries','/home/dbadmin/outputQueries');
                        EXPORT_DIRECTED_QUERIES
-------------------------------------------------------------------------------------------
 1 queries successfully exported.
Queries exported to /home/dbadmin/outputQueries.

(1 row)
```

Input file

The input file that you supply to EXPORT_DIRECTED_QUERIES contains one or more input queries. For each input query, you can optionally specify two fields that are used in the generated directed query:

- DirQueryName provides the directed query's unique identifier, a string that conforms to conventions described in Identifiers .
- DirQueryComment specifies a quote-delimited string, up to 128 characters.

You format each input query as follows:

```
--DirQueryName=query-name
--DirQueryComment='comment'
input-query
```

For example, a file can specify one input query as follows:

```
/* Query: findEmployeesCityJobTitle_OPT */
/* Comment: This query finds all employees of a given city and job title, ordered by employee name */
SELECT employee_dimension.employee_first_name, employee_dimension.employee_last_name, employee_dimension.job_title FROM
public.employee_dimension WHERE (employee_dimension.employee_city = 'Boston'::varchar(6)) ORDER BY employee_dimension.job_title;
```

Output file

EXPORT_DIRECTED_QUERIES generates SQL for creating directed queries, and writes the SQL to the specified file or to standard output. In both cases, output conforms to the following format:

```
/* Query: directed-query-name */
/* Comment: directed-query-comment */
SAVE QUERY input-query;
CREATE DIRECTED QUERY CUSTOM 'directed-query-name'
COMMENT 'directed-query-comment'
OPTVER 'vertica-release-num'
PSDATE 'timestamp'
annotated-query
```

For example, given the previous input, Vertica writes the following output to /home/dbadmin/outputQueries :

```
/* Query: findEmployeesCityJobTitle_OPT */
/* Comment: This query finds all employees of a given city and job title, ordered by employee name */
SAVE QUERY SELECT employee_dimension.employee_first_name, employee_dimension.employee_last_name FROM public.employee_dimension
WHERE ((employee_dimension.employee_city = 'Boston'::varchar(6) /*+:v(1)*/) AND (employee_dimension.job_title = 'Cashier'::varchar(7) /*+:v(2)*/))
ORDER BY employee_dimension.employee_last_name, employee_dimension.employee_first_name;
CREATE DIRECTED QUERY CUSTOM 'findEmployeesCityJobTitle_OPT'
COMMENT 'This query finds all employees of a given city and job title, ordered by employee name'
OPTVER 'Vertica Analytic Database v11.1.0-20220102'
PSDATE '2022-01-06 13:45:17.430254'
SELECT /*+verbatim*/ employee_dimension.employee_first_name AS employee_first_name, employee_dimension.employee_last_name AS
employee_last_name
FROM public.employee_dimension AS employee_dimension/*+projs('public.employee_dimension')*/
WHERE (employee_dimension.employee_city = 'Boston'::varchar(6) /*+:v(1)*/) AND (employee_dimension.job_title = 'Cashier'::varchar(7) /*+:v(2)*/)
ORDER BY 2 ASC, 1 ASC;
```

If a given input query omits DirQueryName and DirQueryComment fields, EXPORT_DIRECTED_QUERIES automatically generates the following output:

- /* Query: Autoname: *timestamp* . *n* */ , where *n* is a zero-based integer index that ensures uniqueness among auto-generated names with the same timestamp.
- /* Comment: Optimizer-generated directed query */

For example, the following input file contains one SELECT statement, and omits the DirQueryName and DirQueryComment fields:

```
SELECT employee_dimension.employee_first_name, employee_dimension.employee_last_name
FROM public.employee_dimension WHERE (employee_dimension.employee_city = 'Boston'::varchar(6))
ORDER BY employee_dimension.job_title;
```

Given this file, EXPORT_DIRECTED_QUERIES returns with a warning about the missing input fields, which it also writes to an error file :

```
> SELECT EXPORT_DIRECTED_QUERIES('/home/dbadmin/inputQueries2','/home/dbadmin/outputQueries3');
                                    EXPORT_DIRECTED_QUERIES
-----------------------------------------------------------------------------------------------------
 1 queries successfully exported.
1 warning message was generated.
Queries exported to /home/dbadmin/outputQueries3.
See error report, /home/dbadmin/outputQueries3.err for details.
(1 row)
```

The output file contains the following content:

```
/* Query: Autoname:2022-01-06 14:11:23.071559.0 */
/* Comment: Optimizer-generated directed query */
SAVE QUERY SELECT employee_dimension.employee_first_name, employee_dimension.employee_last_name FROM public.employee_dimension
WHERE (employee_dimension.employee_city = 'Boston'::varchar(6) /*+:v(1)*/) ORDER BY employee_dimension.job_title;
CREATE DIRECTED QUERY CUSTOM 'Autoname:2022-01-06 14:11:23.071559.0'
COMMENT 'Optimizer-generated directed query'
OPTVER 'Vertica Analytic Database v11.1.0-20220102'
PSDATE '2022-01-06 14:11:23.071559'
SELECT /*+verbatim*/ employee_dimension.employee_first_name AS employee_first_name, employee_dimension.employee_last_name AS
employee_last_name
FROM public.employee_dimension AS employee_dimension/*+projs('public.employee_dimension')*/
WHERE (employee_dimension.employee_city = 'Boston'::varchar(6) /*+:v(1)*/)
ORDER BY employee_dimension.job_title ASC;
```

Error file

If any errors or warnings occur during EXPORT_DIRECTED_QUERIES execution, it returns with a message like this one:

```
1 queries successfully exported.
1 warning message was generated.
Queries exported to /home/dbadmin/outputQueries.
See error report, /home/dbadmin/outputQueries.err for details.
```

EXPORT_DIRECTED_QUERIES writes all errors and warnings to a file that it creates on the same path as the output file, and uses the output file's base name.

In the previous example, the output filename is /home/dbadmin/outputQueries , so EXPORT_DIRECTED_QUERIES writes errors to /home/dbadmin/outputQueries.err .

The error file can capture a number of errors and warnings, such as all instances where EXPORT_DIRECTED_QUERIES was unable to create a directed query. In the following example, the error file contains a warning that no name field was supplied for the specified input query, and records the name that was auto-generated for it:

```

-------------------------------------------------------------------------------------------------
WARNING: Name field not supplied. Using auto-generated name: 'Autoname:2016-10-13 09:44:33.527548.0'
Input Query: SELECT employee_dimension.employee_first_name, employee_dimension.employee_last_name, employee_dimension.job_title FROM
public.employee_dimension WHERE (employee_dimension.employee_city = 'Boston'::varchar(6)) ORDER BY employee_dimension.job_title;
END WARNING
```

Importing directed queries

After you determine which exported query plans you wish to use in the current database, you import them with IMPORT_DIRECTED_QUERIES . You supply this function with the name of the export file that you created with EXPORT_DIRECTED_QUERIES, and the names of directed queries you wish to import. For example:

```

=> SELECT IMPORT_DIRECTED_QUERIES('/home/dbadmin/outputQueries','FindEmployeesBoston');
                            IMPORT_DIRECTED_QUERIES
-------------------------------------------------------------------------------------------------
 1 directed queries successfully imported.
To activate a query named 'my_query1':
=> ACTIVATE DIRECTED QUERY 'my_query1';

(1 row)
```

After importing the desired directed queries, you must activate them with ACTIVATE DIRECTED QUERY before you can use them to create query plans.

Half join and cross join semantics

The Vertica optimizer uses several keywords in directed queries to recreate cross join and half join subqueries. It also supports an additional set of keywords to express complex cross joins and half joins. You can also use these keywords in queries that you execute directly in vsql.

Caution

These keywords do not conform with standard SQL; they are intended for use only by the Vertica optimizer.

For details, see the following topics:

- Half-join subquery semantics
- Complex join semantics

In this section

- Half-join subquery semantics
- Complex join semantics

Half-join subquery semantics

The Vertica optimizer uses several keywords in directed queries to recreate half-join subqueries with certain search operators, such as ANY or NOT IN .

SEMI JOIN

Recreates a query that contains a subquery preceded by an IN , EXISTS , or ANY operator and executes a semi-join.

Input query

```
SELECT product_description FROM product_dimension
  WHERE product_dimension.product_key IN (SELECT qty_in_stock from inventory_fact);
```

Query plan

```
QUERY PLAN DESCRIPTION:
------------------------------

explain SELECT product_description FROM product_dimension WHERE product_dimension.product_key IN (SELECT qty_in_stock from inventory_fact);

 Access Path:
 +-JOIN HASH [Semi] [Cost: 1K, Rows: 30K] (PATH ID: 1) Outer (FILTER) Inner (RESEGMENT)
 |  Join Cond: (product_dimension.product_key = VAL(2))
 |  Materialize at Output: product_dimension.product_description
 |  Execute on: All Nodes
 | +-- Outer -> STORAGE ACCESS for product_dimension [Cost: 152, Rows: 60K] (PATH ID: 2)
 | |      Projection: public.product_dimension
 | |      Materialize: product_dimension.product_key
 | |      Execute on: All Nodes
 | |      Runtime Filter: (SIP1(HashJoin): product_dimension.product_key)
 | +-- Inner -> SELECT [Cost: 248, Rows: 300K] (PATH ID: 3)
 | |      Execute on: All Nodes
 | | +---> STORAGE ACCESS for inventory_fact [Cost: 248, Rows: 300K] (PATH ID: 4)
 | | |      Projection: public.inventory_fact_b0
 | | |      Materialize: inventory_fact.qty_in_stock
 | | |      Execute on: All Nodes
```

Optimizer-generated annotated query

```
SELECT /*+ syntactic_join */ product_dimension.product_description AS product_description
FROM (public.product_dimension AS product_dimension/*+projs('public.product_dimension')*/
SEMI JOIN /*+Distrib(F,R),JType(H)*/ (SELECT inventory_fact.qty_in_stock AS qty_in_stock
FROM public.inventory_fact AS inventory_fact/*+projs('public.inventory_fact')*/) AS subQ_1
ON (product_dimension.product_key = subQ_1.qty_in_stock))
```

NULLAWARE ANTI JOIN

Recreates a query that contains a subquery preceded by a NOT IN or !=ALL operator, and executes a null-aware anti-join.

Input query

```
SELECT product_description FROM product_dimension
WHERE product_dimension.product_key NOT IN (SELECT qty_in_stock from inventory_fact);
```

Query plan

```
 QUERY PLAN DESCRIPTION:
 ------------------------------


 EXPLAIN SELECT product_description FROM product_dimension WHERE product_dimension.product_key not IN (SELECT qty_in_sto
ck from inventory_fact);

 Access Path:
 +-JOIN HASH [Anti][NotInAnti] [Cost: 7K, Rows: 30K] (PATH ID: 1) Inner (BROADCAST)
 |  Join Cond: (product_dimension.product_key = VAL(2))
 |  Materialize at Output: product_dimension.product_description
 |  Execute on: Query Initiator
 | +-- Outer -> STORAGE ACCESS for product_dimension [Cost: 152, Rows: 60K] (PATH ID: 2)
 | |      Projection: public.product_dimension_DBD_2_rep_VMartDesign
 | |      Materialize: product_dimension.product_key
 | |      Execute on: Query Initiator
 | +-- Inner -> SELECT [Cost: 248, Rows: 300K] (PATH ID: 3)
 | |      Execute on: All Nodes
 | | +---> STORAGE ACCESS for inventory_fact [Cost: 248, Rows: 300K] (PATH ID: 4)
 | | |      Projection: public.inventory_fact_DBD_9_seg_VMartDesign_b0
 | | |      Materialize: inventory_fact.qty_in_stock
 | | |      Execute on: All Nodes
```

Optimizer-generated annotated query

```
SELECT /*+ syntactic_join */ product_dimension.product_description AS product_description
FROM (public.product_dimension AS product_dimension/*+projs('public.product_dimension')*/
NULLAWARE ANTI JOIN /*+Distrib(L,B),JType(H)*/ (SELECT inventory_fact.qty_in_stock AS qty_in_stock
FROM public.inventory_fact AS inventory_fact/*+projs('public.inventory_fact')*/) AS subQ_1
ON (product_dimension.product_key = subQ_1.qty_in_stock))
```

SEMIALL JOIN

Recreates a query that contains a subquery preceded by an ALL operator, and executes a semi-all join.

Input query

```
SELECT product_key, product_description FROM product_dimension
  WHERE product_dimension.product_key > ALL (SELECT product_key from inventory_fact);
```

Query plan

```
QUERY PLAN DESCRIPTION:
------------------------------

explain SELECT product_key, product_description FROM product_dimension WHERE product_dimension.product_key > ALL (SELECT product_key from
inventory_fact);

Access Path:
+-JOIN HASH [Semi][All] [Cost: 7M, Rows: 30K] (PATH ID: 1) Outer (FILTER) Inner (BROADCAST)
|  Join Filter: (product_dimension.product_key > VAL(2))
|  Materialize at Output: product_dimension.product_description
|  Execute on: All Nodes
| +-- Outer -> STORAGE ACCESS for product_dimension [Cost: 152, Rows: 60K] (PATH ID: 2)
| |      Projection: public.product_dimension
| |      Materialize: product_dimension.product_key
| |      Execute on: All Nodes
| +-- Inner -> SELECT [Cost: 248, Rows: 300K] (PATH ID: 3)
| |      Execute on: All Nodes
| | +---> STORAGE ACCESS for inventory_fact [Cost: 248, Rows: 300K] (PATH ID: 4)
| | |      Projection: public.inventory_fact_b0
| | |      Materialize: inventory_fact.product_key
| | |      Execute on: All Nodes
```

Optimizer-generated annotated query

```
SELECT /*+ syntactic_join */ product_dimension.product_key AS product_key, product_dimension.product_description AS product_description
FROM (public.product_dimension AS product_dimension/*+projs('public.product_dimension')*/
SEMIALL JOIN /*+Distrib(F,B),JType(H)*/ (SELECT inventory_fact.product_key AS product_key FROM public.inventory_fact AS
inventory_fact/*+projs('public.inventory_fact')*/) AS subQ_1
ON (product_dimension.product_key > subQ_1.product_key))
```

ANTI JOIN

Recreates a query that contains a subquery preceded by a NOT EXISTS operator, and executes an anti-join.

Input query

```
SELECT product_key, product_description FROM product_dimension
WHERE NOT EXISTS (SELECT inventory_fact.product_key FROM inventory_fact
WHERE inventory_fact.product_key=product_dimension.product_key);
```

Query plan

```
QUERY PLAN DESCRIPTION:
------------------------------

 explain SELECT product_key, product_description FROM product_dimension WHERE NOT EXISTS (SELECT inventory_fact.product_
key FROM inventory_fact WHERE inventory_fact.product_key=product_dimension.product_key);

 Access Path:
+-JOIN HASH [Anti] [Cost: 703, Rows: 30K] (PATH ID: 1) Outer (FILTER)
|  Join Cond: (VAL(1) = product_dimension.product_key)
|  Materialize at Output: product_dimension.product_description
|  Execute on: All Nodes
| +-- Outer -> STORAGE ACCESS for product_dimension [Cost: 152, Rows: 60K] (PATH ID: 2)
| |      Projection: public.product_dimension_DBD_2_rep_VMartDesign
| |      Materialize: product_dimension.product_key
| |      Execute on: All Nodes
| +-- Inner -> SELECT [Cost: 248, Rows: 300K] (PATH ID: 3)
| |      Execute on: All Nodes
| | +---> STORAGE ACCESS for inventory_fact [Cost: 248, Rows: 300K] (PATH ID: 4)
| | |      Projection: public.inventory_fact_DBD_9_seg_VMartDesign_b0
| | |      Materialize: inventory_fact.product_key
| | |      Execute on: All Nodes
```

Optimizer-generated annotated query

```
SELECT /*+ syntactic_join */ product_dimension.product_key AS product_key, product_dimension.product_description AS product_description
FROM (public.product_dimension AS product_dimension/*+projs('public.product_dimension')*/
ANTI JOIN /*+Distrib(F,L),JType(H)*/ (SELECT inventory_fact.product_key AS "inventory_fact.product_key"
FROM public.inventory_fact AS inventory_fact/*+projs('public.inventory_fact')*/) AS subQ_1
ON (subQ_1."inventory_fact.product_key" = product_dimension.product_key))
```

Complex join semantics

The optimizer uses a set of keywords to express complex cross joins and half joins . All complex joins are indicated by the keyword COMPLEX , which is inserted before the keyword JOIN —for example, CROSS COMPLEX JOIN . Semantics for complex half joins have an additional requirement, which is detailed below .

Complex cross join

Vertica uses the keyword phrase CROSS COMPLEX JOIN to describe all complex cross joins. For example:

Input query

```
SELECT
 (SELECT max(sales_quantity) FROM store.store_sales_fact) *
 (SELECT max(sales_quantity) FROM online_sales.online_sales_fact);
```

Query plan

```
QUERY PLAN DESCRIPTION:
------------------------------

EXPLAIN SELECT
  (SELECT max(sales_quantity) FROM store.store_sales_fact) *
  (SELECT max(sales_quantity) FROM online_sales.online_sales_fact);

Access Path:
+-JOIN  (CROSS JOIN) [Cost: 4K, Rows: 1 (NO STATISTICS)] (PATH ID: 1)
|  Execute on: Query Initiator
| +-- Outer -> JOIN  (CROSS JOIN) [Cost: 2K, Rows: 1 (NO STATISTICS)] (PATH ID: 2)
| |      Execute on: Query Initiator
| | +-- Outer -> STORAGE ACCESS for dual [Cost: 10, Rows: 1] (PATH ID: 3)
| | |      Projection: v_catalog.dual_p
| | |      Materialize: dual.dummy
| | |      Execute on: Query Initiator
| | +-- Inner -> SELECT [Cost: 2K, Rows: 1 (NO STATISTICS)] (PATH ID: 4)
| | |      Execute on: Query Initiator
| | | +---> GROUPBY NOTHING [Cost: 2K, Rows: 1 (NO STATISTICS)] (PATH ID: 5)
| | | |      Aggregates: max(store_sales_fact.sales_quantity)
| | | |      Execute on: All Nodes
| | | | +---> STORAGE ACCESS for store_sales_fact [Cost: 1K, Rows: 5M (NO STATISTICS)] (PATH ID: 6)
| | | | |      Projection: store.store_sales_fact_super
| | | | |      Materialize: store_sales_fact.sales_quantity
| | | | |      Execute on: All Nodes
| +-- Inner -> SELECT [Cost: 2K, Rows: 1 (NO STATISTICS)] (PATH ID: 7)
| |      Execute on: Query Initiator
| | +---> GROUPBY NOTHING [Cost: 2K, Rows: 1 (NO STATISTICS)] (PATH ID: 8)
| | |      Aggregates: max(online_sales_fact.sales_quantity)
| | |      Execute on: All Nodes
| | | +---> STORAGE ACCESS for online_sales_fact [Cost: 1K, Rows: 5M (NO STATISTICS)] (PATH ID: 9)
| | | |      Projection: online_sales.online_sales_fact_super
| | | |      Materialize: online_sales_fact.sales_quantity
| | | |      Execute on: All Nodes
```

Optimizer-generated annotated query

The following annotated query returns the same results as the input query shown earlier. As with all optimizer-generated annotated queries, you can execute this query directly in vsql, either as written or with modifications:

```
SELECT /*+syntactic_join,verbatim*/ (subQ_1.max * subQ_2.max) AS "?column?"
FROM ((v_catalog.dual AS dual CROSS COMPLEX JOIN /*+Distrib(L,L),JType(H)*/
(SELECT max(store_sales_fact.sales_quantity) AS max
FROM store.store_sales_fact AS store_sales_fact/*+projs('store.store_sales_fact')*/) AS subQ_1 )
CROSS COMPLEX JOIN /*+Distrib(L,L),JType(H)*/ (SELECT max(online_sales_fact.sales_quantity) AS max
FROM online_sales.online_sales_fact AS online_sales_fact/*+projs('online_sales.online_sales_fact')*/) AS subQ_2 )
```

Complex half join

Complex half joins are expressed by one of the following keywords:

- SEMI COMPLEX JOIN
- NULLAWARE ANTI COMPLEX JOIN
- SEMIALL COMPLEX JOIN
- ANTI COMPLEX JOIN

An additional requirement applies to all complex half joins: each subquery's SELECT list ends with a dummy column (labeled as false) that invokes the Vertica meta-function complex_join_marker() . As the subquery processes each row, complex_join_marker() returns true or false to indicate the row's inclusion or exclusion from the result set. The result set returns with this flag to the outer query, which can use the flag from this and other subqueries to filter its own result set.

For example, the query optimizer rewrites the following input query as a NULLAWARE ANTI COMPLEX JOIN . The join returns all rows from the subquery with their complex_join_marker() flag set to the appropriate Boolean value.

Input query

```
SELECT product_dimension.product_description FROM public.product_dimension
  WHERE (NOT (product_dimension.product_key NOT IN (SELECT inventory_fact.qty_in_stock FROM public.inventory_fact)));
```

Query plan

```
 QUERY PLAN DESCRIPTION:
 ------------------------------

 EXPLAIN SELECT product_dimension.product_description FROM public.product_dimension
   WHERE (NOT (product_dimension.product_key NOT IN (SELECT inventory_fact.qty_in_stock FROM public.inventory_fact)));

 Access Path:
+-JOIN HASH [Anti][NotInAnti] [Cost: 3K, Rows: 30K] (PATH ID: 1) Inner (BROADCAST)
|  Join Cond: (product_dimension.product_key = VAL(2))
|  Materialize at Output: product_dimension.product_description
|  Filter: (NOT VAL(2))
|  Execute on: All Nodes
| +-- Outer -> STORAGE ACCESS for product_dimension [Cost: 56, Rows: 60K] (PATH ID: 2)
| | |      Projection: public.product_dimension_super
| | |      Materialize: product_dimension.product_key
| | |      Execute on: All Nodes
| +-- Inner -> SELECT [Cost: 248, Rows: 300K] (PATH ID: 3)
| | |      Execute on: All Nodes
| | +---> STORAGE ACCESS for inventory_fact [Cost: 248, Rows: 300K] (PATH ID: 4)
| | | |      Projection: public.inventory_fact_super
| | | |      Materialize: inventory_fact.qty_in_stock
| | | |      Execute on: All Nodes
```

Optimizer-generated annotated query

The following annotated query returns the same results as the input query shown earlier. As with all optimizer-generated annotated queries, you can execute this query directly in vsql, either as written or with modifications. For example, you can control the outer query's output by modifying how its predicate evaluates the flag subQ_1."false" .

```
SELECT /*+syntactic_join,verbatim*/ product_dimension.product_description AS product_description
FROM (public.product_dimension AS product_dimension/*+projs('public.product_dimension')*/
NULLAWARE ANTI COMPLEX JOIN /*+Distrib(L,B),JType(H)*/
  (SELECT inventory_fact.qty_in_stock AS qty_in_stock, complex_join_marker() AS "false"
   FROM public.inventory_fact AS inventory_fact/*+projs('public.inventory_fact')*/) AS subQ_1
ON (product_dimension.product_key = subQ_1.qty_in_stock)) WHERE (NOT subQ_1."false")
```

Directed query restrictions

In general, directed queries support only SELECT statements as input. Within that broad restriction, a number of specific restrictions apply to input queries. Vertica handles all restrictions through optimizer-generated warnings. The sections below divide these restrictions into several categories.

Tables, views, and projections

Input queries cannot query the following objects:

- Tables without projections
- Tables with access policies
- System and data collector tables, except explicit and implicit references to V_CATALOG.DUAL
- Views
- Projections

Functions

Input queries cannot include the following functions:

- Vertica meta-functions

- Pattern-matching functions
- GROUPING_ID with no arguments

Language elements

Input queries cannot include the following:

- Hints
- WITH clauses when materialization is enabled
- Date/time literals that reference the current time, such as NOW or YESTERDAY

Data types

Input queries cannot include the following data types:

- Spatial data types : GEOMETRY and GEOGRAPHY
- Complex data types : non-native arrays , rows , and sets

Transactions

When transactions in multiple user sessions concurrently access the same data, session-scoped isolation levels determine what data each transaction can access.

A transaction retains its isolation level until it completes, even if the session's isolation level changes during the transaction. Vertica internal processes (such as the Tuple Mover and refresh operations) and DDL operations always run at the SERIALIZABLE isolation level to ensure consistency.

The Vertica query parser supports standard ANSI SQL-92 isolation levels as follows:

- READ COMMITTED (default)
- READ UNCOMMITTED : Automatically interpreted as READ COMMITTED.
- REPEATABLE READ: Automatically interpreted as SERIALIZABLE
- SERIALIZABLE

Transaction isolation levels READ COMMITTED and SERIALIZABLE differ as follows:

Isolation level	**Dirty read**	**Non-repeatable read**	**Phantom read**
READ COMMITTED	Not Possible	Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

You can set separate isolation levels for the database and individual transactions.

Implementation details

Vertica supports conventional SQL transactions with standard ACID properties:

- ANSI SQL 92 style-implicit transactions. You do not need to run a BEGIN or START TRANSACTION command.
- No redo/undo log or two-phase commits.
- The COPY command automatically commits itself and any current transaction (except when loading temporary tables). It is generally good practice to commit or roll back the current transaction before you use COPY. This step is optional for DDL statements, which are auto-committed.

In this section

- Rollback
- Savepoints
- READ COMMITTED isolation
- SERIALIZABLE isolation

Rollback

Transaction rollbacks restore a database to an earlier state by discarding changes made by that transaction. Statement-level rollbacks discard only the changes initiated by the reverted statements. Transaction-level rollbacks discard all changes made by the transaction.

With a ROLLBACK statement, you can explicitly roll back to a named savepoint within the transaction, or discard the entire transaction. Vertica can also initiate automatic rollbacks in two cases:

- An individual statement returns an ERROR message. In this case, Vertica rolls back the statement.

- DDL errors, systemic failures, dead locks, and resource constraints return a ROLLBACK message. In this case, Vertica rolls back the entire transaction.

Explicit and automatic rollbacks always release any locks that the transaction holds.

Savepoints

A *savepoint* is a special marker inside a transaction that allows commands that execute after the savepoint to be rolled back. The transaction is restored to the state that preceded the savepoint.

Vertica supports two types of savepoints:

- An *implicit savepoint* is automatically established after each successful command within a transaction. This savepoint is used to roll back the next statement if it returns an error. A transaction maintains one implicit savepoint, which it rolls forward with each successful command. Implicit savepoints are available to Vertica only and cannot be referenced directly.
- *Named savepoints are* labeled markers within a transaction that you set through SAVEPOINT statements. A named savepoint can later be referenced in the same transaction through RELEASE SAVEPOINT , which destroys it, and ROLLBACK TO SAVEPOINT , which rolls back all operations that followed the savepoint. Named savepoints can be especially useful in nested transactions: a nested transaction that begins with a savepoint can be rolled back entirely, if necessary.

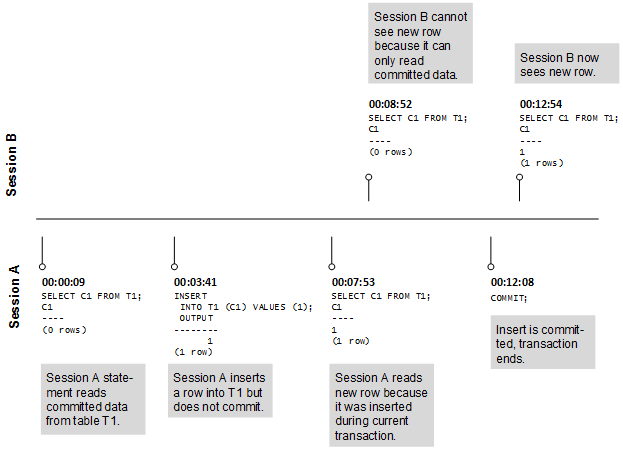
READ COMMITTED isolation

When you use the isolation level READ COMMITTED , a SELECT query obtains a backup of committed data at the transaction's start. Subsequent queries during the current transaction also see the results of uncommitted updates that already executed in the same transaction.

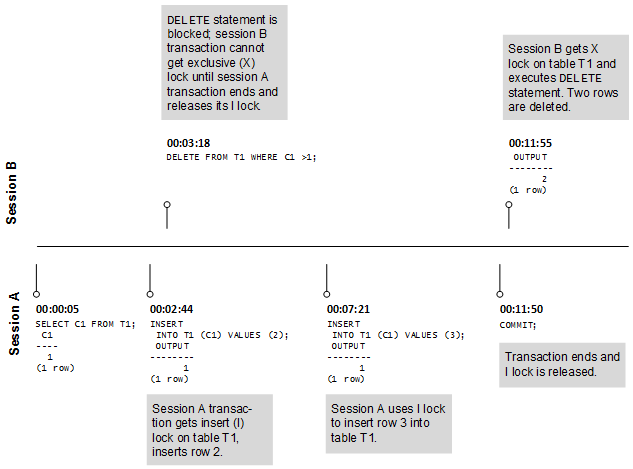
When you use DML statements, your query acquires write locks to prevent other READ COMMITTED transactions from modifying the same data. However, be aware that SELECT statements do not acquire locks, so concurrent transactions can obtain read and write access to the same selection.

READ COMMITTED is the default isolation level. For most queries, this isolation level balances database consistency and concurrency. However, this isolation level can allow one transaction to change the data that another transaction is in the process of accessing. Such changes can yield nonrepeatable and phantom reads . You may have applications with complex queries and updates that require a more consistent view of the database. If so, use SERIALIZABLE isolation instead.

The following figure shows how READ COMMITTED isolation might control how concurrent transactions read and write the same data:



READ COMMITTED isolation maintains exclusive write locks until a transaction ends, as shown in the following graphic:



See also

- Vertica database locks
- LOCKS
- SET SESSION CHARACTERISTICS AS TRANSACTION

SERIALIZABLE isolation

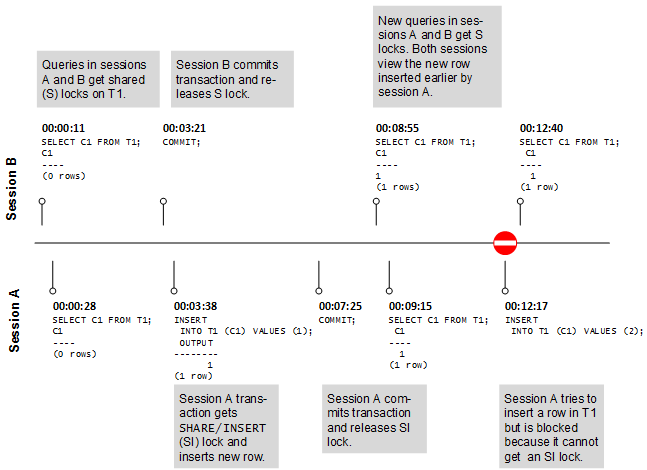
SERIALIZABLE is the strictest SQL transaction isolation level. While this isolation level permits transactions to run concurrently, it creates the effect that transactions are running in serial order. Transactions acquire locks for read and write operations. Thus, successive SELECT commands within a single transaction always produce the same results. Because SERIALIZABLE isolation provides a consistent view of data, it is useful for applications that require complex queries and updates. However, serializable isolation reduces concurrency. For example, it blocks queries during a bulk load.

SERIALIZABLE isolation establishes the following locks:

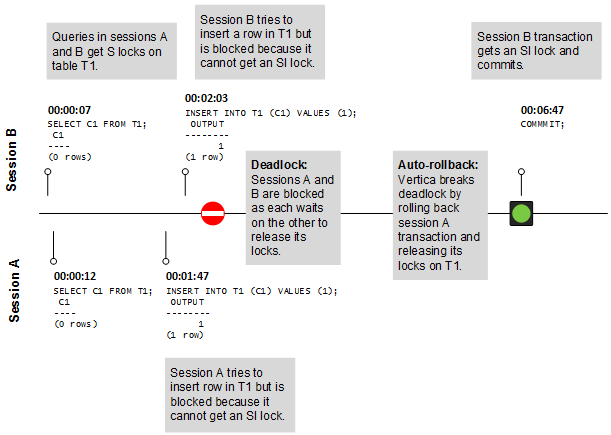
- Table-level read locks: Vertica acquires table-level read locks on selected tables and releases them when the transaction ends. This behavior prevents one transaction from modifying rows while they are being read by another transaction.
- Table-level write lock: Vertica acquires table-level write locks on update and releases them when the transaction ends. This behavior prevents one transaction from reading another transaction's changes to rows before those changes are committed.

At the start of a transaction, a SELECT statement obtains a backup of the selection's committed data. The transaction also sees the results of updates that are run within the transaction before they are committed.

The following figure shows how concurrent transactions that both have SERIALIZABLE isolation levels handle locking:



Applications that use SERIALIZABLE must be prepared to retry transactions due to serialization failures. Such failures often result from deadlocks. When a deadlock occurs, any transaction awaiting a lock automatically times out after 5 minutes. The following figure shows how deadlock might occur and how Vertica handles it:



Note

SERIALIZABLE isolation does not apply to temporary tables. No locks are required for these tables because they are isolated by their transaction scope.

See also Vertica

- Vertica Database Locks
- LOCKS

Vertica database locks

When multiple users concurrently access the same database information, data manipulation can cause conflicts and threaten data integrity. Conflicts occur because some transactions block other operations until the transaction completes. Because transactions committed at the same time should produce consistent results, Vertica uses locks to maintain data concurrency and consistency. Vertica automatically controls locking by limiting the actions a user can take on an object, depending on the state of that object.

Vertica uses object locks and system locks. *Object locks* are used on objects, such as tables and projections. *System locks* include global catalog locks, local catalog locks, and elastic cluster locks. Vertica supports a full range of standard SQL lock modes , such as shared (S) and exclusive (X).

For related information about lock usage in different transaction isolation levels, see READ COMMITTED isolation and SERIALIZABLE isolation .

In this section

- Lock modes
- Lock examples
- Deadlocks
- Troubleshooting locks

Lock modes

Vertica has different lock modes that determine how a lock acts on an object. Each lock mode is unique in its compatibility with other lock modes; each lock mode has its own strength vis a vis other lock modes, which determines whether Each lock mode has a lock compatibility and strength that reflect how it interacts with other locks in the same environment.

Lock mode	Description
Usage (U)	Vertica uses usage (U) locks for Tuple Mover mergeout operations. These Tuple Mover operations run automatically in the background, therefore, other operations on a table except those requiring an O or D locks can run when the object is locked in U mode.
Tuple Mover (T)	Vertica uses Tuple Mover (T) locks for operations on delete vectors. Tuple Mover operations upgrade the table lock mode from U to T when work on delete vectors starts so that no other updates or deletes can happen concurrently.

Shared (S)	Use a shared (S) lock for SELECT queries that run at the serialized transaction isolation level. This allows queries to run concurrently, but the S lock creates the effect that transactions are running in serial order. The S lock ensures that one transaction does not affect another transaction until one transaction completes and its S lock is released. Select operations in READ COMMITTED transaction mode do not require S table locks. See Transactions for more information.
Insert (I)	Vertica requires an insert (I) lock to insert data into a table. Multiple transactions can lock an object in insert mode simultaneously, enabling multiple inserts and bulk loads to occur at the same time. This behavior is critical for parallel loads and high ingestion rates.
Insert Validate (IV)	An insert validate (IV) lock is required for insert operations where the system performs constraint validation for enabled PRIMARY or UNIQUE key constraints.
Shared Insert (SI)	Vertica requires a shared insert (SI) lock when both a read and an insert occur in a transaction. SI mode prohibits delete/update operations. An SI lock also results from lock promotion.
Exclusive (X)	Vertica uses exclusive (X) locks when performing deletes and updates. Only Tuple Mover mergeout operations (U locks) can run concurrently on objects with X locks.
Drop Partition (D)	DROP_PARTITIONS requires a D lock on the target table. This lock is only compatible with I-lock operations, so only table load operations such as INSERT and COPY are allowed during drop partition operations.
Owner (O)	An owner (O) lock is the strongest Vertica lock mode. An object acquires an O lock when it undergoes changes in both data and structure. Such changes can occur in some DDL operations, such as DROP_PARTITIONS, TRUNCATE TABLE, and ADD COLUMN. When an object is locked in O mode, it cannot be locked simultaneously by another transaction in any mode.

Lock compatibility

Bulleted (•) cells in the following matrix shows which locks can be used on the same object simultaneously. Empty cells indicate that a query's requested mode is not granted until the current (granted) mode releases its lock on the object.

Requested mode	**Granted mode**								
	U	**T**	**S**	**I**	**IV**	**SI**	**X**	**D**	**O**
U	•	•	•	•	•	•	•		
T	•	•	•	•	•	•			
S	•	•	•						
I	•	•		•	•			•	
IV	•	•		•					
SI	•	•							
X	•								
D				•					
O									

Lock conversion

Often, the same object is the target of concurrent lock requests from different sessions. The matrix below shows how Vertica responds to multiple lock requests on the same object, one of the following:

- Locks are granted to concurrent requests on an object if the respective lock modes are compatible. For example, D (drop partition) and I (insert)

locks are compatible, so Vertica can grant multiple lock requests on the same table for concurrent load and drop partition operations.

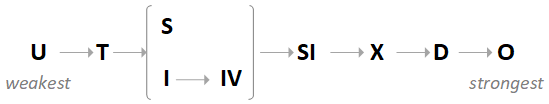
- Lock modes for concurrent requests on an object are incompatible, but the requests also support a higher (stronger) lock mode. In this case, Vertica converts (upgrades) the lock modes for these requests—for example, S and I to SI. The upgraded lock mode enables requests on the object to proceed concurrently.
- Lock modes for concurrent requests on an object are incompatible, and none can be upgraded to a common lock mode. In this case, object lock requests are queued and granted in sequence.

Requested mode	**Granted mode**								
	U	**T**	**S**	**I**	**IV**	**SI**	**X**	**D**	**O**
U	U	T	S	I	IV	SI	X	D	O
T	T	T	S	I	IV	SI	X	D	O
S	S	S	S	SI	SI	SI	X	O	O
I	I	I	SI	I	IV	SI	X	D	O
IV	IV	IV	SI	IV	IV	SI	X	D	O
SI	SI	SI	SI	SI	SI	SI	X	O	O
X	X	X	X	X	X	X	X	O	O
D	D	D	O	D	D	O	O	D	O
O	O	O	O	O	O	O	O	O	O

Lock strength

Lock strength refers to the ability of a lock mode to interact with another lock mode. O locks are strongest and are incompatible with all other locks. Conversely, U locks are weakest and can run concurrently with all other locks except D and O locks.

The following figure depicts the spectrum of lock mode strength:



See also

- Vertica database locks
- LOCKS

Lock examples

Automatic locks

In this example, two sessions (A and B) attempt to perform operations on table T1. These operations automatically acquire the necessary locks.

At the beginning of the example, table T1 has one column (C1) and no rows.

The steps here represent a possible series of transactions from sessions A and B:

1. Transactions in both sessions acquire S locks to read from Table T1.
2. Session B releases its S lock with the COMMIT statement.
3. Session A can upgrade to an SI lock and insert into T1 because Session B released its S lock.
4. Session A releases its SI lock with a COMMIT statement.
5. Both sessions can acquire S locks because Session A released its SI lock.
6. Session A cannot acquire an SI lock because Session B has not released its S lock. SI locks are incompatible with S locks.
7. Session B releases its S lock with the COMMIT statement.
8. Session A can now upgrade to an SI lock and insert into T1.
9. Session B attempts to delete a row from T1 but can't acquire an X lock because Session A has not released its SI lock. SI locks are incompatible with X locks.
10. Session A continues to insert into T1.

11. Session A releases its SI lock.
12. Session B can now acquire an X lock and perform the delete.

This figure illustrates the previous steps:

	Session A Transactions	Session B Transactions	State of Table T1
1	SELECT C1 FROM T1; C1 -- (0 rows)	SELECT C1 FROM T1; C1 -- (0 rows)	C1 -- (0 rows)
2		COMMIT;	
3	INSERT INTO T1 (C1) VALUES (1); OUTPUT -- 1 (1 row)		C1 -- 1 (1 row)
4	COMMIT;		
5	SELECT C1 FROM T1; C1 -- 1 (1 rows)	SELECT C1 FROM T1; C1 -- 1 (1 rows)	
6	INSERT INTO T1 (C1) VALUES (2); OUTPUT -- 1 (1 row)		
7		COMMIT;	
8	OUTPUT ------ (1 row)		C1 -- 1 2 (2 rows)
9		DELETE FROM T1 WHERE C1>1;	
10	INSERT INTO T1 (C1) VALUES (3); OUTPUT -- 1 (1 row)		C1 -- 1 2 3 (3 rows)
11	COMMIT;		
12		OUTPUT ------ (2 rows)	C1 -- 1 (rows)

Manual locks

In this example, Alice attempts to manually lock table customer_info with LOCK TABLE while Bob runs an INSERT statement:

Bob runs the following INSERT statement to acquire an INSERT lock and insert a row:

```
=> INSERT INTO customer_info VALUES(37189,'Albert','Quinlan','Frankfurt',2022);
```

In another session, Alice attempts to acquire a SHARE lock with LOCK TABLE. As shown in the lock compatibility table , the INSERT lock is incompatible with SHARE locks (among others), so Alice cannot acquire a SHARE lock until Bob finishes his transaction:

```
=> LOCK customer_info IN SHARE MODE NOWAIT;
ERROR 5157:  Unavailable: [Txn 0xa00000001c48e3] S lock table - timeout error Timed out S locking Table:public.customer_info. I held by [user Bob (LOCK TABLE)]. Your current transaction isolation level is READ COMMITTED
```

Bob then releases the lock by calling COMMIT:

```
=> COMMIT;
COMMIT
```

Alice can now acquire the SHARE lock:

```
=> LOCK customer_info IN SHARE MODE NOWAIT;
LOCK TABLE
```

Bob tries to insert another row into the table, but because Alice has the SHARE lock, the statement enters a queue and appears to hang; after Alice finishes her transaction, the INSERT statement will automatically acquire the INSERT lock:

```
=> INSERT INTO customer_info VALUES(17441,'Kara','Shen','Cairo',2022);
```

Alice calls COMMIT, ending her transaction and releasing the SHARE lock:

```
=> COMMIT;
COMMIT;
```

Bob's INSERT statement automatically acquires the lock and completes the operation:

```
=> INSERT INTO customer_info VALUES(17441,'Kara','Shen','Cairo',2022);
 OUTPUT
--------
      1
(1 row)
```

Bob calls COMMIT, ending his transaction and releasing the INSERT lock:

```
=> COMMIT;
COMMIT
```

Deadlocks

Deadlocks can occur when two or more sessions with locks on a table attempt to elevate to lock types incompatible with the lock owned by another session. For example, suppose Bob and Alice each run an INSERT statement:

```
--Alice's session
=> INSERT INTO customer_info VALUES(92837,'Alexander','Lamar','Boston',2022);

--Bob's session
=> INSERT INTO customer_info VALUES(76658,'Midori','Tanaka','Osaka',2021);
```

INSERT (I) locks are compatible , so both Alice and Bob have the lock:

```
=> SELECT * FROM locks;
-[ RECORD 1 ]----------+-----------------------------------------------------------------------------------------------
node_names             | v_vmart_node0001,v_vmart_node0002,v_vmart_node0003
object_name            | Table:public.customer_info
object_id              | 45035996274212656
transaction_id         | 45035996275578544
transaction_description | Txn: a00000001c96b0 'INSERT INTO customer_info VALUES(92837,'Alexander','Lamar','Boston',2022);'
lock_mode              | I
lock_scope             | TRANSACTION
request_timestamp      | 2022-10-05 12:57:49.039967-04
grant_timestamp        | 2022-10-05 12:57:49.039971-04
-[ RECORD 2 ]----------+-----------------------------------------------------------------------------------------------
node_names             | v_vmart_node0001,v_vmart_node0002,v_vmart_node0003
object_name            | Table:public.customer_info
object_id              | 45035996274212656
transaction_id         | 45035996275578546
transaction_description | Txn: a00000001c96b2 'INSERT INTO customer_info VALUES(76658,'Midori','Tanaka','Osaka',2021);'
lock_mode              | I
lock_scope             | TRANSACTION
request_timestamp      | 2022-10-05 12:57:56.599637-04
grant_timestamp        | 2022-10-05 12:57:56.599641-04
```

Alice then runs an UPDATE statement, attempting to elevate her existing INSERT lock into an EXCLUSIVE (X) lock. However, because EXCLUSIVE locks are incompatible with the INSERT lock in Bob's session, the UPDATE is added to a queue for the lock and appears to hang:

```
=> UPDATE customer_info SET city='Cambridge' WHERE customer_id=92837;
```

A deadlock occurs when Bob runs an UPDATE statement while Alice's UPDATE is still waiting. Vertica detects the deadlock and terminates Bob's entire transaction (which includes his INSERT), allowing Alice to elevate to an EXCLUSIVE lock and complete her UPDATE:

```
=> UPDATE customer_info SET city='Shibuya' WHERE customer_id=76658;
ROLLBACK 3010:  Deadlock: initiator locks for query - Deadlock X locking Table:public.customer_info. I held by [user Alice (INSERT INTO customer_info
VALUES(92837,'Alexander','Lamar','Boston',2022);)]. Your current transaction isolation level is SERIALIZABLE
```

Preventing deadlocks

You can avoid deadlocks by acquiring the elevated lock earlier in the transaction, ensuring that your session has exclusive access to the table. You can do this in several ways:

- SELECT...FOR UPDATE : You should prefer this method when possible for your transaction.
- LOCK TABLE...IN EXCLUSIVE MODE

To prevent a deadlock in the previous example, Alice and Bob should both begin their transactions with LOCK TABLE. The first session to request the lock will lock the table, and the other waits for the first session to release the lock or until a user-specified timeout :

```
=> LOCK TABLE customer_info IN EXCLUSIVE MODE;
```

Troubleshooting locks

The LOCKS and LOCK_USAGE system tables can help identify problems you may encounter with Vertica database locks.

This example shows one row from the LOCKS system table. From this table you can see what types of locks are active on specific objects and nodes.

```
=> SELECT node_names, object_name, lock_mode, lock_scope FROM LOCKS;
node_names        | object_name                    | lock_mode | lock_scope
------------------+--------------------------------+-----------+-----------
v_vmart_node0001  | Table:public.customer_dimension | X         | TRANSACTION
```

This example shows two rows from the LOCK_USAGE system table. You can also use this table to see what locks are in use on specific objects and nodes.

```
=> SELECT node_name, object_name, mode FROM LOCK_USAGE;
node_name         | object_name      | mode
------------------+------------------+-------
v_vmart_node0001  | Cluster Topology | S
v_vmart_node0001  | Global Catalog   | X
```

Using text search

Text search allows you to quickly search the contents of a single CHAR, VARCHAR, LONG VARCHAR, VARBINARY, or LONG VARBINARY field within a table to locate a specific keyword.

You can use this feature on columns that are queried repeatedly regarding their contents. After you create the text index, DML operations become slightly slower on the source table. This performance change results from syncing the text index and source table. Any time an operation is performed on the source table, the text index updates in the background. Regular queries on the source table are not affected.

The text index contains all of the words from the source table's text field and any other additional columns you included during index creation. Additional columns are not indexed—their values are just passed through to the text index. The text index is like any other Vertica table , except it is linked to the source table internally.

First, create a text index on the table you plan to search. Then, after you have indexed your table, run a query against the text index for a specific keyword. This query returns a doc_id for each instance of the keyword. After querying the text index, joining the text index back to the source table should give a significant performance improvement over directly querying the source table about the contents of its text field.

Important
Do not alter the contents or definitions of the text index. If you alter the contents or definitions of the text index, the results do not appropriately match the source table.

In this section

- Creating a text index
- Creating a text index on a flex table
- Searching a text index
- Dropping a text index
- Stemmers and tokenizers

Creating a text index

In the following example, you perform a text search using a source table called t_log. This source table has two columns:

- One column containing the table's primary key
- Another column containing log file information

You must associate a projection with the source table. Use a projection that is sorted by the primary key and either segmented by hash(id) or unsegmented. You can define this projection on the source table, along with any other existing projections.

Create a text index on the table for which you want to perform a text search.

```
=> CREATE TEXT INDEX text_index ON t_log (id, text);
```

The text index contains two columns:

- doc_id uses the unique identifier from the source table.
- token is populated with text strings from the designated column from the source table. The word column results from tokenizing and stemming the words found in the text column.

If your table is partitioned then your text index also contains a third column named *partition* .

```
=> SELECT * FROM text_index;
        token         | doc_id | partition
----------------------+--------+-----------
<info>                |      6 |      2014
<warning>             |      2 |      2014
<warning>             |      3 |      2014
<warning>             |      4 |      2014
<warning>             |      5 |      2014
database              |      6 |      2014
execute:              |      6 |      2014
object                |      4 |      2014
object                |      5 |      2014
[catalog]             |      4 |      2014
[catalog]             |      5 |      2014
```

You create a text index on a source table only once. In the future, you do not have to re-create the text index each time the source table is updated or changed.

Your text index stays synchronized to the contents of the source table through any operation that is run on the source table. These operations include, but are not limited to:

- COPY
- INSERT
- UPDATE
- DELETE
- DROP PARTITION
- MOVE_PARTITIONS_TO_TABLE
  When you move or swap partitions in a source table that is indexed, verify that the destination table already exists and is indexed in the same way.

Creating a text index on a flex table

In the following example, you create a text index on a flex table. The example assumes that you have created a flex table called mountains. See Getting started in Using Flex Tables to create the flex table used in this example.

Before you can create a text index on your flex table, add a primary key constraint to the flex table.

```
=> ALTER TABLE mountains ADD PRIMARY KEY (__identity__);
```

Create a text index on the table for which you want to perform a text search. Tokenize the __raw__column with the FlexTokenizer and specify the data type as LONG VARBINARY. It is important to use the FlexTokenizer when creating text indices on flex tables because the data type of the __raw__ column differs from the default StringTokenizer.

```
=> CREATE TEXT INDEX flex_text_index ON mountains(__identity__, __raw__) TOKENIZER public.FlexTokenizer(long varbinary);
```

The text index contains two columns:

- doc_id uses the unique identifier from the source table.
- token is populated with text strings from the designated column from the source table. The word column results from tokenizing and stemming the words found in the text column.

If your table is partitioned then your text index also contains a third column named *partition* .

```
=> SELECT * FROM flex_text_index;
   token    | doc_id
------------+--------
 50.6       |      5
 Mt         |      5
 Washington |      5
 mountain   |      5
 12.2       |      3
 15.4       |      2
 17000      |      3
 29029      |      2
 Denali     |      3
 Helen      |      2
 Mt         |      2
 St         |      2
 mountain   |      3
 volcano    |      2
 29029      |      1
 34.1       |      1
 Everest    |      1
 mountain   |      1
 14000      |      4
 Kilimanjaro |      4
 mountain   |      4
(21 rows)
```

You create a text index on a source table only once. In the future, you do not have to re-create the text index each time the source table is updated or changed.

Your text index stays synchronized to the contents of the source table through any operation that is run on the source table. These operations include, but are not limited to:

- COPY
- INSERT
- UPDATE
- DELETE
- DROP PARTITION
- MOVE_PARTITIONS_TO_TABLE
  When you move or swap partitions in a source table that is indexed, verify that the destination table already exists and is indexed in the same way.

Searching a text index

After you create a text index, write a query to run against the index to search for a specific keyword.

In the following example, you use a WHERE clause to search for the keyword <WARNING> in the text index. The WHERE clause should use the stemmer you used to create the text index. When you use the STEMMER keyword, it stems the keyword to match the keywords in your text index. If you did not use the STEMMER keyword, then the default stemmer is v_txtindex.StemmerCaseInsensitive. If you used STEMMER NONE, then you can omit STEMMER keyword from the WHERE clause.

```
=> SELECT * FROM text_index WHERE token = v_txtindex.StemmerCaseInsensitive('<WARNING>');
  token    | doc_id
-----------+--------
<warning> |     2
<warning> |     3
<warning> |     4
<warning> |     5
(4 rows)
```

Next, write a query to display the full contents of the source table that match the keyword you searched for in the text index.

```
=> SELECT * FROM t_log WHERE id IN (SELECT doc_id FROM text_index WHERE token = v_txtindex.StemmerCaseInsensitive('<WARNING>'));
id |    date    |                                  text
---+------------+-------------------------------------------------------------------------------------------
4  | 2014-06-04 | 11:00:49.568 unknown:0x7f9207607700 [Catalog] <WARNING> validateDependencies: Object 45035968
5  | 2014-06-04 | 11:00:49.568 unknown:0x7f9207607700 [Catalog] <WARNING> validateDependencies: Object 45030
2  | 2013-06-04 | 11:00:49.568 unknown:0x7f9207607700 [Catalog] <WARNING> validateDependencies: Object 4503
3  | 2013-06-04 | 11:00:49.568 unknown:0x7f9207607700 [Catalog] <WARNING> validateDependencies: Object 45066
(4 rows)
```

Use the doc_id to find the exact location of the keyword in the source table.The doc_id matches the unique identifier from the source table. This matching allows you to quickly find the instance of the keyword in your table.

Performing a case-sensitive and case-insensitive text search query

Your text index is optimized to match all instances of words depending upon your stemmer. By default, the case insensitive stemmer is applied to all text indices that do not specify a stemmer. Therefore, if the queries you plan to write against your text index are case sensitive, then Vertica recommends you use a case sensitive stemmer to build your text index.

The following examples show queries that match case-sensitive and case-insensitive words that you can use when performing a text search.

This query finds case-insensitive records in a case insensitive text index:

```
=> SELECT * FROM t_log WHERE id IN (SELECT doc_id FROM text_index WHERE token = v_txtindex.StemmerCaseInsensitive('warning'));
```

This query finds case-sensitive records in a case sensitive text index:

```
=> SELECT * FROM t_log_case_sensitive WHERE id IN (SELECT doc_id FROM text_index WHERE token = v_txtindex.StemmerCaseSensitive('Warning'));
```

Including and excluding keywords in a text search query

Your text index also allows you to perform more detailed queries to find multiple keywords or omit results with other keywords. The following example shows a more detailed query that you can use when performing a text search.

In this example, t_log is the source table, and text_index is the text index. The query finds records that either contain:

- Both the words '<WARNING>' and 'validate'
- Only the word '[Log]' and does not contain 'validateDependencies'

```
SELECT * FROM t_log where (
  id IN (SELECT doc_id FROM text_index WHERE token = v_txtindex.StemmerCaseSensitive('<WARNING>'))
    AND (   id IN (SELECT doc_id FROM text_index WHERE token = v_txtindex.StemmerCaseSensitive('validate')
    OR id IN (SELECT doc_id FROM text_index WHERE token = v_txtindex.StemmerCaseSensitive('[Log]')))
    AND NOT (id IN (SELECT doc_id FROM text_index WHERE token = v_txtindex.StemmerCaseSensitive('validateDependencies'))));
```

This query returns the following results:

```
id |   date     |                            text
----+------------+-------------------------------------------------------------------------------------------
11  | 2014-05-04 | 11:00:49.568 unknown:0x7f9207607702 [Log] <WARNING> validate: Object 4503 via fld num_all_roles
13  | 2014-05-04 | 11:00:49.568 unknown:0x7f9207607706 [Log] <WARNING> validate: Object 45035 refers to root_i3
14  | 2014-05-04 | 11:00:49.568 unknown:0x7f9207607708 [Log] <WARNING> validate: Object 4503 refers to int_2
17  | 2014-05-04 | 11:00:49.568 unknown:0x7f9207607700 [Txn] <WARNING> Begin validate Txn: fff0ed17 catalog editor
(4 rows)
```

Dropping a text index

Dropping a text index removes the specified text index from the database.

You can drop a text index when:

- It is no longer queried frequently.
- An administrative task needs to be performed on the source table and requires the text index to be dropped.

Dropping the text index does not drop the source table associated with the text index. However, if you drop the source table associated with a text index, then that text index is also dropped. Vertica considers the text index a dependent object.

The following example illustrates how to drop a text index named text_index:

```
=> DROP TEXT INDEX text_index;
DROP INDEX
```

Stemmers and tokenizers

Vertica provides default stemmers and tokenizers. You can also create your own custom stemmers and tokenizers. The following topics explain the default stemmers and tokenizers, and the requirements for creating custom stemmers and tokenizers in Vertica.

- Vertica stemmers
- Vertica tokenizers
- Configuring a tokenizer
- Requirements for custom stemmers and tokenizers

In this section

- Vertica stemmers
- Vertica tokenizers
- Configuring a tokenizer
- Requirements for custom stemmers and tokenizers

Vertica stemmers

Vertica *stemmers* use the Porter stemming algorithm to find words derived from the same base/root word. For example, if you perform a search on a text index for the keyword *database* , you might also want to get results containing the word *databases* .

To achieve this type of matching, Vertica stores words in their stemmed form when using any of the v_txtindex stemmers.

The Vertica Analytics Platform provides the following stemmers:

Name	**Description**
v_txtindex.Stemmer(long varchar)	Not sensitive to case; outputs lowercase words. Stems strings from a Vertica table. Alias of StemmerCaseInsensitive.
v_txtindex.StemmerCaseSensitive(long varchar)	Sensitive to case. Stems strings from a Vertica table.
v_txtindex.StemmerCaseInsensitive(long varchar)	Default stemmer used if no stemmer is specified when creating a text index. Not sensitive to case; outputs lowercase words. Stems strings from a Vertica table.
v_txtindex.caseInsensitiveNoStemming(long varchar)	Not sensitive to case; outputs lowercase words. Does not use the Porter Stemming algorithm.

Examples

The following examples show how to use a stemmer when creating a text index.

Create a text index using the StemmerCaseInsensitive stemmer:

```
=> CREATE TEXT INDEX idx_100 ON top_100 (id, feedback) STEMMER v_txtindex.StemmerCaseInsensitive(long varchar)
                                                       TOKENIZER v_txtindex.StringTokenizer(long varchar);
```

Create a text index using the StemmerCaseSensitive stemmer:

```
=> CREATE TEXT INDEX idx_unstruc ON unstruc_data (__identity__, __raw__) STEMMER v_txtindex.StemmerCaseSensitive(long varchar)
                                                                         TOKENIZER public.FlexTokenizer(long varbinary);
```

Create a text index without using a stemmer:

```
=> CREATE TEXT INDEX idx_logs FROM sys_logs ON (id, message) STEMMER NONE TOKENIZER v_txtindex.StringTokenizer(long varchar);
```

Vertica tokenizers

A tokenizer does the following:

- Receives a stream of characters.
- Breaks the stream into individual tokens that usually correspond to individual words.
- Returns a stream of tokens.

In this section

- Preconfigured tokenizers
- Advanced log tokenizer
- Basic log tokenizer
- Whitespace log tokenizer
- ICU tokenizer

Preconfigured tokenizers

The Vertica Analytics Platform provides the following preconfigured tokenizers:

Name	Description
public.FlexTokenizer(LONG VARBINARY)	Splits the values in your flex table by white space.
v_txtindex.StringTokenizer(LONG VARCHAR)	Splits the string into words by splitting on white space.
v_txtindex.StringTokenizerDelim(*string* LONG VARCHAR, ' *delimiter* ' CHAR(1))	Splits a string into tokens using the specified delimiter character.
v_txtindex.AdvancedLogTokenizer	Uses the default parameters for all tokenizer parameters. For more information, see Advanced log tokenizer .
v_txtindex.BasicLogTokenizer	Uses the default values for all tokenizer parameters except minorseparator, which is set to an empty list. For more information, see Basic log tokenizer .
v_txtindex.WhitespaceLogTokenizer	Uses default values for tokenizer parameters, except for majorseparators, which uses E' \t\n\f\r' ; and minorseparator, which uses an empty list. For more information, see Whitespace log tokenizer .

Vertica also provides the following tokenizer, which is not preconfigured:

Name	Description
v_txtindex.ICUTokenizer	Supports multiple languages. Tokenizes based on the conventions of the language you set in the locale parameter. For more information, see ICU Tokenizer.

Examples

The following examples show how you can use a preconfigured tokenizer when creating a text index.

Use the StringTokenizer to create an index from the top_100:

```
=> CREATE TEXT INDEX idx_100 FROM top_100 on (id, feedback)
            TOKENIZER v_txtindex.StringTokenizer(long varchar)
             STEMMER v_txtindex.StemmerCaseInsensitive(long varchar);
```

Use the FlexTokenizer to create an index from unstructured data:

```
=> CREATE TEXT INDEX idx_unstruc FROM unstruc_data on (__identity__, __raw__)
                        TOKENIZER public.FlexTokenizer(long varbinary)
                          STEMMER v_txtindex.StemmerCaseSensitive(long varchar);
```

Use the StringTokenizerDelim to split a string at the specified delimiter:

```
=> CREATE TABLE string_table (word VARCHAR(100), delim VARCHAR);
CREATE TABLE
=> COPY string_table FROM STDIN DELIMITER ',';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>
>> SingleWord,dd
>> Break On Spaces,' '
>> Break:On:Colons,:
>> \.
=> SELECT * FROM string_table;
          word | delim
---------------+-------
     SingleWord | dd
 Break On Spaces |
 Break:On:Colons | :
(3 rows)

=> SELECT v_txtindex.StringTokenizerDelim(word,delim) OVER () FROM string_table;
     words
-----------------
 Break
 On
 Colons
 SingleWor
 Break
 On
 Spaces
(7 rows)

=> SELECT v_txtindex.StringTokenizerDelim(word,delim) OVER (PARTITION BY word), word as input FROM string_table;
          words | input
----------------+-----------------
          Break | Break:On:Colons
             On | Break:On:Colons
         Colons | Break:On:Colons
      SingleWor | SingleWord
          Break | Break On Spaces
             On | Break On Spaces
         Spaces | Break On Spaces
(7 rows)
```

Advanced log tokenizer

Returns tokens that can include minor separators. You can use this tokenizer in situations when your tokens are separated by whitespace or various punctuation. The advanced log tokenizer offers more granularity than the basic log tokenizer in defining separators through the addition of minor separators. This approach is frequently appropriate for analyzing log files.

Important

If you create a database with no tables and the k-safety has increased, you must rebalance your data using REBALANCE_CLUSTER before using a Vertica tokenizer.

Parameters

Parameter Name	Parameter Value
stopwordscaseinsensitive	''
minorseparators	E'/:=@.-$#%_'

majorseparators	E' []<>(){}|!:;,'"*&?+\r\n\t'
minLength	'2'
maxLength	'128'
used	'True'

Examples

The following example shows how you can create a text index, from the table foo, using the Advanced Log Tokenizer without a stemmer.

```
=> CREATE TABLE foo (id INT PRIMARY KEY NOT NULL,text VARCHAR(250));
=> COPY foo FROM STDIN;
End with a backslash and a period on a line by itself.
>> 1|2014-05-10 00:00:05.700433 %ASA-6-302013: Built outbound TCP connection 9986454 for outside:101.123.123.111/443 (101.123.123.111/443)
>> \.
=> CREATE PROJECTION foo_projection AS SELECT * FROM foo ORDER BY id
                              SEGMENTED BY HASH(id) ALL NODES KSAFE;
=> CREATE TEXT INDEX indexfoo_AdvancedLogTokenizer ON foo (id, text)
                TOKENIZER v_txtindex.AdvancedLogTokenizer(LONG VARCHAR) STEMMER NONE;
=> SELECT * FROM indexfoo_AdvancedLogTokenizer;
           token            | doc_id
----------------------------+--------
 %ASA-6-302013:             |      1
 00                         |      1
 00:00:05.700433            |      1
 05                         |      1
 10                         |      1
 101                        |      1
 101.123.123.111/443        |      1
 111                        |      1
 123                        |      1
 2014                       |      1
 2014-05-10                 |      1
 302013                     |      1
 443                        |      1
 700433                     |      1
 9986454                    |      1
 ASA                        |      1
 Built                      |      1
 TCP                        |      1
 connection                 |      1
 for                        |      1
 outbound                   |      1
 outside                    |      1
 outside:101.123.123.111/443 |      1
(23 rows)
```

Basic log tokenizer

Returns tokens that exclude specified minor separators. You can use this tokenizer in situations when your tokens are separated by whitespace or various punctuation. This approach is frequently appropriate for analyzing log files.

Important

If you create a database with no tables and the k-safety has increased, you must rebalance your data using REBALANCE_CLUSTER before using a Vertica tokenizer.

Parameters

Parameter Name	Parameter Value
stopwordscaseinsensitive	''
minorseparators	''
majorseparators	E' []<>(){}|!:,'"*&?+\r\n\t'
minLength	'2'
maxLength	'128'
used	'True'

Examples

The following example shows how you can create a text index, from the table foo, using the Basic Log Tokenizer without a stemmer.

```
=> CREATE TABLE foo (id INT PRIMARY KEY NOT NULL,text VARCHAR(250));
=> COPY foo FROM STDIN;
End with a backslash and a period on a line by itself.
>> 1|2014-05-10 00:00:05.700433 %ASA-6-302013: Built outbound TCP connection 9986454 for outside:101.123.123.111/443 (101.123.123.111/443)
>> \.
=> CREATE PROJECTION foo_projection AS SELECT * FROM foo ORDER BY id
                                SEGMENTED BY HASH(id) ALL NODES KSAFE;
=> CREATE TEXT INDEX indexfoo_BasicLogTokenizer ON foo (id, text)
               TOKENIZER v_txtindex.BasicLogTokenizer(LONG VARCHAR) STEMMER NONE;
=> SELECT * FROM indexfoo_BasicLogTokenizer;
           token            | doc_id
--------------------------------+--------
 %ASA-6-302013:               |      1
 00:00:05.700433              |      1
 101.123.123.111/443          |      1
 2014-05-10                   |      1
 9986454                      |      1
 Built                        |      1
 TCP                          |      1
 connection                   |      1
 for                          |      1
 outbound                     |      1
 outside:101.123.123.111/443 |      1
(11 rows)
```

Whitespace log tokenizer

Returns only tokens surrounded by whitespace. You can use this tokenizer in situations where you want to the tokens in your source document to be separated by whitespace characters only. This approach lets you retain the ability to set stop words and token length limits.

Important

If you create a database with no tables and the k-safety has increased, you must rebalance your data using REBALANCE_CLUSTER before using a Vertica tokenizer.

Parameters

Parameter Name	Parameter Value
stopwordscaseinsensitive	''
minorseparators	''

majorseparators	E' \t\n\f\r'
minLength	'2'
maxLength	'128'
used	'True'

Examples

The following example shows how you can create a text index, from the table foo, using the Whitespace Log Tokenizer without a stemmer.

```
=> CREATE TABLE foo (id INT PRIMARY KEY NOT NULL,text VARCHAR(250));
=> COPY foo FROM STDIN;
End with a backslash and a period on a line by itself.
>> 1|2014-05-10 00:00:05.700433 %ASA-6-302013: Built outbound TCP connection 998 6454 for outside:101.123.123.111/443 (101.123.123.111/443)
>> \.
=> CREATE PROJECTION foo_projection AS SELECT * FROM foo ORDER BY id
                              SEGMENTED BY HASH(id) ALL NODES KSAFE;
=> CREATE TEXT INDEX indexfoo_WhitespaceLogTokenizer ON foo (id, text)
               TOKENIZER v_txtindex.WhitespaceLogTokenizer(LONG VARCHAR) STEMMER NONE;
=> SELECT * FROM indexfoo_WhitespaceLogTokenizer;
           token            | doc_id
----------------------------+--------
 %ASA-6-302013:             |      1
 (101.123.123.111/443)      |      1
 00:00:05.700433            |      1
 2014-05-10                 |      1
 6454                       |      1
 998                        |      1
 Built                      |      1
 TCP                        |      1
 connection                 |      1
 for                        |      1
 outbound                   |      1
 outside:101.123.123.111/443 |      1
(12 rows)
```

ICU tokenizer

Supports multiple languages. You can use this tokenizer to identify word boundaries in languages other than English, including Asian languages that are not separated by whitespace.

The ICU Tokenizer is not pre-configured. You configure the tokenizer by first creating a user-defined transform Function (UDTF). Then set the parameter, locale, to identify the language to tokenizer.

Important

If you create a database with no tables and the k-safety has increased, you must rebalance your data using REBALANCE_CLUSTER before using a Vertica tokenizer.

Parameters

Parameter Name	Parameter Value
locale	Uses the POSIX naming convention: language[_COUNTRY] Identify the language using its ISO-639 code, and the country using its ISO-3166 code. For example, the parameter value for simplified Chinese is zh_CN, and the value for Spanish is es_ES. The default value is English if you do not specify a locale.

Example

The following example steps show how you can configure the ICU Tokenizer for simplified Chinese, then create a text index from the table foo, which contains Chinese characters.

For more on how to configure tokenizers, see Configuring a tokenizer .

1. Create the tokenizer using a UDTF. The example tokenizer is named ICUChineseTokenizer.

```
VMart=> CREATE OR REPLACE TRANSFORM FUNCTION v_txtindex.ICUChineseTokenizer AS LANGUAGE 'C++' NAME 'ICUTokenizerFactory'
LIBRARY v_txtindex.logSearchLib NOT FENCED;
CREATE TRANSFORM FUNCTION
```

2. Get the procedure ID of the tokenizer.

```
VMart=> SELECT proc_oid from vs_procedures where procedure_name = 'ICUChineseTokenizer';
    proc_oid
-------------------
 45035996280452894
(1 row)
```

3. Set the parameter, locale, to simplified Chinese. Identify the tokenizer using its procedure ID.

```
VMart=> SELECT v_txtindex.SET_TOKENIZER_PARAMETER('locale','zh_CN' using parameters proc_oid='45035996280452894');
 SET_TOKENIZER_PARAMETER
-------------------------
 t
(1 row)
```

4. Lock the tokenizer.

```
VMart=> SELECT v_txtindex.SET_TOKENIZER_PARAMETER('used','true' using parameters proc_oid='45035996273762696');
 SET_TOKENIZER_PARAMETER
-------------------------
 t
(1 row)
```

5. Create an example table, foo, containing simplified Chinese text to index.

```
VMart=> CREATE TABLE foo(doc_id integer primary key not null,text varchar(250));
CREATE TABLE

VMart=> INSERT INTO foo values(1, u&'\4E2D\534E\4EBA\6C11\5171\548C\56FD');
 OUTPUT
--------
      1
```

6. Create an index, index_example, on the table foo. The example creates the index without a stemmer; Vertica stemmers work only on English text. Using a stemmer for English on non-English text can cause incorrect tokenization.

```
VMart=> CREATE TEXT INDEX index_example ON foo (doc_id, text) TOKENIZER v_txtindex.ICUChineseTokenizer(long varchar) stemmer none;
CREATE INDEX
```

7. View the new index.

```
VMart=> SELECT * FROM index_example ORDER BY token,doc_id;
 token  | doc_id
--------+--------
 中华   |      1
 人民   |      1
 共和国 |      1
(3 rows)
```

Configuring a tokenizer

You configure a tokenizer by creating a user-defined transform function (UDTF) using one of the two base UDTFs in the v_txtindex.AdvTxtSearchLib library. The library contains two base tokenizers: one for Log Words and one for Ngrams. You can configure each base function with or without positional relevance.

In this section

- Tokenizer base configuration

- RetrieveTokenizerproc_oid
- Set tokenizer parameters
- View tokenizer parameters
- Delete tokenizer config file

Tokenizer base configuration

You can choose among several different tokenizer base configurations:

Type	Position	Without Position
Ngram	logNgramTokenizerPositionFactory	logNgramTokenizerFactory
Words	logWordITokenizerPositionFactory	logWordITokenizerFactory

Create a logWord tokenizer without positional relevance:

```
=> CREATE TRANSFORM FUNCTION v_txtindex.fooTokenizer AS LANGUAGE 'C++' NAME 'logWordITokenizerFactory' LIBRARY v_txtindex.logSearchLib NOT FENCED;
```

RetrieveTokenizerproc_oid

After you create the tokenizer, Vertica writes the name and proc_oid to the system table vs_procedures. You must retrieve the tokenizer's proc_oid to perform additional configuration.

Enter the following query, substituting your own tokenizer name:

```
=> SELECT proc_oid FROM vs_procedures WHERE procedure_name = 'fooTokenizer';
```

Set tokenizer parameters

Use the tokenizer's proc_oid to configure the tokenizer. See Configuring a tokenizer for more information about getting the proc_oid of your tokenizer. The following examples show how you can configure each of the tokenizer parameters:

Configure stop words:

```
=> SELECT v_txtindex.SET_TOKENIZER_PARAMETER('stopwordscaseinsensitive','for,the' USING PARAMETERS proc_oid='45035996274128376');
```

Configure major separators:

```
=> SELECT v_txtindex.SET_TOKENIZER_PARAMETER('majorseparators', E'{}()&[]' USING PARAMETERS proc_oid='45035996274128376');
```

Configure minor separators:

```
=> SELECT v_txtindex.SET_TOKENIZER_PARAMETER('minorseparators', '-,$' USING PARAMETERS proc_oid='45035996274128376');
```

Configure minimum length:

```
=> SELECT v_txtindex.SET_TOKENIZER_PARAMETER('minlength', '1' USING PARAMETERS proc_oid='45035996274128376');
```

Configure maximum length:

```
=> SELECT v_txtindex.SET_TOKENIZER_PARAMETER('maxlength', '140' USING PARAMETERS proc_oid='45035996274128376');
```

Configure ngramssize:

```
=> SELECT v_txtindex.SET_TOKENIZER_PARAMETER('ngramssize', '2' USING PARAMETERS proc_oid='45035996274128376');
```

Lock tokenizer parameters

When you finish configuring the tokenizer, set the parameter, used, to True . After changing this setting, you are no longer able to alter the parameters of the tokenizer. At this point, the tokenizer is ready for you to use to create a text index.

Configure the used parameter:

```
=> SELECT v_txtindex.SET_TOKENIZER_PARAMETER('used', 'True' USING PARAMETERS proc_oid='45035996274128376');
```

See also

SET_TOKENIZER_PARAMETER

View tokenizer parameters

After creating a custom tokenizer, you can view the tokenizer's parameter settings in either of two ways:

- Use the GET_TOKENIZER_PARAMETER — View individual tokenizer parameter settings .
- Use the READ_CONFIG_FILE — View all tokenizer parameter settings .

View individual tokenizer parameter settings

If you need to see an individual parameter setting for a tokenizer, you can use GET_TOKENIZER_PARAMETER to see specific tokenizer parameter settings:

```
=> SELECT v_txtindex.GET_TOKENIZER_PARAMETER('majorseparators' USING PARAMETERS proc_oid='45035996274126984');
 getTokenizerParameter
-----------------------
 {}()&[]
(1 row)
```

For more information, see GET_TOKENIZER_PARAMETER .

View all tokenizer parameter settings

If you need to see all of the parameters for a tokenizer, you can use READ_CONFIG_FILE to see all of the parameter settings for your tokenizer:

```
=> SELECT v_txtindex.READ_CONFIG_FILE( USING PARAMETERS proc_oid='45035996274126984') OVER();
             config_key | config_value
--------------------------+---------------
        majorseparators | {}()&[]
              maxlength | 140
              minlength | 1
        minorseparators | -,$
 stopwordscaseinsensitive | for,the
                   type | 1
                   used | true
(7 rows)
```

If the parameter, used, is set to False , then you can only view the parameters that have been applied to the tokenizer.

> **Note**
> Vertica automatically supplies the value for Type, unless you are using an ngram tokenizer, which allows you to set it.

For more information, see READ_CONFIG_FILE .

Delete tokenizer config file

Use the DELETE_TOKENIZER_CONFIG_FILE function to delete a tokenizer configuration file. This function does not delete the User- Defined Transform Function (UDTF). It only deletes the configuration file associated with the UDTF.

Delete the tokenizer configuration file when the parameter, used, is set to False :

```
=> SELECT v_txtindex.DELETE_TOKENIZER_CONFIG_FILE(USING PARAMETERS proc_oid='45035996274127086');
```

Delete the tokenizer configuration file with the parameter, confirm, set to True . This setting forces the configuration file deletion, even if the parameter, used, is also set to True :

```
=> SELECT v_txtindex.DELETE_TOKENIZER_CONFIG_FILE(USING PARAMETERS proc_oid='45035996274126984', confirm='true');
```

For more information, see DELETE_TOKENIZER_CONFIG_FILE .

Requirements for custom stemmers and tokenizers

Sometimes, you may want specific tokenization or stemming behavior that differs from what Vertica provides. In such cases, you can to implement your own custom User Defined Extensions (UDx) to replace the stemmer or tokenizer. For more information about building custom UDxs see Developing user-defined extensions (UDxs) .

Before implementing a custom stemmer or tokenizer in Vertica verify that the UDx extension meets these requirements.

> **Note**
> Custom tokenizers can return multi-column text indices.

Vertica stemmer requirements

Comply with these requirements when you create custom stemmers:

- Must be a User Defined Scalar Function (UDSF) or a SQL Function
- Can be written in C++, Java, or R
- Volatility set to stable or immutable

Supported Data Input Types :

- Varchar
- Long varchar

Supported Data Output Types :

- Varchar
- Long varchar

Vertica tokenizer requirements

To create custom tokenizers, follow these requirements:

- Must be a User Defined Transform Function (UDTF)
- Can be written in C++, Java, or R
- Input type must match the type of the input text

Supported Data Input Types :

- Char
- Varchar
- Long varchar
- Varbinary
- Long varbinary

Supported Data Output Types :

- Varchar
- Long varchar

Managing storage locations

Vertica *storage locations* are paths to file destinations that you designate to store data and temporary files. Each cluster node requires at least two storage locations: one to store data, and another to store database catalog files. You set up these locations as part of installation and setup. (See Prepare disk storage locations for disk space requirements.)

Important

While no technical issue prevents you from using CREATE LOCATION to add one or more Network File System (NFS) storage locations, Vertica does not support NFS data or catalog storage except for MapR mount points. You will be unable to run queries against any other NFS data. When creating locations on MapR file systems, you must specify ALL NODES SHARED.

How Vertica uses storage locations

When you add data to the database or perform a DML operation, the new data is added to storage locations on disk as ROS containers. Depending on the configuration of your database, many ROS containers are likely to exist.

You can label the storage locations that you create, in order to reference them for object storage policies. If an object has no storage policy associated with it, Vertica uses default storage algorithms to store its data in available storage locations. If the object has a storage policy, Vertica stores the data at the object's designated storage location. You can can retire or drop store locations when you no longer need them.

Local storage locations

By default, Vertica stores data in unique locations on each node. Each location is in a directory in a file system that the node can access, and is often in the node's own file system. You can create a local storage location for a single node or for all nodes in the cluster. Cluster-wide storage locations are the most common type of storage. Vertica defaults to using a local cluster-wide storage location for storing all data. If you want it to store data differently, you must create additional storage locations.

Shared storage locations

You can create shared storage locations, where data is stored on a single file system to which all cluster nodes in the cluster have access. This shared file system is often hosted outside of the cluster, such as on a distributed file system like HDFS. Currently, Vertica supports only HDFS shared storage locations. You cannot use NFS as a Vertica shared storage location except when using MapR mount points. See Vertica Storage Location for HDFS for more information.

When you create a shared storage location for DATA and/or TEMP usage, each node in the Vertica cluster creates its own subdirectory in the shared location. The separate directories prevent nodes from overwriting each other's data.

> **Deprecated**
> SHARED DATA and SHARED DATA,TEMP storage locations are deprecated.

For databases running in Eon Mode, the STORAGE_LOCATIONS system table shows a third type of location, communal.

In this section

- Viewing storage locations and policies
- Creating storage locations
- Storage locations on HDFS
- Altering location use
- Altering location labels
- Creating storage policies
- Creating storage policies for low-priority data
- Moving data storage locations
- Clearing storage policies
- Measuring storage performance
- Setting storage performance
- Retiring storage locations
- Restoring retired storage locations
- Dropping storage locations

Viewing storage locations and policies

You can monitor information about available storage location labels and your current storage policies.

Viewing disk storage information

Query the V_MONITOR.DISK_STORAGE system table for disk storage information on each database node. For more information, see Using system tables and Altering location use . The V_MONITOR.DISK_STORAGE system table includes a CATALOG annotation, indicating that the location is used to store catalog files.

> **Note**
> You cannot add or remove a catalog storage location. Vertica creates and manages this storage location internally, and the area exists in the same location on each cluster node.

Viewing location labels

Three system tables have information about storage location labels in their location_labels columns:

- storage_containers
- storage_locations
- partitions

Use a query such as the following for relevant columns of the storage_containers system table:

```
VMART=> select node_name,projection_name, location_label from v_monitor.storage_containers;
   node_name     |   projection_name    | location_label
-----------------+----------------------+----------------
 v_vmart_node0001 | states_p            |
 v_vmart_node0001 | states_p            |
 v_vmart_node0001 | t1_b1               |
 v_vmart_node0001 | newstates_b0        | FAST3
 v_vmart_node0001 | newstates_b0        | FAST3
 v_vmart_node0001 | newstates_b1        | FAST3
 v_vmart_node0001 | newstates_b1        | FAST3
 v_vmart_node0001 | newstates_b1        | FAST3
 .
 .
 .
```

Use a query such as the following for columns of the v_catalog.storage_locations system_table:

```
VMart=> select node_name, location_path, location_usage, location_label from storage_locations;
   node_name     |              location_path              | location_usage | location_label
-----------------+-----------------------------------------+----------------+----------------
 v_vmart_node0001 | /home/dbadmin/VMart/v_vmart_node0001_data | DATA,TEMP      |
 v_vmart_node0001 | home/dbadmin/SSD/schemas                | DATA           |
 v_vmart_node0001 | /home/dbadmin/SSD/tables                | DATA           | SSD
 v_vmart_node0001 | /home/dbadmin/SSD/schemas               | DATA           | Schema
 v_vmart_node0002 | /home/dbadmin/VMart/v_vmart_node0002_data | DATA,TEMP      |
 v_vmart_node0002 | /home/dbadmin/SSD/tables                | DATA           |
 v_vmart_node0002 | /home/dbadmin/SSD/schemas               | DATA           |
 v_vmart_node0003 | /home/dbadmin/VMart/v_vmart_node0003_data | DATA,TEMP      |
 v_vmart_node0003 | /home/dbadmin/SSD/tables                | DATA           |
 v_vmart_node0003 | /home/dbadmin/SSD/schemas               | DATA           |
(10 rows)
```

Use a query such as the following for columns of the v_monitor.partitions system table:

```
VMART=> select partition_key, projection_name, location_label from v_monitor.partitions;
 partition_key |   projection_name    | location_label
---------------+----------------------+----------------
 NH            | states_b0            | FAST3
 MA            | states_b0            | FAST3
 VT            | states_b1            | FAST3
 ME            | states_b1            | FAST3
 CT            | states_b1            | FAST3
 .
 .
 .
```

Viewing storage tiers

Query the storage_tiers system table to see both the labeled and unlabeled storage containers and information about them:

```
VMart=> select * from v_monitor.storage_tiers;
 location_label | node_count | location_count | ros_container_count | total_occupied_size
----------------+------------+----------------+---------------------+---------------------
                |          1 |              2 |                  17 |           297039391
 SSD            |          1 |              1 |                   9 |                1506
 Schema         |          1 |              1 |                   0 |                   0
(3 rows)
```

Viewing storage policies

Query the storage_policies system table to view the current storage policy in place.

```
VMART=> select * from v_monitor.storage_policies;
 schema_name | object_name |  policy_details  | location_label
-------------+-------------+------------------+----------------
             | public      | Schema           | F4
 public      | lineorder   | Partition [4, 4] | M3
(2 rows)
```

Creating storage locations

You can use CREATE LOCATION to add and configure storage locations (other than the required defaults) to provide storage for these purposes:

- Isolating execution engine temporary files from data files.
- Creating labeled locations to use in storage policies.
- Creating storage locations based on predicted or measured access patterns.
- Creating USER storage locations for specific users or user groups.

Important
While no technical issue prevents you from using CREATE LOCATION to add one or more Network File System (NFS) storage locations, Vertica does not support NFS data or catalog storage except for MapR mount points. You will be unable to run queries against any other NFS data. When creating locations on MapR file systems, you must specify ALL NODES SHARED.

You can add a new storage location from one node to another node or from a single node to all cluster nodes. However, do not use a shared directory on one node for other cluster nodes to access.

Planning storage locations

Before adding a storage location, perform the following steps:

1. Verify that the directory you plan to use for a storage location destination is an empty directory with write permissions for the Vertica process.
2. Plan the labels to use if you want to label the location as you create it.
3. Determine the type of information to store in the storage location:
   - DATA,TEMP (default): The storage location can store persistent and temporary DML-generated data, and data for temporary tables.
   - TEMP : A *path* -specified location to store DML-generated temporary data. If *path* is set to S3, then this location is used only when the RemoteStorageForTemp configuration parameter is set to 1, and TEMP must be qualified with ALL NODES SHARED. For details, see S3 Storage of Temporary Data .
   - DATA : The storage location can only store persistent data.
   - USER : Users with READ and WRITE privileges can access data and external tables of this storage location.
   - DEPOT : The storage location is used in Eon Mode to store the depot. Only create DEPOT storage locations on local Linux file systems. Vertica allows a single DEPOT storage location per node. If you want to move your depot to different location (on a different file system, for example) you must first drop the old depot storage location, then create the new location.

   Storing temp and data files in different storage locations is advantageous because the two types of data have different disk I/O access patterns. Temp files are distributed across locations based on available storage space. However, data files can be stored on different storage locations, based on storage policy, to reflect predicted or measured access patterns.

If you plan to place storage locations on HDFS, see Requirements for HDFS storage locations for additional requirements.

Creating unlabeled local storage locations

This example shows a three-node cluster, each with a vertica/SSD directory for storage.



On each node in the cluster, create a directory where the node stores its data. For example:

```
$ mkdir /home/dbadmin/vertica/SSD
```

Vertica recommends that you create the same directory path on each node. Use this path when creating a storage location.

Use the CREATE LOCATION statement to add a storage location. Specify the following information:

- The path on the node where Vertica stores the data.
  Important
  Vertica does not validate the path that you specify. Confirm that the path value points to a valid location.

- The node where the location is available, or ALL NODES. If you specify ALL NODES, the statement creates the storage locations on all nodes in the cluster in a single transaction.
- The type of information to be stored.

To give unprivileged (non-dbadmin) Linux users access to data, you must create a USER storage location. You can also use a USER storage location to give users without their own credentials access to shared file systems and object stores like HDFS and S3. See Creating a Storage Location for USER Access .

The following example shows how to add a location available on all nodes to store only data:

```
=> CREATE LOCATION '/home/dbadmin/vertica/SSD/' ALL NODES USAGE 'DATA';
```

The following example shows how to add a location that is available on the v_vmart_node0001 node to store data and temporary files:

```
=> CREATE LOCATION '/home/dbadmin/vertica/SSD/' NODE 'v_vmart_node0001';
```

Suppose you are using a storage location for data files and want to create ranked storage locations. In this ranking, columns are stored on different disks based on their measured performance. To create ranked storage locations, see Measuring storage performance and Setting storage performance .

After you create a storage location, you can alter the type of information it stores, with some restrictions. See Altering location use .

Storage location subdirectories

You cannot create a storage location in a subdirectory of an existing storage location. Doing so results in an error similar to the following:

```
=> CREATE LOCATION '/tmp/myloc' ALL NODES USAGE 'TEMP';
CREATE LOCATION
=> CREATE LOCATION '/tmp/myloc/ssd' ALL NODES USAGE 'TEMP';
ERROR 5615:  Location [/tmp/myloc/ssd] conflicts with existing location
[/tmp/myloc] on node v_vmart_node0001
```

Creating labeled storage locations

You can add a storage location with a descriptive label using the CREATE LOCATION statement's LABEL keyword. You use labeled locations to set up storage policies. See Creating storage policies .

This example shows how to create a storage location on v_mart_node0002 with the label SSD:

```
=> CREATE LOCATION '/home/dbadmin/SSD/schemas' NODE 'v_vmart_node0002'
   USAGE 'DATA' LABEL 'SSD';
```

This example shows you how to create a storage location on all nodes. Specifying the ALL NODES keyword adds the storage location to all nodes in a single transaction:

```
=> CREATE LOCATION '/home/dbadmin/SSD/schemas' ALL NODES
   USAGE 'DATA' LABEL 'SSD';
```

The new storage location is listed in the DISK_STORAGE system table:

```
=> SELECT * FROM v_monitor.disk_storage;
.
.
-[ RECORD 7 ]-----------+-----------------------------------------------------
node_name               | v_vmart_node0002
storage_path            | /home/dbadmin/SSD/schemas
storage_usage           | DATA
rank                    | 0
throughput              | 0
latency                 | 0
storage_status          | Active
disk_block_size_bytes   | 4096
disk_space_used_blocks  | 1549437
disk_space_used_mb      | 6053
disk_space_free_blocks  | 13380004
disk_space_free_mb      | 52265
disk_space_free_percent | 89%
...
```

Creating a storage location for USER access

To give unprivileged (non-dbadmin) Linux users access to data, you must create a USER storage location.

By default, Vertica uses user-provided credentials to access external file systems such as HDFS and cloud object stores. You can override this default and create a USER storage location to manage access to these locations. To override the default, set the UseServerIdentityOverUserIdentity configuration parameter.

After you create a USER storage location, you can grant one or more users access to it. USER storage locations grant access only to data files, not temp files. You cannot assign a USER storage location to a storage policy. You cannot change an existing storage location to have USER access.

The following example shows how to create a USER storage location on a specific node:

```
=> CREATE LOCATION '/home/dbadmin/UserStorage/BobStore' NODE
  'v_mcdb_node0007' USAGE 'USER';
```

The following example shows how to grant a specific user read and write permissions to the location:

```
=> GRANT ALL ON LOCATION '/home/dbadmin/UserStorage/BobStore' TO Bob;
GRANT PRIVILEGE
```

The following example shows how to use a USER storage location to grant access to locations on S3. Vertica uses the server credential to access the location:

```
   --- set database-level credential (once):
=> ALTER DATABASE DEFAULT SET AWSAuth = 'myaccesskeyid123456:mysecretaccesskey12345678901234567890123456';

=> CREATE LOCATION 's3://datalake' SHARED USAGE 'USER' LABEL 's3user';

=> CREATE ROLE ExtUsers;
   --- Assign users to this role using GRANT (Role).

=> GRANT READ ON LOCATION 's3://datalake' TO ExtUsers;
```

For more information about configuring user privileges, see Database users and privileges and the GRANT (storage location) and REVOKE (storage location) reference pages.

Shared versus local storage

The SHARED keyword indicates that the location is shared by all nodes. Most remote file systems such as HDFS and S3 are shared. For these file systems, the *path* argument represents a single location in the remote file system where all nodes store data. Each node creates its own subdirectory in the shared storage location for its own files. Doing so prevents one node from overwriting files that belong to other nodes.

If using a remote file system, you must specify SHARED, even for one-node clusters. If the location is declared as USER, Vertica does not create subdirectories for each node. The setting of USER takes precedence over SHARED.

If you create a location and omit this keyword, the new storage location is treated as local. Each node must have unique access to the specified path. This location is usually a path in the node's own file system. Storage locations in file systems that are local to each node, such as Linux, are always local.

> **Deprecated**
> SHARED DATA storage locations are deprecated.

S3 storage of temporary data in Eon Mode

If you are using Vertica in Eon Mode and have limited local disk space, that space might be insufficient to handle the large quantities of temporary data that some DML operations can generate. This is especially true for large load operations and refresh operations.

You can leverage S3 storage to handle temporary data, as follows:

1. Create a remote storage location with CREATE LOCATION , where *path* is set to S3 as follows:
   ```
   => CREATE LOCATION 's3://bucket/path' ALL NODES SHARED USAGE 'TEMP';
   ```
2. Set the RemoteStorageForTemp session configuration parameter to 1:
   ```
   => ALTER SESSION SET RemoteStorageForTemp= 1;
   ```
   A temporary storage location must already exist on S3 before you set this parameter to 1; otherwise, Vertica throws an error and hint to create the storage location.
3. Run the queries that require extra temporary storage.
4. Reset RemoteStorageForTemp to its default value:
   ```
   => ALTER SESSION DEFAULT CLEAR RemoteStorageForTemp;
   ```

When you set RemoteStorageForTemp, Vertica redirects temporary data for all DML operations to the specified remote location. The parameter setting remains in effect until it is explicitly reset to its default value (0), or the current session ends.

Important
Redirecting temporary data to S3 is liable to affect performance and require extra S3 API calls. Use it only for DML operations that involve large quantities of data.

Storage locations on HDFS

You can place storage locations in HDFS, in addition to on the local Linux file system. Because HDFS storage locations are not local, querying them can be slower. You might use HDFS storage locations for lower-priority data or data that is rarely queried (cold data). Moving lower-priority data to HDFS frees space on your Vertica cluster for higher-priority data.

If you are using Vertica for SQL on Apache Hadoop, you typically place storage locations only on HDFS.

In this section

- Requirements for HDFS storage locations
- How the HDFS storage location stores data
- Best practices for Vertica for SQL on Apache Hadoop
- Troubleshooting HDFS storage locations

Requirements for HDFS storage locations

> **Caution**
> If you use HDFS storage locations, the HDFS data must be available when you start Vertica. Your HDFS cluster must be operational, and the ROS files must be present. If you moved data files, or they are corrupted, or your HDFS cluster is not responsive, Vertica cannot start.

To store Vertica's data on HDFS, verify that:

- Your Hadoop cluster has WebHDFS enabled.
- All of the nodes in your Vertica cluster can connect to all of the nodes in your Hadoop cluster. Any firewall between the two clusters must allow connections on the ports used by HDFS.

- If your HDFS cluster is unsecured, you have a Hadoop user whose username matches the name of the Vertica database superuser (usually named dbadmin). This Hadoop user must have read and write access to the HDFS directory where you want Vertica to store its data.
- If your HDFS cluster uses Kerberos authentication:
  - You have a Kerberos principal for Vertica, and it has read and write access to the HDFS directory that will be used for the storage location. See Kerberos below for instructions.
  - The Kerberos KDC is running.
- Your HDFS cluster has enough storage available for Vertica data. See Space Requirements below for details.
- The data you store in an HDFS-backed storage location does not expand your database's size beyond any data allowance in your Vertica license. Vertica counts data stored in an HDFS-backed storage location as part of any data allowance set by your license. See Managing licenses in the Administrator's Guide for more information.

Backup/Restore has additional requirements .

Space requirements

If your Vertica database is K-safe , HDFS-based storage locations contain two copies of the data you store in them. One copy is the primary projection, and the other is the buddy projection. If you have enabled HDFS's data-redundancy feature, Hadoop stores both projections multiple times. This duplication might seem excessive. However, it is similar to how a RAID level 1 or higher stores redundant copies of both the primary and buddy projections. The redundant copies also help the performance of HDFS by enabling multiple nodes to process a request for a file.

Verify that your HDFS installation has sufficient space available for redundant storage of both the primary and buddy projections of your K-safe data. You can adjust the number of duplicates stored by HDFS by setting the HadoopFSReplication configuration parameter. See Troubleshooting HDFS Storage Locations for details.

Kerberos

To use a storage location in HDFS with Kerberos, take the following additional steps:

1. Create a Kerberos principal for each Vertica node as explained in Using Kerberos with Vertica .
2. Give all node principals read and write permission to the HDFS directory you will use as a storage location.

If you plan to use vbr to back up and restore the location, see additional requirements in Requirements for backing up and restoring HDFS storage locations .

Adding HDFS storage locations to new nodes

If you add nodes to your Vertica cluster, they do not automatically have access to existing HDFS storage locations. You must manually create the storage location for the new node using the CREATE LOCATION statement. Do not use the ALL NODES option in this statement. Instead, use the NODE option with the name of the new node to tell Vertica that just that node needs to add the shared location.

> **Caution**
> You must manually create the storage location. Otherwise, the new node uses the default storage policy (usually, storage on the local Linux file system) to store data that the other nodes store in HDFS. As a result, the node can run out of disk space.

Consider an HDFS storage location that was created on a three-node cluster with the following statements:

```
=> CREATE LOCATION 'hdfs://hadoopNS/vertica/colddata' ALL NODES SHARED
   USAGE 'data' LABEL 'coldstorage';

=> SELECT SET_OBJECT_STORAGE_POLICY('SchemaName','coldstorage');
```

The following example shows how to add the storage location to a new cluster node:

```
=> CREATE LOCATION 'hdfs://hadoopNS/vertica/colddata' NODE 'v_vmart_node0004'
  SHARED USAGE 'data' LABEL 'coldstorage';
```

Any active standby nodes in your cluster when you create an HDFS storage location automatically create their own instances of the location. When the standby node takes over for a down node, it uses its own instance of the location to store data for objects using the HDFS storage policy. Treat standby nodes added after you create the storage location as any other new node. You must manually define the HDFS storage location.

How the HDFS storage location stores data

Vertica stores data in storage locations on HDFS similarly to the way it stores data in the Linux file system. When you create a storage location on HDFS, Vertica stores the ROS containers holding its data on HDFS. You can choose which data uses the HDFS storage location: from the data for just a single table or partition to all of the database's data.

When Vertica reads data from or writes data to an HDFS storage location, the node storing or retrieving the data contacts the Hadoop cluster directly to transfer the data. If a single ROS container file is split among several HDFS nodes, the Vertica node connects to each of them. The Vertica node retrieves the pieces and reassembles the file. Because each node fetches its own data directly from the source, data transfers are parallel, increasing their efficiency. Having the Vertica nodes directly retrieve the file splits also reduces the impact on the Hadoop cluster.

What you can store in HDFS

Use HDFS storage locations to store only data. You cannot store catalog information in an HDFS storage location.

> **Caution**
> While it is possible to use an HDFS storage location for temporary data storage, you must never do so. Using HDFS for temporary storage causes severe performance issues.

What HDFS storage locations cannot do

Because Vertica uses storage locations to store ROS containers in a proprietary format, MapReduce and other Hadoop components cannot access your Vertica ROS data stored in HDFS. Never allow another program that has access to HDFS to write to the ROS files. Any outside modification of these files can lead to data corruption and loss. Applications must use the Vertica client libraries to access Vertica data. If you want to share ROS data with other Hadoop components, you can export it (see File export).

Best practices for Vertica for SQL on Apache Hadoop

If you are using the Vertica for SQL on Apache Hadoop product, Vertica recommends the following best practices for storage locations:

- Place only data type storage locations on HDFS storage.
- Place temp space directly on the local Linux file system, not in HDFS.
- For the best performance, place the Vertica catalog directly on the local Linux file system.
- Create the database first on a local Linux file system. Then, you can extend the database to HDFS storage locations and set storage policies that exclusively place data blocks on the HDFS storage location.
- For better performance, if you are running Vertica only on a subset of the HDFS nodes, do not run the HDFS balancer on them. The HDFS balancer can move data blocks farther away, causing Vertica to read non-local data during query execution. Queries run faster if they do not require network I/O.

Generally, HDFS requires approximately 2 GB of memory for each node in the cluster. To support this requirement in your Vertica configuration:

1. Create a 2-GB resource pool.
2. Do not assign any Vertica execution resources to this pool. This approach reserves the space for use by HDFS.

Alternatively, use Ambari or Cloudera Manager to find the maximum heap size required by HDFS and set the size of the resource pool to that value.

For more about how to configure resource pools, see Managing workloads .

Troubleshooting HDFS storage locations

This topic explains some common issues with HDFS storage locations.

HDFS storage disk consumption

By default, HDFS makes three copies of each file it stores. This replication helps prevent data loss due to disk or system failure. It also helps increase performance by allowing several nodes to handle a request for a file.

A Vertica database with a K-safety value of 1 or greater also stores its data redundantly using buddy projections.

When a K-Safe Vertica database stores data in an HDFS storage location, its data redundancy is compounded by HDFS's redundancy. HDFS stores three copies of the primary projection's data, plus three copies of the buddy projection for a total of six copies of the data.

If you want to reduce the amount of disk storage used by HDFS locations, you can alter the number of copies of data that HDFS stores. The Vertica configuration parameter named HadoopFSReplication controls the number of copies of data HDFS stores.

You can determine the current HDFS disk usage by logging into the Hadoop NameNode and issuing the command:

```
$ hdfs dfsadmin -report
```

This command prints the usage for the entire HDFS storage, followed by details for each node in the Hadoop cluster. The following example shows the beginning of the output from this command, with the total disk space highlighted:

```
$ hdfs dfsadmin -report
Configured Capacity: 51495516981 (47.96 GB)
Present Capacity: 32087212032 (29.88 GB)
DFS Remaining: 31565144064 (29.40 GB)
DFS Used: 522067968 (497.88 MB)
DFS Used%: 1.63%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
. . .
```

After loading a simple million-row table into a table stored in an HDFS storage location, the report shows greater disk usage:

```
Configured Capacity: 51495516981 (47.96 GB)
Present Capacity: 32085299338 (29.88 GB)
DFS Remaining: 31373565952 (29.22 GB)
DFS Used: 711733386 (678.76 MB)
DFS Used%: 2.22%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
. . .
```

The following Vertica example demonstrates:

1. Creating the storage location on HDFS.
2. Dropping the table in Vertica.
3. Setting the HadoopFSReplication configuration option to 1. This tells HDFS to store a single copy of an HDFS storage location's data.
4. Recreating the table and reloading its data.

```
=> CREATE LOCATION 'hdfs://hadoopNS/user/dbadmin' ALL NODES SHARED
    USAGE 'data' LABEL 'hdfs';
CREATE LOCATION

=> DROP TABLE messages;
DROP TABLE

=> ALTER DATABASE DEFAULT SET PARAMETER HadoopFSReplication = 1;

=> CREATE TABLE messages (id INTEGER, text VARCHAR);
CREATE TABLE

=> SELECT SET_OBJECT_STORAGE_POLICY('messages', 'hdfs');
 SET_OBJECT_STORAGE_POLICY
---------------------------
Object storage policy set.
(1 row)

=> COPY messages FROM '/home/dbadmin/messages.txt';
 Rows Loaded
-------------
1000000
```

Running the HDFS report on Hadoop now shows less disk space use:

```
$ hdfs dfsadmin -report
Configured Capacity: 51495516981 (47.96 GB)
Present Capacity: 32086278190 (29.88 GB)
DFS Remaining: 31500988416 (29.34 GB)
DFS Used: 585289774 (558.18 MB)
DFS Used%: 1.82%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
. . .
```

Caution

Reducing the number of copies of data stored by HDFS increases the risk of data loss. It can also negatively impact the performance of HDFS by reducing the number of nodes that can provide access to a file. This slower performance can impact the performance of Vertica queries that involve data stored in an HDFS storage location.

ERROR 6966: StorageBundleWriter

You might encounter Error 6966 when loading data into a storage location on a small Hadoop cluster (5 or fewer data nodes). This error is caused by the way HDFS manages the write pipeline and replication. You can mitigate this problem by reducing the number of replicas as explained in HDFS Storage Disk Consumption . For configuration changes you can make in the Hadoop cluster instead, see this blog post from Hortonworks .

Kerberos authentication when creating a storage location

If HDFS uses Kerberos authentication, then the CREATE LOCATION statement authenticates using the Vertica keytab principal, not the principal of the user performing the action. If the creation fails with an authentication error, verify that you have followed the steps described in Kerberos to configure this principal.

When creating an HDFS storage location on a Hadoop cluster using Kerberos, CREATE LOCATION reports the principal being used as in the following example:

```
=> CREATE LOCATION 'hdfs://hadoopNS/user/dbadmin' ALL NODES SHARED
        USAGE 'data' LABEL 'coldstorage';
NOTICE 0: Performing HDFS operations using kerberos principal [vertica/hadoop.example.com]
CREATE LOCATION
```

Backup or restore fails

For issues with backup/restore of HDFS storage locations, see Troubleshooting backup and restore .

Altering location use

ALTER_LOCATION_USE lets you change the type of files that Vertica stores at a storage location. You typically use labels only for DATA storage locations, not TEMP.

This example shows how to alter the storage location on v_vmartdb_node0004 to store only data files:

```
=> SELECT ALTER_LOCATION_USE ('/thirdVerticaStorageLocation/' , 'v_vmartdb_node0004' , 'DATA');
```

Altering HDFS storage locations

When altering an HDFS storage location, you must make the change for all nodes in the Vertica cluster. To do so, specify a node value of '', as in the following example:

```
=> SELECT ALTER_LOCATION_USE('hdfs:///user/dbadmin/v_vmart',
  '','TEMP');
```

Restrictions

You cannot change a storage location from a USER usage type if you created the location that way, or to a USER type if you did not. You can change a USER storage location to specify DATA (storing TEMP files is not supported). However, doing so does not affect the primary objective of a USER storage location, to be accessible by non-dbadmin users with assigned privileges.

You cannot change a storage location from SHARED TEMP or SHARED USER to SHARED DATA or the reverse.

Effects of altering storage location use

Before altering a storage location use type, be aware that at least one location must remain for storing data and temp files on a node. You can store data and temp files in the same, or separate, storage locations.

Altering an existing storage location has the following effects:

Alter use from...	**To store only...**	**Has this effect...**
Temp and data files (or data only)	Temp files	Data content is eventually merged out by the Tuple Mover.You can also manually merge out data from the storage location using DO_TM_TASK . The location stores only temp files from that point forward.
Temp and data files (or temp only)	Data files	Vertica continues to run all statements that use temp files (such as queries and loads). Subsequent statements no longer use the changed storage location for temp files, and the location stores only data files from that point forward.

Altering location labels

ALTER_LOCATION_LABEL lets you change the label for a storage location in several ways:

- Add a label to an unlabeled storage location .
- Remove a label .
- Change an existing label .

You can perform these operations on individual nodes or cluster-wide.

Adding a location label

You add a location label to an unlabeled storage location with ALTER_LOCATION_LABEL. For example, unlabeled storage location /home/dbadmin/Vertica/SSD is defined on a three-node cluster:



You can label this storage location as SSD on all nodes as follows:

```
=> SELECT ALTER_LOCATION_LABEL('/home/dbadmin/vertica/SSD', '', 'SSD');
```

> **Caution**
> If you label a storage location that contains data, Vertica moves the data to an unlabeled location, if one exists. To prevent data movement between storage locations, labels should be applied either to all storage locations or none.

Removing a location label

You can remove a location label only if the following conditions are both true:

- No database object has a storage policy that specifies this label.
- The labeled location is not the last available storage for the objects associated with it.

The following statement removes the SSD label from the specified storage location on all nodes:

```
=> SELECT ALTER_LOCATION_LABEL('/home/dbadmin/SSD/tables','', '');
        ALTER_LOCATION_LABEL
-------------------------------------------
 /home/dbadmin/SSD/tables label changed.
(1 row)
```

Altering a storage location label

You can relabel a storage location only if no database object has a storage policy that specifies this label.

Creating storage policies

Vertica meta-function SET_OBJECT_STORAGE_POLICY creates a storage policy that associates a database object with a labeled storage location. When an object has a storage policy, Vertica uses the labeled location as the default storage location for that object's data.

You can create storage policies for the database, schemas, tables, and partition ranges. Each object can be associated with one storage policy. Each time data is loaded and updated, Vertica checks whether the object has a storage policy. If it does, Vertica uses the labeled storage location. If no storage policy exists for an object or its parent entities, data storage processing continues using standard storage algorithms on available storage locations. If all storage locations are labeled, Vertica uses one of them.

Storage policies let you determine where to store critical data. For example, you might create a storage location with the label SSD that represents the fastest available storage on the cluster nodes. You then create storage policies to associate tables with that labeled location. For example, the following SET_OBJECT_STORAGE_POLICY statement sets a storage policy on table test to use the storage location labeled SSD as its default location:

```
=>  SELECT SET_OBJECT_STORAGE_POLICY('test','ssd', true);
          SET_OBJECT_STORAGE_POLICY
-----------------------------------------------------
Object storage policy set.
Task: moving storages
(Table: public.test) (Projection: public.test_b0)
(Table: public.test) (Projection: public.test_b1)

(1 row)
```

> **Note**
> You cannot include temporary files in storage policies. Storage policies are for use only with data files on storage locations for DATA. Storage policies are not valid for USER locations.

Creating one or more storage policies does not require that policies exist for all database objects. A site can support objects with or without storage policies. You can add storage policies for a discrete set of priority objects, while letting other objects exist without a policy, so they use available storage.

Creating policies based on storage performance

You can measure the performance of any disk storage location (see Measuring storage performance). Then, using the performance measurements, set the storage location performance. Vertica uses the performance measurements you set to rank its storage locations and, through ranking, to determine which key projection columns to store on higher performing locations, as described in Setting storage performance .

If you already set the performance of your site's storage locations, and decide to use storage policies, any storage location with an associated policy has a higher priority than the storage ranking setting.

You can use storage policies to move older data to less-expensive storage locations while keeping it available for queries. See Creating storage policies for low-priority data .

Storage hierarchy and precedence

Vertica determines where to store object data according to the following hierarchy of storage policies, listed below in ascending order of precedence:

1. Database
2. Schema
3. Table
4. Table partition

If an object lacks its own storage policy, it uses the storage policy of its parent object. For example, table Region.Income in database Sales is

partitioned by months. Labeled storage policies FAST and STANDARD are assigned to the table and database, respectively. No storage policy is assigned to the table's partitions or its parent schema, so these use the storage policies of their parent objects, FAST and STANDARD , respectively:

Object	Storage policy	Policy precedence	Labeled location	Default location
Sales (database)	YES	4	STANDARD	STANDARD
Region (schema)	NO	3	N/A	STANDARD
Income (table)	YES	2	FAST	FAST
MONTH (partitions)	NO	1	N/A	FAST

When Tuple Mover operations such as mergeout occur, all Income data moves to the FAST storage location. Other tables in the Region schema use their own storage policy. If a Region table lacks its own storarage policy, Tuple Mover uses the next storage policy above it—in this case, it uses database storage policy and moves the table data to STANDARD .

Querying existing storage policies

You can query existing storage policies, listed in the location_label column of system table STORAGE_CONTAINERS :

```
=> SELECT node_name, projection_name, location_label FROM v_monitor.storage_containers;
   node_name    |   projection_name    | location_label
----------------+----------------------+----------------
 v_vmart_node0001 | states_p           |
 v_vmart_node0001 | states_p           |
 v_vmart_node0001 | t1_b1              |
 v_vmart_node0001 | newstates_b0       | LEVEL3
 v_vmart_node0001 | newstates_b0       | LEVEL3
 v_vmart_node0001 | newstates_b1       | LEVEL3
 v_vmart_node0001 | newstates_b1       | LEVEL3
 v_vmart_node0001 | newstates_b1       | LEVEL3
 v_vmart_node0001 | states_p_v1_node0001 | LEVEL3
 v_vmart_node0001 | states_p_v1_node0001 | LEVEL3
 v_vmart_node0001 | states_p_v1_node0001 | LEVEL3
 v_vmart_node0001 | states_p_v1_node0001 | LEVEL3
 v_vmart_node0001 | states_p_v1_node0001 | LEVEL3
 v_vmart_node0001 | states_p_v1_node0001 | LEVEL3
 v_vmart_node0001 | states_b0          | SSD
 v_vmart_node0001 | states_b0          | SSD
 v_vmart_node0001 | states_b1          | SSD
 v_vmart_node0001 | states_b1          | SSD
 v_vmart_node0001 | states_b1          | SSD
...
```

Forcing existing data storage to a new storage location

By default, the Tuple Mover enforces object storage policies after all pending mergeout operations are complete. SET_OBJECT_STORAGE_POLICY moves existing data storage to a new location immediately, if you set its parameter *enforce-storage-move* to true . You might want to force a move, even though it means waiting for the operation to complete before continuing, if the data being moved is old. The Tuple Mover runs less frequently on older data.

> **Note**
> If parameter *enforce-storage-move* is set to true , SET_OBJECT_STORAGE_POLICY performs a cluster-wide operation. If an error occurs on any node, the function displays a warning message and skips that node. It then continues executing the operation on the remaining nodes.

Creating storage policies for low-priority data

If some of your data is in a partitioned table, you can move less-queried partitions to less-expensive storage such as HDFS. The data is still accessible in queries, just at a slower speed. In this scenario, the faster storage is often referred to as "hot storage," and the slower storage is referred to as "cold storage."

Suppose you have a table named messages (containing social-media messages) that is partitioned by the year and month of the message's timestamp. You can list the partitions in the table by querying the PARTITIONS system table.

```
=> SELECT partition_key, projection_name, node_name, location_label FROM partitions
   ORDER BY partition_key;
partition_key | projection_name |    node_name     | location_label
--------------+-----------------+------------------+----------------
201309        | messages_b1     | v_vmart_node0001 |
201309        | messages_b0     | v_vmart_node0003 |
201309        | messages_b1     | v_vmart_node0002 |
201309        | messages_b1     | v_vmart_node0003 |
201309        | messages_b0     | v_vmart_node0001 |
201309        | messages_b0     | v_vmart_node0002 |
201310        | messages_b0     | v_vmart_node0002 |
201310        | messages_b1     | v_vmart_node0003 |
201310        | messages_b0     | v_vmart_node0001 |
. . .
201405        | messages_b0     | v_vmart_node0002 |
201405        | messages_b1     | v_vmart_node0003 |
201405        | messages_b1     | v_vmart_node0001 |
201405        | messages_b0     | v_vmart_node0001 |
(54 rows)
```

Next, suppose you find that most queries on this table access only the latest month or two of data. You might decide to move the older data to cold storage in an HDFS-based storage location. After you move the data, it is still available for queries, but with lower query performance.

To move partitions to the HDFS storage location, supply the lowest and highest partition key values to be moved in the SET_OBJECT_STORAGE_POLICY function call. The following example shows how to move data between two dates. In this example:

- The partition key value 201309 represents September 2013.
- The partition key value 201403 represents March 2014.
- The name, coldstorage, is the label of the HDFS-based storage location.
- The final argument, which is optional, is true , meaning that the function does not return until the move is complete. By default the function returns immediately and the data is moved when the Tuple Mover next runs. When data is old, however, the Tuple Mover runs less frequently, which would delay recovering the original storage space.

```
=> SELECT SET_OBJECT_STORAGE_POLICY('messages','coldstorage', '201309', '201403', 'true');
```

The partitions within the specified range are moved to the HDFS storage location labeled coldstorage the next time the Tuple Mover runs. This location name now displays in the PARTITIONS system table's location_label column.

```
=> SELECT partition_key, projection_name, node_name, location_label
   FROM partitions ORDER BY partition_key;
partition_key | projection_name |    node_name     | location_label
--------------+-----------------+------------------+----------------
201309        | messages_b0     | v_vmart_node0003 | coldstorage
201309        | messages_b1     | v_vmart_node0001 | coldstorage
201309        | messages_b1     | v_vmart_node0002 | coldstorage
201309        | messages_b0     | v_vmart_node0001 | coldstorage
. . .
201403        | messages_b0     | v_vmart_node0002 | coldstorage
201404        | messages_b0     | v_vmart_node0001 |
201404        | messages_b0     | v_vmart_node0002 |
201404        | messages_b1     | v_vmart_node0001 |
201404        | messages_b1     | v_vmart_node0002 |
201404        | messages_b0     | v_vmart_node0003 |
201404        | messages_b1     | v_vmart_node0003 |
201405        | messages_b0     | v_vmart_node0001 |
201405        | messages_b1     | v_vmart_node0002 |
201405        | messages_b0     | v_vmart_node0002 |
201405        | messages_b0     | v_vmart_node0003 |
201405        | messages_b1     | v_vmart_node0001 |
201405        | messages_b1     | v_vmart_node0003 |
(54 rows)
```

After your initial data move, you can move additional data to the HDFS storage location periodically. You can move individual partitions or a range of partitions from the "hot" storage to the "cold" storage location using the same method:

```
=> SELECT SET_OBJECT_STORAGE_POLICY('messages', 'coldstorage', '201404', '201404', 'true');

=> SELECT projection_name, node_name, location_label
   FROM PARTITIONS WHERE PARTITION_KEY = '201404';
 projection_name |    node_name     | location_label
-----------------+------------------+----------------
 messages_b0     | v_vmart_node0002 | coldstorage
 messages_b0     | v_vmart_node0003 | coldstorage
 messages_b1     | v_vmart_node0003 | coldstorage
 messages_b0     | v_vmart_node0001 | coldstorage
 messages_b1     | v_vmart_node0002 | coldstorage
 messages_b1     | v_vmart_node0001 | coldstorage
(6 rows)
```

Moving partitions to a table stored on HDFS

Another method of moving partitions from hot storage to cold storage is to move the partitions' data to a separate table in the other storage location. This method breaks the data into two tables, one containing hot data and the other containing cold data. Use this method if you want to prevent queries from inadvertently accessing data stored in cold storage. To query the older data, you must explicitly query the cold table.

To move partitions:

1. Create a new table whose schema matches that of the existing partitioned table.
2. Set the storage policy of the new table to use the HDFS-based storage location.
3. Use the MOVE_PARTITIONS_TO_TABLE function to move a range of partitions from the hot table to the cold table. The partitions migrate when the Tuple Mover next runs.

The following example demonstrates these steps. You first create a table named cold_messages. You then assign it the HDFS-based storage location named coldstorage, and, finally, move a range of partitions.

```
=> CREATE TABLE cold_messages LIKE messages INCLUDING PROJECTIONS;
=> SELECT SET_OBJECT_STORAGE_POLICY('cold_messages', 'coldstorage');
=> SELECT MOVE_PARTITIONS_TO_TABLE('messages','201309','201403','cold_messages');
```

Moving data storage locations

SET_OBJECT_STORAGE_POLICY moves data storage from an existing location (labeled and unlabeled) to another labeled location. This function performs two tasks:

- Creates a storage policy for an object, or changes its current policy.
- Moves all existing data for the specified objects to the target storage location.

Before it moves object data to the specified storage location, Vertica calculates the required storage and checks available space at the target. Before calling SET_OBJECT_STORAGE_POLICY , check available space on the new target location. Be aware that checking does not guarantee that this space remains available when the Tuple Mover actually executes the move. If the storage location lacks sufficient free space, the function returns an error.

> **Note**
> Moving an object's current storage to a new target is a cluster-wide operation. If a node is unavailable, the function returns a warning message, and then continues to implement the move on other nodes. When the node rejoins the cluster, the Tuple Mover updates it with the storage data.

By default, the Tuple Mover moves object data to the new storage location after all pending mergeout tasks return. You can force the data to move immediately by setting the function's *enforce-storage-move* argument to true. For example, the following statement sets the storage policy for a table and implements the move immediately:

```
=> SELECT SET_OBJECT_STORAGE_POLICY('states', 'SSD', 'true');
                        SET_OBJECT_STORAGE_POLICY
-----------------------------------------------------------------------------------------------
 Object storage policy set.
Task: moving storages
(Table: public.states) (Projection: public.states_p1)
(Table: public.states) (Projection: public.states_p2)
(Table: public.states) (Projection: public.states_p3)
(1 row)
```

> **Tip**
> Consider using the ENFORCE_OBJECT_STORAGE_POLICY meta-function to relocate the data of multiple database objects as needed, to bring them into compliance with current storage policies. Using this function is equivalent to calling SET_OBJECT_STORAGE_POLICY successively on multiple database objects and setting the *enforce-storage-move* argument to true.

Clearing storage policies

The CLEAR_OBJECT_STORAGE_POLICY meta-function clears a storage policy from a database, schema, table, or table partition. For example, the following statement clears the storage policy for a table:

```
=> SELECT CLEAR_OBJECT_STORAGE_POLICY ('store.store_sales_fact');
  CLEAR_OBJECT_STORAGE_POLICY
--------------------------------
 Object storage policy cleared.
(1 row)
```

The Tuple Mover moves existing storage containers to the parent storage policy's location, or the default storage location if there is no parent policy. By default, this move occurs after all pending mergeout tasks return.

You can force the data to move immediately by setting the function's *enforce-storage-move* argument to true. For example, the following statement clears the storage policy for a table and implements the move immediately:

```
=> SELECT CLEAR_OBJECT_STORAGE_POLICY ('store.store_orders_fact', 'true');
                        CLEAR_OBJECT_STORAGE_POLICY
-----------------------------------------------------------------------------
 Object storage policy cleared.
Task: moving storages
(Table: store.store_orders_fact) (Projection: store.store_orders_fact_b0)
(Table: store.store_orders_fact) (Projection: store.store_orders_fact_b1)

(1 row)
```

> **Tip**
> Consider using the ENFORCE_OBJECT_STORAGE_POLICY meta-function to relocate the data of multiple database objects as needed, to bring them into compliance with current storage policies. Using this function is equivalent to calling CLEAR_OBJECT_STORAGE_POLICY successively on multiple database objects and setting *enforce-storage-move* to true.

Effects on related elements

Clearing a storage policy at one level, such as a table, does not necessarily affect storage policies at other levels, such as that table's partitions.

For example, the lineorder table has a storage policy to store table data at a location labeled F2 . Various partitions in this table are individually assigned their own storage locations, as verified by querying the STORAGE_POLICIES system table:

```
=> SELECT * from v_monitor.storage_policies;
 schema_name | object_name |  policy_details  | location_label
-------------+-------------+------------------+----------------
             | public      | Schema           | F4
 public      | lineorder   | Partition [0, 0] | F1
 public      | lineorder   | Partition [1, 1] | F2
 public      | lineorder   | Partition [2, 2] | F4
 public      | lineorder   | Partition [3, 3] | M1
 public      | lineorder   | Partition [4, 4] | M3
(6 rows)
```

Clearing the current storage policy from the lineorder table has no effect on the storage policies of its individual partitions. For example, given the following CLEAR_OBJECT_STORAGE_POLICY statement:

```
=> SELECT CLEAR_OBJECT_STORAGE_POLICY ('lineorder');
     CLEAR_OBJECT_STORAGE_POLICY
-------------------------------------
 Default storage policy cleared.
(1 row)
```

The individual partitions in the table retain their storage policies:

```

=> SELECT * from v_monitor.storage_policies;
 schema_name | object_name |  policy_details  | location_label
-------------+-------------+------------------+----------------
             | public      | Schema           | F4
 public      | lineorder   | Partition [0, 0] | F1
 public      | lineorder   | Partition [1, 1] | F2
 public      | lineorder   | Partition [2, 2] | F4
 public      | lineorder   | Partition [3, 3] | M1
 public      | lineorder   | Partition [4, 4] | M3
(6 rows)
```

If you clear storage policies from a range of partitions key in a table, the storage policies of parent objects and other partition ranges are unaffected. For example, the following statement clears storage policies from partition keys 0 through 3:

```
=> SELECT CLEAR_OBJECT_STORAGE_POLICY ('lineorder','0','3');
     clear_object_storage_policy
-------------------------------------
 Default storage policy cleared.
(1 row)
=> SELECT * from storage_policies;
 schema_name | object_name |  policy_details  | location_label
-------------+-------------+------------------+----------------
             | public      | Schema           | F4
 public      | lineorder   | Table            | F2
 public      | lineorder   | Partition [4, 4] | M3
(2 rows)
```

Measuring storage performance

Vertica lets you measure disk I/O performance on any storage location at your site. You can use the returned measurements to set performance, which automatically provides rank. Depending on your storage needs, you can also use performance to determine the storage locations needed for critical data as part of your site's storage policies. Storage performance measurements apply only to data storage locations, not temporary storage locations.

Measuring storage location performance calculates the time it takes to read and write 1 MB of data from the disk, which equates to:

```
IO time = time to read/write 1MB + time to seek = 1/throughput + 1/Latency
```

- Throughput is the average throughput of sequential reads/writes (expressed in megabytes per second).
- Latency is for random reads only in seeks (units in seeks per second).

Thus, the I/O time of a faster storage location is less than that of a slower storage location.

> **Note**
> Measuring storage location performance requires extensive disk I/O, which is a resource-intensive operation. Consider starting this operation when fewer other operations are running.

Vertica gives you two ways to measure storage location performance, depending on whether the database is running. You can either:

- Measure performance on a running database.
- Measure performance before a cluster is set up.

Both methods return the throughput and latency for the storage location. Record or capture the throughput and latency information so you can use it to set the location performance (see Setting storage performance).

Measuring performance on a running Vertica database

Use the MEASURE_LOCATION_PERFORMANCE() function to measure performance for a storage location when the database is running. This function has the following requirements:

- The storage path must already exist in the database.
- You need RAM*2 free space available in a storage location to measure its performance. For example, if you have 16 GB RAM, you need 32 GB of available disk space. If you do not have enough disk space, the function returns an error.

Use the system table DISK_STORAGE to obtain information about disk storage on each database node.

The following example shows how to measure the performance of a storage location on v_vmartdb_node0004 :

```
=> SELECT MEASURE_LOCATION_PERFORMANCE('/secondVerticaStorageLocation/','v_vmartdb_node0004');
WARNING:  measure_location_performance can take a long time. Please check logs for progress
        measure_location_performance
-----------------------------------------------------
 Throughput : 122 MB/sec. Latency : 140 seeks/sec
```

Measuring performance before a cluster is set up

You can measure disk performance before setting up a cluster. This approach is useful when you want to verify that the disk is functioning within normal parameters. To perform this measurement, you must already have Vertica installed.

To measure disk performance, use the following command:

```
opt/vertica/bin/vertica -m <path to disk mount>
```

For example:

```
opt/vertica/bin/vertica -m /secondVerticaStorageLocation/node0004_data
```

Setting storage performance

You can use the measurements returned from the MEASURE_LOCATION_PERFORMANCE function as input values to the SET_LOCATION_PERFORMANCE() function.

Note
You must set the throughput and latency parameters of this function to 1 or more.

The following example shows how to set the performance of a storage location on v_vmartdb_node0004 , using values for this location returned from the MEASURE_LOCATION_PERFORMANCE function. Set the throughput to 122 MB/second and the latency to 140 seeks/second.
MEASURE_LOCATION_PERFORMANCE

```
=> SELECT SET_LOCATION_PERFORMANCE('/secondVerticaStorageLocation/','node2','122','140');
```

Sort order ranking by location performance settings

After you set performance-data parameters, Vertica automatically uses performance data to rank storage locations whenever it stores projection columns.

Vertica stores columns included in the projection sort order on the fastest available storage locations. Columns not included in the projection sort order are stored on slower disks. Columns for each projection are ranked as follows:

- Columns in the sort order are given the highest priority (numbers > 1000).
- The last column in the sort order is given the rank number 1001.
- The next-to-last column in the sort order is given the rank number 1002, and so on until the first column in the sort order is given 1000 + # of sort columns.
- The remaining columns are given numbers from 1000–1, starting with 1000 and decrementing by one per column.

Vertica then stores columns on disk from the highest ranking to the lowest ranking. It places highest-ranking columns on the fastest disks and the lowest-ranking columns on the slowest disks.

Using location performance settings with storage policies

You initially measure location performance and set it in the Vertica database. Then, you can use the performance results to determine the fastest storage to use in your storage policies.

- Set the locations with the highest performance as the default locations for critical data.
- Use slower locations as default locations for older, or less-important data. Such slower locations may not require policies at all, if you do not want to specify default locations.

Vertica determines data storage as follows, depending on whether a storage policy exists:

Storage policy	Label	# Locations	Vertica action
No	No	Multiple	Uses ranking (as described), choosing a location from all locations that exist.
Yes	Yes	Single	Uses that storage location exclusively.
Yes	Yes	Multiple	Ranks storage (as described) among all same-name labeled locations.

Retiring storage locations

You can retire a storage location to stop using it. Retiring a storage location prevents Vertica from storing data or temp files to it, but does not remove the actual location. Any data previously stored on the retired location is eventually merged out by the Tuple Mover. Use the RETIRE_LOCATION function to retire a location.

The following example retires a location from a single node:

```
=> SELECT RETIRE_LOCATION('/secondStorageLocation/' , 'v_vmartdb_node0004');
```

To retire a storage location on all nodes, use an empty string ('') for the second argument. If the location is SHARED, you can retire it only on all nodes.

You can expedite retiring and then dropping a storage location by passing an optional third argument, *enforce* , as true . With this directive, the function moves the data out of the storage location instead of waiting for the Tuple Mover, allowing you to drop the location immediately.

You can also use the ENFORCE_OBJECT_STORAGE_POLICY function to trigger the move for all storage locations at once, which allows you to drop the locations. This approach is equivalent to using the *enforce* argument.

The following example shows how to retire a storage location on all nodes so that it can be immediately dropped:

```
=> SELECT RETIRE_LOCATION('/secondStorageLocation/' , '', true);
```

> **Note**
> If the location used in a storage policy is the last available storage for its associated objects, you cannot retire it *unless* you set *enforce* to true .

Data and temp files can be stored in one, or multiple separate, storage locations.

For further information on dropping a location after retiring it, see Dropping storage locations .

Restoring retired storage locations

You can restore a previously retired storage location. After the location is restored, Vertica re-ranks the storage location and uses the restored location to process queries as determined by its rank.

Use the RESTORE_LOCATION function to restore a retired storage location.

The following example shows how to restore a retired storage location on a single node:

```
=> SELECT RESTORE_LOCATION('/secondStorageLocation/' , 'v_vmartdb_node0004');
```

To restore a storage location on all nodes, use an empty string ('') for the second argument. The following example demonstrates creating, retiring, and restoring a location on all nodes:

```
=> CREATE LOCATION '/tmp/ab1' ALL NODES USAGE 'TEMP';
CREATE LOCATION

=> SELECT RETIRE_LOCATION('/tmp/ab1', '');
retire_location
------------------------
/tmp/ab1 retired.
   (1 row)

=> SELECT location_id, node_name, location_path, location_usage, is_retired
        FROM STORAGE_LOCATIONS WHERE location_path ILIKE '/tmp/ab1';
location_id       | node_name           | location_path | location_usage | is_retired
------------------+---------------------+---------------+----------------+------------
45035996273736724 | v_vmart_node0001    | /tmp/ab1      | TEMP           | t
45035996273736726 | v_vmart_node0002    | /tmp/ab1      | TEMP           | t
45035996273736728 | v_vmart_node0003    | /tmp/ab1      | TEMP           | t
45035996273736730 | v_vmart_node0004    | /tmp/ab1      | TEMP           | t
   (4 rows)

=> SELECT RESTORE_LOCATION('/tmp/ab1', '');
restore_location
-------------------------
/tmp/ab1 restored.
   (1 row)

=> SELECT location_id, node_name, location_path, location_usage, is_retired
        FROM STORAGE_LOCATIONS WHERE location_path ILIKE '/tmp/ab1';
location_id       | node_name           | location_path | location_usage | is_retired
------------------+---------------------+---------------+----------------+------------
45035996273736724 | v_vmart_node0001    | /tmp/ab1      | TEMP           | f
45035996273736726 | v_vmart_node0002    | /tmp/ab1      | TEMP           | f
45035996273736728 | v_vmart_node0003    | /tmp/ab1      | TEMP           | f
45035996273736730 | v_vmart_node0004    | /tmp/ab1      | TEMP           | f
   (4 rows)
```

RESTORE_LOCATION restores the location only on the nodes where the location exists and is retired. The meta-function does not propagate the storage location to nodes where that location did not previously exist.

Restoring on all nodes fails if the location has been dropped on any of them. If you have dropped the location on some nodes, you have two options:

- If you no longer want to use the node where the location was dropped, restore the location individually on each of the other nodes.
- Alternatively, you can re-create the location on the node where you dropped it. To do so, use CREATE LOCATION . After you re-create the location, you can then restore it on all nodes.

The following example demonstrates the failure if you try to restore on nodes where you have dropped the location:

```
=> SELECT RETIRE_LOCATION('/tmp/ab1', '');
retire_location
-----------------------
/tmp/ab1 retired.
   (1 row)

=> SELECT DROP_LOCATION('/tmp/ab1', 'v_vmart_node0002');
drop_location
-----------------------
/tmp/ab1 dropped.
   (1 row)

==> SELECT location_id, node_name, location_path, location_usage, is_retired
         FROM STORAGE_LOCATIONS WHERE location_path ILIKE '/tmp/ab1';
location_id       | node_name           | location_path | location_usage | is_retired
------------------+---------------------+---------------+----------------+------------
45035996273736724 | v_vmart_node0001    | /tmp/ab1      | TEMP           | t
45035996273736728 | v_vmart_node0003    | /tmp/ab1      | TEMP           | t
45035996273736730 | v_vmart_node0004    | /tmp/ab1      | TEMP           | t
   (3 rows)

=> SELECT RESTORE_LOCATION('/tmp/ab1', '');
ERROR 2081:  [/tmp/ab1] is not a valid storage location on node v_vmart_node0002
```

Dropping storage locations

To drop a storage location, use the DROP_LOCATION function. You cannot drop locations with DATA usage, only TEMP or USER usage. (See Altering Storage Locations Before Dropping Them .)

Because dropping a storage location cannot be undone, Vertica recommends that you first retire a storage location (see Retiring storage locations). Retiring a storage location before dropping it lets you verify that there will be no adverse effects on any data access. If you decide not to drop it, you can restore it (see Restoring retired storage locations).

The following example shows how to drop a storage location on a single node:

```
=> SELECT DROP_LOCATION('/secondStorageLocation/' , 'v_vmartdb_node0002');
```

When you drop a storage location, the operation cascades to associated objects including any granted privileges to the storage.

> **Caution**
> Dropping a storage location is a permanent operation and cannot be undone. Subsequent queries on storage used for external table access fail with a COPY COMMAND FAILED message.

Altering storage locations before dropping them

You can drop only storage locations containing temp files. Thus, you must alter a storage location to the TEMP usage type before you can drop it. However, if data files still exist in the storage location, Vertica prevents you from dropping it. Deleting data files does not clear the storage location and can result in database corruption. To handle a storage area containing data files so that you can drop it, use one of these options:

- Manually merge out the data files.
- Wait for the Tuple Mover to merge out the data files automatically.
- Retire the location, and force changes to take effect immediately.
- Manually drop partitions .

Dropping HDFS storage locations

After dropping a storage location on HDFS, clean up residual files and snapshots on HDFS as explained in Removing HDFS storage locations .

Dropping USER storage locations

Storage locations that you create with the USER usage type can contain only data files, not temp files. However, you can drop a USER location, regardless of any remaining data files. This behavior differs from that of a storage location not designated for USER access.

Checking location properties

You can check the properties of a storage location, such as whether it is a USER location or is being used only for TEMP files, in the STORAGE_LOCATIONS system table. You can also use this table to verify that a location has been retired.

Analyzing workloads

If queries perform suboptimally, use Workload Analyzer to get tuning recommendations for them and hints about optimizing database objects. Workload Analyzer is a Vertica utility that analyzes system information in Vertica system tables .

Workload Analyzer identifies the root causes of poor query performance through intelligent monitoring of query execution, workload history, resources, and configurations. It then returns a set of tuning recommendations based on statistics, system and data collector events, and database/table/projection design. Use these recommendations to tune query performance, quickly and easily.

You can run Workload Analyzer in two ways:

- Call the Vertica function ANALYZE_WORKLOAD .
- Use the Management Console interface .

See Workload analyzer recommendations for common issues that Workload Analyzer finds, and recommendations.

In this section

- Getting tuning recommendations
- Workload analyzer recommendations

Getting tuning recommendations

Call the function ANALYZE_WORKLOAD to get tuning recommendations for queries and database objects. The function arguments specify what events to analyze and when.

Setting scope and time span

ANALYZE_WORKLOAD 's scope argument determines what to analyze:

This argument...	Returns Workload Analyzer recommendations for...
'' (empty string)	All database objects
Table name	A specific table
Schema name	All objects in the specified schema

The optional since-time argument specifies to return values from all in -scope events starting from since-time and continuing to the current system status. If you omit since_time , ANALYZE_WORKLOAD returns recommendations for events since the last recorded time that you called the function. You must explicitly cast the since-time string value to either TIMESTAMP or TIMESTAMPTZ .

The following examples show four ways to express the since-time argument with different formats. All queries return the same result for workloads on table t1 since October 4, 2012:

```
=> SELECT ANALYZE_WORKLOAD('t1', TIMESTAMP '2012-10-04 11:18:15');
=> SELECT ANALYZE_WORKLOAD('t1', '2012-10-04 11:18:15'::TIMESTAMPTZ);
=> SELECT ANALYZE_WORKLOAD('t1', 'October 4, 2012'::TIMESTAMP);
=> SELECT ANALYZE_WORKLOAD('t1', '10-04-12'::TIMESTAMPTZ);
```

Saving function results

Instead of analyzing events since a specific time, you can save results from ANALYZE_WORKLOAD , by setting the function's second argument to true . The default is false , and no results are saved. After saving function results, subsequent calls to ANALYZE_WORKLOAD analyze only events since you last saved returned data, and ignore all previous events.

For example, the following statement returns recommendations for all database objects in all schemas and records this analysis invocation.

```
=> SELECT ANALYZE_WORKLOAD('', true);
```

The next invocation of ANALYZE_WORKLOAD analyzes events from this point forward.

Observation count and time

The observation_count column returns an integer that represents the total number of events Workload Analyzerobserved for this tuning recommendation. In each case above, Workload Analyzer is making its first recommendation. Null results in observation_time only mean that the recommendations are from the current system status instead of from a prior event.

Tuning targets

The tuning_parameter column returns the object on which Workload Analyzer recommends that you apply the tuning action. The parameter of release in the example above notifies the DBA to set a password for user release.

Tuning recommendations and costs

Workload Analyzer's output returns a brief description of tasks you should consider in the tuning_description column, along with a SQL command you can run, where appropriate, in the tuning_command column . In records 1 and 2 above, Workload Analyzer recommends that you run the Database Designer on two tables, and in record 3 recommends setting a user's password. Record 3 also provides the ALTER USER command to run because the tuning action is a SQL command.

Output in the tuning_cost column indicates the cost of running the recommended tuning command:

- LOW : Running the tuning command has minimal impact on resources. You can perform the tuning operation at any time, like changing the user's password in Record 3 above.
- MEDIUM : Running the tuning command has moderate impact on resources.
- HIGH : Running the tuning command has maximum impact on resources. Depending on the size of your database or table, consider running high-cost operations during off-peak load times.

Examples

The following statement tells Workload Analyzer to analyze all events for the locations table:

```
=> SELECT ANALYZE_WORKLOAD('locations');
```

Workload Analyzer returns with a recommendation that you run the Database Designer on the table, an operation that, depending on the size of locations , might incur a high cost:

```
-[ RECORD 1 ]----------+-------------------------------------------------
observation_count      | 1
first_observation_time |
last_observation_time  |
tuning_parameter       | public.locations
tuning_description     | run database designer on table public.locations
tuning_command         |
tuning_cost            | HIGH
```

The following statement analyzes workloads on all tables in the VMart example database since one week before today:

```
=> SELECT ANALYZE_WORKLOAD('', NOW() - INTERVAL '1 week');
```

Workload Analyzer returns with the following results:

```
-[ RECORD 1 ]----------+-----------------------------------------------------
observation_count      | 4
first_observation_time | 2012-02-17 13:57:17.799003-04
last_observation_time  | 2011-04-22 12:05:26.856456-04
tuning_parameter       | store.store_orders_fact.date_ordered
tuning_description     | analyze statistics on table column store.store_orders_fact.date_ordered
tuning_command         | select analyze_statistics('store.store_orders_fact.date_ordered');
tuning_cost            | MEDIUM
-[ RECORD 2 ]----------+-----------------------------------------------------
...
-[ RECORD 14 ]---------+-----------------------------------------------------
observation_count      | 2
first_observation_time | 2012-02-19 17:52:03.022644-04
last_observation_time  | 2012-02-19 17:52:03.02301-04
tuning_parameter       | SELECT x FROM t WHERE x > (SELECT SUM(DISTINCT x) FROM
                       | t GROUP BY y) OR x < 9;
tuning_description     | consider incremental design on query
tuning_command         |
tuning_cost            | HIGH
```

Workload Analyzer finds two issues:

- In record 1, the date_ordered column in the store.store_orders_fact table likely has stale statistics, so Workload Analyzer suggests running ANALYZE_STATISTICS on that column. The function output also returns the query to run. For example:
  ```
  => SELECT ANALYZE_STATISTICS('store.store_orders_fact.date_ordered');
  ```
- In record 14, Workload Analyzer identifies an under-performing query in the tuning_parameter column. It recommends to use the Database Designer to run an incremental design. Workload Analyzer rates the potential cost as HIGH .

System table recommendations

You can also get tuning recommendations by querying system table TUNING_RECOMMENDATIONS , which returns tuning recommendation results from the last ANALYZE_WORKLOAD call.

```
=> SELECT * FROM tuning_recommendations;
```

System information that Workload Analyzer uses for its recommendations is held in SQL system tables , so querying the TUNING_RECOMMENDATIONS system table does not run Workload Analyzer.

See also

Collecting database statistics

Workload analyzer recommendations

Workload Analyzer monitors database activity and logs recommendations as needed in system table TUNING_RECOMMENDATIONS . When you run Workload Analyzer, the utility returns the following information:

- Description of the object that requires tuning
- Recommended action
- SQL command to implement the recommendation

Common issues and recommendations

Issue	Recommendation
No custom resource pools, user queries are typically handled by the GENERAL resource pool.	Create custom resource pools to handle queries from specific users.
A projection is identified as rarely or never used to execute queries:	Remove the projection with DROP PROJECTION
User with admin privileges has empty password.	Set the password for user with ALTER USER...IDENTIFIED BY .
Table has too many partitions.	Alter the table's partition expression with ALTER TABLE . Also consider grouping partitions and hierarchical partitioning .

Partitioned table data is not fully reorganized after repartitioning.	Reorganize data in the partitioned table with ALTER TABLE...REORGANIZE .
Table has multiple partition keys within the same ROS container.	
Tuple Mover 's MoveOutInterval parameter setting is greater than the default value.	Decrease the parameter setting, or reset the parameter to its default setting.
Average CPU usage exceeds 95% for 20 minutes.	Check system processes, or change resource pool settings of parameters PLANNEDCONCURRENCY and/or MAXCONCURRENCY. For details, see ALTER RESOURCE POOL and Built-in resource pools configuration .
Excessive swap activity; average memory usage exceeds 99% for 10 minutes.	Check system processes
A table does not have any Database Designer-designed projections.	Run database designer on the table. For details, see Incremental Design .
Table statistics are stale.	Run ANALYZE_STATISTICS on table columns. See also Collecting database statistics .
Data distribution in segmented projection is skewed.	Resegment projection on high-cardinality columns. For details, see Designing for segmentation .
Attempts to execute a query generated a GROUP BY spill event.	Consider running an incremental design on the query.
Internal configuration parameter is not the same across nodes.	Reset configuration parameter with ALTER DATABASE...SET
LGE threshold setting is lower than the default setting.	Workload Analyzer does not trigger a tuning recommendation for this scenario unless you altered settings and/or services under the guidance of technical support.
Tuple Mover is disabled.	
Too many ROS containers since the last mergeout operation; configuration parameters are set lower than the default.	
Too many ROS containers since the last mergeout operation; the TM Mergeout service is disabled.	

Managing the database

This section describes how to manage the Vertica database. It includes the following topics:

- Connection load balancing
- Managing nodes
- Adding disk space to a node
- Tuple mover
- Managing the tuple mover
- Managing workloads

In this section

- Managing nodes
- Managing disk space
- Memory usage reporting
- Memory trimming
- Tuple mover
- Managing workloads

- HTTPS service

Managing nodes

Vertica provides the ability to add , remove , and replace nodes on a live cluster that is actively processing queries. This ability lets you scale the database without interrupting users.

In this section

In this section

- Stop Vertica on a node
- Restart Vertica on a node
- Setting node type
- Active standby nodes
- Large cluster
- Multiple databases on a cluster
- Fault groups
- Terrace routing
- Elastic cluster
- Adding nodes
- Removing nodes
- Replacing nodes
- Rebalancing data across nodes
- Redistributing configuration files to nodes
- Stopping and starting nodes on MC
- Upgrading your operating system on nodes in your Vertica cluster
- Reconfiguring node messaging
- Adjusting Spread Daemon timeouts for virtual environments

Stop Vertica on a node

In some cases, you need to take down a node to perform maintenance tasks, or upgrade hardware. You can do this with one of the following:

- Administration Tools
- Command line admintools stop_node

Important

Before removing a node from a cluster, check that the cluster has the minimum number of nodes required to comply with K-safety. If necessary, temporarily lower the database K-safety level .

Administration tools

1. Run Administration Tools, select **Advanced Menu** , and click **OK** .
2. Select **Stop Vertica on Host** and click **OK** .
3. Choose the host that you want to stop and click **OK** .
4. Return to the Main Menu, select **View Database Cluster State** , and click **OK** . The host you previously stopped should appear DOWN.
5. You can now perform maintenance.

See Restart Vertica on a Node for details about restarting Vertica on a node.

Command line

You can use the command line tool stop_node to stop Vertica on one or more nodes. stop_node takes one or more node IP addresses as arguments. For example, the following command stops Vertica on two nodes:

```
$ admintools -t stop_node -s 192.0.2.1,192.0.2.2
```

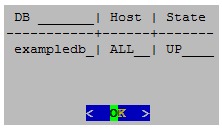
Restart Vertica on a node

After stopping a node to perform maintenance tasks such as upgrading hardware, you need to restart the node so it can reconnect with the database cluster.

1. Run Administration Tools. From the Main Menu select **Restart Vertica on Host** and click **OK** .
2. Select the database and click **OK** .
3. Select the host that you want to restart and click **OK** .

> **Note**
> This process may take a few moments.

4. Return to the Main Menu, select **View Database Cluster State** , and click **OK** . The host you restarted now appears as UP, as shown.



Setting node type

When you create a node, Vertica automatically sets its type to PERMANENT . This enables Vertica to use this node to store data. You can change a node's type with ALTER NODE , to one of the following:

- PERMANENT: (default): A node that stores data.
- EPHEMERAL: A node that is in transition from one type to another—typically, from PERMANENT to either STANDBY or EXECUTE.
- STANDBY: A node that is reserved to replace any node when it goes down. A standby node stores no segments or data until it is called to replace a down node. When used as a replacement node, Vertica changes its type to PERMANENT. For more information, see Active standby nodes .
- EXECUTE: A node that is reserved for computation purposes only. An execute node contains no segments or data.

> **Note**
> STANDBY and EXECUTE node types are supported only in Enterprise Mode.

Active standby nodes

An *active standby node* exists is a node in an Enterprise Mode database that is available to replace any failed node. Unlike permanent Vertica nodes, an standby node does not perform computations or contain data. If a permanent node fails, an active standby node can replace the failed node, after the failed node exceeds the failover time limit. After replacing the failed node, the active standby node contains the projections and performs all calculations of the node it replaced.

In this section

In this section

- Creating an active standby node
- Replace a node with an active standby node
- Revert active standby nodes

Creating an active standby node

You can create active standby nodes in an Enterprise Mode database at the same time that you create the database, or later.

> **Note**
> When you create an active standby node, be sure to add any necessary storage locations. For more information, refer to Adding Storage Locations .

Creating an active standby node in a new database

1. Create a database , including the nodes that you intend to use as active standby nodes.
2. Using vsql, connect to a node **other than** the node that you want to use as an active standby node.
3. Use ALTER NODE to convert the node from a permanent node to an active standby node. For example:

```
=> ALTER NODE v_mart_node5 STANDBY;
```

   After you issue the ALTER NODE statement, the affected node goes down and restarts as an active standby node.

Creating an active standby node in an existing database

When you create a node to be used as an active standby node, change the new node to ephemeral status as quickly as possible to prevent the cluster from moving data to it.

1. Add a node to the database .
   Important
   Do not rebalance the database at this stage.

2. Using vsql, connect to any other node.
3. Use ALTER NODE to convert the new node from a permanent node to an ephemeral node. For example:
   ```
   => ALTER NODE v_mart_node5 EPHEMERAL;
   ```
4. Rebalance the cluster to remove all data from the ephemeral node.
5. Use ALTER NODE on the ephemeral node to convert it to an active standby node. For example:
   ```
   => ALTER NODE v_mart_node5 STANDBY;
   ```

Replace a node with an active standby node

A failed node on an Enterprise Mode database can be replaced with an active standby node automatically, or manually.

Important
A node must be down before it can be replaced with an active standby node. Attempts to replace a node that is up return with an error.

Automatic replacement

You can configure automatic replacement of failed nodes with parameter FailoverToStandbyAfter . If enabled, this parameter specifies the length of time that an active standby node waits before taking the place of a failed node. If possible, Vertica selects a standby node from the same fault group as the failed node. Otherwise, Vertica randomly selects an available active standby node.

Manual replacement

As an administrator, you can manually replace a failed node with ALTER NODE :

1. Connect to the database with Administration Tools or vsql .
2. Replace the node with ALTER NODE...REPLACE. The REPLACE option can specify a standby node. If REPLACE is unqualified, then Vertica selects a standby node from the same fault group as the failed node, if one is available; otherwise, it randomly selects an available active standby node.

Revert active standby nodes

When a down node in an Enterprise Mode database is ready for reactivation, you can restore it by reverting its replacement to standby status. You can perform this operation on individual nodes or the entire database, with ALTER NODE and ALTER DATABASE , respectively:

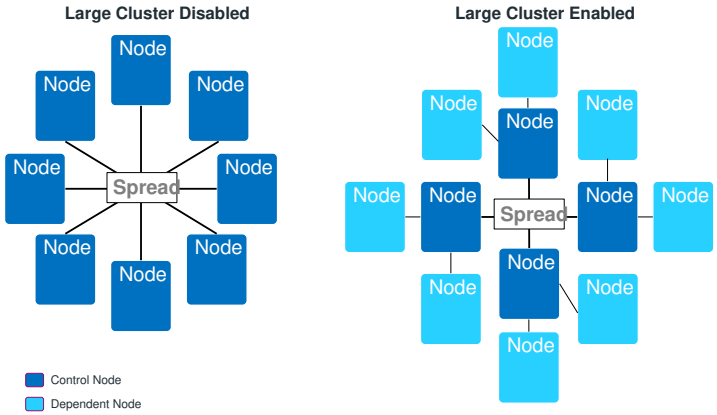
1. Connect to the database with Administration Tools or via vsql .
2. Revert the standby nodes.
   - Individually with ALTER NODE:
     ```
     ALTER NODE node-name RESET;
     ```
   - Collectively across the database cluster with ALTER DATABASE:
     ```
     ALTER DATABASE DEFAULT RESET STANDBY;
     ```

If a down node cannot resume operation, Vertica ignores the reset request and leaves the standby node in place.

Large cluster

Vertica uses the Spread service to broadcast control messages between database nodes. This service can limit the growth of a Vertica database cluster. As you increase the number of cluster nodes, the load on the Spread service also increases as more participants exchange messages. This increased load can slow overall cluster performance. Also, network addressing limits the maximum number of participants in the Spread service to 120 (and often far less). In this case, you can use *large cluster* to overcome these Spread limitations.

When large cluster is enabled, a subset of cluster nodes, called *control nodes* , exchange messages using the Spread service. Other nodes in the cluster are assigned to one of these control nodes, and depend on them for cluster-wide communication. Each control node passes messages from the Spread service to its dependent nodes. When a dependent node needs to broadcast a message to other nodes in the cluster, it passes the message to its control node, which in turn sends the message out to its other dependent nodes and the Spread service.



By setting up dependencies between control nodes and other nodes, you can grow the total number of database nodes, and remain in compliance with the Spread limit of 120 nodes.

> **Note**
> Technically, when large cluster is disabled, all of the nodes in the cluster are control nodes. In this case, all nodes connect to Spread. When large cluster is enabled, some nodes become dependent on control nodes.

A downside of the large cluster feature is that if a control node fails, its dependent nodes are cut off from the rest of the database cluster. These nodes cannot participate in database activities, and Vertica considers them to be down as well. When the control node recovers, it re-establishes communication between its dependent nodes and the database, so all of the nodes rejoin the cluster.

> **Note**
> The Spread service demon runs as an independent process on the control node host. It is not part of the Vertica process. If the Vertica process goes down on the node—for example, you use admintools to stop the Vertica process on the host—Spread continues to run. As long as the Spread demon runs on the control node, the node's dependents can communicate with the database cluster and participate in database activity. Normally, the control node only goes down if the node's host has an issue—or example, you shut it down, it becomes disconnected from the network, or a hardware failure occurs.

Large cluster and database growth

When your database has large cluster enabled, Vertica decides whether to make a newly added node into a control or a dependent node as follows:

- In Enterprise Mode, if the number of control nodes configured for the database cluster is greater than the current number of nodes it contains, Vertica makes the new node a control node. In Eon Mode, the number of control nodes is set at the subcluster level. If the number of control nodes set for the subcluster containing the new node is less than this setting, Vertica makes the new node a control node.
- If the Enterprise Mode cluster or Eon Mode subcluster has reached its limit on control nodes, a new node becomes a dependent of an existing control node.

When a newly-added node is a dependent node, Vertica automatically assigns it to a control node. Which control node it chooses is guided by the database mode:

- Enterprise Mode database: Vertica assigns the new node to the control node with the least number of dependents. If you created fault groups in your database, Vertica chooses a control node in the same fault group as the new node. This feature lets you use fault groups to organize control nodes and their dependents to reflect the physical layout of the underlying host hardware. For example, you might want dependent nodes to be in the same rack as their control nodes. Otherwise, a failure that affects the entire rack (such as a power supply failure) will not only cause nodes in the rack to go down, but also nodes in other racks whose control node is in the affected rack. See Fault groups for more information.
- Eon Mode database: Vertica always adds new nodes to a subcluster. Vertica assigns the new node to the control node with the fewest dependent nodes in that subcluster. Every subcluster in an Eon Mode database with large cluster enabled has at least one control node. Keeping dependent nodes in the same subcluster as their control node maintains subcluster isolation.

Important

In versions of Vertica prior to 10.0.1, nodes in an Eon Mode database with large cluster enabled were not necessarily assigned a control node in their subcluster. If you have upgraded your Eon Mode database from a version of Vertica earlier than 10.0.1 and have large cluster enabled, realign the control nodes in your database. This process reassigns dependent nodes and fixes any cross-subcluster control node dependencies. See Realigning Control Nodes and Reloading Spread for more information.

Spread's upper limit of 120 participants can cause errors when adding a subcluster to an Eon Mode database. If your database cluster has 120 control nodes, attempting to create a subcluster fails with an error. Every subcluster must have at least one control node. When your cluster has 120 control nodes , Vertica cannot create a control node for the new subcluster. If this error occurs, you must reduce the number of control nodes in your database cluster before adding a subcluster.

When to enable large cluster

Vertica automatically enables large cluster in two cases:

- The database cluster contains 120 or more nodes. This is true for both Enterprise Mode and Eon Mode.
- You create an Eon Mode subcluster (either a primary subcluster or a secondary subcluster) with an initial node count of 16 or more. Vertica does not automatically enable large cluster if you expand an existing subcluster to 16 or more nodes by adding nodes to it.

> **Note**
> You can prevent Vertica from automatically enabling large cluster when you create a subcluster with 16 or more nodes by setting the control-set-size parameter to -1. See Creating subclusters for details.

You can choose to manually enable large cluster mode before Vertica automatically enables it. Your best practice is to enable large cluster when your database cluster size reaches a threshold:

- For cloud-based databases, enable large cluster when the cluster contains 16 or more nodes. In a cloud environment, your database uses point-to-point network communications. Spread scales poorly in point-to-point communications mode. Enabling large cluster when the database cluster reaches 16 nodes helps limit the impact caused by Spread being in point-to-point mode.
- For on-premises databases, enable large cluster when the cluster reaches 50 to 80 nodes. Spread scales better in an on-premises environment. However, by the time the cluster size reaches 50 to 80 nodes, Spread may begin exhibiting performance issues.

In either cloud or on-premises environments, enable large cluster if you begin to notice Spread-related performance issues. Symptoms of Spread performance issues include:

- The load on the spread service begins to cause performance issues. Because Vertica uses Spread for cluster-wide control messages, Spread performance issues can adversely affect database performance. This is particularly true for cloud-based databases, where Spread performance problems becomes a bottleneck sooner, due to the nature of network broadcasting in the cloud infrastructure. In on-premises databases, broadcast messages are usually less of a concern because messages usually remain within the local subnet. Even so, eventually, Spread usually becomes a bottleneck before Vertica automatically enables large cluster automatically when the cluster reaches 120 nodes.
- The compressed list of addresses in your cluster is too large to fit in a maximum transmission unit (MTU) packet (1478 bytes). The MTU packet has to contain all of the addresses for the nodes participating in the Spread service. Under ideal circumstances (when your nodes have the IP addresses 1.1.1.1, 1.1.1.2 and so on) 120 addresses can fit in this packet. This is why Vertica automatically enables large cluster if your database cluster reaches 120 nodes. In practice, the compressed list of IP addresses will reach the MTU packet size limit at 50 to 80 nodes.

In this section

- Planning a large cluster
- Enabling large cluster
- Changing the number of control nodes and realigning
- Monitoring large clusters

Planning a large cluster

There are two factors you should consider when planning to expand your database cluster to the point that it needs to use large cluster:

- How many control nodes should your database cluster have?
- How should those control nodes be distributed?

Determining the number of control nodes

When you manually enable large cluster or add enough nodes to trigger Vertica to enable it automatically, a subset of the cluster nodes become control nodes. In subclusters with fewer than 16 nodes, all nodes are control nodes. In many cases, you can set the number of control nodes to the square root of the total number of nodes in the entire Enterprise Mode cluster, or in Eon Mode subclusters with more than 16 nodes. However, this formula for calculating the number of control is not guaranteed to always meet your requirements.

When choosing the number of control nodes in a database cluster, you must balance two competing considerations:

- If a control node fails or is shut down, all nodes that depend on it are cut off from the database. They are also down until the control node rejoins the database. You can reduce the impact of a control node failure by increasing the number of control nodes in your cluster.

- The more control nodes in your cluster, the greater the load on the spread service. In cloud environments, increased complexity of the network environment broadcast can contribute to high latency. This latency can cause messages sent over the spread service to take longer to reach all of the nodes in the cluster.

In a cloud environment, experience has shown that 16 control nodes balances the needs of reliability and performance. In an Eon Mode database, you must have at least one control node per subcluster. Therefore, if you have more than 16 subclusters, you must have more than 16 control nodes.

In an Eon Mode database, whether on-premises or in the cloud, consider adding more control nodes to your primary subclusters than to secondary subclusters. Only nodes in primary subclusters are responsible for maintaining K-safety in an Eon Mode database . Therefore, a control node failure in a primary subcluster can have greater impact on your database than a control node failure in a secondary subcluster.

In an on-premises Enterprise Mode database, consider the physical layout of the hosts running your database when choosing the number of control nodes. If your hosts are spread across multiple server racks, you want to have enough control nodes to distribute them across the racks. Distributing the control nodes helps ensure reliability in the case of a failure that involves the entire rack (such as a power supply or network switch failure). You can configure your database so no node depends on a control node that is in a separate rack. Limiting dependency to within a rack prevents a failure that affects an entire rack from causing additional node loss outside the rack due to control node loss.

Selecting the number of control nodes based on the physical layout also lets you reduce network traffic across switches. By having dependent nodes on the same racks as their control nodes, the communications between them remain in the rack, rather that traversing a network switch.

You might need to increase the number of control nodes to evenly distribute them across your racks. For example, on-premises Enterprise Mode database has 64 total nodes, spread across three racks. The square root of the number of nodes yields 8 control nodes for this cluster. However, you cannot evenly distribute eight control nodes among the three racks. Instead, you can have 9 control nodes and evenly distribute three control nodes per rack.

Influencing control node placement

After you determine the number of nodes for your cluster, you need to determine how to distribute them among the cluster nodes. Vertica chooses which nodes become control nodes. You can influence how Vertica chooses the control nodes and which nodes become their dependents. The exact process you use depends on your database's mode:

- Enterprise Mode on-premises database: Define fault groups to influence control node placement. Dependent nodes are always in the same fault group as their control node. You usually define fault groups that reflect the physical layout of the hosts running your database. For example, you usually define one or more fault groups for the nodes in a single rack of servers. When the fault groups reflect your physical layout, Vertica places control nodes and dependents in a way that can limit the impact of rack failures. See Fault groups for more information.
- Eon Mode database: Use subclusters to control the placement of control nodes. Each subcluster must have at least one control node. Dependent nodes are always in the same subcluster as their control nodes. You can set the number of control nodes for each subcluster. Doing so lets you assign more control nodes to primary subclusters, where it's important to minimize the impact of a control node failure.

How Vertica chooses a default number of control nodes

Vertica can automatically choose the number of control nodes in the entire cluster (when in Enterprise Mode) or for a subcluster (when in Eon Mode). It sets a default value in these circumstances:

- When you pass the default keyword to the --large-cluster option of the install_vertica script (see Enable Large Cluster When Installing Vertica).
- Vertica automatically enables large cluster when your database cluster grows to 120 or more nodes.
- Vertica automatically enables large cluster for an Eon Mode subcluster if you create it with more than 16 nodes. Note that Vertica does not enable large cluster on a subcluster you expand past the 16 node limit. It only enables large clusters that start out larger than 16 nodes.

The number of control nodes Vertica chooses depends on what triggered Vertica to set the value.

If you pass the --large-cluster default option to the install_vertica script, Vertica sets the number of control nodes to the square root of the number of nodes in the initial cluster.

If your database cluster reaches 120 nodes, Vertica enables large cluster by making any newly-added nodes into dependents. The default value for the limit on the number of control nodes is 120. When you reach this limit, any newly-added nodes are added as dependents. For example, suppose you have a 115 node Enterprise Mode database cluster where you have not manually enabled large cluster. If you add 10 nodes to this cluster, Vertica adds 5 of the nodes as control nodes (bringing you up to the 120-node limit) and the other 5 nodes as dependents.

Important

You should manually enable large cluster before your database reaches 120 nodes.

In an Eon Mode database, each subcluster has its own setting for the number of control nodes. Vertica only automatically sets the number of control nodes when you create a subcluster with more than 16 nodes initially. When this occurs, Vertica sets the number of control nodes for the subcluster to the square root of the number of nodes in the subcluster.

For example, suppose you add a new subcluster with 25 nodes in it. This subcluster starts with more than the 16 node limit, so Vertica sets the number of control nodes for subcluster to 5 (which is the square root of 25). Five of the nodes are added as control nodes, and the remaining 20 are added as dependents of those five nodes.

Even though each subcluster has its own setting for the number of control nodes, an Eon Mode database cluster still has the 120 node limit on the total number of control nodes that it can have.

Enabling large cluster

Vertica enables the large cluster feature automatically when:

- The total number of nodes in the database cluster exceeds 120.
- You create an Eon Mode subcluster with more than 16 nodes.

In most cases, you should consider manually enabling large cluster before your cluster size reaches either of these thresholds. See Planning a large cluster for guidance on when to enable large cluster.

You can enable large cluster on a new Vertica database , or on an existing database .

Enable large cluster when installing Vertica

You can enable large cluster when installing Vertica onto a new database cluster. This option is useful if you know from the beginning that your database will benefit from large cluster.

The install_vertica script's --large-cluster argument enables large cluster during installation. It takes a single integer value between 1 and 120 that specifies the number of control nodes to create in the new database cluster. Alternatively, this option can take the literal argument default . In this case, Vertica enables large cluster mode and sets the number of control nodes to the square root of the number nodes you provide in the --hosts argument. For example, if --hosts specifies 25 hosts and --large-cluster is set to default , the install script creates a database cluster with 5 control nodes.

The --large-cluster argument has a slightly different effect depending on the database mode you choose when creating your database:

- Enterprise Mode: --large-cluster sets the total number of control nodes for the entire database cluster.
- Eon Mode : --large-cluster sets the number of control nodes in the initial default subcluster . This setting has no effect on subclusters that you create later.

> **Note**
> You cannot use --large-cluster to set the number of control nodes in your initial database to be higher than the number of you pass in the --hosts argument. The installer sets the number of control nodes to whichever is the lower value: the value you pass to the --large-cluster option or the number of hosts in the --hosts option.
>
> You can set the number of control nodes to be higher than the number of nodes currently in an existing database, with the meta-function SET_CONTROL_SET_SIZE function. You choose to set a higher number to preallocate control nodes when planning for future expansion. For details, see Changing the number of control nodes and realigning .

After the installation process completes, use the Administration tools or the [%=Vertica.MC%] to create a database. See Create an empty database for details.

If your database is on-premises and running in Enterprise Mode, you usually want to define fault groups that reflect the physical layout of your hosts. They let you define which hosts are in the same server racks, and are dependent on the same infrastructure (such power supplies and network switches). With this knowledge, Vertica can realign the control nodes to make your database better able to cope with hardware failures. See Fault groups for more information.

After creating a database, any nodes that you add are, by default, dependent nodes. You can change the number of control nodes in the database with the meta-function SET_CONTROL_SET_SIZE .

Enable large cluster in an existing database

You can manually enable large cluster in an existing database. You usually choose to enable large cluster manually before your database reaches the point where Vertica automatically enables it. See When To Enable Large Cluster for an explanation of when you should consider enabling large cluster.

Use the meta-function SET_CONTROL_SET_SIZE to enable large cluster and set the number of control nodes . You pass this function an integer value that sets the number of control nodes in the entire Enterprise Mode cluster, or in an Eon Mode subcluster.

Changing the number of control nodes and realigning

You can change the number of control nodes in the entire database cluster in Enterprise Mode, or the number of control nodes in a subcluster in Eon Mode. You may choose to change the number of control nodes in a cluster or subcluster to reduce the impact of control node loss on your database. See Planning a large cluster to learn more about when you should change the number of control nodes in your database.

You change the number of control nodes by calling the meta-function SET_CONTROL_SET_SIZE . If large cluster was not enabled before the call to SET_CONTROL_SET_SIZE, the function enables large cluster in your database. See Enabling large cluster for more information.

When you call SET_CONTROL_SET_SIZE in an Enterprise Mode database, it sets the number of control nodes in the entire database cluster. In an Eon Mode database, you must supply SET_CONTROL_SET_SIZE with the name of a subcluster in addition to the number of control nodes. The function sets the number of control nodes for that subcluster. Other subclusters in the database cluster are unaffected by this call.

Before changing the number of control nodes in an Eon Mode subcluster, verify that the subcluster is running. Changing the number of control nodes of a subcluster while it is down can cause configuration issues that prevent nodes in the subcluster from starting.

> **Note**
> You can set the number of control nodes to a value that is higher than the number of nodes currently in the cluster or subcluster. When the number of control nodes is higher than the current node count, newly-added nodes become control nodes until the number of nodes in the cluster or subcluster reaches the number control nodes you set.
>
> You may choose to set the number of control nodes higher than the current node count to plan for future expansion. For example, suppose you have a 4-node subcluster in an Eon Mode database that you plan to expand in the future. You determine that you want limit the number of control nodes in this cluster to 8, even if you expand it beyond that size. In this case, you can choose to set the control node size for the subcluster to 8 now. As you add new nodes to the subcluster, they become control nodes until the size of the subcluster reaches 8. After that point, Vertica assigns newly-added nodes as a dependent of an existing control node in the subcluster.

Realigning control nodes and reloading spread

After you call the SET_CONTROL_SET_SIZE function, there are several additional steps you must take before the new setting takes effect.

Important
Follow these steps if you have upgraded your large-cluster enabled Eon Mode database from a version prior to 10.0.1. Earlier versions of Vertica did not restrict control node assignments to be within the same subcluster. When you realign the control nodes after an upgrade, Vertica configures each subcluster to have at least one control node, and assigns nodes to a control node in their own subcluster.

1. Call the REALIGN_CONTROL_NODES function. This function tells Vertica to re-evaluate the assignment of control nodes and their dependents in your cluster or subcluster. When calling this function in an Eon Mode database, you must supply the name of the subcluster where you changed the control node settings.
2. Call the RELOAD_SPREAD function. This function updates the control node assignment information in configuration files and triggers Spread to reload.
3. Restart the nodes affected by the change in control nodes. In an Enterprise Mode database, you must restart the entire database to ensure all nodes have updated configuration information. In Eon Mode, restart the subcluster or subclusters affected by your changes. You must restart the entire Eon Mode database if you changed a critical subcluster (such as the only primary subcluster).

   > **Note**
   > You do not need to restart nodes if the earlier steps didn't change control node assignments. This case usually only happens when you set the number of control nodes in an Eon Mode subcluster to higher than the subcluster's current node count, and all nodes in the subcluster are already control nodes. In this case, no control nodes are added or removed, so node dependencies do not change. Because the dependencies did not change, the nodes do not need to reload the Spread configuration.

4. In an Enterprise Mode database, call START_REBALANCE_CLUSTER to rebalance the cluster. This process improves your database's fault tolerance by shifting buddy projection assignments to limit the impact of a control node failure. You do not need to take this step in an Eon Mode database.

Enterprise Mode example

The following example makes 4 out of the 8 nodes in an Enterprise Mode database into control nodes. It queries the LARGE_CLUSTER_CONFIGURATION_STATUS system table which shows control node assignments for each node in the database. At the start, all nodes are their own control nodes. See Monitoring large clusters for more information the system tables associated with large cluster.

```
=> SELECT * FROM V_CATALOG.LARGE_CLUSTER_CONFIGURATION_STATUS;
    node_name     | spread_host_name | control_node_name
------------------+------------------+-------------------
 v_vmart_node0001 | v_vmart_node0001 | v_vmart_node0001
 v_vmart_node0002 | v_vmart_node0002 | v_vmart_node0002
 v_vmart_node0003 | v_vmart_node0003 | v_vmart_node0003
 v_vmart_node0004 | v_vmart_node0004 | v_vmart_node0004
 v_vmart_node0005 | v_vmart_node0005 | v_vmart_node0005
 v_vmart_node0006 | v_vmart_node0006 | v_vmart_node0006
 v_vmart_node0007 | v_vmart_node0007 | v_vmart_node0007
 v_vmart_node0008 | v_vmart_node0008 | v_vmart_node0008
(8 rows)

=> SELECT SET_CONTROL_SET_SIZE(4);
 SET_CONTROL_SET_SIZE
----------------------
 Control size set
(1 row)

=> SELECT REALIGN_CONTROL_NODES();
                    REALIGN_CONTROL_NODES
-------------------------------------------------------------------
 The new control node assignments can be viewed in vs_nodes.
 Check vs_cluster_layout to see the proposed new layout. Reboot
 all the nodes and call rebalance_cluster now

(1 row)

=> SELECT RELOAD_SPREAD(true);
 RELOAD_SPREAD
---------------
 Reloaded
(1 row)

=> SELECT SHUTDOWN();
```

After restarting the database, the final step is to rebalance the cluster and query the LARGE_CLUSTER_CONFIGURATION_STATUS table to see the current control node assignments:

```
=> SELECT START_REBALANCE_CLUSTER();
 START_REBALANCE_CLUSTER
-------------------------
 REBALANCING
(1 row)

=> SELECT * FROM V_CATALOG.LARGE_CLUSTER_CONFIGURATION_STATUS;
    node_name     | spread_host_name | control_node_name
------------------+------------------+-------------------
 v_vmart_node0001 | v_vmart_node0001 | v_vmart_node0001
 v_vmart_node0002 | v_vmart_node0002 | v_vmart_node0002
 v_vmart_node0003 | v_vmart_node0003 | v_vmart_node0003
 v_vmart_node0004 | v_vmart_node0004 | v_vmart_node0004
 v_vmart_node0005 | v_vmart_node0001 | v_vmart_node0001
 v_vmart_node0006 | v_vmart_node0002 | v_vmart_node0002
 v_vmart_node0007 | v_vmart_node0003 | v_vmart_node0003
 v_vmart_node0008 | v_vmart_node0004 | v_vmart_node0004
(8 rows)
```

Eon Mode example

The following example configures 4 control nodes in an 8-node secondary subcluster named analytics. The primary subcluster is not changed. The primary differences between this example and the previous Enterprise Mode example is the need to specify a subcluster when calling SET_CONTROL_SET_SIZE, not having to restart the entire database, and not having to call START_REBALANCE_CLUSTER.

```
=>  SELECT * FROM V_CATALOG.LARGE_CLUSTER_CONFIGURATION_STATUS;
     node_name       |   spread_host_name   |  control_node_name
---------------------+----------------------+---------------------
 v_verticadb_node0001 | v_verticadb_node0001 | v_verticadb_node0001
 v_verticadb_node0002 | v_verticadb_node0002 | v_verticadb_node0002
 v_verticadb_node0003 | v_verticadb_node0003 | v_verticadb_node0003
 v_verticadb_node0004 | v_verticadb_node0004 | v_verticadb_node0004
 v_verticadb_node0005 | v_verticadb_node0005 | v_verticadb_node0005
 v_verticadb_node0006 | v_verticadb_node0006 | v_verticadb_node0006
 v_verticadb_node0007 | v_verticadb_node0007 | v_verticadb_node0007
 v_verticadb_node0008 | v_verticadb_node0008 | v_verticadb_node0008
 v_verticadb_node0009 | v_verticadb_node0009 | v_verticadb_node0009
 v_verticadb_node0010 | v_verticadb_node0010 | v_verticadb_node0010
 v_verticadb_node0011 | v_verticadb_node0011 | v_verticadb_node0011
(11 rows)


=> SELECT subcluster_name,node_name,is_primary,control_set_size FROM
   V_CATALOG.SUBCLUSTERS;
  subcluster_name   |      node_name       | is_primary | control_set_size
--------------------+----------------------+------------+------------------
 default_subcluster | v_verticadb_node0001 | t          |               -1
 default_subcluster | v_verticadb_node0002 | t          |               -1
 default_subcluster | v_verticadb_node0003 | t          |               -1
 analytics          | v_verticadb_node0004 | f          |               -1
 analytics          | v_verticadb_node0005 | f          |               -1
 analytics          | v_verticadb_node0006 | f          |               -1
 analytics          | v_verticadb_node0007 | f          |               -1
 analytics          | v_verticadb_node0008 | f          |               -1
 analytics          | v_verticadb_node0009 | f          |               -1
 analytics          | v_verticadb_node0010 | f          |               -1
 analytics          | v_verticadb_node0011 | f          |               -1
(11 rows)

=> SELECT SET_CONTROL_SET_SIZE('analytics',4);
 SET_CONTROL_SET_SIZE
----------------------
 Control size set
(1 row)

=> SELECT REALIGN_CONTROL_NODES('analytics');
                    REALIGN_CONTROL_NODES
-------------------------------------------------------------------------------
 The new control node assignments can be viewed in vs_nodes. Call
reload_spread(true). If the subcluster is critical, restart the database.
Otherwise, restart the subcluster

(1 row)

=> SELECT RELOAD_SPREAD(true);
 RELOAD_SPREAD
---------------
 Reloaded
(1 row)
```

At this point, the analytics subcluster needs to restart. You have several options to restart it. See Starting and stopping subclusters for details. This example uses the admintools command line to stop and start the subcluster.

```
$ admintools -t stop_subcluster -d verticadb -c analytics -p password
*** Forcing subcluster shutdown ***
Verifying subcluster 'analytics'
    Node 'v_verticadb_node0004' will shutdown
    Node 'v_verticadb_node0005' will shutdown
    Node 'v_verticadb_node0006' will shutdown
    Node 'v_verticadb_node0007' will shutdown
    Node 'v_verticadb_node0008' will shutdown
    Node 'v_verticadb_node0009' will shutdown
    Node 'v_verticadb_node0010' will shutdown
    Node 'v_verticadb_node0011' will shutdown
Shutdown subcluster command successfully sent to the database

$ admintools -t restart_subcluster -d verticadb -c analytics -p password
*** Restarting subcluster for database verticadb ***
    Restarting host [10.11.12.19] with catalog [v_verticadb_node0004_catalog]
    Restarting host [10.11.12.196] with catalog [v_verticadb_node0005_catalog]
    Restarting host [10.11.12.51] with catalog [v_verticadb_node0006_catalog]
    Restarting host [10.11.12.236] with catalog [v_verticadb_node0007_catalog]
    Restarting host [10.11.12.103] with catalog [v_verticadb_node0008_catalog]
    Restarting host [10.11.12.185] with catalog [v_verticadb_node0009_catalog]
    Restarting host [10.11.12.80] with catalog [v_verticadb_node0010_catalog]
    Restarting host [10.11.12.47] with catalog [v_verticadb_node0011_catalog]
    Issuing multi-node restart
    Starting nodes:
        v_verticadb_node0004 (10.11.12.19) [CONTROL]
        v_verticadb_node0005 (10.11.12.196) [CONTROL]
        v_verticadb_node0006 (10.11.12.51) [CONTROL]
        v_verticadb_node0007 (10.11.12.236) [CONTROL]
        v_verticadb_node0008 (10.11.12.103)
        v_verticadb_node0009 (10.11.12.185)
        v_verticadb_node0010 (10.11.12.80)
        v_verticadb_node0011 (10.11.12.47)
    Starting Vertica on all nodes. Please wait, databases with a large catalog may take a while to initialize.
    Node Status: v_verticadb_node0004: (DOWN) v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
                 v_verticadb_node0007: (DOWN) v_verticadb_node0008: (DOWN) v_verticadb_node0009: (DOWN)
                 v_verticadb_node0010: (DOWN) v_verticadb_node0011: (DOWN)
    Node Status: v_verticadb_node0004: (DOWN) v_verticadb_node0005: (DOWN) v_verticadb_node0006: (DOWN)
                 v_verticadb_node0007: (DOWN) v_verticadb_node0008: (DOWN) v_verticadb_node0009: (DOWN)
                 v_verticadb_node0010: (DOWN) v_verticadb_node0011: (DOWN)
    Node Status: v_verticadb_node0004: (INITIALIZING) v_verticadb_node0005: (INITIALIZING) v_verticadb_node0006:
                 (INITIALIZING) v_verticadb_node0007: (INITIALIZING) v_verticadb_node0008: (INITIALIZING)
                 v_verticadb_node0009: (INITIALIZING) v_verticadb_node0010: (INITIALIZING) v_verticadb_node0011: (INITIALIZING)
    Node Status: v_verticadb_node0004: (UP) v_verticadb_node0005: (UP) v_verticadb_node0006: (UP)
                 v_verticadb_node0007: (UP) v_verticadb_node0008: (UP) v_verticadb_node0009: (UP)
                 v_verticadb_node0010: (UP) v_verticadb_node0011: (UP)
Syncing catalog on verticadb with 2000 attempts.
```

Once the subcluster restarts, you can query the system tables to see the control node configuration:

```
=>  SELECT * FROM V_CATALOG.LARGE_CLUSTER_CONFIGURATION_STATUS;
     node_name       |   spread_host_name   |  control_node_name
---------------------+----------------------+---------------------
 v_verticadb_node0001 | v_verticadb_node0001 | v_verticadb_node0001
 v_verticadb_node0002 | v_verticadb_node0002 | v_verticadb_node0002
 v_verticadb_node0003 | v_verticadb_node0003 | v_verticadb_node0003
 v_verticadb_node0004 | v_verticadb_node0004 | v_verticadb_node0004
 v_verticadb_node0005 | v_verticadb_node0005 | v_verticadb_node0005
 v_verticadb_node0006 | v_verticadb_node0006 | v_verticadb_node0006
 v_verticadb_node0007 | v_verticadb_node0007 | v_verticadb_node0007
 v_verticadb_node0008 | v_verticadb_node0004 | v_verticadb_node0004
 v_verticadb_node0009 | v_verticadb_node0005 | v_verticadb_node0005
 v_verticadb_node0010 | v_verticadb_node0006 | v_verticadb_node0006
 v_verticadb_node0011 | v_verticadb_node0007 | v_verticadb_node0007
(11 rows)

=> SELECT subcluster_name,node_name,is_primary,control_set_size FROM subclusters;
  subcluster_name   |      node_name       | is_primary | control_set_size
--------------------+----------------------+------------+------------------
 default_subcluster | v_verticadb_node0001 | t          |               -1
 default_subcluster | v_verticadb_node0002 | t          |               -1
 default_subcluster | v_verticadb_node0003 | t          |               -1
 analytics          | v_verticadb_node0004 | f          |                4
 analytics          | v_verticadb_node0005 | f          |                4
 analytics          | v_verticadb_node0006 | f          |                4
 analytics          | v_verticadb_node0007 | f          |                4
 analytics          | v_verticadb_node0008 | f          |                4
 analytics          | v_verticadb_node0009 | f          |                4
 analytics          | v_verticadb_node0010 | f          |                4
 analytics          | v_verticadb_node0011 | f          |                4
(11 rows)
```

Disabling large cluster

To disable large cluster, call SET_CONTROL_SET_SIZE with a value of -1. This value is the default for non-large cluster databases. It tells Vertica to make all nodes into control nodes.

In an Eon Mode database, to fully disable large cluster you must to set the number of control nodes to -1 in every subcluster that has a set number of control nodes. You can see which subclusters have a set number of control nodes by querying the CONTROL_SET_SIZE column of the V_CATALOG.SUBCLUSTERS system table.

The following example resets the number of control nodes set in the previous Eon Mode example.

```
=> SELECT subcluster_name,node_name,is_primary,control_set_size FROM subclusters;
  subcluster_name   |      node_name       | is_primary | control_set_size
--------------------+----------------------+------------+------------------
 default_subcluster | v_verticadb_node0001 | t          |               -1
 default_subcluster | v_verticadb_node0002 | t          |               -1
 default_subcluster | v_verticadb_node0003 | t          |               -1
 analytics          | v_verticadb_node0004 | f          |                4
 analytics          | v_verticadb_node0005 | f          |                4
 analytics          | v_verticadb_node0006 | f          |                4
 analytics          | v_verticadb_node0007 | f          |                4
 analytics          | v_verticadb_node0008 | f          |                4
 analytics          | v_verticadb_node0009 | f          |                4
 analytics          | v_verticadb_node0010 | f          |                4
 analytics          | v_verticadb_node0011 | f          |                4
(11 rows)



=> SELECT SET_CONTROL_SET_SIZE('analytics',-1);
 SET_CONTROL_SET_SIZE
```

```
---------------------
 Control size set
(1 row)

=> SELECT REALIGN_CONTROL_NODES('analytics');
                    REALIGN_CONTROL_NODES
-------------------------------------------------------------------------------------------
 The new control node assignments can be viewed in vs_nodes. Call reload_spread(true).
 If the subcluster is critical, restart the database. Otherwise, restart the subcluster

(1 row)

=> SELECT RELOAD_SPREAD(true);
 RELOAD_SPREAD
---------------
 Reloaded
(1 row)

-- After restarting the analytics subcluster...

=> SELECT * FROM V_CATALOG.LARGE_CLUSTER_CONFIGURATION_STATUS;
     node_name       |   spread_host_name   |  control_node_name
---------------------+----------------------+----------------------
 v_verticadb_node0001 | v_verticadb_node0001 | v_verticadb_node0001
 v_verticadb_node0002 | v_verticadb_node0002 | v_verticadb_node0002
 v_verticadb_node0003 | v_verticadb_node0003 | v_verticadb_node0003
 v_verticadb_node0004 | v_verticadb_node0004 | v_verticadb_node0004
 v_verticadb_node0005 | v_verticadb_node0005 | v_verticadb_node0005
 v_verticadb_node0006 | v_verticadb_node0006 | v_verticadb_node0006
 v_verticadb_node0007 | v_verticadb_node0007 | v_verticadb_node0007
 v_verticadb_node0008 | v_verticadb_node0008 | v_verticadb_node0008
 v_verticadb_node0009 | v_verticadb_node0009 | v_verticadb_node0009
 v_verticadb_node0010 | v_verticadb_node0010 | v_verticadb_node0010
 v_verticadb_node0011 | v_verticadb_node0011 | v_verticadb_node0011
(11 rows)

=> SELECT subcluster_name,node_name,is_primary,control_set_size FROM subclusters;
  subcluster_name   |      node_name       | is_primary | control_set_size
--------------------+----------------------+------------+------------------
 default_subcluster | v_verticadb_node0001 | t          |               -1
 default_subcluster | v_verticadb_node0002 | t          |               -1
 default_subcluster | v_verticadb_node0003 | t          |               -1
 analytics          | v_verticadb_node0004 | f          |               -1
 analytics          | v_verticadb_node0005 | f          |               -1
 analytics          | v_verticadb_node0006 | f          |               -1
 analytics          | v_verticadb_node0007 | f          |               -1
 analytics          | v_verticadb_node0008 | f          |               -1
 analytics          | v_verticadb_node0009 | f          |               -1
 analytics          | v_verticadb_node0010 | f          |               -1
 analytics          | v_verticadb_node0011 | f          |               -1
(11 rows)
```

Monitoring large clusters

Monitor large cluster traits by querying the following system tables:

- V_CATALOG.LARGE_CLUSTER_CONFIGURATION_STATUS —Shows the current spread hosts and the control designations in the catalog so you can see if they match.
- V_MONITOR.CRITICAL_HOSTS —Lists the hosts whose failure would cause the database to become unsafe and force a shutdown.

> **Tip**

> The CRITICAL_HOSTS view is especially useful for large cluster arrangements. For non-large clusters, query the CRITICAL_NODES table.

- In an Eon Mode database, the CONTROL_SET_SIZE column of the V_CATALOG.SUBCLUSTERS system table shows the number of control nodes set for each subcluster.

You might also want to query the following system tables:

- V_CATALOG.FAULT_GROUPS —Shows fault groups and their hierarchy in the cluster.
- V_CATALOG.CLUSTER_LAYOUT —Shows the relative position of the actual arrangement of the nodes participating in the database cluster and the fault groups that affect them.

Multiple databases on a cluster

Vertica allows you to manage your database workloads by running multiple databases on a single cluster. However, databases cannot share the same node while running.

Example

If you have an 8-node cluster, with database 1 running on nodes 1, 2, 3, 4 and database 2 running on nodes 5, 6, 7, 8, you cannot create a new database in this cluster because all nodes are occupied. But if you stop database 1, you can create a database 3 using nodes 1, 2, 3, 4. Or if you stop both databases 1 and 2, you can create a database 3 using nodes 3, 4, 5, 6. In this latter case, database 1 and database 2 cannot be restarted unless you stop database 3, as they occupy the same nodes.

Fault groups

> **Note**
> You cannot create fault groups for an Eon Mode database. Rather, Vertica automatically creates fault groups on a large cluster Eon database; these fault groups are configured around the control nodes and their dependents of each subcluster. These fault groups are managed internally by Vertica and are not accessible to users.

Fault groups let you configure an Enterprise Mode database for your physical cluster layout. Sharing your cluster topology lets you use terrace routing to reduce the buffer requirements of large queries. It also helps to minimize the risk of correlated failures inherent in your environment, usually caused by shared resources.

Vertica automatically creates fault groups around control nodes (servers that run spread) in large cluster arrangements, placing nodes that share a control node in the same fault group. Automatic and user-defined fault groups do not include ephemeral nodes because such nodes hold no data.

Consider defining your own fault groups specific to your cluster's physical layout if you want to:

- Use terrace routing to reduce the buffer requirements of large queries.
- Reduce the risk of correlated failures. For example, by defining your rack layout, Vertica can better tolerate a rack failure.
- Influence the placement of control nodes in the cluster.

Vertica supports complex, hierarchical fault groups of different shapes and sizes. The database platform provides a fault group script (DDL generator), SQL statements, system tables, and other monitoring tools.

See High availability with fault groups for an overview of fault groups with a cluster topology example.

In this section

- About the fault group script
- Creating a fault group input file
- Creating fault groups
- Monitoring fault groups
- Dropping fault groups

About the fault group script

To help you define fault groups on your cluster, Vertica provides a script named fault_group_ddl_generator.py in the /opt/vertica/scripts directory. This script generates the SQL statements you need to run to create fault groups.

The fault_group_ddl_generator.py script does not create fault groups for you, but you can copy the output to a file. Then, when you run the helper script, you can use \i or vsql –f commands to pass the cluster topology to Vertica.

The fault group script takes the following arguments:

- The database name
- The fault group input file

For example:

```
$ python /opt/vertica/scripts/fault_group_ddl_generator.py VMartdb fault_grp_input.out
```

See also

- Creating a fault group input file
- Creating fault groups
- Dropping fault groups
- Monitoring fault groups
- Fault groups

Creating a fault group input file

Use a text editor to create a fault group input file for the targeted cluster.

The following example shows how you can create a fault group input file for a cluster that has 8 racks with 8 nodes on each rack—for a total of 64 nodes in the cluster.

1. On the first line of the file, list the parent (top-level) fault groups, delimited by spaces.
   ```
   rack1 rack2 rack3 rack4 rack5 rack6 rack7 rack8
   ```
2. On the subsequent lines, list the parent fault group followed by an equals sign (=). After the equals sign, list the nodes or fault groups delimited by spaces.
   ```
   <parent> = <child_1> <child_2> <child_n...>
   ```

   Such as:
   ```
   rack1 = v_vmart_node0001 v_vmart_node0002 v_vmart_node0003 v_vmart_node0004
   rack2 = v_vmart_node0005 v_vmart_node0006 v_vmart_node0007 v_vmart_node0008
   rack3 = v_vmart_node0009 v_vmart_node0010 v_vmart_node0011 v_vmart_node0012
   rack4 = v_vmart_node0013 v_vmart_node0014 v_vmart_node0015 v_vmart_node0016
   rack5 = v_vmart_node0017 v_vmart_node0018 v_vmart_node0019 v_vmart_node0020
   rack6 = v_vmart_node0021 v_vmart_node0022 v_vmart_node0023 v_vmart_node0024
   rack7 = v_vmart_node0025 v_vmart_node0026 v_vmart_node0027 v_vmart_node0028
   rack8 = v_vmart_node0029 v_vmart_node0030 v_vmart_node0031 v_vmart_node0032
   ```

   After the first row of parent fault groups, the order in which you write the group descriptions does not matter. All fault groups that you define in this file must refer back to a parent fault group. You can indicate the parent group directly or by specifying the child of a fault group that is the child of a parent fault group.
   Such as:
   ```
   rack1 rack2 rack3 rack4 rack5 rack6 rack7 rack8
   rack1 = v_vmart_node0001 v_vmart_node0002 v_vmart_node0003 v_vmart_node0004
   rack2 = v_vmart_node0005 v_vmart_node0006 v_vmart_node0007 v_vmart_node0008
   rack3 = v_vmart_node0009 v_vmart_node0010 v_vmart_node0011 v_vmart_node0012
   rack4 = v_vmart_node0013 v_vmart_node0014 v_vmart_node0015 v_vmart_node0016
   rack5 = v_vmart_node0017 v_vmart_node0018 v_vmart_node0019 v_vmart_node0020
   rack6 = v_vmart_node0021 v_vmart_node0022 v_vmart_node0023 v_vmart_node0024
   rack7 = v_vmart_node0025 v_vmart_node0026 v_vmart_node0027 v_vmart_node0028
   rack8 = v_vmart_node0029 v_vmart_node0030 v_vmart_node0031 v_vmart_node0032
   ```

After you create your fault group input file, you are ready to run the fault_group_ddl_generator.py . This script generates the DDL statements you need to create fault groups in Vertica.

If your Vertica database is co-located on a Hadoop cluster, and that cluster uses more than one rack, you can use fault groups to improve performance. See Configuring rack locality .

See also

Creating fault groups

Creating fault groups

When you define fault groups, Vertica distributes data segments across the cluster. This allows the cluster to be aware of your cluster topology so it can tolerate correlated failures inherent in your environment, such as a rack failure. For an overview, see High Availability With Fault Groups .

Important

Defining fault groups requires careful and thorough network planning, and a solid understanding of your network topology.

Prerequisites

To define a fault group, you must have:

- Superuser privileges
- A fault group input file
- An existing database

Run the fault group script

1. As the database administrator, run the fault_group_ddl_generator.py script:

   ```
   python /opt/vertica/scripts/fault_group_ddl_generator.py databasename fault-group-inputfile > sql-filename
   ```

   For example, the following command writes the Python script output to the SQL file fault_group_ddl.sql .

   ```
   $ python /opt/vertica/scripts/fault_group_ddl_generator.py
     VMart fault_groups_VMart.out > fault_group_ddl.sql
   ```

   After the script returns, you can run the SQL file, instead of multiple DDL statements individually.

   > **Tip**
   > Consider saving the input file so you can modify fault groups later—for example, after expanding the cluster or changing the distribution of control nodes.

2. Using vsql , run the DDL statements in fault_group_ddl.sql or execute the commands in the file using vsql .

   ```
   => \i fault_group_ddl.sql
   ```

3. If large cluster is enabled, realign control nodes with REALIGN_CONTROL_NODES . Otherwise, skip this step.

   ```
   => SELECT REALIGN_CONTROL_NODES();
   ```

4. Save cluster changes to the Spread configuration file by calling RELOAD_SPREAD :

   ```
   => SELECT RELOAD_SPREAD(true);
   ```

5. Use Administration tools to restart the database.
6. Save changes to the cluster's data layout by calling REBALANCE_CLUSTER :

   ```
   => SELECT REBALANCE_CLUSTER();
   ```

See also

- Cluster Management Functions
- Terrace routing
- CREATE FAULT GROUP
- ALTER FAULT GROUP
- DROP FAULT GROUP
- ALTER DATABASE

Monitoring fault groups

You can monitor fault groups by querying Vertica system tables or by logging in to the Management Console (MC) interface.

Monitor fault groups using system tables

Use the following system tables to view information about fault groups and cluster vulnerabilities, such as the nodes the cluster cannot lose without the database going down:

- V_CATALOG.FAULT_GROUPS : View the hierarchy of all fault groups in the cluster.
- V_CATALOG.CLUSTER_LAYOUT : Observe the arrangement of the nodes participating in the data business and the fault groups that affect them. Ephemeral nodes do not appear in the cluster layout ring because they hold no data.

Monitoring fault groups using Management Console

An MC administrator can monitor and highlight fault groups of interest by following these steps:

1. Click the running database you want to monitor and click **Manage** in the task bar.
2. Open the **Fault Group View** menu, and select the fault groups you want to view.
3. (Optional) Hide nodes that are not in the selected fault group to focus on fault groups of interest.

Nodes assigned to a fault group each have a colored bubble attached to the upper-left corner of the node icon. Each fault group has a unique color.If the number of fault groups exceeds the number of colors available, MC recycles the colors used previously.

Because Vertica supports complex, hierarchical fault groups of different shapes and sizes, MC displays multiple fault group participation as a stack of different-colored bubbles. The higher bubbles represent a lower-tiered fault group, which means that bubble is closer to the parent fault group, not the child or grandchild fault group.

For more information about fault group hierarchy, see High Availability With Fault Groups .

Dropping fault groups

When you remove a fault group from the cluster, be aware that the drop operation removes the specified fault group and its child fault groups. Vertica places all nodes under the parent of the dropped fault group. To see the current fault group hierarchy in the cluster, query system table FAULT_GROUPS .

Drop a fault group

Use the DROP FAULT GROUP statement to remove a fault group from the cluster. The following example shows how you can drops the group2 fault group:

```
=> DROP FAULT GROUP group2;
DROP FAULT GROUP
```

Drop all fault groups

Use the ALTER DATABASE statement to drop all fault groups, along with any child fault groups, from the specified database cluster.

The following command drops all fault groups from the current database.

```
=> ALTER DATABASE DEFAULT DROP ALL FAULT GROUP;
ALTER DATABASE
```

Add nodes back to a fault group

To add a node back to a fault group, you must manually reassign it to a new or existing fault group. To do so, use the CREATE FAULT GROUP and ALTER FAULT GROUP..ADD NODE statements.

See also

- DROP FAULT GROUP
- CREATE FAULT GROUP
- ALTER FAULT GROUP..ADD NODE
- Creating fault groups
- About the fault group script
- Creating a fault group input file

Terrace routing

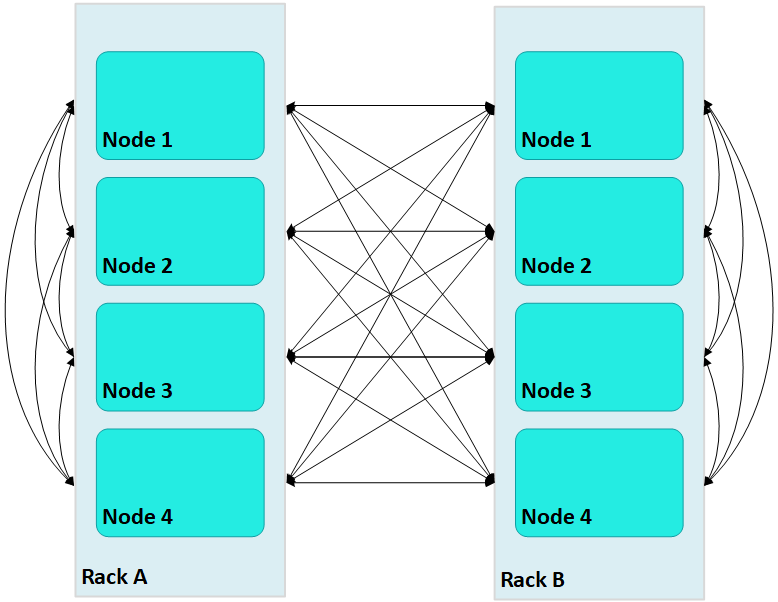
Important

Before you apply terrace routing to your database, be sure you are familiar with large cluster and fault groups .

Terrace routing can significantly reduce message buffering on a large cluster database. The following sections describe how Vertica implements terrace routing on Enterprise Mode and Eon Mode databases.

Terrace routing on Enterprise Mode

Terrace routing on an Enterprise Mode database is implemented through fault groups that define a rack-based topology. In a large cluster with terrace routing disabled, nodes in a Vertica cluster form a fully connected network, where each non-dependent (control) node sends messages across the database cluster through connections with all other non-dependent nodes, both within and outside its own rack/fault group:

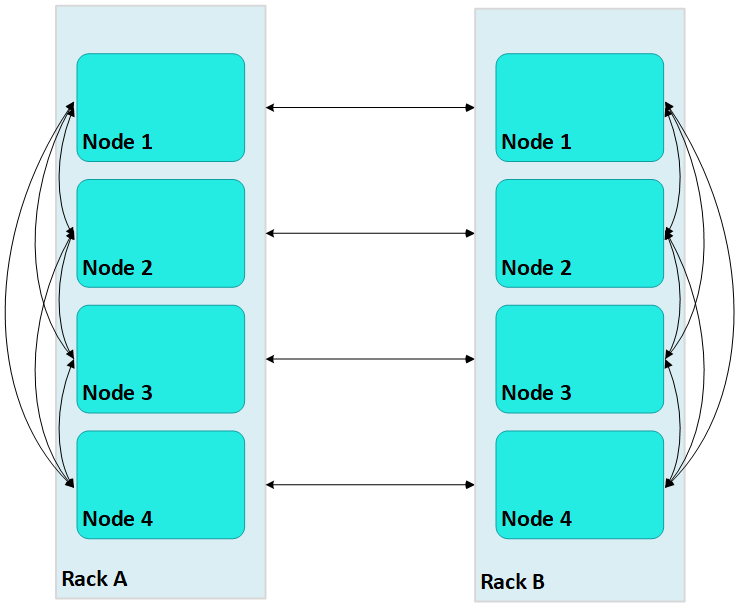


In this case, large Vertica clusters can require many connections on each node, where each connection incurs its own network buffering requirements. The total number of buffers required for each node is calculated as follows:

```
(numRacks * numRackNodes) - 1
```

In a two-rack cluster with 4 nodes per rack as shown above, this resolves to 7 buffers for each node.

With terrace routing enabled, you can considerably reduce large cluster network buffering. Each *n* th node in a rack/fault group is paired with the corresponding *n* th node of all other fault groups. For example, with terrace routing enabled, messaging in the same two-rack cluster is now implemented as follows:



Thus, a message that originates from node 2 on rack A (A2) is sent to all other nodes on rack A; each rack A node then conveys the message to its corresponding node on rack B—A1 to B1, A2 to B2, and so on.

With terrace routing enabled, each node of a given rack avoids the overhead of maintaining message buffers to all other nodes. Instead, each node is only responsible for maintaining connections to:

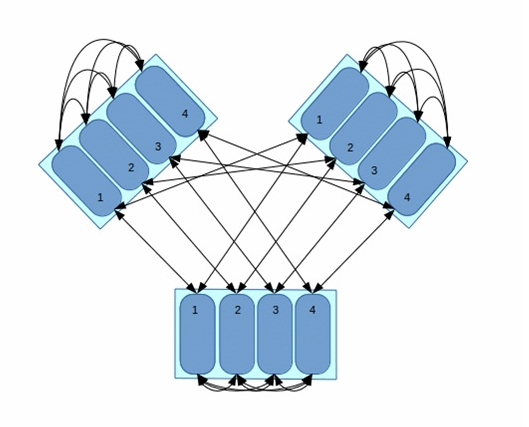
- All other nodes of the same rack (*numRackNodes* - 1)
- One node on each of the other racks (*numRacks* - 1)

Thus, the total number of message buffers required for each node is calculated as follows:

```
(numRackNodes-1) + (numRacks-1)
```

In a two-rack cluster with 4 nodes as shown earlier, this resolves to 4 buffers for each node.

Terrace routing trades time (intra-rack hops) for space (network message buffers). As a cluster expands with additional racks and nodes, the argument favoring this trade off becomes increasingly persuasive:



In this three-rack cluster with 4 nodes per rack, without terrace routing the number of buffers required by each node would be 11. With terrace routing, the number of buffers per node is 5. As a cluster expands with the addition of racks and nodes per rack, the disparity between buffer requirements widens. For example, given a six-rack cluster with 16 nodes per rack, without terrace routing the number of buffers required per node is 95; with terrace routing, 20.

Enabling terrace routing

Terrace routing depends on fault group definitions that describe a cluster network topology organized around racks and their member nodes. As noted earlier, when terrace routing is enabled, Vertica first distributes data within the rack/fault group; it then uses *n* th node-to- *n* th node mappings to forward this data to all other racks in the database cluster.

You enable (or disable) terrace routing for any Enterprise Mode large cluster that implements rack-based fault groups through configuration parameter TerraceRoutingFactor . To enable terrace routing, set this parameter as follows:

$$\text{TerraceRoutingFactor} < \frac{(numRackNodes-1) + (numRacks-1)}{(numRacks * numRackNodes) - 1}$$

where:

- *numRackNodes* : Number of nodes in a rack
- *numRacks* : Number of racks in the cluster

For example:

#Racks	Nodes/rack	#Connections		Terrace routing enabled if TerraceRoutingFactor less than:
		Without terrace routing	With terrace routing	
2	16	31	16	1.94
4	16	63	18	3.5
6	16	95	20	4.75
8	16	127	22	5.77

By default, TerraceRoutingFactor is set to 2, which generally ensures that terrace routing is enabled for any Enterprise Mode large cluster that implements rack-based fault groups. Vertica recommends enabling terrace routing for any cluster that contains 64 or more nodes, or if queries often require excessive buffer space.

To disable terrace routing, set TerraceRoutingFactor to a large integer such as 1000:

```
=> ALTER DATABASE DEFAULT SET TerraceRoutingFactor = 1000;
```

Terrace routing on Eon Mode

As in Enterprise Mode mode, terrace routing is enabled by default on an Eon Mode database, and is implemented through fault groups. However, you do not create fault groups for an Eon Mode database. Rather, Vertica automatically creates fault groups on a large cluster database; these fault groups are configured around the control nodes and their dependents of each subcluster. These fault groups are managed internally by Vertica and are not

accessible to users.

Elastic cluster

> **Note**
> Elastic Cluster is an Enterprise Mode-only feature. For scaling your database under Eon Mode, see Scaling your Eon Mode database .

You can scale your cluster up or down to meet the needs of your database. The most common case is to add nodes to your database cluster to accommodate more data and provide better query performance. However, you can scale down your cluster if you find that it is over-provisioned, or if you need to divert hardware for other uses.

You scale your cluster by adding or removing nodes. Nodes can be added or removed without shutting down or restarting the database. After adding a node or before removing a node, Vertica begins a rebalancing process that moves data around the cluster to populate the new nodes or move data off nodes about to be removed from the database. During this process, nodes can exchange data that are not being added or removed to maintain robust intelligent K-safety . If Vertica determines that the data cannot be rebalanced in a single iteration due to lack of disk space, then the rebalance operation spans multiple iterations.

To help make data rebalancing due to cluster scaling more efficient, Vertica locally segments data storage on each node so it can be easily moved to other nodes in the cluster. When a new node is added to the cluster, existing nodes in the cluster give up some of their data segments to populate the new node. They also exchange segments to minimize the number of nodes that any one node depends upon. This strategy minimizes the number of nodes that might become critical when a node fails . When a node is removed from the cluster, its storage containers are moved to other nodes in the cluster (which also relocates data segments to minimize how many nodes might become critical when a node fails). This method of breaking data into portable segments is referred to as elastic cluster, as it facilitates enlarging or shrinking the cluster.

The alternative to elastic cluster is re-segmenting all projection data and redistributing it evenly among all database nodes any time a node is added or removed. This method requires more processing and more disk space, as it requires all data in all projections to be dumped and reloaded.

Elastic cluster scaling factor

In a new installation, each node has a *scaling factor* that specifies the number of local segments (see Scaling factor). Rebalance efficiently redistributes data by relocating local segments provided that, after nodes are added or removed, there are sufficient local segments in the cluster to redistribute the data evenly (determined by MAXIMUM_SKEW_PERCENT). For example, if the scaling factor = 8, and there are initially 5 nodes, then there are a total of 40 local segments cluster-wide.

If you add two additional nodes (seven nodes) Vertica relocates five local segments on two nodes, and six such segments on five nodes, resulting in roughly a 16.7 percent skew. Rebalance relocates local segments only if the resulting skew is less than the allowed threshold, as determined by MAXIMUM_SKEW_PERCENT. Otherwise, segmentation space (and hence data, if uniformly distributed over this space) is evenly distributed among the seven nodes, and new local segment boundaries are drawn for each node, such that each node again has eight local segments.

> **Note**
> By default, the scaling factor only has an effect while Vertica rebalances the database. While rebalancing, each node breaks the projection segments it contains into storage containers, which it then moves to other nodes if necessary. After rebalancing, the data is recombined into ROS containers. It is possible to have Vertica always group data into storage containers. See Local data segmentation for more information.

Enabling elastic cluster

You enable elastic cluster with ENABLE_ELASTIC_CLUSTER . Query the ELASTIC_CLUSTER system table to verify that elastic cluster is enabled:

```
=> SELECT is_enabled FROM ELASTIC_CLUSTER;
 is_enabled
------------
 t
(1 row)
```

In this section

- Scaling factor
- Viewing scaling factor settings
- Setting the scaling factor
- Local data segmentation
- Elastic cluster best practices

Scaling factor

To avoid an increased number of ROS containers, do not enable local segmentation and do not change the scaling factor.

Viewing scaling factor settings

To view the scaling factor, query the ELASTIC_CLUSTER table:

```
=> SELECT scaling_factor FROM ELASTIC_CLUSTER;
scaling_factor
---------------
             4
(1 row)

=> SELECT SET_SCALING_FACTOR(6);
 SET_SCALING_FACTOR
--------------------
 SET
(1 row)

=> SELECT scaling_factor FROM ELASTIC_CLUSTER;
 scaling_factor
---------------
             6
(1 row)
```

Setting the scaling factor

The scaling factor determines the number of storage containers that Vertica uses to store each projection across the database during rebalancing when local segmentation is enabled. When setting the scaling factor, follow these guidelines:

- The number of storage containers should be greater than or equal to the number of partitions multiplied by the number of local segments: *num-storage-containers* >= (*num-partitions* * *num-local-segments*)
- Set the scaling factor high enough so rebalance can transfer local segments to satisfy the skew threshold, but small enough so the number of storage containers does not result in too many ROS containers, and cause ROS pushback . The maximum number of ROS containers (by default 1024) is set by configuration parameter ContainersPerProjectionLimit .

Use the SET_SCALING_FACTOR function to change your database's scaling factor. The scaling factor can be an integer between 1 and 32.

```
=> SELECT SET_SCALING_FACTOR(12);
SET_SCALING_FACTOR
--------------------
 SET
(1 row)
```

Local data segmentation

By default, the scaling factor only has an effect when Vertica rebalances the database. During rebalancing, nodes break the projection segments they contain into storage containers which they can quickly move to other nodes.

This process is more efficient than re-segmenting the entire projection (in particular, less free disk space is required), but it still has significant overhead, since storage containers have to be separated into local segments, some of which are then transferred to other nodes. This overhead is not a problem if you rarely add or remove nodes from your database.

However, if your database is growing rapidly and is constantly busy, you may find the process of adding nodes becomes disruptive. In this case, you can enable local segmentation, which tells Vertica to always segment its data based on the scaling factor, so the data is always broken into containers that are easily moved. Having the data segmented in this way dramatically speeds up the process of adding or removing nodes, since the data is always in a state that can be quickly relocated to another node. The rebalancing process that Vertica performs after adding or removing a node just has to decide which storage containers to relocate, instead of first having to first break the data into storage containers.

Local data segmentation increases the number of storage containers stored on each node. This is not an issue unless a table contains many partitions. For example, if the table is partitioned by day and contains one or more years. If local data segmentation is enabled, then each of these table partitions is broken into multiple local storage segments, which potentially results in a huge number of files which can lead to ROS "pushback."

Consider your table partitions and the effect enabling local data segmentation may have before enabling the feature.

In this section

- Enabling and disabling local segmentation

Enabling and disabling local segmentation

To enable local segmentation, use the ENABLE_LOCAL_SEGMENTS function. To disable local segmentation, use the DISABLE_LOCAL_SEGMENTATION function:

```
=> SELECT ENABLE_LOCAL_SEGMENTS();
ENABLE_LOCAL_SEGMENTS
----------------------
 ENABLED
(1 row)


=> SELECT is_local_segment_enabled FROM elastic_cluster;
 is_enabled
------------
 t
(1 row)

=> SELECT DISABLE_LOCAL_SEGMENTS();
 DISABLE_LOCAL_SEGMENTS
------------------------
 DISABLED
(1 row)

=> SELECT is_local_segment_enabled FROM ELASTIC_CLUSTER;
 is_enabled
------------
 f
(1 row)
```

Elastic cluster best practices

The following are some best practices with regard to local segmentation.

> **Note**
> You should always perform a database backup before and after performing any of the operations discussed in this topic. You need to back up before changing any elastic cluster or local segmentation settings to guard against a hardware failure causing the rebalance process to leave the database in an unusable state. You should perform a full backup of the database after the rebalance procedure to avoid having to rebalance the database again if you need to restore from a backup.

When to enable local data segmentation

Local data segmentation can significantly speed up the process of resizing your cluster. You should enable local data segmentation if:

- your database does not contain tables with hundreds of partitions.
- the number of nodes in the database cluster is a power of two.
- you plan to expand or contract the size of your cluster.

Local segmentation can result in an excessive number of storage containers with tables that have hundreds of partitions, or in clusters with a non-power-of-two number of nodes. If your database has these two features, take care when enabling local segmentation.

Adding nodes

There are many reasons for adding one or more nodes to an installation of Vertica:

- **Increase system performance.** Add additional nodes due to a high query load or load latency or increase disk space without adding storage locations to existing nodes.

> **Note**
> The database response time depends on factors such as type and size of the application query, database design, data size and data types stored, available computational power, and network bandwidth. Adding nodes to a database cluster does not necessarily improve the system response time for every query, especially if the response time is already short, e.g., less then 10 seconds, or the response time is not hardware bound.

- **Make the database K-safe** (K-safety =1) or increase K-safety to 2. See Failure recovery for details.
- **Swap a node for maintenance.** Use a spare machine to temporarily take over the activities of an existing node that needs maintenance. The node that requires maintenance is known ahead of time so that when it is temporarily removed from service, the cluster is not vulnerable to additional node failures.
- **Replace a node.** Permanently add a node to remove obsolete or malfunctioning hardware.

Important
If you install Vertica on a single node without specifying the IP address or host name (or you used localhost), you cannot expand the cluster. You must reinstall Vertica and specify an IP address or host name that is not localhost/127.0.0.1 .

Adding nodes consists of the following general tasks:

1. Back up the database .
   Vertica strongly recommends that you back up the database before you perform this significant operation because it entails creating new projections, refreshing them, and then deleting the old projections. See Backing up and restoring the database for more information.
   The process of migrating the projection design to include the additional nodes could take a while; however during this time, all user activity on the database can proceed normally, using the old projections.
2. Configure the hosts you want to add to the cluster.
   See Before you Install Vertica . You will also need to edit the hosts configuration file on all of the existing nodes in the cluster to ensure they can resolve the new host.
3. Add one or more hosts to the cluster .
4. Add the hosts you added to the cluster (in step 3) to the database.

> **Note**
> When you add a "host" to the database, it becomes a "node." You can add nodes to your database using either the Administration tools or the Management Console (See Monitoring with MC).
>
> You can also add nodes using the admintools command line, which allows you to preserve the specific order of the nodes you add.

After you add nodes to the database, Vertica automatically distributes updated configuration files to the rest of the nodes in the cluster and starts the process of rebalancing data in the cluster. See Rebalancing data across nodes for details.

In this section

- Adding hosts to a cluster
- Adding nodes to a database
- Add nodes to a cluster in AWS

Adding hosts to a cluster

After you have backed up the database and configured the hosts you want to add to the cluster, you can now add hosts to the cluster using the update_vertica script.

You cannot use the MC to add hosts to a cluster in an on-premises environment. However, after the hosts are added to the cluster, the MC does allow you to add the hosts to a database as nodes.

Prerequisites and restrictions

If you installed Vertica on a single node without specifying the IP address or hostname (you used localhost), it is not possible to expand the cluster. You must reinstall Vertica and specify an IP address or hostname.

Procedure to add hosts

From one of the existing cluster hosts, run the update_vertica script with a minimum of the --add-hosts *host(s)* parameter (where *host(s)* is the hostname or IP address of the system(s) that you are adding to the cluster) and the --rpm or --deb parameter:

```
# /opt/vertica/sbin/update_vertica --add-hosts host(s) --rpm package
```

> **Note**
> See Install Vertica with the installation script for the full list of parameters. You must also provide the same options you used when originally installing the cluster.

The update_vertica ** script uses all the same options as install_vertica and:

- Installs the Vertica RPM on the new host.
- Performs post-installation checks, including RPM version and N-way network connectivity checks.
- Modifies spread to encompass the larger cluster.
- Configures the Administration Tools to work with the larger cluster.

Important Tips:

- Consider using --large-cluster with more than 50 nodes.
- A host can be specified by the hostname or IP address of the system you are adding to the cluster. However, internally Vertica stores all host addresses as IP addresses.
- Do not use include spaces in the hostname/IP address list provided with --add-hosts if you specified more than one host.
- If a package is specified with --rpm/--deb , and that package is newer than the one currently installed on the existing cluster, then, Vertica first installs the new package on the existing cluster hosts before the newly-added hosts.
- Use the same command line parameters for the database administrator username, password, and directory path you used when you installed the cluster originally. Alternatively, you can create a properties file to save the parameters during install and then re-using it on subsequent install and update operations. See Installing Vertica Silently .
- If you are installing using sudo, the database administrator user (dbadmin) must already exist on the hosts you are adding and must be configured with passwords and home directory paths identical to the existing hosts. Vertica sets up passwordless ssh from existing hosts to the new hosts, if needed.
- If you initially used the --point-to-point option to configure spread to use direct, point-to-point communication between nodes on the subnet, then use the --point-to-point option whenever you run install_vertica or update_vertica . Otherwise, your cluster's configuration is reverted to the default (*broadcast*), which may impact future databases.
- The maximum number of spread daemons supported in point-to-point communication and broadcast traffic is 80. It is possible to have more than 80 nodes by using large cluster mode, which does not install a spread daemon on each node.

Examples

```
--add-hosts host01 --rpm
--add-hosts 192.168.233.101
--add-hosts host02,host03
```

Adding nodes to a database

After you add one or more hosts to the cluster, you can add them as nodes to the database with one of the following:

- admintools command line, to ensure nodes are added in a specific order
- Administration Tools
- Management Console

Command line

With the admintools db_add_node tool, you can control the order in which nodes are added to the database cluster. It specifies the hosts of new nodes with its -s or --hosts option, which takes a comma-delimited argument list. Vertica adds new nodes in the list-specified order. For example, the following command adds three nodes:

```
$ admintools -t db_add_node \
    -d VMart \
    -p 'password' \
    -s 192.0.2.1,192.0.2.2,192.0.2.3
```

> **Tip**
> When adding nodes to an Eon Mode database, you can also specify the subcluster that the new nodes should belong to. See Adding and removing nodes from subclusters for more information.

Administration tools

You add nodes to a database with the Administration Tools as follows:

1. Open the Administration Tools.
2. On the **Main Menu** , select **View Database Cluster State** to verify that the database is running. If it is not, start it.
3. From the **Main Menu** , select **Advanced Menu** and click **OK** .
4. In the **Advanced Menu** , select **Cluster Management** and click **OK** .
5. In the **Cluster Management** menu, select **Add Host(s)** and click **OK** .
6. Select the database to which you want to add one or more hosts, and then select **OK** .
   A list of unused hosts is displayed.
7. Select the hosts you want to add to the database and click **OK** .
8. When prompted, click **Yes** to confirm that you want to add the hosts.
9. When prompted, enter the password for the database, and then select **OK** .
10. When prompted that the hosts were successfully added, select **OK** .
11. Vertica now automatically starts the rebalancing process to populate the new node with data. When prompted, enter the path to a temporary directory that the Database Designer can use to rebalance the data in the database and select **OK** .
12. Either press Enter to accept the default K-safety value, or enter a new higher value for the database and select **OK** .
13. Select whether to rebalance the database immediately, or later. In both cases, Vertica creates a script, which you can use to rebalance at any time. Review the summary of the rebalancing process and select **Proceed** .
    If you choose to automatically rebalance, the rebalance process runs. If you chose to create a script, the script is generated and saved. In either case, you are shown a success screen.
14. Select **OK** to complete the Add Node process.

Management Console

To add nodes to an Eon Mode database using MC, see Add nodes to a cluster in AWS using Management Console .

To add hosts to an Enterprise Mode database using MC, see Adding hosts to a cluster

Add nodes to a cluster in AWS

This section gives an overview on how to add nodes if you are managing your cluster using admintools. Each main step points to another topic with the complete instructions.

Step 1: before you start

Before you add nodes to a cluster, verify that you have an AWS cluster up and running and that you have:

- Created a database.
- Defined a database schema.
- Loaded data.
- Run the Database Designer.
- Connected to your database.

Step 2: launch new instances to add to an existing cluster

Perform the procedure in Configure and launch an instance to create new instances (hosts) that you then will add to your existing cluster. Be sure to choose the same details you chose when you created the original instances (VPC, placement group, subnet, and security group).

Step 3: include new instances as cluster nodes

You need the IP addresses when you run the install_vertica script to include new instances as cluster nodes.

If you are configuring Amazon Elastic Block Store (EBS) volumes, be sure to configure the volumes on the node before you add the node to your cluster.

To add the new instances as nodes to your existing cluster:

1. Configure and launch your new instances .
2. Connect to the instance that is assigned to the Elastic IP. See Connect to an instance if you need more information.
3. Run the Vertica installation script to add the new instances as nodes to your cluster. Specify the internal IP addresses for your instances and your *.pem file name.

```
$ sudo /opt/vertica/sbin/install_vertica --add-hosts instance-ip --dba-user-password-disabled \
  --point-to-point --data-dir /vertica/data --ssh-identity ~/name-of-pem.pem
```

Step 4: add the nodes

After you have added the new instances to your existing cluster, add them as nodes to your cluster, as described in Adding nodes to a database .

Step 5: rebalance the database

After you add nodes to a database, always rebalance the database.

Removing nodes

Although less common than adding a node, permanently removing a node is useful if the host system is obsolete or over-provisioned.

Important
Before removing a node from a cluster, check that the cluster has the minimum number of nodes required to comply with K-safety. If necessary, temporarily lower the database K-safety level .

In this section

- Automatic eviction of unhealthy nodes
- Lowering K-Safety to enable node removal
- Removing nodes from a database
- Removing hosts from a cluster
- Remove nodes from an AWS cluster

Automatic eviction of unhealthy nodes

To manage the health of the nodes in your cluster, Vertica performs regular health checks by sending and receiving "heartbeats." During a health check, each node in the cluster verifies read-write access to its catalog, catalog disk, and local storage locations ('TEMP, DATA', TEMP, DATA, and DEPOT). Upon verification, the node sends a heartbeat. If a node fails to send a heartbeat after five intervals (fails five health checks), then the node is evicted from the cluster.

You can control the time between each health check with the DatabaseHeartBeatInterval parameter. By default, DatabaseHeartBeatInterval is set to 120, which allows five 120-second intervals to pass without a heartbeat.

The amount of time allowed before an eviction is:

*TOT = DHBI * 5*

where *TOT* is the total time (in seconds) allowed without a heartbeat before eviction, and *DHBI* is equal to the value of DatabaseHeartBeatInterval.

If you set the DatabaseHeartBeatInterval too low, it can cause evictions in cases of brief node health issues. Sometimes, such premature evictions result in lower availability and performance of the Vertica database.

See also

DatabaseHeartbeatInterval in General parameters

Lowering K-Safety to enable node removal

A database with a K-safety level of 1 requires at least three nodes to operate, and a database with a K-safety level 2 requires at least 5 nodes to operate. You can check the cluster's current K-safety level as follows:

```
=> SELECT current_fault_tolerance FROM system;
 current_fault_tolerance
-------------------------
                       1
(1 row)
```

To remove a node from a cluster with the minimum number of nodes that it requires for K-safety, first lower the K-safety level with MARK_DESIGN_KSAFE .

Caution
Lowering the K-safety level of a database to 0 eliminates Vertica's fault tolerance features. If you must reduce K-safety to 0, first back up the database .

1. Connect to the database with Administration Tools or vsql .
2. Call the function MARK_DESIGN_KSAFE :

```
SELECT MARK_DESIGN_KSAFE(n);
```

where *n* is the new K-safety level for the database.

Removing nodes from a database

> **Note**
> In an Eon Mode database, you remove nodes from the subcluster that contains them, rather than from the database. See Removing Nodes for more information.

As long as there are enough nodes remaining to satisfy the K-Safety requirements, you can remove the node from a database. You cannot drop nodes that are critical for K-safety . See Lowering K-Safety to enable node removal .

You can remove nodes from a database using one of the following:

- Management Console interface
- Administration Tools

Prerequisites
Before removing a node from the database, verify that the database complies with the following requirements:

- It is running.
- It has been backed up .
- The database has the minimum number of nodes required to comply with K-safety. If necessary, temporarily lower the database K-safety level .
- All of the nodes in your database must be either up or in active standby. Vertica reports the error "All nodes must be UP or STANDBY before dropping a node" if you attempt to remove a node while a database node is down. You will get this error, even if you are trying to remove the node that is down.

Management Console

Remove nodes with Management Console from its Manage page:

Remove database nodes as follows:

1. Choose the node to remove.
2. Click **Remove node** in the Node List.

The following restrictions apply:

- You can only remove nodes that belong to the database cluster.
- You cannot remove DOWN nodes.

When you remove a node, its state changes to STANDBY. You can later add STANDBY nodes back to the database.

Administration tools

To remove unused hosts from the database using Administration Tools:

1. Open the Administration Tools. See Using the administration tools for information about accessing the Administration Tools.
2. On the Main Menu, select View Database Cluster State to verify that the database is running. If the database is not running, start it.
3. From the Main Menu, choose Advanced Menu and choose OK.
4. In the Advanced menu, choose Cluster Management and choose OK.
5. In the Cluster Management menu, choose Remove Host(s) from Database and choose OK.
6. When warned that you must redesign your database and create projections that exclude the hosts you are going to drop, choose Yes.
7. Select the database from which you want to remove the hosts and choose OK.
   A list of currently active hosts appears.
8. Select the hosts you want to remove from the database and choose OK.
9. When prompted, choose OK to confirm that you want to remove the hosts.
10. When informed that the hosts were successfully removed, choose OK.
11. If you removed a host from a Large Cluster configuration, open a vsql session and run realign_control_nodes:
    ```
    SELECT realign_control_nodes();
    ```
    For more details, see REALIGN_CONTROL_NODES .
12. If this host is not used by any other database in the cluster, you can remove the host from the cluster. See Removing hosts from a cluster .

Removing hosts from a cluster

If a host that you removed from the database is not used by any other database, you can remove it from the cluster with update_vertica . You can leave the database running during this operation.

When you use update_vertica to reduce the size of the cluster, it also performs these tasks:

- Modifies the spread to match the smaller cluster.
- Configures Administration tools to work with the smaller cluster.

> **Note**
> You can use Management Console to remove hosts from a database , but you cannot remove those hosts from a cluster.

From one of the Vertica cluster hosts, run update_vertica with the —remove-hosts switch. This switch takes an list of comma-separated hosts to remove from the cluster. You can reference hosts by their names or IP addresses. For example, you can remove hosts host01 , host02 , and host03 as follows:

```
# /opt/vertica/sbin/update_vertica --remove-hosts host01,host02,host03 \
        --rpm /tmp/vertica-10.1.1-0.x86_64.RHEL6.rpm \
        --dba-user mydba
```

If --rpm specifies a new RPM, then Vertica installs it on the existing cluster hosts before proceeding.

update_vertica uses the same options as install_vertica .For all options, see Install Vertica with the installation script .

Requirements

- If -remove-host s specifies a list of multiple hosts, the list must not embed any spaces between hosts.
- Use the same command line options as in the original installation. If you used non-default values for the database administrator username, password, or directory path, provide the same settings when you remove hosts; otherwise; the procedure fails. Consider saving the original installation options in a properties file that you can reuse on subsequent installation and update operations. See Install Vertica silently .

Remove nodes from an AWS cluster

Use the following procedures to remove instances/nodes from an AWS cluster.

To avoid data loss, Vertica strongly recommends that you back up your database before removing a node. For details, see Backing up and restoring the database .

Remove hosts from the database

Before you remove hosts from the database, verify that you have:

- Backed up the database.
- Lowered the K-safety of the database.

> **Note**
> Do not stop the database.

To remove a host from the database:

1. While logged on as dbadmin, launch Administration Tools.
   $ /opt/vertica/bin/admintools
2. From the **Main Menu** , select **Advanced Menu** .
3. From **Advanced Menu** , select **Cluster Management** . Click **OK** .
4. From **Cluster Management** , select **Remove Host(s)** . Click **OK** .
5. From **Select Database** , choose the database from which you plan to remove hosts. Click **OK** .
6. Select the host(s) to remove. Click **OK** .
7. Click **Yes** to confirm removal of the hosts.

> **Note**
> Enter a password if necessary. Leave blank if there is no password.

8. Click **OK** . The system displays a message telling you that the hosts have been removed. Automatic rebalancing also occurs.
9. Click **OK** to confirm. Administration Tools brings you back to the **Cluster Management** menu.

Remove nodes from the cluster

To remove nodes from a cluster, run the update_vertica script and specify:

- The option --remove-hosts , followed by the IP addresses of the nodes you are removing.
- The option --ssh-identity , followed by the location and name of your *pem file.
- The option --dba-user-password-disabled .

The following example removes one node from the cluster:

```
$ sudo /opt/vertica/sbin/update_vertica  --remove-hosts 10.0.11.165  --point-to-point  \
  --ssh-identity ~/name-of-pem.pem --dba-user-password-disabled
```

Stop the AWS instances

After you have removed one or more nodes from your cluster, to save costs associated with running instances, you can choose to stop the AWS instances that were previously part of your cluster.

To stop an instance in AWS:

1. On AWS, navigate to your **Instances** page.
2. Right-click the instance, and choose **Stop** .

This step is optional because, after you have removed the node from your Vertica cluster, Vertica no longer sees the node as part of the cluster, even though it is still running within AWS.

Replacing nodes

If you have a K-Safe database, you can replace nodes, as necessary, without bringing the system down. For example, you might want to replace an existing node if you:

- Need to repair an existing host system that no longer functions and restore it to the cluster
- Want to exchange an existing host system for another more powerful system

> **Note**
> Vertica does not support replacing a node on a K-safe=0 database. Use the procedures to add and remove nodes instead.

The process you use to replace a node depends on whether you are replacing the node with:

- A host that uses the same name and IP address
- A host that uses a different name and IP address
- An active standby node

Prerequisites

- Configure the replacement hosts for Vertica. See Before you Install Vertica .
- Read the Important **Tips** sections under Adding hosts to a cluster and Removing hosts from a cluster .
- Ensure that the database administrator user exists on the new host and is configured identically to the existing hosts. Vertica will setup passwordless ssh as needed.
- Ensure that directories for Catalog Path, Data Path, and any storage locations are added to the database when you create it and/or are mounted correctly on the new host and have read and write access permissions for the database administrator user. Also ensure that there is sufficient disk space.
- Follow the best practice procedure below for introducing the failed hardware back into the cluster to avoid spurious full-node rebuilds.

Best practice for restoring failed hardware

Following this procedure will prevent Vertica from misdiagnosing missing disk or bad mounts as data corruptions, which would result in a time-consuming, full-node recovery.

If a server fails due to hardware issues, for example a bad disk or a failed controller, upon repairing the hardware:

1. Reboot the machine into runlevel 1, which is a root and console-only mode.
   Runlevel 1 prevents network connectivity and keeps Vertica from attempting to reconnect to the cluster.
2. In runlevel 1, validate that the hardware has been repaired, the controllers are online, and any RAID recover is able to proceed.

   > **Note**
   > You do not need to initialize RAID recover in runlevel 1; simply validate that it can recover.

3. Once the hardware is confirmed consistent, only then reboot to runlevel 3 or higher.

At this point, the network activates, and Vertica rejoins the cluster and automatically recovers any missing data. Note that, on a single-node database, if any files that were associated with a projection have been deleted or corrupted, Vertica will delete all files associated with that projection, which could result in data loss.

In this section

- Replacing a host using the same name and IP address
- Replacing a failed node using a node with a different IP address
- Replacing a functioning node using a different name and IP address
- Using the administration tools to replace nodes

Replacing a host using the same name and IP address

If a host of an existing Vertica database is removed you can replace it while the database is running.

> **Note**
> Remember a host in Vertica consists of the hardware and operating system on which Vertica software resides, as well as the same network configurations.

You can replace the host with a new host that has the following same characteristics as the old host:

- Name
- IP address
- Operating system
- The OS administrator user
- Directory location

Replacing the host while your database is running prevents system downtime. Before replacing a host, backup your database. See Backing up and restoring the database for more information.

Replace a host using the same characteristics as follows:

1. Run install_vertica from a functioning host using the --rpm or --deb parameter:
   ```
   $ /opt/vertica/sbin/install_vertica --rpm rpm_package
   ```
   For more information see Install Vertica using the command line .
2. Use Administration Tools from an existing node to restart the new host. See Restart Vertica on a node .

The node automatically joins the database and recovers its data by querying the other nodes in the database. It then transitions to an UP state.

Replacing a failed node using a node with a different IP address

Replacing a failed node with a host system that has a different IP address from the original consists of the following steps:

1. Back up the database .
   Vertica recommends that you back up the database before you perform this significant operation because it entails creating new projections, deleting old projections, and reloading data.
2. Add the new host to the cluster. See Adding hosts to a cluster .
3. If Vertica is still running in the node being replaced, then use the Administration Tools to Stop *Vertica on Host* on the host being replaced.
4. Use the Administration Tools to replace the original host with the new host. If you are using more than one database, replace the original host in all the databases in which it is used. See Replacing Hosts .
5. Use the procedure in Distributing Configuration Files to the New Host to transfer metadata to the new host.

6. Remove the host from the cluster .
7. Use the Administration Tools to restart Vertica on the host. On the **Main Menu** , select **Restart Vertica on Host** , and click **OK** . See Starting the database for more information.

Once you have completed this process, the replacement node automatically recovers the data that was stored in the original node by querying other nodes within the database.

Replacing a functioning node using a different name and IP address

Replacing a node with a host system that has a different IP address and host name from the original consists of the following general steps:

1. Back up the database .
   Vertica recommends that you back up the database before you perform this significant operation because it entails creating new projections, deleting old projections, and reloading data.
2. Add the replacement hosts to the cluster .
   At this point, both the original host that you want to remove and the new replacement host are members of the cluster.
3. Use the Administration Tools to Stop *Vertica on Host* on the host being replaced.
4. Use the Administration Tools to replace the original host with the new host. If you are using more than one database, replace the original host in all the databases in which it is used. See Replacing Hosts .
5. Remove the host from the cluster .
6. Restart Vertica on the host.

Once you have completed this process, the replacement node automatically recovers the data that was stored in the original node by querying the other nodes within the database. It then transitions to an UP state.

> **Note**
> If you do not remove the original host from the cluster and you attempt to restart the database, the host is not invited to join the database because its node address does not match the new address stored in the database catalog. Therefore, it remains in the INITIALIZING state.

Using the administration tools to replace nodes

If you are replacing a node with a host that uses a different name and IP address, use the Administration Tools to replace the original host with the new host. Alternatively, you can use the Management Console to replace a node .

Replace the original host with a new host using the administration tools

To replace the original host with a new host using the Administration Tools:

1. Back up the database. See Backing up and restoring the database .
2. From a node that is up, and is not going to be replaced, open the Administration tools .
3. On the **Main Menu** , select **View Database Cluster State** to verify that the database is running. If it's not running, use the Start Database command on the Main Menu to restart it.
4. On the **Main Menu** , select **Advanced Menu** .
5. In the **Advanced Menu** , select **Stop Vertica on Host** .
6. Select the host you want to replace, and then click **OK** to stop the node.
7. When prompted if you want to stop the host, select **Yes** .
8. In the **Advanced Menu** , select **Cluster Management** , and then click **OK** .
9. In the **Cluster Management** menu, select **Replace Host** , and then click **OK** .
10. Select the database that contains the host you want to replace, and then click **OK** .
    A list of all the hosts that are currently being used displays.
11. Select the host you want to replace, and then click **OK** .
12. Select the host you want to use as the replacement, and then click **OK** .
13. When prompted, enter the password for the database, and then click **OK** .
14. When prompted, click **Yes** to confirm that you want to replace the host.
15. When prompted that the host was successfully replaced, click **OK** .
16. In the **Main Menu** , select **View Database Cluster State** to verify that all the hosts are running. You might need to start Vertica on the host you just replaced. Use **Restart Vertica on Host** .
    The node enters a RECOVERING state.

> **Caution**
> If you are using a K-Safe database, keep in mind that the recovering node counts as one node down even though it might not yet contain a

complete copy of the data. This means that if you have a database in which K safety=1, the current fault tolerance for your database is at a critical level. If you lose one more node, the database shuts down. Be sure that you do not stop any other nodes.

Rebalancing data across nodes

Vertica can rebalance your database when you add or remove nodes. As a superuser, you can manually trigger a rebalance with Administration Tools , SQL functions , or the Management Console .

A rebalance operation can take some time, depending on the cluster size, and the number of projections and the amount of data they contain. You should allow the process to complete uninterrupted. If you must cancel the operation, call CANCEL_REBALANCE_CLUSTER .

Why rebalance?

Rebalancing is useful or even necessary after you perform one of the following operations:

- Change the size of the cluster by adding or removing nodes.
- Mark one or more nodes as ephemeral in preparation of removing them from the cluster.
- Change the scaling factor of an elastic cluster, which determines the number of storage containers used to store a projection across the database.
- Set the control node size or realign control nodes on a large cluster layout.
- Specify more than 120 nodes in your initial Vertica cluster configuration.
- Modify a fault group by adding or removing nodes.

General rebalancing tasks

When you rebalance a database cluster, Vertica performs the following tasks for all projections, segmented and unsegmented alike:

- Distributes data based on:
  - User-defined fault groups , if specified
  - Large cluster automatic fault groups
- Ignores node-specific distribution specifications in projection definitions. Node rebalancing always distributes data across all nodes.
- When rebalancing is complete, sets the Ancient History Mark the greatest allowable epoch (now).

Vertica rebalances segmented and unsegmented projections differently, as described below.

Rebalancing segmented projections

For each segmented projection, Vertica performs the following tasks:

1. Copies and renames projection buddies and distributes them evenly across all nodes. The renamed projections share the same base name.
2. Refreshes the new projections.
3. Drops the original projections.

Rebalancing unsegmented projections

For each unsegmented projection, Vertica performs the following tasks:

If adding nodes:

- Creates projection buddies on them.
- Maps the new projections to their shared name in the database catalog.

If dropping nodes : drops the projection buddies from them.

K-safety and rebalancing

Until rebalancing completes, Vertica operates with the existing K-safe value. After rebalancing completes, Vertica operates with the K-safe value specified during the rebalance operation. The new K-safe value must be equal to or higher than current K-safety. Vertica does not support downgrading K-safety and returns a warning if you try to reduce it from its current value. For more information, see Lowering K-Safety to enable node removal .

Rebalancing failure and projections

If a failure occurs while rebalancing the database, you can rebalance again. If the cause of the failure has been resolved, the rebalance operation continues from where it failed. However, a failed data rebalance can result in projections becoming out of date.

To locate out-of-date projections, query the system table PROJECTIONS as follows:

```
=> SELECT projection_name, anchor_table_name, is_up_to_date FROM projections
  WHERE is_up_to_date = false;
```

To remove out-of-date projections, use DROP PROJECTION .

Temporary tables

Node rebalancing has no effect on projections of temporary tables.

For Detailed Information About Rebalancing

See the Knowledge Base articles:

- What Happens During Rebalancing
- Optimizing for Rebalancing

In this section

- Rebalancing data using the administration tools UI
- Rebalancing data using SQL functions

Rebalancing data using the administration tools UI

To rebalance the data in your database:

1. Open the Administration Tools. (See Using the administration tools .)
2. On the **Main Menu** , select **View Database Cluster State** to verify that the database is running. If it is not, start it.
3. From the **Main Menu** , select **Advanced Menu** and click **OK** .
4. In the **Advanced Menu** , select **Cluster Management** and click **OK** .
5. In the **Cluster Management** menu, select **Re-balance Data** and click **OK** .
6. Select the database you want to rebalance, and then select **OK** .
7. Enter the directory for the Database Designer outputs (for example /tmp) and click **OK** .
8. Accept the proposed K-safety value or provide a new value. Valid values are 0 to 2.
9. Review the message and click **Proceed** to begin rebalancing data.
   The Database Designer modifies existing projections to rebalance data across all database nodes with the K-safety you provided. A script to rebalance data, which you can run manually at a later time, is also generated and resides in the path you specified; for example /tmp/extend_catalog_rebalance.sql .
   Important
   Rebalancing data can take some time, depending on the number of projections and the amount of data they contain. Vertica recommends that you allow the process to complete. If you must cancel the operation, use Ctrl+C.

   The terminal window notifies you when the rebalancing operation is complete.
10. Press **Enter** to return to the Administration Tools.

Rebalancing data using SQL functions

Vertica has three SQL functions for starting and stopping a cluster rebalance. You can call these functions from a script that runs during off-peak hours, rather than manually trigger a rebalance through Administration Tools.

- REBALANCE_CLUSTER rebalances the database cluster synchronously as a session foreground task.
- START_REBALANCE_CLUSTER asynchronously rebalances the database cluster as a background task.
- CANCEL_REBALANCE_CLUSTER stops any rebalance task that is currently in progress or is waiting to execute.

Redistributing configuration files to nodes

The add and remove node processes automatically redistribute the Vertica configuration files. You rarely need to redistribute the configuration files to help resolve configuration issues.

To distribute configuration files to a host:

1. Log on to a host that contains these files and start Administration Tools .
2. On the Administration Tools **Main Menu** , select **Configuration Menu** and click **OK** .
3. On the **Configuration Menu** , select **Distribute Config Files** and click **OK** .
4. Select **Database Configuration** .
5. Select the database where you want to distribute the files and click **OK** .
   Vertica configuration files are distributed to all other database hosts. If the files already existed on a host, they are overwritten.
6. On the **Configuration Menu** , select **Distribute Config Files** and click **OK** .
7. Select **SSL Keys** .

Certifications and keys are distributed to all other database hosts. If they already existed on a host, they are overwritten.

8. On the **Configuration Menu** , select **Distribute Config Files** and click **OK** .
   Select **AdminTools Meta-Data** .
   Administration Tools metadata is distributed to every host in the cluster.
9. Restart the database .

> **Note**
> To distribute the configuration file admintools.conf via the command line or scripts, use the admintools option distribute_config_files :
>
> ```
> $ admintools -t distribute_config_files
> ```

Stopping and starting nodes on MC

You can start and stop one or more database nodes through the **Manage** page by clicking a specific node to select it and then clicking the Start or Stop button in the Node List.

> **Note**
> The Stop and Start buttons in the toolbar start and stop the database, not individual nodes.

On the **Databases and Clusters** page, you must click a database first to select it. To stop or start a node on that database, click the **View** button. You'll be directed to the Overview page. Click **Manage** in the applet panel at the bottom of the page and you'll be directed to the database node view.

The Start and Stop database buttons are always active, but the node Start and Stop buttons are active only when one or more nodes of the same status are selected; for example, all nodes are UP or DOWN.

After you click a Start or Stop button, Management Console updates the status and message icons for the nodes or databases you are starting or stopping.

Upgrading your operating system on nodes in your Vertica cluster

If you need to upgrade the operating system on the nodes in your Vertica cluster, check with the documentation for your Linux distribution to make sure they support the particular upgrade you are planning.

For example, the following articles provide information about upgrading Red Hat:

- How do I upgrade from Red Hat Enterprise Linux 6 to Red Hat Enterprise Linux 7?
- Does Red Hat support upgrades between major versions of Red Hat Enterprise Linux?

After you confirm that you can perform the upgrade, follow the steps at Best Practices for Upgrading the Operating System on Nodes in a Vertica Cluster .

Reconfiguring node messaging

Sometimes, nodes of an existing, operational Vertica database cluster require new IP addresses . Cluster nodes might also need to change their messaging protocols —for example, from broadcast to point-to-point. The admintools re_ip utility performs both tasks.

> **Note**
> You cannot change from one address family—IPv4 or IPv6—to another. For example, if hosts in the database cluster are identified by IPv4 network addresses, you can only change host addresses to another set of IPv4 addresses.

Changing IP addresses

You can use re_ip to perform two tasks:

- Update node IP addresses
- Change node control and broadcast addresses

In both cases, re_ip requires a mapping file that identifies the current node IP addresses, which are stored in admintools.conf . You can get these addresses in two ways:

- Use the admintools utility list_allnodes :

```
$ admintools -t list_allnodes
Node             | Host          | State | Version        | DB
-----------------+---------------+-------+----------------+-----------
v_vmart_node0001 | 192.0.2.254   | UP    | vertica-12.0.1 | VMart
v_vmart_node0002 | 192.0.2.255   | UP    | vertica-12.0.1 | VMart
v_vmart_node0003 | 192.0.2.256   | UP    | vertica-12.0.1 | VMart
```

> **Tip**
> list_allnodes can help you identify issues that you might have to access Vertica. For example, if hosts are not communicating with each other, the Version column displays Unavailable.

- Print the content of admintools.conf :

```
$ cat /opt/vertica/config/admintools.conf
...
[Cluster]
hosts = 192.0.2.254, 192.0.2.255, 192.0.2.256

[Nodes]
node0001 = 192.0.2.254/home/dbadmin,/home/dbadmin
node0002 = 192.0.2.255/home/dbadmin,/home/dbadmin
node0003 = 192.0.2.256/home/dbadmin,/home/dbadmin
...
```

Update node IP addresses

You can update IP addresses with re_ip as described below. re_ip automatically backs up admintools.conf so you can recover the original settings if necessary.

1. Create a mapping file with lines in the following format:

   ```
   oldIPaddress newIPaddress[, controlAddress, controlBroadcast]
   ...
   ```

   For example:

   ```
   192.0.2.254 198.51.100.255, 198.51.100.255, 203.0.113.255
   192.0.2.255 198.51.100.256, 198.51.100.256, 203.0.113.255
   192.0.2.256 198.51.100.257, 198.51.100.257, 203.0.113.255
   ```

   controlAddress and *controlBroadcast* are optional. If omitted:
   - *controlAddress* defaults to *newIPaddress* .
   - *controlBroadcast* defaults to the host of *newIPaddress* 's broadcast IP address.
2. Stop the database.
3. Run re_ip to map old IP addresses to new IP addresses:

   ```
   $ admintools -t re_ip -f mapfile
   ```

   re_ip issues warnings for the following mapping file errors:
   - IP addresses are incorrectly formatted.
   - Duplicate IP addresses, whether old or new.

   If re_ip finds no syntax errors, it performs the following tasks:
   - Remaps the IP addresses as listed in the mapping file.
   - If the -i option is omitted, asks to confirm updates to the database.
   - Updates required local configuration files with the new IP addresses.
   - Distributes the updated configuration files to the hosts using new IP addresses.

   For example:

```
Parsing mapfile...
New settings for Host 192.0.2.254 are:
address: 198.51.100.255
New settings for Host 192.0.2.255 are:
address: 198.51.100.256
New settings for Host 192.0.2.254 are:
address: 198.51.100.257

The following databases would be affected by this tool: Vmart

Checking DB status ...
Enter "yes" to write new settings or "no" to exit > yes
Backing up local admintools.conf ...
Writing new settings to local admintools.conf ...

Writing new settings to the catalogs of database Vmart ...
The change was applied to all nodes.
Success. Change committed on a quorum of nodes.

Initiating admintools.conf distribution ...
Success. Local admintools.conf sent to all hosts in the cluster.
```

4. Restart the database.

re_ip and export IP address

By default, a node's IP address and its export IP address are identical. For example:

```
=> SELECT node_name, node_address, export_address FROM nodes;
   node_name      | node_address    | export_address
------------------------------------------------------------
v_VMartDB_node0001 | 192.168.100.101 | 192.168.100.101
v_VMartDB_node0002 | 192.168.100.102 | 192.168.100.101
v_VMartDB_node0003 | 192.168.100.103 | 192.168.100.101
v_VMartDB_node0004 | 192.168.100.104 | 192.168.100.101
(4 rows)
```

The export address is the IP address of the node on the network. This address provides access to other DBMS systems, and enables you to import and export data across the network.

If node IP and export IP addresses are the same, then running re_ip changes both to the new address. Conversely, if you manually change the export address , subsequent re_ip operations leave your export address changes untouched.

Change node control and broadcast addresses

You can map IP addresses for the database only, by using the re_ip option -O (or --db-only). Database-only operations are useful for error recovery. The node names and IP addresses that are specified in the mapping file must be the same as the node information in admintools.conf . In this case, admintools.conf is not updated. Vertica updates only spread.conf and the catalog with the changes.

You can also use re_ip to change the node control and broadcast addresses. In this case the mapping file must contain the control messaging IP address and associated broadcast address. This task allows nodes on the same host to have different data and control addresses.

1. Create a mapping file with lines in the following format:

```
nodeName nodeIPaddress, controlAddress, controlBroadcast
  ...
```

> **Tip**
> Query the system table NODES for node names.

For example:

```
vertica_node001 192.0.2.254, 203.0.113.255, 203.0.113.258
vertica_node002 192.0.2.255, 203.0.113.256, 203.0.113.258
vertica_node003 192.0.2.256, 203.0.113.257, 203.0.113.258
```

2. Stop the database.
3. Run the following command to map the new IP addresses:

```
$ admintools -t re_ip -f mapfile -O -d dbname
```

4. Restart the database.

Changing node messaging protocols

You can use re_ip to reconfigure spread messaging between Vertica nodes. re_ip configures node messaging to broadcast or point-to-point (unicast) messaging with these options:

- -U , --broadcast (default)
- -T , --point-to-point

Both options support up to 80 spread daemons. You can exceed the 80-node limit by using large cluster mode, which does not install a spread daemon on each node.

For example, to set the database cluster messaging protocol to point-to-point:

```
$ admintools -t re_ip -d dbname -T
```

To set the messaging protocol to broadcast:

```
$ admintools -t re_ip -d dbname -U
```

Setting re_ip timeout

You can configure how long re_ip executes a given task before it times out, by editing the setting of prepare_timeout_sec in admintools.conf . By default, this parameter is set to 7200 (seconds).

In this section

- re_ip command
- Restarting a node with new host IPs

re_ip command

Updates database cluster node IP addresses and reconfigures spread messaging between nodes.

Syntax

```
admintools -t re_ip { -h
  | -f mapfile [-O -d dbname]
  | -d dbname { -T | U }
} [-i]
```

Options

Option	Description
-h | --help	Displays online help.
-f *mapfile* | --file= *mapfile*	Name of the mapping text file used to map old addresses to new ones .
-O | --dba-only	Used for error recovery, updates and replaces data on the database cluster catalog and control messaging system. If the mapping file fails, Vertica automatically recreates it when you re-run the command. For details, see Change Node Control and Broadcast Addresses . This option updates only one database at a time, so it requires the -d option.

-T | --point-to-point	Sets control messaging to the point-to-point (unicast) protocol. Vertica can change the messaging protocol on only one database at a time, so you must specify the target database with the -d option. Use point-to-point if nodes are not located on the same subnet. Point-to-point supports up to 80 spread daemons. You can exceed the 80-node limit by using large cluster mode, which does not install a spread daemon on each node.
-U | --broadcast	Sets control messaging to the broadcast protocol, the default setting. Vertica can change the messaging protocol on only one database at a time, so you must specify the target database with the -d option. Broadcast supports up to 80 spread daemons. You can exceed the 80-node limit by using large cluster mode, which does not install a spread daemon on each node.
-d *dbname* | --database= *dbname*	Database name, required with the following re_ip options: • -O • -T • -U
-i | --noprompts	System does not prompt to validate new settings before executing the re_ip operation. Prompting is on by default.

Restarting a node with new host IPs

Kubernetes only

> **Note**
> For information about remapping node IP addresses on a non-Kubernetes database, see Reconfiguring node messaging .

The node IP addresses of an Eon Mode database on Kubernetes must occasionally be updated—for example, a pod fails, or is added to the cluster or rescheduled. When this happens, you must update the Vertica catalog with the new IP addresses of affected nodes and restart the node.

> **Note**
> You cannot switch an existing database cluster from one address family to another. For example, you cannot change the IP addresses of the nodes in your database from IPv4 to IPv6.

Vertica's restart_node tool addresses these requirements with its --new-host-ips option, which lets you change the node IP addresses of an Eon Mode database running on Kubernetes, and restart the updated nodes. Unlike remapping node IP addresses on other (non-Kubernetes) databases, you can perform this task on individual nodes in a running database:

```
admintools -t restart_node \
{-d db-name |--database=db-name} [-p password | --password=password] \
{{-s nodes-list | --hosts=nodes-list} --new-host-ips=ip-address-list}
```

- *nodes-list* is a comma-delimited list of nodes to restart. All nodes in the list must be down, otherwise admintools returns an error.
- *ip-address-list* is a comma-delimited list of new IP addresses or host names to assign to the specified nodes.

> **Note**
> Because a host name resolves to an IP address, Vertica recommends that you use the IP address to eliminate unneeded complexity.

The following requirements apply to *nodes-list* and *ip-address-list* :

- The number of node hosts and IP addresses or host names must be the same.

- The lists must not include any embedded spaces.

For example, you can restart node v_k8s_node0003 with a new IP address:

```
$ admintools -t list_allnodes
 Node          | Host       | State    | Version        | DB
---------------+------------+----------+----------------+-----
 v_k8s_node0001 | 172.28.1.4 | UP       | vertica-10.1.1 | K8s
 v_k8s_node0002 | 172.28.1.5 | UP       | vertica-10.1.1 | K8s
 v_k8s_node0003 | 172.28.1.6 | DOWN     | vertica-10.1.1 | K8s

$ admintools -t restart_node -s v_k8s_node0003 --new-host-ips 172.28.1.7 -d K8s
 Info: no password specified, using none
*** Updating IP addresses for nodes of database K8s ***
      Start update IP addresses for nodes
       Updating node IP addresses
       Generating new configuration information and reloading spread
*** Restarting nodes for database K8s ***
       Restarting host [172.28.1.7] with catalog [v_k8s_node0003_catalog]
       Issuing multi-node restart
       Starting nodes:
            v_k8s_node0003 (172.28.1.7)
       Starting Vertica on all nodes. Please wait, databases with a large catalog may take a while to initialize.
       Node Status: v_k8s_node0003: (DOWN)
       Node Status: v_k8s_node0003: (DOWN)
       Node Status: v_k8s_node0003: (DOWN)
       Node Status: v_k8s_node0003: (DOWN)
       Node Status: v_k8s_node0003: (RECOVERING)
       Node Status: v_k8s_node0003: (UP)
$ admintools -t list_allnodes
 Node          | Host       | State | Version        | DB
---------------+------------+-------+----------------+-----
 v_k8s_node0001 | 172.28.1.4 | UP    | vertica-10.1.1 | K8s
 v_k8s_node0002 | 172.28.1.5 | UP    | vertica-10.1.1 | K8s
 v_k8s_node0003 | 172.28.1.7 | UP    | vertica-10.1.1 | K8s
```

Adjusting Spread Daemon timeouts for virtual environments

Vertica relies on Spread daemons to pass messages between database nodes. Occasionally, nodes fail to respond to messages within the specified Spread timeout. These failures might be caused by spikes in network latency or brief pauses in the node's VM—for example, scheduled Azure maintenance timeouts . In either case, Vertica assumes that the non-responsive nodes are down and starts to remove them from the database, even though they might still be running. You can address this issue by adjusting the Spread timeout as needed.

Adjusting spread timeout

By default, the Spread timeout depends on the number of configured Spread segments:

Configured Spread segments	**Default timeout**
1	8 seconds
> 1	25 seconds

Important

If you deploy your Vertica cluster with Azure Marketplace, the default Spread timeout is set to 35 seconds. If you manually create your cluster in Azure, the default Spread timeout is set to 8 or 25 seconds.

If the Spread timeout is likely to elapse before the network or database nodes can respond, increase the timeout to the maximum length of non-responsive time plus five seconds. For example, if Azure memory-preserving maintenance pauses node VMs for up to 30 seconds, set the Spread timeout to 35 seconds.

If you are unsure how long network or node disruptions are liable to last, gradually increase the Spread timeout until fewer instances of UP nodes leave the database.

Important
Vertica cannot react to a node going down or being shut down improperly before the timeout period elapses. Changing Spread's timeout to a value too high can result in longer query restarts if a node goes down.

To see the current setting of the Spread timeout, query system table SPREAD_STATE . For example, the following query shows that the current timeout setting (token_timeout) is set to 8000ms:

```
=> SELECT * FROM V_MONITOR.SPREAD_STATE;
   node_name     | token_timeout
-----------------+---------------
 v_vmart_node0003 |         8000
 v_vmart_node0001 |         8000
 v_vmart_node0002 |         8000
(3 rows)
```

To change the Spread timeout, call the meta-function SET_SPREAD_OPTION and set the token timeout to a new value. The following example sets the timeout to 35000ms (35 seconds):

```
=> SELECT SET_SPREAD_OPTION( 'TokenTimeout', '35000');
NOTICE 9003:  Spread has been notified about the change
                SET_SPREAD_OPTION
-------------------------------------------------------------
 Spread option 'TokenTimeout' has been set to '35000'.

(1 row)

=> SELECT * FROM V_MONITOR.SPREAD_STATE;
   node_name     | token_timeout
-----------------+---------------
 v_vmart_node0001 |         35000
 v_vmart_node0002 |         35000
 v_vmart_node0003 |         35000
(3 rows);
```

> **Note**
> Changing Spread settings with SET_SPREAD_OPTION has minor impact on your cluster as it pauses while the new settings are propagated across the cluster. Because of this delay, changes to the Spread timeout are not immediately visible in system table SPREAD_STATE .

Azure maintenance and spread timeouts

Azure scheduled maintenance on virtual machines might pause nodes longer than the Spread timeout period. If so, Vertica is liable to view nodes that do not respond to Spread messages as down and remove them from the database.

The length of Azure maintenance tasks is usually well-defined. For example, memory-preserving updates can pause a VM for up to 30 seconds while performing maintenance on the system hosting the VM. This pause does not disrupt the node, which resumes normal operation after maintenance is complete. To prevent Vertica from removing nodes while they undergo Azure maintenance, adjust the Spread timeout as needed.

See also

- Configure and launch a new instance
- Configure storage
- Connect to a virtual machine
- Deploy Vertica from the Azure Marketplace
- Eon Mode on Azure prerequisites

Managing disk space

Vertica detects and reports low disk space conditions in the log file so you can address the issue before serious problems occur. It also detects and reports low disk space conditions via SNMP traps if enabled.

Critical disk space issues are reported sooner than other issues. For example, running out of catalog space is fatal; therefore, Vertica reports the condition earlier than less critical conditions. To avoid database corruption when the disk space falls beyond a certain threshold, Vertica begins to reject transactions that update the catalog or data.

> **Caution**
> A low disk space report indicates one or more hosts are running low on disk space or have a failing disk. It is imperative to add more disk space (or replace a failing disk) as soon as possible.

When Vertica reports a low disk space condition, use the DISK_RESOURCE_REJECTIONS system table to determine the types of disk space requests that are being rejected and the hosts on which they are being rejected.

To add disk space, see Adding disk space to a node . To replace a failed disk, see Replacing failed disks .

Monitoring disk space usage

You can use these system tables to monitor disk space usage on your cluster:

System table	Description
DISK_STORAGE	Monitors the amount of disk storage used by the database on each node.
COLUMN_STORAGE	Monitors the amount of disk storage used by each column of each projection on each node.
PROJECTION_STORAGE	Monitors the amount of disk storage used by each projection on each node.

In this section

- Adding disk space to a node
- Replacing failed disks
- Catalog and data files
- Understanding the catalog directory
- Reclaiming disk space from deleted table data

Adding disk space to a node

This procedure describes how to add disk space to a node in the Vertica cluster.

> **Note**
> If you are adding disk space to multiple nodes in the cluster, then use the following procedure for each node, one node at a time.

To add disk space to a node:

1. If you must shut down the hardware to which you are adding disk space, then first shut down Vertica on the host where disk space is being added.
2. Add the new disk to the system as required by the hardware environment. Boot the hardware if it is was shut down.
3. Partition, format, and mount the new disk, as required by the hardware environment.
4. Create a data directory path on the new volume.
   For example:
   ```
   mkdir –p /myNewPath/myDB/host01_data2/
   ```
5. If you shut down the hardware, then restart Vertica on the host.
6. Open a database connection to Vertica and add a storage location to add the new data directory path. Specify the node in the CREATE LOCATION , otherwise Vertica assumes you are creating the storage location on all nodes.
   See Creating storage locations in this guide and the CREATE LOCATION statement in the SQL Reference Manual.

Replacing failed disks

If the disk on which the data or catalog directory resides fails, causing full or partial disk loss, perform the following steps:

1. Replace the disk and recreate the data or catalog directory.
2. Distribute the configuration file (vertica.conf) to the new host. See Distributing Configuration Files to the New Host for details.

3. Restart the Vertica on the host, as described in Restart Vertica On Host .

See Catalog and data files for information about finding your DATABASE_HOME_DIR.

Catalog and data files

For the recovery process to complete successfully, it is essential that catalog and data files be in the proper directories.

In Vertica, the catalog is a set of files that contains information (metadata) about the objects in a database, such as the nodes, tables, constraints, and projections. The catalog files are replicated on all nodes in a cluster, while the data files are unique to each node. These files are installed by default in the following directories:

```
/DATABASE_HOME_DIR/DATABASE_NAME/v_db_nodexxxx_catalog/ /DATABASE_HOME_DIR/DATABASE_NAME/v_db_nodexxxx_catalog/
```

> **Note**
> DATABASE_HOME_DIR is the path, which you can see from the Administration Tools. See Using the administration tools in the Administrator's Guide for details on using the interface.

To view the path of your database:

1. Run the Administration tools .
   ```
   $ /opt/vertica/bin/admintools
   ```
2. From the Main Menu, select **Configuration Menu** and click **OK** .
3. Select **View Database** and click **OK** .
4. Select the database you want would like to view and click **OK** to see the database profile.

See Understanding the catalog directory for an explanation of the contents of the catalog directory.

Understanding the catalog directory

The catalog directory stores metadata and support files for your database. Some of the files within this directory can help you troubleshoot data load or other database issues. See Catalog and data files for instructions on locating your database's catalog directory. By default, it is located in the database directory. For example, if you created the VMart database in the database administrator's account, the path to the catalog directory is:

```
/home/dbadmin/VMart/v_vmart_node*nnnn*_catalog
```

where node *nnnn* is the name of the node you are logged into. The name of the catalog directory is unique for each node, although most of the contents of the catalog directory are identical on each node.

The following table explains the files and directories that may appear in the catalog directory.

> **Note**
> Do not change or delete any of the files in the catalog directory unless asked to do so by Vertica support.

File or Directory	Description
bootstrap-catalog.log	A log file generated as the Vertica server initially creates the database (in which case, the log file is only created on the node used to create the database) and whenever the database is restored from a backup.
Catalog/	Contains catalog information about the database, such as checkpoints.
CopyErrorLogs/	The default location for the COPY exceptions and rejections files generated when data in a bulk load cannot be inserted into the database. See Handling messy data for more information.
DataCollector/	Log files generated by the Data collector .
debug_log.conf	Debugging information configuration file. For Vertica use only.

Epoch.log	Used during recovery to indicate the latest epoch that contains a complete set of data.
ErrorReport.txt	A stack trace written by Vertica if the server process exits unexpectedly.
Libraries/	Contains user defined library files that have been loaded into the database See Developing user-defined extensions (UDxs) . Do not change or delete these libraries through the file system. Instead, use the CREATE LIBRARY , DROP LIBRARY , and ALTER LIBRARY statements.
Snapshots/	The location where backups are stored.
tmp/	A temporary directory used by Vertica's internal processes.
UDxLogs/	Log files written by user defined functions that run in fenced mode .
vertica.conf	The configuration file for Vertica.
vertica.log	The main log file generated by the Vertica server process.
vertica.pid	The process ID and path to the catalog directory of the Vertica server process running on this node.

Reclaiming disk space from deleted table data

You can reclaim disk space from deleted table data in several ways:

- Purge deleted records .
- Rebuild the table .
- Drop table partition .

Memory usage reporting

Vertica periodically polls its own memory usage to determine whether it is below the threshold that is set by configuration parameter MemoryPollerReportThreshold .Polling occurs at regular intervals—by default, every 2 seconds—as set by configuration parameter MemoryPollerIntervalSec .

The memory poller compares MemoryPollerReportThreshold with the following expression:

```
RSS / available-memory
```

When this expression evaluates to a value higher than MemoryPollerReportThreshold —by default, set to 0.93, then the memory poller writes a report to MemoryReport.log , in the Vertica working directory. This report includes information about Vertica memory pools, how much memory is consumed by individual queries and session, and so on. The memory poller also logs the report as an event in system table MEMORY_EVENTS , where it sets EVENT_TYPE to MEMORY_REPORT .

The memory poller also checks for excessive glibc allocation of free memory (glibc memory bloat). For details, see Memory trimming .

Memory trimming

Under certain workloads, glibc can accumulate a significant amount of free memory in its allocation arena. This memory consumes physical memory as indicated by its usage of resident set size (RSS), which glibc does not always return to the operating system. High retention of physical memory by glibc— *glibc memory bloat* —can adversely affect other processes, and, under high workloads, can sometimes cause Vertica to run out of memory.

Vertica provides two configuration parameters that let you control how frequently Vertica detects and consolidates much of the glibc-allocated free memory, and then returns it to the operating system:

- MemoryPollerTrimThreshold : Sets the threshold for the memory poller to start checking whether to trim glibc -allocated memory.
  The memory poller compares MemoryPollerTrimThreshold —by default, set to 0.83— with the following expression:
  ```
  RSS / available-memory
  ```
  If this expression evaluates to a value higher than MemoryPollerTrimThreshold , then the memory poller starts checking the next threshold—set in MemoryPollerMallocBloatThreshold —for glibc memory bloat.

> **Note**
> On high-memory machines where very large Vertica RSS values are atypical, consider a higher setting for MemoryPollerTrimThreshold . To turn off auto-trimming, set this parameter to 0.

- MemoryPollerMallocBloatThreshold : Sets the threshold of glibc memory bloat.
  The memory poller calls glibc function malloc_info() to obtain the amount of free memory in malloc. It then compares MemoryPollerMallocBloatThreshold —by default, set to 0.3—with the following expression:

  ```
  free-memory-in-malloc / RSS
  ```

  If this expression evaluates to a value higher than MemoryPollerMallocBloatThreshold , the memory poller calls glibc function malloc_trim() . This function reclaims free memory from malloc and returns it to the operating system. Details on calls to malloc_trim() are written to system table MEMORY_EVENTS .
  For example, the memory poller calls malloc_trim() when the following conditions are true:
  - MemoryPollerMallocBloatThreshold is set to 0.5.
  - malloc_info() returns 15GB memory in malloc free.
  - RSS is 30GB.

> **Note**
> This parameter is ignored if MemoryPollerTrimThreshold is set to 0 (disabled).

Trimming memory manually

If auto-trimming is disabled, you can manually reduce glibc-allocated memory by calling Vertica function MEMORY_TRIM . This function calls malloc_trim() .

Tuple mover

The Tuple Mover manages ROS data storage. On mergeout , it combines small ROS containers into larger ones and purges deleted data. The Tuple Mover automatically performs these tasks in the background.

The database mode affects which nodes perform Tuple Mover operations:

- In an Enterprise Mode database, all nodes run the Tuple Mover to perform mergeout operations on the data they store.
- In Eon Mode, the primary subscriber to each shard plans Tuple Mover mergeout operations on the ROS containers in the shard. It can delegate the execution of this plan to another node in the cluster.

Tuple Mover operations typically require no intervention. However, Vertica provides various ways to adjust Tuple Mover behavior. For details, see Managing the tuple mover .

The tuple mover in Eon Mode databases

In Eon Mode, the Tuple Mover's operations are broken into two parts: mergeout planning and mergeout execution. Mergeout planning is always carried out by the primary subscribers of the shards involved in the mergeout. These primary subscribers are part of same the primary subcluster. As part of its mergeout planning, the primary subscriber chooses a node to execute the mergeout plan. It uses two criteria to decide which node should execute the mergeout:

- Only nodes that have memory allocated to their TM resource pool are eligible to perform a mergeout. The primary subscriber ignores all nodes in subclusters whose TM pool's MEMORYSIZE and MAXMEMORYSIZE settings are 0.
- From the group of nodes able to execute a mergeout, the primary subscriber chooses the node that has the most ROS containers in its depot that are involved in the mergeout.

Limiting which subclusters perform mergeout tasks

You can prevent a secondary subcluster from being assigned mergeout tasks by changing the MEMORYSIZE and MAXMEMORYSIZE settings of the its TM pool to 0. These settings prevent the primary subscribers from assigning mergeout tasks to nodes in the subcluster.

Important
Primary subclusters must always be able to execute mergeout tasks. Only change these settings on secondary subclusters.

For example, this statement prevents the subcluster named dashboard from running mergeout tasks.

```
=> ALTER RESOURCE POOL TM FOR SUBCLUSTER dashboard MEMORYSIZE '0%'
  MAXMEMORYSIZE '0%';
```

In this section

- Mergeout
- Managing the tuple mover

Mergeout

Mergeout is a Tuple Mover process that consolidates ROS containers and purges deleted records. DML activities such as COPY and data partitioning generate new ROS containers that typically require consolidation, while deleting and repartitioning data requires reorganization of existing containers. The Tuple Mover constantly monitors these activities, and executes mergeout as needed to consolidate and reorganize containers. By doing so, the Tuple Mover seeks to avoid two problems:

- Performance degradation when column data is fragmented across multiple ROS containers.
- Risk of ROS pushback when ROS containers for a given projection increase faster than the Tuple Mover can handle them. A projection can have up to 1024 ROS containers; when it reaches that limit, Vertica starts to return ROS pushback errors on all attempts to query the projection.

In this section

- Mergeout request types and precedence
- Scheduled mergeout
- User-invoked mergeout
- Partition mergeout
- Deletion marker mergeout
- Disabling mergeout on specific tables
- Purging ROS containers
- Mergeout strata algorithm

Mergeout request types and precedence

The Tuple Mover constantly monitors all activity that generates new ROS containers. As it does so, it creates mergeout requests and queues them according to type. These types include, in descending order of precedence:

1. RECOMPUTE_LIMITS: Sets criteria used by the Tuple Mover to determine when to queue new merge requests for a projection. This request type is queued in two cases:
   - When a projection is created.
   - When an existing projection changes—for example, a column is added or dropped, or a configuration parameter changes that affects ROS storage for that projection, such as ActivePartitionCount .
2. MERGEOUT: Consolidate new containers. These containers typically contain data from recent load activity or table partitioning.
3. DVMERGEOUT: Consolidate data marked for deletion, or *delete vectors* .
4. PURGE: Purge aged-out delete vectors from containers.

The Tuple Mover also monitors how frequently containers are created for each projection, to determine which projections might be at risk from ROS pushback. Intense DML activity on projections typically causes a high rate of container creation. The Tuple Mover monitors MERGEOUT and DVMERGEOUT requests and, within each set, prioritizes them according to their level of projection activity. Mergeout requests for projections with the highest rate of container creation get priority for immediate execution.

> **Note**
> The Tuple Mover often postpones mergeout for projections with a low level of load activity. Until a projection meets the internal threshold for queuing mergeout requests, mergeout from those projections is liable to remain on hold.

Scheduled mergeout

At regular intervals set by configuration parameter MergeOutInterval , the Tuple Mover checks the mergeout request queue for pending requests:

1. If the queue contains mergeout requests, the Tuple Mover does nothing and goes back to sleep.
2. If the queue is empty, the Tuple Mover:
   - Processes pending storage location move requests.
   - Checks for new unqueued purge requests and adds them to the queue.

   It then goes back to sleep.

By default, this parameter is set to 600 (seconds).

Important
Scheduled mergeout is independent of the Tuple Mover service that continuously monitors mergeout requests and executes them as needed.

User-invoked mergeout

You can invoke mergeout at any time on one or more projections, by calling Vertica meta-function DO_TM_TASK :

```
DO_TM_TASK('mergeout'[, '[[database.]schema.]{table | projection} ]')
```

The function scans the database catalog within the specified scope to identify outstanding mergeout tasks. If no table or projection is specified, DO_TM_TASK scans the entire catalog. Unlike the continuous TM service, which runs in the TM resource pool, DO_TM_TASK runs in the GENERAL pool. If DO_TM_TASK executes mergeout tasks that are pending in the merge request queue, the TM service removes these tasks from the queue with no action taken.

Partition mergeout

Vertica keeps data from different table partitions or partition groups separate on disk. The Tuple Mover adheres to this separation policy when it consolidates ROS containers. When a partition is first created, it typically has frequent data loads and requires regular activity from the Tuple Mover. As a partition ages, it commonly transitions to a mostly read-only workload and requires much less activity.

The Tuple Mover has two different policies for managing these different partition workloads:

- *Active partition* is the partition that was most recently created. The Tuple Mover uses a strata-based algorithm that seeks to minimize the number of times individual tuples undergo mergeout. A table's active partition count identifies how many partitions are active for that table.
- *Inactive partitions* are those that were not most recently created. The Tuple Mover consolidates ROS containers to a minimal set while avoiding merging containers whose size exceeds MaxMrgOutROSSizeMB .

> **Note**
> If you invoke mergeout with the Vertica meta-function DO_TM_TASK , all partitions are consolidated into the smallest possible number of containers, including active partitions.

For details on how the Tuple Mover identifies active partitions, see Active and inactive partitions .

Partition mergeout thread allocation

The TM resource pool sets the number of threads that are available for mergeout with its MAXCONCURRENCY parameter. By default , this parameter is set to 7. Vertica allocates half the threads to active partitions, and the remaining half to active and inactive partitions. If MAXCONCURRENCY is set to an uneven integer, Vertica rounds up to favor active partitions.

For example, if MAXCONCURRENCY is set to 7, then Vertica allocates four threads exclusively to active partitions, and allocates the remaining three threads to active and inactive partitions as needed. If additional threads are required to avoid ROS pushback, increase MAXCONCURRENCY with ALTER RESOURCE POOL .

Deletion marker mergeout

When you delete data from the database, Vertica does not remove it. Instead, it marks the data as deleted. Using many DELETE statements to mark a small number of rows relative to the size of a table can result in creating many small containers— *delete vectors* —to hold data marked for deletion. Each delete vector container consumes resources, so a large number of such containers can adversely impact performance, especially during recovery.

After the Tuple Mover performs a mergeout, it looks for deletion marker containers that hold few entries. If such containers exist, the Tuple Mover merges them together into a single, larger container. This process helps lower the overhead of tracking deleted data by freeing resources used by multiple, individual containers. The Tuple Mover does not purge or otherwise affect the deleted data, but consolidates delete vectors for greater efficiency.

> **Tip**
> Query system table DELETE_VECTORS to view the number and size of containers that store deleted data.

Disabling mergeout on specific tables

By default, mergeout is enabled for all tables and their projections. You can disable mergeout on a table with ALTER TABLE . For example:

```
=> ALTER TABLE public.store_orders_temp SET MERGEOUT 0;
ALTER TABLE
```

In general, it is useful to disable mergeout on tables that you create to serve a temporary purpose—for example, staging tables that are used to archive old partition data , or swap partitions between tables—which are deleted soon after the task is complete. By doing so, you avoid the mergeout-related overhead that the table would otherwise incur.

You can query system table TABLES to identify tables that have mergeout disabled:

```
=> SELECT table_schema, table_name, is_mergeout_enabled FROM v_catalog.tables WHERE is_mergeout_enabled= 0;
 table_schema |    table_name     | is_mergeout_enabled
--------------+-------------------+---------------------
 public       | store_orders_temp | f
(1 row)
```

Purging ROS containers

Vertica periodically checks ROS storage containers to determine whether delete vectors are eligible for purge, as follows:

1. Counts the number of aged-out delete vectors in each container—that is, delete vectors that are equal to or earlier than the ancient history mark (AHM) epoch.
2. Calculates the percentage of aged-out delete vectors relative to the total number of records in the same ROS container.
3. If this percentage exceeds the threshold set by configuration parameter PurgeMergeoutPercent (by default, 20 percent), Vertica automatically performs a mergeout on the ROS container that permanently removes all aged-out delete vectors. Vertica uses the TM resource pool's MAXCONCURRENCY setting to determine how many threads are available for the mergeout operation.

You can also manually purge all aged-out delete vectors from ROS containers with two Vertica meta-functions:

- DO_TM_TASK('mergeout')
- PURGE

Both functions remove all aged-out delete vectors from ROS containers, regardless of how many are in a given container.

Mergeout strata algorithm

The mergeout operation uses a strata-based algorithm to verify that each tuple is subjected to a mergeout operation a small, constant number of times, despite the process used to load the data. The mergeout operation uses this algorithm to choose which ROS containers to merge for non-partitioned tables and for active partitions in partitioned tables.

Vertica builds strata for each active partition and for projections anchored to non-partitioned tables. The number of strata, the size of each stratum, and the maximum number of ROS containers in a stratum is computed based on disk size, memory, and the number of columns in a projection.

Merging small ROS containers before merging larger ones provides the maximum benefit during the mergeout process. The algorithm begins at stratum 0 and moves upward. It checks to see if the number of ROS containers in a stratum has reached a value equal to or greater than the maximum ROS containers allowed per stratum. The default value is 32. If the algorithm finds that a stratum is full, it marks the projections and the stratum as eligible for mergeout.

Managing the tuple mover

The Tuple Mover is preconfigured to handle typical workloads. However, some situations might require you to adjust Tuple Mover behavior. You can do so in various ways:

- [Configure the TM resource pool](#Configur).
- Manage active data partitions .

Configuring the TM resource pool

The Tuple Mover uses the built-in TM resource pool to handle its workload. Several settings of this resource pool can be adjusted to facilitate handling of high volume loads:

- MEMORYSIZE

- MAXMEMORYSIZE
- MAXCONCURRENCY
- PLANNEDCONCURRENCY

MEMORYSIZE

Specifies how much memory is reserved for the TM pool per node. The TM pool can grow beyond this lower limit by borrowing from the GENERAL pool. By default, this parameter is set to 5% of available memory. If MEMORYSIZE of the GENERAL resource pool is also set to a percentage, the TM pool can compete with it for memory. This value must always be less than or equal to MAXMEMORYSIZE setting.

> **Caution**
> Increasing MEMORYSIZE to a large percentage can cause regressions in memory-sensitive queries that run in the GENERAL pool.

MAXMEMORYSIZE

Sets the upper limit of memory that can be allocated to the TM pool. The TM pool can grow beyond the value set by MEMORYSIZE by borrowing memory from the GENERAL pool. This value must always be equal to or greater than the MEMORYSIZE setting.

In an Eon Mode database, if you set this value to 0 on a subcluster level, the Tuple Mover is disabled on the subcluster.

Important
Never set the TM pool's MAXMEMORYSIZE to 0 on a primary subcluster . Primary subclusters must always run the Tuple Mover.

MAXCONCURRENCY

Sets across all nodes the maximum number of concurrent execution slots available to TM pool. In databases created in Vertica releases ≥9.3, the default value is 7. In databases created in earlier versions, the default is 3.This setting specifies the maximum number of merges that can occur simultaneously on multiple threads.

PLANNEDCONCURRENCY

Specifies the preferred number queries to execute concurrently in the resource pool, across all nodes, by default set to 6. The Resource Manager uses PLANNEDCONCURRENCY to calculate the target memory that is available to a given query:

TM-memory-size / PLANNEDCONCURRENCY

The PLANNEDCONCURRENCY setting must be proportional to the size of RAM, the CPU, and the storage subsystem. Depending on the storage type, increasing PLANNEDCONCURRENCY for Tuple Mover threads might create a storage I/O bottleneck. Monitor the storage subsystem; if it becomes saturated with long I/O queues, more than two I/O queues, and long latency in read and write, adjust the PLANNEDCONCURRENCY parameter to keep the storage subsystem resources below saturation level.

Managing active data partitions

The Tuple Mover assumes that all loads and updates to a partitioned table are targeted to one or more partitions that it identifies as *active* . In general, the partitions with the largest partition keys—typically, the most recently created partitions—are regarded as active. As the partition ages, its workload typically shrinks and becomes mostly read-only.

You can specify how many partitions are active for partitioned tables at two levels, in ascending order of precedence:

- Configuration parameter ActivePartitionCount determines how many partitions are active for partitioned tables in the database. By default, ActivePartitionCount is set to 1. The Tuple Mover applies this setting to all tables that do not set their own active partition count.
- Individual tables can supersede ActivePartitionCount by setting their own active partition count with CREATE TABLE and ALTER TABLE .

For details, see Active and inactive partitions .

See also

Best practices for managing workload resources

Managing workloads

Vertica's resource management scheme allows diverse, concurrent workloads to run efficiently on the database. For basic operations, Vertica pre-configures the built-in GENERAL pool based on RAM and machine cores. You can customize the General pool to handle specific concurrency requirements.

You can also define new resource pools that you configure to limit memory usage, concurrency, and query priority. You can then optionally assign each database user to use a specific resource pool, which controls memory resources used by their requests.

User-defined pools are useful if you have competing resource requirements across different classes of workloads. Example scenarios include:

- A large batch job takes up all server resources, leaving small jobs that update a web page without enough resources. This can degrade user experience.
  In this scenario, create a resource pool to handle web page requests and ensure users get resources they need. Another option is to create a limited resource pool for the batch job, so the job cannot use up all system resources.
- An application has lower priority than other applications and you want to limit the amount of memory and number of concurrent users for the low-priority application.
  In this scenario, create a resource pool with an upper limit on the query's memory and associate the pool with users of the low-priority application.

You can also use resource pools to manage resources assigned to running queries. You can assign a run-time priority to a resource pool, as well as a threshold to assign different priorities to queries with different durations. See Managing resources at query run time for more information.

Enterprise Mode and Eon Mode

In Enterprise Mode, there is one global set of resource pools for the entire database. In Eon Mode, you can allocate resources globally or per subcluster. See Managing workload resources in an Eon Mode database for more information.

In this section

- Resource manager
- Resource pool architecture
- Managing resources at query run time
- Restoring resource manager defaults
- Best practices for managing workload resources
- Managing system resource usage

Resource manager

On a single-user environment, the system can devote all resources to a single query, getting the most efficient execution for that one query. More likely, your environment needs to run several queries at once, which can cause tension between providing each query the maximum amount of resources (fastest run time) and serving multiple queries simultaneously with a reasonable run time.

The Vertica Resource Manager lets you resolve this tension, while ensuring that every query is eventually serviced and that true system limits are respected at all times.

For example, when the system experiences resource pressure, the Resource Manager might queue queries until the resources become available or a timeout value is reached. In addition, when you configure various Resource Manager settings, you can tune each query's target memory based on the expected number of concurrent queries running against the system.

Resource manager impact on query execution

The Resource Manager impacts individual query execution in various ways. When a query is submitted to the database, the following series of events occur:

1. The query is parsed, optimized to determine an execution plan, and distributed to the participating nodes.
2. The Resource Manager is invoked on each node to estimate resources required to run the query and compare that with the resources currently in use. One of the following will occur:
   - If the memory required by the query alone would exceed the machine's physical memory, the query is rejected - it cannot possibly run. Outside of significantly under-provisioned nodes, this case is very unlikely.
   - If the resource requirements are not currently available, the query is queued. The query will remain on the queue until either sufficient resources are freed up and the query runs or the query times out and is rejected.
   - Otherwise the query is allowed to run.
3. The query starts running when all participating nodes allow it to run.

> **Note**
> Once the query is running, the Resource Manager further manages resource allocation using RUNTIMEPRIORITY and RUNTIMEPRIORITYTHRESHOLD parameters for the resource pool . See Managing resources at query run time for more information.

Apportioning resources for a specific query and the maximum number of queries allowed to run depends on the resource pool configuration. See Resource pool architecture .

On each node, no resources are reserved or held while the query is in the queue. However, multi-node queries queued on some nodes will hold resources on the other nodes. Vertica makes every effort to avoid deadlocks in this situation.

Resource pool architecture

The Resource Manager handles resources as one or more resource pools, which are a pre-allocated subset of the system resources with an associated queue.

In Enterprise Mode, there is one global set of resource pools that apply to all subclusters in the entire database. In Eon Mode, you can allocate resources globally or per subcluster. Global-level resource pools apply to all subclusters. Subcluster-level resource pools allow you to fine-tune resources for the type of workloads that the subcluster does. If you have both global- and subcluster-level resource pool settings, you can override any memory-related global setting for that subcluster. Global settings are applied to subclusters that do not have subcluster-level resource pool settings. See Managing workload resources in an Eon Mode database for more information about fine-tuning resource pools per subcluster.

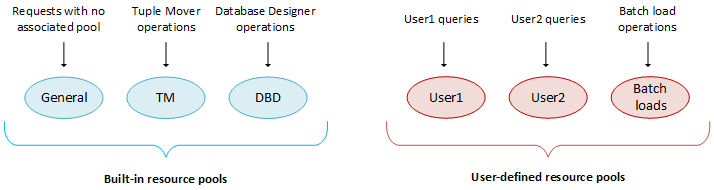
Vertica is preconfigured with a set of Built-in pools that allocate resources to different request types, where the GENERAL pool allows for a certain concurrency level based on the RAM and cores in the machines.

Modifying and creating resource pools

You can configure the built-in GENERAL pool based on actual concurrency and performance requirements, as described in Built-in pools . You can also create custom pools to handle various classes of workloads and optionally restrict user requests to your custom pools.

You create and modify user-defined resource pools with CREATE RESOURCE POOL and ALTER RESOURCE POOL , respectively. You can configure these resource pools for memory usage, concurrency, and queue priority. You can also restrict a database user or user session to use a specific resource pool. Doing so allows you to control how memory, CPU, and other resources are allocated.

The following graphic illustrates what database operations are executed in which resource pool. Only three built-in pools are shown.



In this section

- Defining secondary resource pools
- Querying resource pool settings
- User resource allocation
- Query budgeting

Defining secondary resource pools

You can define secondary resource pools to which running queries can cascade if they exceed their primary pool's RUNTIMECAP .

Identifying a secondary pool

Secondary resource pools designate a place where queries that exceed the RUNTIMECAP of the pool on which they are running can continue execution. If a query exceeds a pool's RUNTIMECAP, the query can cascade to a pool with a larger RUNTIMECAP instead of returning with an error. When a query cascades to another pool, the original pool regains the memory used by that query.

Unlike a user's primary resource pool, which requires USAGE privileges, Vertica does not check for user privileges on secondary resource pools. Thus, a user whose query cascades to a secondary resource pool requires no USAGE privileges on that resource pool.

You can define a secondary pool so it queues long-running queries if the pool lacks sufficient memory to handle that query immediately, by setting two parameters:

- Set PRIORITY to HOLD, to queue queries until the pool's QUEUETIMEOUT interval elapses.
- Set QUEUETIMEOUT to the appropriate length.

Eon Mode restrictions

In Eon Mode, you can associate user-defined resource pools with a subcluster. The following restrictions apply:

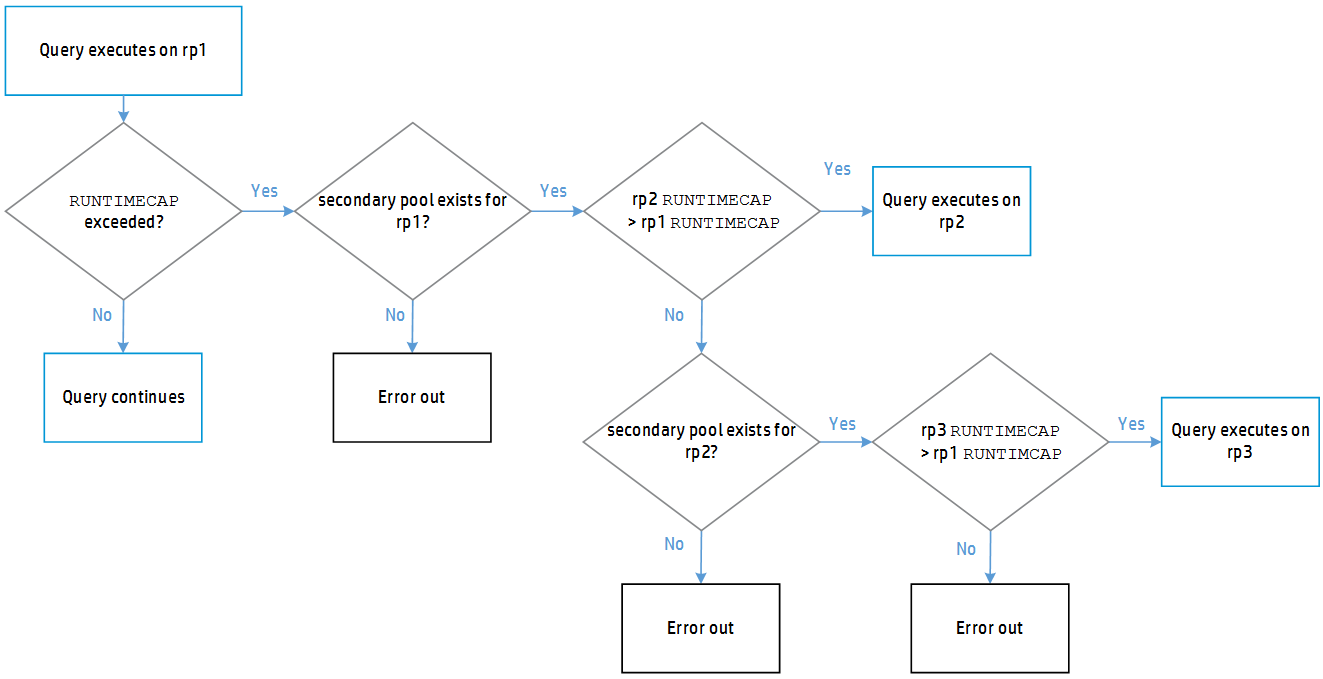
- Global resource pools can cascade only to other global resource pools.
- A subcluster resource pool can cascade to a global resource pool, or to another subcluster-specific resource pool that belongs to the same subcluster. If a subcluster-specific resource pool cascades to a user-defined resource pool that exists on both the global and subcluster level, the subcluster-level resource pool has priority. For example:

```
=> CREATE RESOURCE POOL billing1;
=> CREATE RESOURCE POOL billing1 FOR CURRENT SUBCLUSTER;
=> CREATE RESOURCE POOL billing2 FOR CURRENT SUBCLUSTER CASCADE TO billing1;
WARNING 9613:  Resource pool billing1 both exists at both subcluster level and global level, assuming subcluster level
CREATE RESOURCE POOL
```

Query cascade path

Vertica routes queries to a secondary pool when the RUNTIMECAP on an initial pool is reached. Vertica then checks the secondary pool's RUNTIMECAP value. If the secondary pool's RUNTIMECAP is greater than the initial pool's value, the query executes on the secondary pool. If the secondary pool's RUNTIMECAP is less than or equal to the initial pool's value, Vertica retries the query on the next pool in the chain until it finds a pool on which the RUNTIMECAP is greater than the initial pool's value. If the secondary pool does not have sufficient resources available to execute the query at that time, SELECT queries may re-queue, re-plan, and abort on that pool. Other types of queries will fail due to insufficient resources. If no appropriate secondary pool exists for a query, the query will error out.

The following diagram demonstrates the path a query takes to execution.



Query execution time allocation

After Vertica finds an appropriate pool on which to run the query, it continues to execute that query uninterrupted. The query now has the difference of the two pools' RUNTIMECAP limits in which to complete:

query execution time allocation = rp2 RUNTIMECAP - rp1 RUNTIMECAP

Using CASCADE TO

As a superuser , you can identify an existing resource pool—either user-defined pool or the GENERAL pool—by using the CASCADE TO parameter in the CREATE RESOURCE POOL or ALTER RESOURCE POOL statement.

In the following example, two resource pools are created and associated with a user as follows:

1. The shortUserQueries resource pool is created with a one-minute RUNTIMECAP
2. The userOverflow resource pool is created with a RUNTIMECAP of five minutes.
3. shortUserQueries is modified with ALTER RESOURCE POOL...CASCADE to use userOverflow to handle queries that require more than one minute to process.
4. The user molly is created and configured to use shortUserQueries to handle that user's queries.

Given this scenario, queries issued by molly are initially directed to shortUserQueries for handling; queries that require more than one minute of processing time automatically cascade to the userOverflow pool to complete execution. Using the secondary pool frees up space in the primary pool, which is configured to handle short queries:

```
=> CREATE RESOURCE POOL shortUserQueries RUNTIMECAP '1 minutes'
=> CREATE RESOURCE POOL userOverflow RUNTIMECAP '5 minutes';
=> ALTER RESOURCE POOL shortUserQueries CASCADE TO userOverflow;
=> CREATE USER molly RESOURCE POOL shortUserQueries;
```

If desired, you can modify this scenario so userOverflow can queue long-running queries until it is available to handle them, by setting the PRIORITY and QUEUETIMEOUT parameters:

```
=> ALTER RESOURCE POOL userOverflow PRIORITY HOLD QUEUETIMEOUT '10 minutes';
```

In this scenario, a query that cascades to userOverflow can be queued up to 10 minutes until userOverflow acquires the memory it requires to handle it. After 10 minutes elapse, the query is rejected and returns with an error.

Dropping a secondary pool

If you try to drop a resource pool that is the secondary pool for another resource pool, Vertica returns an error. The error lists the resource pools that depend on the secondary pool you tried to drop. To drop a secondary resource pool, first set the CASCADE TO parameter to DEFAULT on the primary resource pool, and then drop the secondary pool.

For example, you can drop resource pool rp2 , which is a secondary pool for rp1 , as follows:

```
=> ALTER RESOURCE POOL rp1 CASCADE TO DEFAULT;
=> DROP RESOURCE POOL rp2;
```

Secondary pool parameter dependencies

In general, a secondary pool's parameters are applied to an incoming query. In the case of RUNTIMEPRIORITY , the following dependencies apply:

- If the RUNTIMEPRIORITYTHRESHOLD timer was not started when the query was running in the primary pool, the query adopts the secondary resource pools' RUNTIMEPRIORITY when it cascades. This happens either when the RUNTIMEPRIORITYTHRESHOLD is not set for the primary pool or the RUNTIMEPRIORITY is set to HIGH for the primary pool.
- If the RUNTIMEPRIORITYTHRESHOLD was reached in the primary pool, the query adopts the secondary resource pools' RUNTIMEPRIORITY when it cascades.
- If the RUNTIMEPRIORITYTHRESHOLD was not reached in the primary pool and the secondary pool has no threshold, the query adopts the new pool's RUNTIMEPRIORITY when it cascades.
- If the RUNTIMEPRIORITYTHRESHOLD was not reached in the primary pool and the secondary pool has a threshold set.
- If the primary pool's RUNTIMEPRIORITYTHRESHOLD is greater than or equal to the secondary pool's RUNTIMEPRIORITYTHRESHOLD , the query adopts the secondary pool's RUNTIMEPRIORITY after the query reaches the RUNTIMEPRIORITYTHRESHOLD of the primary pool.
  For example:
  ```
  RUNTIMECAP of primary pool = 5 sec
  RUNTIMEPRIORITYTHRESHOLD of primary pool = 8 sec
  RUNTIMTPRIORITYTHRESHOLD of secondary pool = 7 sec
  ```
  In this case, the query runs for 5 seconds on the primary pool and then cascades to the secondary pool. After another 3 seconds, 8 seconds total, the query adopts the RUNTIMEPRIORITY of the secondary pool.
- If the primary pool's RUNTIMEPRIORITYTHRESHOLD is less than the secondary pool's RUNTIMEPRIORITYTHRESHOLD, the query adopts the secondary pool's RUNTIMEPRIORITY after the query reaches the RUNTIMEPRIORITYTHRESHOLD of the secondary pool.
  In this case, the query runs for 5 seconds on the primary pool and then cascades to the secondary pool. After another 7 seconds, 12 seconds total, the query adopts the RUNTIMEPRIORITY of the secondary pool:
  ```
  RUNTIMECAP of primary pool = 5 sec
  RUNTIMEPRIORITYTHRESHOLD of primary pool = 8 sec
  RUNTIMTPRIORITYTHRESHOLD of secondary pool = 12 se
  ```

CASCADE errors

A query that cascades to a secondary resource pool typically returns with an error in the following cases:

- The resource pool cannot acquire the memory it needs to finish processing the query.
- The secondary resource pool parameter MAXCONCURRENCY is set to 1 and it is already processing another query.

Querying resource pool settings

You can use the following to get information about resource pools:

- RESOURCE_POOLS returns resource pool settings from the Vertica database catalog, as set by CREATE RESOURCE POOL and ALTER RESOURCE POOL .

- SUBCLUSTER_RESOURCE_POOL_OVERRIDES displays all subcluster level overrides of global resource pool settings.

For runtime information about resource pools, see Monitoring resource pools .

Querying resource pool settings

The following example queries various settings of two internal resource pools, GENERAL and TM:

```
=> SELECT name, subcluster_oid, subcluster_name, maxmemorysize, memorysize, runtimepriority, runtimeprioritythreshold, queuetimeout
   FROM RESOURCE_POOLS WHERE name IN('general', 'tm');
  name   | subcluster_oid | subcluster_name | maxmemorysize | memorysize | runtimepriority | runtimeprioritythreshold | queuetimeout
---------+----------------+-----------------+---------------+------------+-----------------+--------------------------+--------------
 general |              0 |                 | Special: 95%  |            | MEDIUM          |                        2 | 00:05
 tm      |              0 |                 |               | 3G         | MEDIUM          |                       60 | 00:05
(2 rows)
```

Viewing overrides to global resource pools

In Eon Mode, you can query SUBCLUSTER_RESOURCE_POOL_OVERRIDES in the system tables to view any overrides to global resource pools for individual subclusters. The following query shows an override that sets MEMORYSIZE and MAXMEMORYSIZE for the built-in resource pool TM to 0% in the analytics_1 subcluster. These settings prevent the subcluster from performing Tuple Mover mergeout tasks.

```
=> SELECT * FROM SUBCLUSTER_RESOURCE_POOL_OVERRIDES;
     pool_oid      | name |  subcluster_oid   | subcluster_name | memorysize | maxmemorysize | maxquerymemorysize
-------------------+------+-------------------+-----------------+------------+---------------+--------------------
 45035996273705058 | tm   | 45035996273843504 | analytics_1     | 0%         | 0%            |
(1 row)
```

User resource allocation

You can allocate resources to users in two ways:

- Customize allocation of resources for individual users by setting the appropriate user parameters .
- Assign users to a resource pool. This resource pool is used to process all queries from its assigned users, allocating resources as set by resource pool parameters.

The two methods can complement each other. For example, you can set the RUNTIMECAP parameter in the user-defined resource pool user_rp to 20 minutes. This setting applies to the queries of all users who are assigned to user_rp , including user Bob:

```
=> ALTER RESOURCE POOL user_rp RUNTIMECAP '20 minutes';
ALTER RESOURCE POOL
=> GRANT USAGE ON RESOURCE POOL user_rp to Bob;
GRANT PRIVILEGE
=> ALTER USER Bob RESOURCE POOL pool user_rp;
ALTER USER
```

When Vertica directs any query from user Bob to the user_rp resource pool for processing, it allocates resources to the query as configured in the resource pool, including RUNTIMECAP. Accordingly, queries in user_rp that do not complete execution within 20 minutes cascade to a secondary resource pool (if one is designated), or return to Bob with an error.

You can also edit Bob's user profile by setting its user-level parameter RUNTIMECAP to 10 minutes:

```
=> ALTER USER Bob RUNTIMECAP '10 minutes';
ALTER USER
```

On receiving queries from Bob after this change, the resource pool compares the two RUNTIMECAP settings—its own, and Bob's profile setting—and applies the shorter of the two. If you subsequently reassign Bob to another resource pool, the same logic applies, where the new resource pool continues to apply the shorter of the two RUNTIMECAP settings.

Precedence of user resource pools

Resource pools can be assigned to users at three levels, in ascending order of precedence:

1. Default user resource pool, set in configuration parameter DefaultResourcePoolForUsers . When a database user is created with CREATE USER , Vertica automatically sets the new user's profile to use this resource pool unless the CREATE USER statement specifies otherwise.
   Important

By default, DefaultResourcePoolForUsers is set to the GENERAL resource pool, on which all new users have USAGE privileges. If you reconfigure DefaultResourcePoolForUsers to specify a user-defined resource pool, be sure that new users have USAGE privileges on it.

2. User resource pool, set in the user's profile by CREATE USER or ALTER USER with its RESOURCE POOL parameter. If you try to drop a user's resource pool, Vertica checks whether it can assign that user to the default user resource pool. If the user cannot be assigned to this resource pool—typically, for lack of USAGE privileges—Vertica rolls back the drop operation.
3. Current user session resource pool, set by SET SESSION RESOURCE_POOL .

In all cases, users must have USAGE privileges on their assigned resource pool; otherwise, they cannot log in to the database.

Resource pool usage in Eon Mode

In an Eon Mode database, you can assign a given resource pool to a subcluster, and then configure user profiles to use that resource pool. When users connect to a subcluster, Vertica determines which resource pool handles their queries as follows:

1. If a user's resource pool and the subcluster resource pool are the same, then the subcluster resource pool handles queries from that user.
2. If a user's resource pool and the subcluster resource pool are different, and the user has privileges on the default user resource pool, then that resource pool handles queries from the user.
3. If a user's resource pool and the subcluster resource pool are different, and the user lacks privileges on the default user resource pool, then no resource pool is available on any node to handle queries from that user, and the queries return with an error.

Examples

For examples of different use cases for managing user resources, see Managing workloads with resource pools and user profiles .

Query budgeting

Before it can execute a query, Vertica devises a query plan, which it sends to each node that will participate in executing the query. The Resource Manager evaluates the plan on each node and estimates how much memory and concurrency the node needs to execute its part of the query. This is the *query budget* , which Vertica stores in the query_budget_kb column of system table V_MONITOR.RESOURCE_POOL_STATUS .

A query budget is based on several parameter settings of the resource pool where the query will execute:

- MEMORYSIZE
- MAXMEMORYSIZE
- PLANNEDCONCURRENCY

You can modify MAXMEMORYSIZE and PLANNEDCONCURRENCY for the GENERAL resource pool with ALTER RESOURCE POOL . This resource pool typically executes queries that are not assigned to a user-defined resource pool. You can set all three parameters for any user-defined resource pool when you create it with CREATE RESOURCE POOL , or later with ALTER RESOURCE POOL .

Important

You can also limit how much memory that a pool can allocate at runtime to its queries, by setting parameter MAXQUERYMEMORYSIZE on that pool. For more information, see CREATE RESOURCE POOL .

Computing the GENERAL pool query budget

Vertica calculates query budgets in the GENERAL pool with the following formula:

queryBudget = *queuingThresholdPool* / PLANNEDCONCURRENCY

Note

Vertica calculates the GENERAL pool's queuing threshold as 95 percent of its MAXMEMORYSIZE setting.

Computing query budgets for user-defined resource pools

For user-defined resource pools, Vertica uses the following algorithm:

1. If MEMORYSIZE is set to 0 and MAXMEMORYSIZE is not set:
   queryBudget = *queuingThresholdGeneralPool* / PLANNEDCONCURRENCY
2. If MEMORYSIZE is set to 0 and MAXMEMORYSIZE is set to a non-default value:
   query-budget = *queuingThreshold* / PLANNEDCONCURRENCY

> **Note**
> Vertica calculates a user-defined pool's queuing threshold as 95 percent of its MAXMEMORYSIZE setting.

3. If MEMORYSIZE is set to a non-default value:
   queryBudget = MEMORYSIZE / PLANNEDCONCURRENCY

By carefully tuning a resource pool's MEMORYSIZE and PLANNEDCONCURRENCY parameters, you can control how much memory can be budgeted for queries.

> **Caution**
> Query budgets do not typically require tuning, However, if you reduce the MAXMEMORYSIZE because you need memory for other purposes, be aware that doing so also reduces the query budget. Reducing the query budget negatively impacts the query performance, particularly if the queries are complex.
>
> To maintain the original query budget for the resource pool, be sure to reduce parameters MAXMEMORYSIZE and PLANNEDCONCURRENCY together.

See also

Do You Need to Put Your Query on a Budget? in the Vertica User Community .

Managing resources at query run time

The Resource Manager estimates the resources required for queries to run, and then prioritizes them. You can control how the Resource Manager prioritizes query execution in several ways:

- Set a resource pool's run-time priority relative to other resource pools.
- Change a running query's priority relative to other queries in the same resource pool.
- Move a running query to another resource pool.

In this section

- Setting runtime priority for the resource pool
- Changing runtime priority of a running query
- Manually moving queries to different resource pools

Setting runtime priority for the resource pool

For each resource pool, you can manage resources that are assigned to queries that are already running. You assign each resource pool a runtime priority of HIGH, MEDIUM, or LOW. These settings determine the amount of runtime resources (such as CPU and I/O bandwidth) assigned to queries in the resource pool when they run. Queries in a resource pool with a HIGH priority are assigned greater runtime resources than those in resource pools with MEDIUM or LOW runtime priorities.

Prioritizing queries within a resource pool

While runtime priority helps to manage resources for the resource pool, there may be instances where you want some flexibility within a resource pool. For instance, you may want to ensure that very short queries run at a high priority, while also ensuring that all other queries run at a medium or low priority.

The Resource Manager allows you this flexibility by letting you set a *runtime priority threshold* for the resource pool. With this threshold, you specify a time limit (in seconds) by which a query must finish before it is assigned the runtime priority of the resource pool. All queries begin running with a HIGH priority; once a query's duration exceeds the time limit specified in the runtime priority threshold, it is assigned the runtime priority of the resource pool.

Setting runtime priority and runtime priority threshold

You specify runtime priority and runtime priority threshold by setting two resource pool parameters with CREATE RESOURCE POOL or ALTER RESOURCE POOL :

- RUNTIMEPRIORITY
- RUNTIMEPRIORITYTHRESHOLD

Changing runtime priority of a running query

CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY lets you to change a query's runtime priority. You can change the runtime priority of a query that is already executing.

This function takes two arguments:

- The query's transaction ID, obtained from the system table SESSIONS
- The desired priority, one of the following string values: HIGH , MEDIUM , or LOW

Restrictions

Superusers can change the runtime priority of any query to any priority level. The following restrictions apply to other users:

- They can only change the runtime priority of their own queries.
- They cannot raise the runtime priority of a query to a level higher than that of the resource pools.

Procedure

Changing a query's runtime priority is a two-step procedure:

1. Get the query's transaction ID by querying the system table SESSIONS . For example, the following statement returns information about all running queries:
```
=> SELECT transaction_id, runtime_priority, transaction_description from SESSIONS;
```
2. Run `CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY```, specifying the query's transaction ID and desired runtime priority:
```
=> SELECT CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY(45035996273705748, 'low')
```

Manually moving queries to different resource pools

If you are the database administrator, you can move queries to another resource pool mid-execution using the MOVE_STATEMENT_TO_RESOURCE_POOL meta-function.

You might want to use this feature if a single query is using a large amount of resources, preventing smaller queries from executing.

What happens when a query moves to a different resource pool

When a query is moved from one resource pool to another, it continues executing, provided the target pool has enough resources to accommodate the incoming query. If sufficient resources cannot be assigned in the target pool on at least one node, Vertica cancels the query and attempts to re-plan the query. If Vertica cannot re-plan the query, the query is canceled indefinitely.

When you successfully move a query to a target resource pool, its resources will be accounted for by the target pool and released on the first pool.

If you move a query to a resource pool with PRIORITY HOLD, Vertica cancels the query and queues it on the target pool. This cancellation remains in effect until you change the PRIORITY or move the query to another pool without PRIORITY HOLD. You can use this option if you want to store long-running queries for later use.

You can view the RESOURCE_ACQUISITIONS or RESOURCE_POOL_STATUS system tables to determine if the target pool can accommodate the query you want to move. Be aware that the system tables may change between the time you query the tables and the time you invoke the MOVE_STATEMENT_TO_RESOURCE_POOL meta-function.

When a query successfully moves from one resource pool to another mid-execution, it executes until the greater of the existing and new RUNTIMECAP is reached. For example, if the RUNTIMECAP on the initial pool is greater than that on the target pool, the query can execute until the initial RUNTIMECAP is reached. When a query successfully moves from one resource pool to another mid-execution the CPU affinity will change.

Using the MOVE_STATEMENT_TO_RESOURCE_POOL function

To manually move a query from its current resource pool to another resource pool, use the MOVE_STATEMENT_TO_RESOURCE_POOL meta-function. Provide the session id, transaction id, statement id, and target resource pool name, as shown:

```
=> SELECT MOVE_STATEMENT_TO_RESOURCE_POOL ('v_vmart_node0001.example.-31427:0x82fbm', 45035996273711993, 1, 'my_target_pool');
```

See also:

- Defining secondary resource pools
- MOVE_STATEMENT_TO_RESOURCE_POOL
- RESOURCE_POOL_MOVE

Restoring resource manager defaults

System table RESOURCE_POOL_DEFAULTS stores default values for all parameters for all built-in and user-defined resource pools.

If you have changed the value of any parameter in any of your resource pools and want to restore it to its default, you can simply alter the table and set the parameter to DEFAULT. For example, the following statement sets the RUNTIMEPRIORITY for the resource pool sysquery back to its default value:

```
=> ALTER RESOURCE POOL sysquery RUNTIMEPRIORITY DEFAULT;
```

Best practices for managing workload resources

This section provides general guidelines and best practices on how to set up and tune resource pools for various common scenarios.

> **Note**
> The exact settings for resource pool parameters are heavily dependent on your query mix, data size, hardware configuration, and concurrency requirements. Vertica recommends performing your own experiments to determine the optimal configuration for your system.

In this section

- Basic principles for scalability and concurrency tuning
- Setting a runtime limit for queries
- Handling session socket blocking
- Managing workloads with resource pools and user profiles
- Tuning built-in pools
- Reducing query run time
- Managing workload resources in an Eon Mode database

Basic principles for scalability and concurrency tuning

A Vertica database runs on a cluster of commodity hardware. All loads and queries running against the database take up system resources, such as CPU, memory, disk I/O bandwidth, file handles, and so forth. The performance (run time) of a given query depends on how much resource it has been allocated.

When running more than one query concurrently on the system, both queries are sharing the resources; therefore, each query could take longer to run than if it was running by itself. In an efficient and scalable system, if a query takes up all the resources on the machine and runs in X time, then running two such queries would double the run time of each query to 2X. If the query runs in > 2X, the system is not linearly scalable, and if the query runs in < 2X then the single query was wasteful in its use of resources. Note that the above is true as long as the query obtains the minimum resources necessary for it to run and is limited by CPU cycles. Instead, if the system becomes bottlenecked so the query does not get enough of a particular resource to run, then the system has reached a limit. In order to increase concurrency in such cases, the system must be expanded by adding more of that resource.

In practice, Vertica should achieve near linear scalability in run times, with increasing concurrency, until a system resource limit is reached. When adequate concurrency is reached without hitting bottlenecks, then the system can be considered as ideally sized for the workload.

> **Note**
> Typically Vertica queries on segmented tables run on multiple (likely all) nodes of the cluster. Adding more nodes generally improves the run time of the query almost linearly.

Setting a runtime limit for queries

You can set a limit for the amount of time a query is allowed to run. You can set this limit at three levels, listed in descending order of precedence:

1. The resource pool to which the user is assigned.
2. User profile with RUNTIMECAP configuredby CREATE USER / ALTER USER
3. Session queries, set by SET SESSION RUNTIMECAP

In all cases, you set the runtime limit with an interval value that does not exceed one year. When you set runtime limit at multiple levels, Vertica always uses the shortest value. If a runtime limit is set for a non-superuser, that user cannot set any session to a longer runtime limit. Superusers can set the runtime limit for other users and for their own sessions, to any value up to one year, inclusive.

Example

user1 is assigned to the ad_hoc_queries resource pool:

```
=> CREATE USER user1 RESOURCE POOL ad_hoc_queries;
```

RUNTIMECAP for user1 is set to 1 hour:

```
=> ALTER USER user1 RUNTIMECAP '60 minutes';
```

RUNTIMECAP for the ad_hoc_queries resource pool is set to 30 minutes:

```
=> ALTER RESOURCE POOL ad_hoc_queries RUNTIMECAP '30 minutes';
```

In this example, Vertica terminates user1 's queries if they exceed 30 minutes. Although the user1 's runtime limit is set to one hour, the pool on which the query runs, which has a 30-minute runtime limit, has precedence.

> **Note**
> If a secondary pool for the ad_hoc_queries pool is specified using the CASCADE TO function, the query executes on that pool when the RUNTIMECAP on the ad_hoc_queries pool is surpassed.

See also

- RESOURCE_POOLS
- Defining secondary resource pools

Handling session socket blocking

A session socket can be blocked while awaiting client input or output for a given query. Session sockets are typically blocked for numerous reasons—for example, when the Vertica execution engine transmits data to the client, or a COPY LOCAL operation awaits load data from the client.

In rare cases, a session socket can remain indefinitely blocked. For example, a query times out on the client, which tries to forcibly cancel the query, or relies on the session RUNTIMECAP setting to terminate it. In either case, if the query ends while awaiting messages or data, the socket can remain blocked and the session hang until it is forcibly closed.

Configuring a grace period

You can configure the system with a grace period, during which a lagging client or server can catch up and deliver a pending response. If the socket is blocked for a continuous period that exceeds the grace period setting, the server shuts down the socket and throws a fatal error. The session is then terminated. If no grace period is set, the query can maintain its block on the socket indefinitely.

You should set the session grace period high enough to cover an acceptable range of latency and avoid closing sessions prematurely—for example, normal client-side delays in responding to the server. Very large load operations might require you to adjust the session grace period as needed.

You can set the grace period at four levels, listed in descending order of precedence:

1. Session (highest)
2. User
3. Node
4. Database

Setting grace periods for the database and nodes

At the database and node levels, you set the grace period to any interval up to 20 days, through configuration parameter BlockedSocketGracePeriod :

- ALTER DATABASE *db-name* SET BlockedSocketGracePeriod = ' *interval* ';
- ALTER NODE *node-name* SET BlockedSocketGracePeriod = ' *interval* ';

By default, the grace period for both levels is set to an empty string, which allows unlimited blocking.

Setting grace periods for users and sessions

You can set the grace period for individual users and for a given session, as follows:

- { CREATE | ALTER USER } *user-name* GRACEPERIOD {' *interval* ' | NONE };
- SET SESSION GRACEPERIOD { ' *interval* ' | = DEFAULT | NONE };

A user can set a session to any interval equal to or less than the grace period set for that user. Superusers can set the grace period for other users, and for their own sessions, to any value up to 20 days, inclusive.

Examples

Superuser dbadmin sets the database grace period to 6 hours. This limit only applies to non-superusers. dbadmin can sets the session grace period for herself to any value up to 20 days—in this case, 10 hours:

```
=> ALTER DATABASE VMart SET BlockedSocketGracePeriod = '6 hours';
ALTER DATABASE
=> SHOW CURRENT BlockedSocketGracePeriod;
  level   |           name           | setting
----------+--------------------------+---------
 DATABASE | BlockedSocketGracePeriod | 6 hours
(1 row)

=> SET SESSION GRACEPERIOD '10 hours';
SET
=> SHOW GRACEPERIOD;
    name     | setting
-------------+---------
 graceperiod | 10:00
(1 row)
```

dbadmin creates user user777 created with no grace period setting. Thus, the effective grace period for user777 is derived from the database setting of BlockedSocketGracePeriod , which is 6 hours. Any attempt by user777 to set the session grace period to a value greater than 6 hours returns with an error:

```
=> CREATE USER user777;
=> \c - user777
You are now connected as user "user777".
=> SHOW GRACEPERIOD;
    name     | setting
-------------+---------
 graceperiod | 06:00
(1 row)

=> SET SESSION GRACEPERIOD '7 hours';
ERROR 8175:  The new period 07:00 would exceed the database limit of 06:00
```

dbadmin sets a grace period of 5 minutes for user777 . Now, user777 can set the session grace period to any value equal to or less than the user-level setting:

```
=> \c
You are now connected as user "dbadmin".
=> ALTER USER user777 GRACEPERIOD '5 minutes';
ALTER USER
=> \c - user777
You are now connected as user "user777".
=> SET SESSION GRACEPERIOD '6 minutes';
ERROR 8175:  The new period 00:06 would exceed the user limit of 00:05
=> SET SESSION GRACEPERIOD '4 minutes';
SET
```

Managing workloads with resource pools and user profiles

The scenarios in this section describe common workload-management issues, and provide solutions with examples.

In this section

- Periodic batch loads
- CEO query
- Preventing runaway queries
- Restricting resource usage of ad hoc query application
- Setting a hard limit on concurrency for an application
- Handling mixed workloads: batch versus interactive

- Setting priorities on queries issued by different users
- Continuous load and query
- Prioritizing short queries at run time
- Dropping the runtime priority of long queries

Periodic batch loads

Scenario

You do batch loads every night, or occasionally (infrequently) during the day. When loads are running, it is acceptable to reduce resource usage by queries, but at all other times you want all resources to be available to queries.

Solution

Create a separate resource pool for loads with a higher priority than the preconfigured setting on the build-in GENERAL pool.

In this scenario, nightly loads get preference when borrowing memory from the GENERAL pool. When loads are not running, all memory is automatically available for queries.

Example

Create a resource pool that has higher priority than the GENERAL pool:

1. Create resource pool load_pool with PRIORITY set to 10:
```
=> CREATE RESOURCE POOL load_pool PRIORITY 10;
```
2. Modify user load_user to use the new resource pool:
```
=> ALTER USER load_user RESOURCE POOL load_pool;
```

CEO query

Scenario

The CEO runs a report every Monday at 9AM, and you want to be sure that the report always runs.

Solution

To ensure that a certain query or class of queries always gets resources, you could create a dedicated pool for it as follows:

1. Using the PROFILE command, run the query that the CEO runs every week to determine how much memory should be allocated:
```
=> PROFILE SELECT DISTINCT s.product_key, p.product_description
-> FROM store.store_sales_fact s, public.product_dimension p
-> WHERE s.product_key = p.product_key AND s.product_version = p.product_version
-> AND s.store_key IN (
->  SELECT store_key FROM store.store_dimension
->  WHERE store_state = 'MA')
-> ORDER BY s.product_key;
```
2. At the end of the query, the system returns a notice with resource usage:
```
NOTICE:  Statement is being profiled.HINT:  select * from v_monitor.execution_engine_profiles where
transaction_id=45035996273751349 and statement_id=6;
NOTICE:  Initiator memory estimate for query: [on pool general: 1723648 KB,
minimum: 355920 KB]
```
3. Create a resource pool with MEMORYSIZE reported by the above hint to ensure that the CEO query has at least this memory reserved for it:

```
=> CREATE RESOURCE POOL ceo_pool MEMORYSIZE '1800M' PRIORITY 10;
CREATE RESOURCE POOL
=> \x
Expanded display is on.
=> SELECT * FROM resource_pools WHERE name = 'ceo_pool';
-[ RECORD 1 ]-------+-------------
name               | ceo_pool
is_internal        | f
memorysize         | 1800M
maxmemorysize      |
priority           | 10
queuetimeout       | 300
plannedconcurrency | 4
maxconcurrency     |
singleinitiator    | f
```

4. Assuming the CEO report user already exists, associate this user with the above resource pool using ALTER USER statement.

```
=> ALTER USER ceo_user RESOURCE POOL ceo_pool;
```

5. Issue the following command to confirm that the ceo_user is associated with the ceo_pool:

```
=> SELECT * FROM users WHERE user_name ='ceo_user';
-[ RECORD 1 ]-+-----------------
user_id       | 45035996273713548
user_name     | ceo_user
is_super_user | f
resource_pool | ceo_pool
memory_cap_kb | unlimited
```

If the CEO query memory usage is too large, you can ask the Resource Manager to reduce it to fit within a certain budget. See Query budgeting .

Preventing runaway queries

Scenario

Joe, a business analyst often runs big reports in the middle of the day that take up the whole machine's resources.You want to prevent Joe from using more than 100MB of memory, and you want to also limit Joe's queries to run for less than 2 hours.

Solution

User resource allocation provides a solution to this scenario. To restrict the amount of memory Joe can use at one time, set a MEMORYCAP for Joe to 100MB using the ALTER USER command. To limit the amount of time that Joe's query can run, set a RUNTIMECAP to 2 hours using the same command. If any query run by Joe takes up more than its cap, Vertica rejects the query.

If you have a whole class of users whose queries you need to limit, you can also create a resource pool for them and set RUNTIMECAP for the resource pool. When you move these users to the resource pool, Vertica limits all queries for these users to the RUNTIMECAP you specified for the resource pool.

Example

```
=> ALTER USER analyst_user MEMORYCAP '100M' RUNTIMECAP '2 hours';
```

If Joe attempts to run a query that exceeds 100MB, the system returns an error that the request exceeds the memory session limit, such as the following example:

```
\i vmart_query_04.sqlvsql:vmart_query_04.sql:12: ERROR:  Insufficient resources to initiate plan
on pool general [Request exceeds memory session limit: 137669KB > 102400KB]
```

Only the system database administrator (dbadmin) can increase only the MEMORYCAP setting. Users cannot increase their own MEMORYCAP settings and will see an error like the following if they attempt to edit their MEMORYCAP or RUNTIMECAP settings:

```
ALTER USER analyst_user MEMORYCAP '135M';
ROLLBACK:  permission denied
```

Restricting resource usage of ad hoc query application

Scenario

You recently made your data warehouse available to a large group of users who are inexperienced with SQL. Some users run reports that operate on

a large number of rows and overwhelm the system. You want to throttle system usage by these users.

Solution

1. Create a resource pool for ad hoc applications where MAXMEMORYSIZE is equal to MEMORYSIZE. This prevents queries in that resource pool from borrowing resources from the GENERAL pool. Also, set RUNTIMECAP to limit the maximum duration of ad hoc queries:

```
=> CREATE RESOURCE POOL adhoc_pool
   MEMORYSIZE '200M'
     MAXMEMORYSIZE '200M'
   RUNTIMECAP '20 seconds'
   PRIORITY 0
   QUEUETIMEOUT 300
   PLANNEDCONCURRENCY 4;
=> SELECT pool_name, memory_size_kb, queueing_threshold_kb
   FROM V_MONITOR.RESOURCE_POOL_STATUS WHERE pool_name='adhoc_pool';
 pool_name  | memory_size_kb | queueing_threshold_kb
------------+----------------+-----------------------
 adhoc_pool |         204800 |                153600
(1 row)
```

2. Associate this resource pool with database users who use the application to connect to the database.

```
=> ALTER USER app1_user RESOURCE POOL adhoc_pool;
```

Tip
Other solutions include limiting the memory usage of individual users such as in the Preventing runaway queries .

Setting a hard limit on concurrency for an application

Scenario

For billing purposes, analyst Jane would like to impose a hard limit on concurrency for this application. How can she achieve this?

Solution

The simplest solution is to create a separate resource pool for the users of that application and set its MAXCONCURRENCY to the desired concurrency level. Any queries beyond MAXCONCURRENCY are queued.

Tip
Vertica recommends leaving PLANNEDCONCURRENCY to the default level so the queries get their maximum amount of resources. The system as a whole thus runs with the highest efficiency.

Example

In this example, there are four billing users associated with the billing pool. The objective is to set a hard limit on the resource pool so a maximum of three concurrent queries can be executed at one time. All other queries will queue and complete as resources are freed.

```
=> CREATE RESOURCE POOL billing_pool MAXCONCURRENCY 3 QUEUETIMEOUT 2;
=> CREATE USER bill1_user RESOURCE POOL billing_pool;
=> CREATE USER bill2_user RESOURCE POOL billing_pool;
=> CREATE USER bill3_user RESOURCE POOL billing_pool;
=> CREATE USER bill4_user RESOURCE POOL billing_pool;
=> \x
Expanded display is on.

=> select maxconcurrency,queuetimeout from resource_pools where name = 'billing_pool';
maxconcurrency | queuetimeout
----------------+--------------
             3 |            2
(1 row)
> SELECT reason, resource_type, rejection_count  FROM RESOURCE_REJECTIONS
WHERE pool_name = 'billing_pool' AND node_name ilike '%node0001';
reason                              | resource_type | rejection_count
----------------------------------------+---------------+-----------------
Timedout waiting for resource request | Queries       |              16
(1 row)
```

If queries are running and do not complete in the allotted time (default timeout setting is 5 minutes), the next query requested gets an error similar to the following:

```
ERROR:  Insufficient resources to initiate plan on pool billing_pool [Timedout waiting for resource request: Request exceeds limits:
Queries Exceeded: Requested = 1, Free = 0 (Limit = 3, Used = 3)]
```

The table below shows that there are three active queries on the billing pool.

```
=> SELECT pool_name, thread_count, open_file_handle_count, memory_inuse_kb FROM RESOURCE_ACQUISITIONS
WHERE pool_name = 'billing_pool';
pool_name    | thread_count | open_file_handle_count | memory_inuse_kb
-------------+--------------+------------------------+-----------------
billing_pool |            4 |                      5 |          132870
billing_pool |            4 |                      5 |          132870
billing_pool |            4 |                      5 |          132870
(3 rows)
```

Handling mixed workloads: batch versus interactive

Scenario

You have a web application with an interactive portal. Sometimes when IT is running batch reports, the web page takes a long time to refresh and users complain, so you want to provide a better experience to your web site users.

Solution

The principles learned from the previous scenarios can be applied to solve this problem. The basic idea is to segregate the queries into two groups associated with different resource pools. The prerequisite is that there are two distinct database users issuing the different types of queries. If this is not the case, do consider this a best practice for application design.

**Method 1
**Create a dedicated pool for the web page refresh queries where you:

- Size the pool based on the average resource needs of the queries and expected number of concurrent queries issued from the portal.
- Associate this pool with the database user that runs the web site queries. See CEO query for information about creating a dedicated pool.

This ensures that the web site queries always run and never queue behind the large batch jobs. Leave the batch jobs to run off the GENERAL pool.

For example, the following pool is based on the average resources needed for the queries running from the web and the expected number of concurrent queries. It also has a higher PRIORITY to the web queries over any running batch jobs and assumes the queries are being tuned to take 250M each:

```
=> CREATE RESOURCE POOL web_pool
   MEMORYSIZE '250M'
   MAXMEMORYSIZE NONE
   PRIORITY 10
   MAXCONCURRENCY 5
   PLANNEDCONCURRENCY 1;
```

**Method 2
**Create a resource pool with fixed memory size. This limits the amount of memory available to batch reports so memory is always left over for other purposes. For details, see Restricting resource usage of ad hoc query application .

For example:

```
=> CREATE RESOURCE POOL batch_pool
   MEMORYSIZE '4G'
   MAXMEMORYSIZE '4G'
   MAXCONCURRENCY 10;
```

The same principle can be applied if you have three or more distinct classes of workloads.

Setting priorities on queries issued by different users

Scenario

You want user queries from one department to have a higher priority than queries from another department.

Solution

The solution is similar to the mixed workload case . In this scenario, you do not limit resource usage; you set different priorities. To do so, create two different pools, each with MEMORYSIZE=0% and a different PRIORITY parameter. Both pools borrow from the GENERAL pool, however when competing for resources, the priority determine the order in which each pool's request is granted. For example:

```
=> CREATE RESOURCE POOL dept1_pool PRIORITY 5;
=> CREATE RESOURCE POOL dept2_pool PRIORITY 8;
```

If you find this solution to be insufficient, or if one department's queries continuously starves another department's users, you can add a reservation for each pool by setting MEMORYSIZE so some memory is guaranteed to be available for each department.

For example, both resources use the GENERAL pool for memory, so you can allocate some memory to each resource pool by using ALTER RESOURCE POOL to change MEMORYSIZE for each pool:

```
=> ALTER RESOURCE POOL dept1_pool MEMORYSIZE '100M';
=> ALTER RESOURCE POOL dept2_pool MEMORYSIZE '150M';
```

Continuous load and query

Scenario

You want your application to run continuous load streams, but many have up concurrent query streams. You want to ensure that performance is predictable.

Solution

The solution to this scenario depends on your query mix. In all cases, the following approach applies:

1. Determine the number of continuous load streams required. This may be related to the desired load rate if a single stream does not provide adequate throughput, or may be more directly related to the number of sources of data to load. Create a dedicated resource pool for the loads, and associate it with the database user that will perform them. See CREATE RESOURCE POOL for details.
   In general, concurrency settings for the load pool should be less than the number of cores per node. Unless the source processes are slow, it is more efficient to dedicate more memory per load, and have additional loads queue. Adjust the load pool's QUEUETIMEOUT setting if queuing is expected.
2. Run the load workload for a while and observe whether the load performance is as expected. If the Tuple Mover is not tuned adequately to cover the load behavior, see Managing the tuple mover .
3. If there is more than one kind of query in the system—for example, some queries must be answered quickly for interactive users, while others are part of a batch reporting process—follow the guidelines in Handling mixed workloads: batch versus interactive .
4. Let the queries run and observe performance. If some classes of queries do not perform as desired, then you might need to tune the GENERAL pool as outlined in Restricting resource usage of ad hoc query application , or create more dedicated resource pools for those queries. For more information, see CEO query and Handling mixed workloads: batch versus interactive .

See the sections on Managing workloads and CREATE RESOURCE POOL for information on obtaining predictable results in mixed workload environments.

Prioritizing short queries at run time

Scenario

You recently created a resource pool for users who are inexperienced with SQL and who frequently run ad hoc reports. Until now, you managed resource allocation by creating a resource pool where MEMORYSIZE and MAXMEMORYSIZE are equal. This prevented queries in that resource pool from borrowing resources from the GENERAL pool. Now you want to manage resources at run time and prioritize short queries so they are never queued as a result of limited run-time resources.

Solution

- Set RUNTIMEPRIORITY for the resource pool to MEDIUM or LOW.
- Set RUNTIMEPRIORITYTHRESHOLD for the resource pool to the duration of queries you want to ensure always run at a high priority.

For example:

```
=> ALTER RESOURCE POOL ad_hoc_pool RUNTIMEPRIORITY medium RUNTIMEPRIORITYTHRESHOLD 5;
```

Because RUNTIMEPRIORITYTHRESHOLD is set to 5, all queries in resource pool ad_hoc_pool that complete within 5 seconds run at high priority. Queries that exceeds 5 seconds drop down to the RUNTIMEPRIORITY assigned to the resource pool, MEDIUM.

Dropping the runtime priority of long queries

Scenario

You want most queries in a resource pool to run at a HIGH runtime priority; however, you'd like to be able to drop jobs longer than 1 hour to a lower priority.

Solution

Set the RUNTIMEPRIORITY for the resource pool to LOW and set the RUNTIMEPRIORITYTHRESHOLD to a number that cuts off only the longest jobs.

Example

To ensure that all queries with a duration of more than 3600 seconds (1 hour) are assigned a low runtime priority, modify the resource pool as follows:

- Set the RUNTIMEPRIORITY to LOW.
- Set the RUNTIMETHRESHOLD to 3600

```
=> ALTER RESOURCE POOL ad_hoc_pool RUNTIMEPRIORITY low RUNTIMEPRIORITYTHRESHOLD 3600;
```

Tuning built-in pools

The scenarios in this section describe how to tune built-in pools.

- Restricting Vertica to take only 60% of memory
- Tuning for recovery
- Tuning for refresh
- Tuning tuple mover pool settings
- Tuning for machine learning

In this section

- Restricting Vertica to take only 60% of memory
- Tuning for recovery
- Tuning for refresh
- Tuning tuple mover pool settings
- Tuning for machine learning

Restricting Vertica to take only 60% of memory

Scenario

You have a single node application that embeds Vertica, and some portion of the RAM needs to be devoted to the application process. In this scenario, you want to limit Vertica to use only 60% of the available RAM.

Solution

Set the MAXMEMORYSIZE parameter of the GENERAL pool to the desired memory size. See Resource pool architecture for a discussion on resource limits.

Tuning for recovery

Scenario

You have a large database that contains a single large table with two projections, and with default settings, recovery is taking too long. You want to give recovery more memory to improve speed.

Solution

Set PLANNEDCONCURRENCY and MAXCONCURRENCY in the RECOVERY pool to 1, so recovery can take as much memory as possible from the GENERAL pool and run only one thread at once.

> **Caution**
> This setting can slow down other queries in your system.

Tuning for refresh

Scenario

When a refresh operation is running, system performance is affected and user queries are rejected. You want to reduce the memory usage of the refresh job.

Solution

Set MEMORYSIZE in the REFRESH pool to a fixed value. The Resource Manager then tunes the refresh query to only use this amount of memory.

Important
Remember to reset MEMORYSIZE in the REFRESH pool to 0% after the refresh operation completes, so memory can be used for other operations.

Tuning tuple mover pool settings

Scenario 1

During heavy load operations, you occasionally notice spikes in the number of ROS containers. You would like the Tuple Mover to perform mergeout more aggressively to consolidate ROS containers, and avoid ROS pushback.

Solution

Use ALTER RESOURCE POOL to increase the setting of MAXCONCURRENCY in the TM resource pools . This setting determines how many threads are available for mergeout. By default , this parameter is set to 7. Vertica allocates half the threads to active partitions, and the remaining half to active and inactive partitions as needed. If MAXCONCURRENCY is set to an uneven integer, Vertica rounds up to favor active partitions.

For example, if you increase MAXCONCURRENCY to 9, then Vertica allocates five threads exclusively to active partitions, and allocates the remaining four threads to active and inactive partitions.

Scenario 2

You have a secondary subcluster that is dedicated to time-sensitive analytic queries. You want to limit any other workloads on this subcluster that could interfere with it processing queries while also freeing up memory to perform queries.

By default, each subcluster has a built-in TM resource pool for Tuple Mover operations that makes it eligible to execute Tuple Mover mergeout operations. The TM pool consumes memory that could be used for queries. In addition, the mergeout operation could add a slight overhead to your subcluster's processing. You want to reallocate the memory consumed by the TM pool, and prevent the subcluster from running mergeout operations.

Solution

Use ALTER RESOURCE POOL to override the global TM resource pool for the secondary subcluster, and set both its MAXMEMORYSIZE and MEMORYSIZE to 0. This allows you to use the memory consumed by the global TM pool for use running analytic queries and prevents the subcluster being assigned TM mergeout operations to execute.

Tuning for machine learning

Scenario

A large number of machine learning functions are running, and you want to give them more memory to improve performance.

Solution

Vertica executes machine learning functions in the BLOBDATA resource pool. To improve performance of machine learning functions and avoid spilling queries to disk, increase the pool's MAXMEMORYSIZE setting with ALTER RESOURCE POOL .

For more about tuning query budgets, see Query budgeting .

See also

- Managing resources at query run time
- Built-in resource pools configuration

Reducing query run time

Query run time depends on the complexity of the query, the number of operators in the plan, data volumes, and projection design. I/O or CPU bottlenecks can cause queries to run slower than expected. You can often remedy high CPU usage with better projection design . High I/O can often be traced to contention caused by joins and sorts that spill to disk. However, no single solution addresses all queries that incur high CPU or I/O usage. You must analyze and tune each queryindividually.

You can evaluate a slow-running query in two ways:

- Prefix the query with EXPLAIN to view the optimizer's query plan.
- Examine the execution profile by querying system tables QUERY_CONSUMPTION or EXECUTION_ENGINE_PROFILES .

Examining the query plan can reveal one or more of the following:

- Suboptimal projection sort order
- Predicate evaluation on an unsorted or unencoded column
- Use of GROUPBY HASH instead of GROUPBY PIPE

Profiling

Vertica provides profiling mechanisms that help you evaluate database performance at different levels. For example, you can collect profiling data for a single statement, a single session, or for all sessions on all nodes. For details, see Profiling database performance .

Managing workload resources in an Eon Mode database

You primarily control workloads in an Eon Mode database using subclusters. For example, you can create subclusters for specific use cases, such as ETL or query workloads, or you can create subclusters for different groups of users to isolate workloads. Within each subcluster, you can create individual resource pools to optimize resource allocation according to workload. See Managing subclusters for more information about how Vertica uses subclusters.

Global and subcluster-specific resource pools

You can define global resource pool allocations that affect all nodes in the database. You can also create resource pool allocations at the subcluster level. If you create both, the subcluster-level settings override the global settings.

> **Note**
> The GENERAL pool requires at least 25% of available memory to function properly. If you attempt to set MEMORYSIZE for a user-defined resource pool to more than 75%, Vertica returns an error.

You can use this feature to remove global resource pools that the subcluster does not need. Additionally, you can create a resource pool with settings that are adequate for most subclusters, and then tailor the settings for specific subclusters as needed.

Optimizing ETL and query subclusters

Overriding resource pool settings at the subcluster level allows you to isolate built-in and user-defined resource pools and optimize them by workload. You often assign specific roles to different subclusters:

- Subclusters dedicated to ETL workloads and DDL statements that alter the database.
- Subclusters dedicated to running in-depth, long-running analytics queries. These queries need more resources allocated for the best performance.
- Subclusters that run many short-running "dashboard" queries that you want to finish quickly and run in parallel.

After you define the type of queries executed by each subcluster, you can create a subcluster-specific resource pool that is optimized to improve efficiency for that workload.

The following scenario optimizes 3 subclusters by workload:

- etl: A subcluster that performs ETL that you want to optimize for Tuple Mover operations.
- dashboard: A subcluster that you want to designate for short-running queries executed by a large number of users to refresh a web page.
- analytics: A subcluster that you want to designate for long-running queries.

See Best practices for managing workload resources for additional scenarios about resource pool tuning.

Configure an ETL subcluster to improve TM performance

Vertica chooses the subcluster that has the most ROS containers involved in a mergeout operation in its depot to execute a mergeout (see The Tuple Mover in Eon Mode Databases). Often, a subcluster performing ETL will be the best candidate to perform a mergeout because the data it loaded is involved in the mergeout. You can choose to improve the performance of mergeout operations on a subcluster by altering the TM pool's MAXCONCURRENCY setting to increase the number of threads available for mergeout operations. You cannot change this setting at the subcluster level, so you must set it globally:

```
=> ALTER RESOURCE POOL TM MAXCONCURRENCY 10;
```

See Tuning tuple mover pool settings for additional information about Tuple Mover resources.

Configure the dashboard query subcluster

By default, secondary subclusters have memory allocated to Tuple Mover resource pools. This pool setting allows Vertica to assign mergeout operations to the subcluster, which can add a small overhead. If you primarily use a secondary subcluster for queries, the best practice is to reclaim the memory used by the TM pool and prevent mergeout operations being assigned to the subcluster.

To optimize your dashboard query secondary subcluster, set their TM pool's MEMORYSIZE and MAXMEMORYSIZE settings to 0:

```
=> ALTER RESOURCE POOL TM FOR SUBCLUSTER dashboard MEMORYSIZE '0%'
  MAXMEMORYSIZE '0%';
```

Important

Do not set the TM pool's MEMORYSIZE and MAXMEMORYSIZE settings to 0 on primary subclusters . They must always be able to run the Tuple Mover.

To confirm the overrides, query the SUBCLUSTER_RESOURCE_POOL_OVERRIDES table:

```
=> SELECT pool_oid, name, subcluster_name, memorysize, maxmemorysize
        FROM SUBCLUSTER_RESOURCE_POOL_OVERRIDES;

    pool_oid      | name | subcluster_name | memorysize | maxmemorysize
------------------+------+-----------------+------------+---------------
 45035996273705046 | tm   | dashboard       | 0%         | 0%
(1 row)
```

To optimize the dashboard subcluster for short-running queries on a web page, create a dash_pool subcluster-level resource pool that uses 70% of the subcluster's memory. Additionally, increase PLANNEDCONCURRENCY to use all of the machine's logical cores, and limit EXECUTIONPARALLELISM to no more than half of the machine's available cores:

```
=> CREATE RESOURCE POOL dash_pool FOR SUBCLUSTER dashboard
    MEMORYSIZE '70%'
    PLANNEDCONCURRENCY 16
    EXECUTIONPARALLELISM 8;
```

Configure the analytic query subcluster

To optimize the analytics subcluster for long-running queries, create an analytics_pool subcluster-level resource pool that uses 60% of the subcluster's memory. In this scenario, you cannot allocate more memory to this pool because the nodes in this subcluster still have memory assigned to their TM pools. Additionally, set EXECUTIONPARALLELISM to AUTO to use all cores available on the node to process a query, and limit PLANNEDCONCURRENCY to no more than 8 concurrent queries:

```
=> CREATE RESOURCE POOL analytics_pool FOR SUBCLUSTER analytics
     MEMORYSIZE '60%'
     EXECUTIONPARALLELISM AUTO
     PLANNEDCONCURRENCY 8;
```

Managing system resource usage

You can use the Using system tables to track overall resource usage on your cluster. These and the other system tables are described in the Vertica system tables .

If your queries are experiencing errors due to resource unavailability, you can use the following system tables to obtain more details:

System Table	Description
RESOURCE_REJECTIONS	Monitors requests for resources that are rejected by the Resource manager .
DISK_RESOURCE_REJECTIONS	Monitors requests for resources that are rejected due to disk space shortages. See Managing disk space for more information.

When requests for resources of a certain type are being rejected, do one of the following:

- Increase the resources available on the node by adding more memory, more disk space, and so on. See Managing disk space .
- Reduce the demand for the resource by reducing the number of users on the system (see Managing sessions), rescheduling operations, and so on.

The LAST_REJECTED_VALUE field in RESOURCE_REJECTIONS indicates the cause of the problem. For example:

- The message Usage of a single requests exceeds high limit means that the system does not have enough of the resource available for the single request. A common example occurs when the file handle limit is set too low and you are loading a table with a large number of columns.
- The message Timed out or Canceled waiting for resource reservation usually means that there is too much contention for the resource because the hardware platform cannot support the number of concurrent users using it.

In this section

- Managing sessions
- Managing load streams

Managing sessions

Vertica provides several methods for database administrators to view and control sessions. The methods vary according to the type of session:

- External (user) sessions are initiated by vsql or programmatic (ODBC or JDBC) connections and have associated client state.
- Internal (system) sessions are initiated by Vertica and have no client state.

Configuring maximum sessions

The maximum number of per-node user sessions is set by the configuration parameter MaxClientSessions parameter, by default 50. You can set MaxClientSessions parameter to any value between 0 and 1000. In addition to this maximum, Vertica also allows up to five administrative sessions per node.

For example:

```
=> ALTER DATABASE DEFAULT SET MaxClientSessions = 100;
```

> **Note**
> If you use the Administration Tools "Connect to Database" option, Vertica will attempt connections to other nodes if a local connection does not succeed. These cases can result in more successful "Connect to Database" commands than you would expect given the MaxClientSessions value.

Viewing sessions

The system table SESSIONS contains detailed information about user sessions and returns one row per session. Superusers have unrestricted access to all database metadata. Access for other users varies according to their privileges .

Interrupting and closing sessions

You can interrupt a running statement with the Vertica function INTERRUPT_STATEMENT . Interrupting a running statement returns a session to an idle state:

- No statements or transactions are running.
- No locks are held.
- The database is doing no work on behalf of the session.

Closing a user session interrupts the session and disposes of all state related to the session, including client socket connections for the target sessions. The following Vertica functions close one or more user sessions:

- CLOSE_SESSION
- CLOSE_ALL_SESSIONS
- CLOSE_USER_SESSIONS
- SHUTDOWN

SELECT statements that call these functions return after the interrupt or close message is delivered to all nodes. The function might return before Vertica completes execution of the interrupt or close operation. Thus, there might be a delay after the statement returns and the interrupt or close takes effect throughout the cluster. To determine if the session or transaction ended, query the SESSIONS system table.

In order to shut down a database, you must first close all user sessions. For more about database shutdown, see Stopping the database .

Managing load streams

You can use system table LOAD_STREAMS to monitor data as it is loaded on your cluster. Several columns in this table showmetrics for each load stream on each node, including the following:

Column name	Value...
ACCEPTED_ROW_COUNT	Increases during parsing, up to the maximum number of rows in the input file.
PARSE_COMPLETE_PERCENT	Remains zero (0) until all named pipes return an EOF. While COPY awaits an EOF from multiple pipes, it can appear to be hung. However, before canceling the COPY statement, check your system CPU and disk accesses to determine if any activity is in progress. In a typical load, the PARSE_COMPLETE_PERCENT value can either increase slowly or jump quickly to 100%, if you are loading from named pipes or STDIN.
SORT_COMPLETE_PERCENT	Remains at 0 when loading from named pipes or STDIN. After PARSE_COMPLETE_PERCENT reaches 100 percent, SORT_COMPLETE_PERCENT increases to 100 percent.

Depending on the data sizes, a significant lag can occur between the time PARSE_COMPLETE_PERCENT reaches 100 percent and the time SORT_COMPLETE_PERCENT begins to increase.

HTTPS service

The HTTPS service lets clients securely access and manage a Vertica database with a REST API. This service listens on port 8443 and runs on all nodes.

Only the dbadmin user can authenticate to the HTTPS service.

This service encrypts communications with mutual TLS (mTLS) . To configure mTLS, you must alter the server TLS configuration with a server certificate and a trusted Certificate Authority (CA). For mTLS authentication, each client request must include a certificate that is signed by the CA in the server TLS configuration and specifies the dbadmin user in the Common Name (CN). For additional details about these TLS components and Vertica, see TLS protocol .

Important

During installation, the install_vertica script generates self-signed certificates in the /opt/vertica/config/https_certs directory. Vertica uses these certificates to bootstrap the HTTPS service on a new cluster—they are not suitable for production. Certificates in the TLS configuration supersede those in the /opt/vertica/config/https_certs directory.

Password authentication

The following command connects to the HTTPS service from outside the cluster with the username and password:

```
$ curl --insecure --user dbadmin:db-password https://10.20.30.40:8443/endpoint
```

Important

Due to security concerns, this request method is not recommended. For example, the command history can save the dbadmin password.

Certificate authentication

Client requests authenticate to the HTTPS service with a private key and certificate:

```
$ curl https://10.20.30.40:8443/endpoint \
  --key path/to/client_key.key \
  --cert path/to/client_cert.pem \
```

When the Vertica server receives the request, it verifies that the client certificate is signed by a trusted CA and specifies the dbadmin user. To establish this workflow, you must complete the following:

- Alter the server TLS configuration with a server certificate and a CA.
- Generate a client certificate that is signed by the CA in the server TLS configuration. The client certificate SUBJECT must specify the dbadmin user.
- Grant TLS access to the database.

> **Note**
> To demonstrate a comprehensive setup, the following sections use a self-signed CA certificate that signs both the client and server certificates. In a production environment, you should replace the self-signed CA with a trusted CA.
>
> For details about importing a CA certificate, see Generating TLS certificates and keys .

Create a CA certificate

Important
A self-signed CA certificate is convenient for development purposes, but you should always use a proper certificate authority in a production environment.

A CA is a trusted entity that signs and validates other certificates with its own certificate. The following example generates a self-signed root CA certificate:

1. Generate or import a private key. The following command generates a new private key:

   ```
   => CREATE KEY ca_private_key TYPE 'RSA' LENGTH 4096;
   CREATE KEY
   ```

2. Generate the certificate with the following format. Sign the certificate the with the private key that you generated or imported in the previous step:

   ```
   => CREATE CA CERTIFICATE ca_certificate
   SUBJECT '/C=country_code/ST=state_or_province/L=locality/O=organization/OU=org_unit/CN=Vertica Root CA'
   VALID FOR days_valid
   EXTENSIONS 'authorityKeyIdentifier' = 'keyid:always,issuer', 'nsComment' = 'Vertica generated root CA cert'
   KEY ca_private_key;
   ```

   > **Note**
   > The CA certificate SUBJECT must be different from the SUBJECT of any certificate that it signs.

   For example:

   ```
   => CREATE CA CERTIFICATE SSCA_cert
   SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=OpenText/OU=Vertica/CN=Vertica Root CA'
   VALID FOR 3650
   EXTENSIONS 'nsComment' = 'Self-signed root CA cert'
   KEY SSCA_key;
   ```

Create the server certificate

The server private key and certificate verify the Vertica server's identity for clients:

1. Generate the server private key:

```
=> CREATE KEY server_private_key TYPE 'RSA' LENGTH 2048;
CREATE KEY
```

2. Generate the server certificate with the following format. Include the *server_private_key* , and sign it with the CA certificate:

```
=> CREATE CERTIFICATE server_certificate
   SUBJECT '/C=country_code/ST=state_or_province/L=locality/O=organization/OU=org_unit/CN=Vertica server certificate'
   SIGNED BY ca_certificate
   KEY server_private_key;
CREATE CERTIFICATE
```

   For example:

```
=> CREATE CERTIFICATE server_certificate
   SUBJECT '/C=US/ST=Massachusetts/L=Burlington/O=OpenText/OU=Vertica/CN=Vertica server certificate'
   SIGNED BY ca_certificate
   KEY server_private_key;
CREATE CERTIFICATE
```

Alter the TLS configuration

After you generate the server certificate, you must alter the server's default TLS configuration with the server certificate and its CA. When you change the server TLS configuration, the HTTPS service restarts, and the new keys and certificates are added to the catalog and distributed to the nodes in the cluster:

1. Alter the default server configuration . Mutual TLS requires that you set TLSMODE to TRY_VERIFY or VERIFY_CA :

```
=> ALTER TLS CONFIGURATION server CERTIFICATE server_certificate ADD CA CERTIFICATES ca_certificate TLSMODE 'VERIFY_CA';
```

2. Verify the changes on the TLS configuration object :

```
=> SELECT name, certificate, ca_certificate, mode FROM TLS_CONFIGURATIONS WHERE name='server';
 name  |    certificate     | ca_certificate |    mode
-------+--------------------+----------------+-----------
server | server_certificate | ca_certificate | VERIFY_CA
(1 row)
```

Create the client certificate

The client private key and certificate verify the client's identity for requests. Generate a client private key and a client certificate that specifies the dbadmin user, and sign the client certificate with the same CA that signed the server certificate.

The following steps generate a client key and certificate, and then make them available to the client:

1. Generate the client key:

```
=> CREATE KEY client_private_key TYPE 'RSA' LENGTH 2048;
CREATE KEY
```

2. Generate the client certificate. Mutual TLS requires that the Common Name (CN) in the SUBJECT specifies a database username:

```
=> CREATE CERTIFICATE client_certificate
SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=OpenText/OU=Vertica/CN=dbadmin/emailAddress=example@example.com'
SIGNED BY ca_certificate
EXTENSIONS 'nsComment' = 'Vertica client cert', 'extendedKeyUsage' = 'clientAuth'
KEY client_private_key;
CREATE CERTIFICATE
```

3. On the client machine, export the client key and client certificate to the client filesystem. The following commands use the vsql client :

```
$ vsql -At -c "SELECT key FROM cryptographic_keys WHERE name = 'client_private_key';" -o client_private_key.key
$ vsql -At -c "SELECT certificate_text FROM certificates WHERE name = 'client_certificate';" -o client_cert.pem
```

   In the preceding command:
   - -A : enables unaligned output.
   - -t : prevents the command from outputting metadata, such as column names.
   - -c : instructs the shell to run one command and then exit.
   - -o : writes the query output to the specified filename.

   For details about all vsql command line options, see Command-line options
4. Copy or move the client key and certificate to a location that your client recognizes.
   The following commands move the client key and certificate to the hidden directory ~/.client-creds , and then grants the file owner read and write permissions with chmod :

```
$ mkdir ~/.client-creds
$ mv client_private_key.key ~/.client-creds/client_key.key
$ mv client_cert.pem ~/.client-creds/client_cert.pem
$ chmod 600 ~/.client-creds/client_key.key ~/.client-creds/client_cert.pem
```

Create an authentication record

Next, you must create an authentication record in the database. An authentication record defines a set of authentication and the access methods for the database. You grant this record to a user or role to control how they authenticate to the database:

1. Create the authentication record . The tls method requires that clients authenticate with a certificate whose Common Name (CN) specifies a database username:

```
=> CREATE AUTHENTICATION auth_record METHOD 'tls' HOST TLS '0.0.0.0/0';
CREATE AUTHENTICATION
```

2. Grant the authentication record to a user or to a role. The following example grants the authentication record to PUBLIC , the default role for all users:

```
=> GRANT AUTHENTICATION auth_record TO PUBLIC;
GRANT AUTHENTICATION
```

After you grant the authentication record, the user or role can access HTTPS endpoints .

In this section

- HTTPS endpoints
- Prometheus metrics

HTTPS endpoints

The HTTPS service exposes general-purpose endpoints for interacting with your database.

/v1/metrics (GET)

Vertica exposes time series metrics for Prometheus monitoring and alerting. These metrics create a detailed model of your database behavior over time to provide valuable performance and troubleshooting insights.

Send a GET request to /v1/metrics :

```
$ curl https://host:8443/v1/metrics \
  --key path/to/client_key.key \
  --cert path/to/client_cert.pem \
```

Vertica scrapes metrics and outputs the metrics in Prometheus text-based exposition format . This format applies context-specific labels to each metric to help group metrics when you visualize your data. It also describes the metric type—Vertica provides counter, gauge, and histogram metric types . The following example outlines the output format:

```
# HELP metric-name metric-defintion
# TYPE metric-name metric-type
metric-name{label-key="label-value"[, ...]} metric-value
```

For example, the following example shows a snippet of the request response that provides details about the vertica_resource_pool_memory_size_actual_kb metric:

```
$ curl https://10.20.30.40:8443/v1/metrics \
  --key path/to/client_key.key \
  --cert path/to/client_cert.pem \
...
# HELP vertica_resource_pool_memory_size_actual_kb Current amount of memory (in kilobytes) allocated to the resource pool by the resource manager.
# TYPE vertica_resource_pool_memory_size_actual_kb gauge
vertica_resource_pool_memory_size_actual_kb{node_name="v_vmart_node0001",pool_name="metadata",revive_instance_id="114b25c4aab6fec8c26b121c
vertica_resource_pool_memory_size_actual_kb{node_name="v_vmart_node0001",pool_name="blobdata",revive_instance_id="114b25c4aab6fec8c26b121cff
vertica_resource_pool_memory_size_actual_kb{node_name="v_vmart_node0001",pool_name="jvm",revive_instance_id="114b25c4aab6fec8c26b121cff2b52
vertica_resource_pool_memory_size_actual_kb{node_name="v_vmart_node0001",pool_name="sysquery",revive_instance_id="114b25c4aab6fec8c26b121cf
vertica_resource_pool_memory_size_actual_kb{node_name="v_vmart_node0001",pool_name="tm",revive_instance_id="114b25c4aab6fec8c26b121cff2b52"
vertica_resource_pool_memory_size_actual_kb{node_name="v_vmart_node0001",pool_name="general",revive_instance_id="114b25c4aab6fec8c26b121cff2
vertica_resource_pool_memory_size_actual_kb{node_name="v_vmart_node0001",pool_name="recovery",revive_instance_id="114b25c4aab6fec8c26b121cff
vertica_resource_pool_memory_size_actual_kb{node_name="v_vmart_node0001",pool_name="dbd",revive_instance_id="114b25c4aab6fec8c26b121cff2b52
vertica_resource_pool_memory_size_actual_kb{node_name="v_vmart_node0001",pool_name="refresh",revive_instance_id="114b25c4aab6fec8c26b121cff2
...
```

For a comprehensive list of metrics, see Prometheus metrics .

Prometheus metrics

The following table describes the metrics available at https:// *host* :8443/v1/metrics/ .

Name	Type	Description
vertica_allocator_total_size_bytes	gauge	Amount of bytes consumed in an allocator pool.
vertica_build_info	gauge	Shows information about the Vertica build through labels.
vertica_data_size_compressed_mb	gauge	Total compressed size (in megabytes) of the data.
vertica_data_size_estimation_error_mb	gauge	Margin of error (in megabytes) of the estimated raw data size.
vertica_db_info	gauge	Shows information about the current database through labels.
vertica_depot_evictions_bytes	counter	Total size (in bytes) of depot evictions.
vertica_depot_evictions_total	counter	Number of depot evictions.
vertica_depot_fetch_queue_size	gauge	Number of files in the depot's fetch queue.
vertica_depot_fetches_bytes	counter	Total size (in bytes) of successful depot fetches.

vertica_depot_fetches_failures_total	counter	Number of failed depot fetch requests.
vertica_depot_fetches_ms	histogram	Time (in milliseconds) that it takes to fetch files into the depot.
vertica_depot_fetches_requests_total	counter	Number of depot fetch requests.
vertica_depot_lookup_hits_total	counter	Number of cache hits when finding a file in the depot.
vertica_depot_lookup_requests_total	counter	Number of attempts to find a file in the depot.
vertica_depot_max_size_bytes	gauge	Maximum size (in bytes) of the depot.
vertica_depot_size_bytes	gauge	Number of bytes currently used in the depot.
vertica_depot_uploads_bytes	counter	Number of bytes uploaded to persistent storage.
vertica_depot_uploads_failures_total	counter	Number of failures during upload attempts to persistent storage.
vertica_depot_uploads_in_progress_bytes	gauge	Number of bytes in running requests that are uploading a file to persistent storage.
vertica_depot_uploads_in_progress_counter	gauge	Number of requests currently uploading a file to persistent storage.
vertica_depot_uploads_ms	histogram	Time (in milliseconds) it took to upload files to persistent storage.
vertica_depot_uploads_queued_bytes	gauge	Number of bytes in queued requests to upload a file to persistent storage.
vertica_depot_uploads_queued_counter	gauge	Number of queued requests to upload a file to persistent storage.
vertica_depot_uploads_requests_total	counter	Number of file upload attempts to persistent storage.
vertica_depot_usage_percent	gauge	Current size of the depot, expressed as a percentage of maximum depot size.
vertica_disk_storage_free_mb	gauge	Number of megabytes of free storage available.
vertica_disk_storage_free_percent	gauge	Amount of free storage available, expressed as a percentage of total disk storage.
vertica_disk_storage_latency_seek_per_second	gauge	Measures a storage location's performance in seeks/sec. 1/latency is the time that it takes to seek to the data.
vertica_disk_storage_throughput_mb_per_second	gauge	Measures a storage location's performance in MBps. 1/throughput is the time that it takes to read 1MB of data.
vertica_disk_storage_total_mb	gauge	Number of megabytes of total disk storage.
vertica_disk_storage_used_mb	gauge	Number of megabytes of disk storage in use.
vertica_errors	counter	Number of errors, by error level and error code.
vertica_estimated_data_size_raw_mb	gauge	Estimation (in megabytes) of the total raw data size. This is computed each time there is an audit.
vertica_file_system_attempted_operations_total	gauge	Number of attempted file system operations.
vertica_file_system_data_reads_total	gauge	Number of read operations, such as S3 GET requests, to download files.
vertica_file_system_data_writes_total	gauge	Number of write operations, such as S3 PUT requests, to upload files.

vertica_file_system_downstream_bytes	gauge	Number of bytes received.
vertica_file_system_failed_operations_total	gauge	Number of failed filesystem operations.
vertica_file_system_metadata_reads_total	gauge	Number of requests to read metadata. For example, S3 list bucket and HEAD requests are metadata reads.
vertica_file_system_metadata_writes_total	gauge	Number of requests to write metadata. For example, S3 POST and DELETE requests are metadata writes.
vertica_file_system_open_files_counter	gauge	Number of currently open files.
vertica_file_system_reader_counter	gauge	Number of currently running read operations.
vertica_file_system_retries_total	gauge	Number of retry events.
vertica_file_system_upstream_bytes	gauge	Number of bytes sent.
vertica_file_system_writer_counter	gauge	Number of currently running writer operations.
vertica_is_readonly	gauge	Returns whether the nodes are read-only.
vertica_last_audit_end_time	gauge	The time (in milliseconds) that the last audit ended.
vertica_last_catalog_sync_seconds	gauge	Number of seconds elapsed since the most recent catalog sync.
vertica_license_node_count	gauge	If the license limits the number of nodes, the number of nodes that the license allows.
vertica_license_size_mb	gauge	If the license limits the size of the database, the number of megabytes that license allows.
vertica_locked_users	gauge	Number of users that are locked out of their accounts.
vertica_login_attempted_total	counter	Number of login attempts.
vertica_login_failure_total	counter	Number of failed login attempts.
vertica_login_success_total	counter	Number of successful login attempts.
vertica_planned_file_reads_bytes	counter	Total number of bytes read in requests for files (estimated during query planning).
vertica_planned_file_reads_requests_total	counter	Total number of read requests for files (estimated during query planning).
vertica_query_requests_attempted_total	counter	Number of attempted query requests.
vertica_query_requests_failed_total	counter	Number of failed query requests.
vertica_query_requests_processed_rows_total	counter	Number of processed rows for each query type.
vertica_query_requests_succeeded_total	counter	Number of successful query requests.
vertica_query_requests_time_ms	histogram	Time (in milliseconds) that it takes to execute query requests in the resource pool.
vertica_queued_requests_failed_reservation_total	counter	Number of queued requests whose resource reservation failed in the resource pool.

vertica_queued_requests_max_memory_kb	gauge	Maximum memory requested for a single queued request in the resource pool.
vertica_queued_requests_total	gauge	Number of requests that are queued in the resource pool.
vertica_queued_requests_total_memory_kb	gauge	Total memory requested for all queued requests in the resource pool.
vertica_queued_requests_wait_time_ms	histogram	Length of time (in microseconds) that a resource pool queues queries.
vertica_resource_pool_general_memory_borrowed_kb	gauge	Amount of memory (in kilobytes) that running requests borrow from the GENERAL pool.
vertica_resource_pool_max_concurrency	gauge	MAXCONCURRENCY parameter setting for the resource pool. When set to -1, the resource pool can have an unlimited number of concurrent execution slots. When set to 0, queries are prevented from running in the pool.
vertica_resource_pool_max_memory_size_kb	gauge	MAXMEMORYSIZE parameter setting (in kilobytes) for the resource pool.
vertica_resource_pool_max_query_memory_size_kb	gauge	MAXQUERYMEMORYSIZE parameter setting (in kilobytes) for the resource pool. When set to -1, the resource pool borrows any amount of available memory from the GENERAL pool, up to vertica_resource_pool_max_memory_size_kb.
vertica_resource_pool_memory_inuse_kb	gauge	Amount of memory (in kilobytes) acquired by requests running against the resource pool.
vertica_resource_pool_memory_size_actual_kb	gauge	Current amount of memory (in kilobytes) allocated to the resource pool by the resource manager.
vertica_resource_pool_planned_concurrency	gauge	PLANNEDCONCURRENCY parameter setting for the resource pool.
vertica_resource_pool_priority	gauge	PRIORITY parameter setting for the resource pool.
vertica_resource_pool_query_budget_kb	gauge	Amount of resource pool memory (in kilobytes) that queries are currently tuned to use. When equal to -1, queries are prevented from running in the pool.
vertica_resource_pool_queue_timeout	gauge	QUEUETIMEOUT parameter setting for the resource pool.
vertica_resource_pool_queueing_threshold_kb	gauge	Limits the amount of memory (in kilobytes) that a resource pool makes available to all requests before it queues requests.
vertica_resource_pool_running_query_count	gauge	Number of queries currently executing in the pool.
vertica_resource_pool_runtime_priority_threshold	gauge	RUNTIMEPRIORITYTHRESHOLD parameter setting for the resource pool.
vertica_sessions_blocked_counter	gauge	Number of sessions that are blocked waiting for locks.
vertica_sessions_running_counter	gauge	Number of active sessions.
vertica_storage_containers_count	gauge	Total number of storage containers.
vertica_subcluster_info	gauge	Shows information about a subcluster through labels.
vertica_total_nodes_count	gauge	Total number of nodes.
vertica_transactions_completed_total	counter	Number of completed transactions.

vertica_transactions_failed_total	counter	Number of failed transactions.
vertica_transactions_started_total	counter	Number of transactions that have started.
vertica_up_nodes_count	gauge	Number of nodes that have Vertica running and can accept connections.

Monitoring Vertica

You can monitor the activity and health of a Vertica database through various log files and system tables. Vertica provides various configuration parameters that control monitoring options. You can also use the Management Console to observe database activity.

In this section

- Monitoring log files
- Rotating log files
- Monitoring process status (ps)
- Monitoring Linux resource usage
- Monitoring disk space usage
- Monitoring elastic cluster rebalancing
- Monitoring events
- Using system tables
- Data collector utility
- Monitoring partition reorganization
- Monitoring resource pools
- Monitoring recovery
- Clearing projection refresh history
- Monitoring Vertica using notifiers

Monitoring log files

When a database is running

When a Vertica database is running, each node in the cluster writes messages into a file named vertica.log . For example, the Tuple Mover and the transaction manager write INFO messages into vertica.log at specific intervals even when there is no mergeout activity.

You configure the location of the vertica.log file. By default, the log file is in:

```
catalog-path/database-name/node-name_catalog/vertica.log
```

- *catalog-path* is the path shown in the NODES system table minus the Catalog directory at the end.
- *database-name* is the name of your database.
- *node-name* is the name of the node shown in the NODES system table.

> **Note**
> Vertica often changes the format or content of log files in subsequent releases to benefit both customers and customer support.

To monitor one node in a running database in real time:

1. Log in to the database administrator account on any node in the cluster.
2. In a terminal window enter:

```
$ tail -f catalog-path/database-name/node-name_catalog/vertica.log
```

> **Note**
> To monitor your overall database (rather than an individual node/host), use the Data Collector, which records system activities and performance. See Data collector utility for more on Data Collector.

catalog-path	The catalog pathname specified when you created the database. See Creating a database .

database-name	The database name (case sensitive)
node-name	The node name, as specified in the database definition. See Viewing a database .

When the database/node is starting up

During system startup, before the Vertica log has been initialized to write messages, each node in the cluster writes messages into a file named dbLog . This log is useful to diagnose situations where the database fails to start before it can write messages into vertica.log . The dblog is located at the following path, using catalog-path and database-name as described above:

```
catalog-path/database-name/dbLog
```

See also

- Rotating log files

Rotating log files

Most Linux distributions include the logrotate utility. Using this utility simplifies log file administration. By setting up a logrotate configuration file, you can use the utility to complete one or more of these tasks automatically:

- Compress and rotate log files
- Remove log files automatically
- Email log files to named recipients

You can configure logrotate to complete these tasks at specific intervals, or when log files reach a particular size.

If logrotate is present when Vertica is installed, then Vertica automatically sets this utility to look for configuration files. Thus, logrotate searches for configuration files in the /opt/vertica/config/logrotate directory on each node.

When you create a database, Vertica creates database-specific logrotate configurations on each node in your cluster, which are used by the logrotate utility. It then creates a file with the path /opt/vertica/config/logrotate/ for each individual database.

For information about additional settings, use the man logrotate command.

Executing the Python script through the dbadmin logrotate cron job

During the installation of Vertica, the installer configures a cron job for the dbadmin user. This cron job is configured to execute a Python script that runs the logrotate utility. You can view the details of this cron job by viewing the dbadmin.cron file, which is located in the /opt/vertica/config directory.

If you want to customize a cron job to configure logrotate for your Vertica database, you *must* create the cron job under the dbadmin user.

Using the administration tools logrotate utility

You can use the admintools logrotate option to help configure logrotate scripts for a database and distribute the scripts across the cluster. The logrotate option allows you to specify:

- How often to rotate logs
- How large logs can become before being rotated
- How long to keep the logs

Example:

The following example shows you how to set up log rotation on a weekly schedule and keeps for three months (12 logs).

```
$ admintools -t logrotate -d <dbname> -r weekly -k 12
```

See Writing administration tools scripts for more usage information.

Configure logrotate for MC

The Management Console log file is:

```
/opt/vconsole/log/mc/mconsole.log
```

To configure logrotate for MC, configure the following file:

```
/opt/vconsole/temp/webapp/WEB-INF/classes/log4j.xml
```

Edit the log4j.xml file and set these parameters as follows:

1. Restrict the size of the log:
   ```
   <param name="MaxFileSize" value="1MB"/>
   ```
2. Restrict the number of file backups for the log:
   ```
   <param name="MaxBackupIndex" value="1"/>
   ```
3. Restart MC as the root user:
   ```
   # etc/init.d/vertica-consoled restart
   ```

Rotating logs manually

To implement a custom log rotation process, follow these steps:

1. Rename or archive the existing *vertica.log* file. For example:
   ```
   $ mv vertica.log vertica.log.1
   ```
2. Send the Vertica process the USR1 signal, using either of the following approaches:
   ```
   $ killall -USR1 vertica
   ```
   or
   ```
   $ ps -ef | grep -i vertica
   $ kill -USR1 process-id
   ```

See also

- Monitoring log files

Monitoring process status (ps)

You can use ps to monitor the database and Spread processes running on each node in the cluster. For example:

```
$ ps aux | grep /opt/vertica/bin/vertica
$ ps aux | grep /opt/vertica/spread/sbin/spread
```

You should see one Vertica process and one Spread process on each node for common configurations. To monitor Administration Tools and connector processes:

```
$ ps aux | grep vertica
```

There can be many connection processes but only one Administration Tools process.

Monitoring Linux resource usage

You should monitor system resource usage on any or all nodes in the cluster. You can use System Activity Reporting (SAR) to monitor resource usage.

> **Note**
> OpenText recommends that you install pstack and sysstat to help monitor Linux resources. The SYSSTAT package contains utilities for monitoring system performance and usage activity, such as sar , as well as tools you can schedule via cron to collect performance and activity data. See the SYSSTAT Web page for details.
>
> The pstack utility lets you print a stack trace of a running process. See the PSTACK man page for details.

1. Log in to the database administrator account on any node.
2. Run the top utility
   ```
   $ top
   ```

   A high CPU percentage in top indicates that Vertica is CPU-bound. For example:

```
top - 11:44:28 up 53 days, 23:47,  9 users,  load average: 0.91, 0.97, 0.81
Tasks: 123 total,   1 running, 122 sleeping,   0 stopped,   0 zombie
Cpu(s):  26.9%us,  1.3%sy,  0.0%ni, 71.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   4053136 total,  3882020k used,  171116 free,   407688 buffers
Swap:  4192956 total,   176k used,  4192780 free,  1526436 cached
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
13703 dbadmin    1   0 1374m 678m  55m S 99.9 17.1   6:21.70 vertica
 2606 root      16   0 32152  11m 2508 S  1.0  0.3   0:16.97 X
    1 root      16   0  4748  552  456 S  0.0  0.0   0:01.51 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   0:04.92 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:11.75 ksoftirqd/0
...
```

Some possible reasons for high CPU usage are:

- The Tuple Mover runs automatically and thus consumes CPU time even if there are no connections to the database.
- The swappiness kernel parameter may not be set to 0. Execute the following command from the Linux command line to see the value of this parameter:

  ```
  $ cat /proc/sys/vm/swappiness
  ```

  If this value is not 0, change it by following the steps in Check for swappiness .
- Some information sources:
  - Red Hat
  - Indiana University Unix Systems Support Group

3. Run the iostat utility. A high idle time in top at the same time as a high rate of blocks read in iostat indicates that Vertica is disk-bound. For example:

```
$ /usr/bin/iostat
Linux 2.6.18-164.el5 (qa01)     02/05/2011
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          0.77    2.32    0.76    0.68    0.00   95.47
Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
hda               0.37         3.40        10.37    2117723    6464640
sda               0.46         1.94        18.96    1208130   11816472
sdb               0.26         1.79        15.69    1114792    9781840
sdc               0.24         1.80        16.06    1119304   10010328
sdd               0.22         1.79        15.52    1117472    9676200
md0               8.37         7.31        66.23    4554834   41284840
```

Monitoring disk space usage

You can use these system tables to monitor disk space usage on your cluster:

System table	Description
DISK_STORAGE	Monitors the amount of disk storage used by the database on each node.
COLUMN_STORAGE	Monitors the amount of disk storage used by each column of each projection on each node.
PROJECTION_STORAGE	Monitors the amount of disk storage used by each projection on each node.

Monitoring elastic cluster rebalancing

Vertica includes system tables that can be used to monitor the rebalance status of an elastic cluster and gain general insight to the status of elastic cluster on your nodes.

- REBALANCE_TABLE_STATUS provides general information about a rebalance. It shows, for each table, the amount of data that has been separated, the amount that is currently being separated, and the amount to be separated. It also shows the amount of data transferred, the amount that is currently being transferred, and the remaining amount to be transferred (or an estimate if storage is not separated).

> **Note**
> If multiple rebalance methods were used for a single table (for example, the table has unsegmented and segmented projections), the table

> may appear multiple times - once for each rebalance method.

- REBALANCE_PROJECTION_STATUS can be used to gain more insight into the details for a particular projection that is being rebalanced. It provides the same type of information as above, but in terms of a projection instead of a table.

In each table, the columns SEPARATED_PERCENT and TRANSFERRED_PERCENT can be used to determine overall progress.

Historical rebalance information

Historical information about work completed is retained, so query with the table column IS_LATEST to restrict output to only the most recent or current rebalance activity. Historical data may include information about dropped projections or tables. If a table or projection has been dropped and information about the anchor table is not available, then NULL is displayed for the table ID and <unknown> for table name. Information on dropped tables is still useful, for example, in providing justification for the duration of a task.

Monitoring events

To help you monitor your database system, Vertica traps and logs significant events that affect database performance and functionality if you do not address their root causes. This section describes where events are logged, the types of events that Vertica logs, how to respond to these events, the information that Vertica provides for these events, and how to configure event monitoring.

In this section

- Event logging mechanisms
- Event codes
- Event data
- Configuring event reporting
- Event reporting examples

Event logging mechanisms

Vertica posts events to the following mechanisms:

Mechanism	Description
vertica.log	All events are automatically posted to vertica.log . See Monitoring the Log Files .
ACTIVE_EVENTS	This SQL system table provides information about all open events. See Using system tables and ACTIVE_EVENTS .
SNMP	To post traps to SNMP, enable global reporting in addition to each individual event you want trapped. See Configuring event reporting .
Syslog	To log events to syslog, enable event reporting for each individual event you want logged. See Configuring event reporting .

Event codes

The following table lists the event codes that Vertica logs to the events system tables.

Event Code	Severity	Event Code Description	Description/Action

0	Warning	Low Disk Space	Warning indicates one of the following issues: • Lack of disk space for database • Disk failure • I/O hardware failure **Action:** Add more disk space, or replace the failing disk or hardware as soon as possible. Check dmesg to see what caused the problem. Also, use the DISK_RESOURCE_REJECTIONS system table to determine the types of disk space requests that are being rejected and the hosts where they are rejected. See Managing disk space for details.
1	Warning	Read Only File System	Database lacks write access to the file system for data or catalog paths. This sometimes occurs if Linux remounts a drive due to a kernel issue. **Action:** Give the database write access.
2	Emergency	Loss Of K Safety	The database is no longer K-safe because insufficient nodes are functioning within the cluster. Loss of K-safety causes the database to shut down. **Action:** Recover the system .
3	Critical	Current Fault Tolerance at Critical Level	One or more nodes in the cluster failed. If the database loses one more node, it will no longer be K-safe and shut down. **Action:** Restore nodes that failed or shut down.
4	Warning	Too Many ROS Containers	Heavy load activity on one or more projections sometimes generates more ROS containers than the Tuple Mover can handle. Vertica allows up to 1024 ROS containers per projection before it rolls back additional load jobs and returns a ROS pushback error message. **Action:** The Tuple Mover typically catches up with pending mergeout requests and the Optimizer can resume executing queries on affected tables (see Mergeout). If this problem does not resolve quickly, or if it occurs frequently, it is probably related to insufficient RAM allocated to MAXMEMORY in the TM resource pool .
5	Informational	WOS Over Flow	Deprecated
6	Informational	Node State Change	The node state changed. **Action:** Check node status.
7	Warning	Recovery Failure	Database was not restored to a functional state after a hardware or software related failure. **Action:** Reasons for the warning can vary, see the event description for details.
8	Warning	Recovery Error	Database encountered an error while attempting to recover. If the number of recovery errors exceeds Max Tries, the Recovery Failure event is triggered. **Action:** Reasons for the warning can vary, see the event description for details.
9	n/a	Recovery Lock Error	Unused

10	n/a	Recovery Projection Retrieval Error	Unused
11	Warning	Refresh Error	The database encountered an error while attempting to refresh. **Action:** Reasons for the warning can vary, see the event description for details.
12	n/a	Refresh Lock Error	Unused
13	n/a	Tuple Mover Error	Deprecated
14	Warning	Timer Service Task Error	Error occurred in an internal scheduled task. **Action:** None, internal use only
15	Warning	Stale Checkpoint	Deprecated
16	Notice	License Size Compliance	Database size exceeds license size allowance. **Action:** See Monitoring database size for license compliance .
17	Notice	License Term Compliance	Database is not in compliance with your Vertica license. **Action:** Check compliance status with GET_COMPLIANCE_STATUS .
18	Error	CRC Mismatch	Cyclic Redundancy Check (CRC) returned an error or errors while fetching data. **Action:** Review the vertica.log file or the SNMP trap utility to evaluate CRC errors .
19	Critical/Warning	Catalog Sync Exceeds Durability Threshold	Severity: Critical when exceeding hard limit, Warning when exceeding soft limit.
20	Critical	Cluster Read-only	Eon Mode) Quorum or primary shard coverage loss forced database into read-only mode . **Action:** Restart down nodes .

Event data

To help you interpret and solve the issue that triggered an event, each event provides a variety of data, depending upon the event logging mechanism used.

The following table describes the event data and indicates where it is used.

vertica.log	**ACTIVE_EVENTS (column names)**	**SNMP**	**Syslog**	**Description**
N/A	NODE_NAME	N/A	N/A	The node where the event occurred.
Event Code	EVENT_CODE	Event Type	Event Code	A numeric ID that indicates the type of event. See Event Types in the previous table for a list of event type codes.

Event Id	EVENT_ID	Event OID	Event Id	A unique numeric ID that identifies the specific event.
Event Severity	EVENT_ SEVERITY	Event Severity	Event Severity	The severity of the event from highest to lowest. These events are based on standard syslog severity types: 0 – Emergency 1 – Alert 2 – Critical 3 – Error 4 – Warning 5 – Notice 6 – Info 7 – Debug
PostedTimestamp	EVENT_ POSTED_ TIMESTAMP	N/A	PostedTimestamp	The year, month, day, and time the event was reported. Time is provided as military time.
ExpirationTimestamp	EVENT_ EXPIRATION	N/A	ExpirationTimestamp	The time at which this event expires. If the same event is posted again prior to its expiration time, this field gets updated to a new expiration time.
EventCodeDescription	EVENT_CODE_ DESCRIPTION	Description	EventCodeDescription	A brief description of the event and details pertinent to the specific situation.
ProblemDescription	EVENT_PROBLEM_ DESCRIPTION	Event Short Description	ProblemDescription	A generic description of the event.
N/A	REPORTING_ NODE	Node Name	N/A	The name of the node within the cluster that reported the event.
DatabaseName	N/A	Database Name	DatabaseName	The name of the database that is impacted by the event.
N/A	N/A	Host Name	Hostname	The name of the host within the cluster that reported the event.
N/A	N/A	Event Status	N/A	The status of the event. It can be either: 1 – Open 2 – Clear

Configuring event reporting

Event reporting is automatically configured for vertica.log , and current events are automatically posted to the ACTIVE_EVENTS system table. You can also configure Vertica to post events to syslog and SNMP .

In this section

- Configuring reporting for the simple notification service (SNS)
- Configuring reporting for syslog
- Configuring reporting for SNMP
- Configuring event trapping for SNMP
- Verifying SNMP configuration

Configuring reporting for the simple notification service (SNS)

You can monitor Data collector (DC) components and send new rows to Amazon Web Services (AWS) Simple Notification Service (SNS). SNS notifiers are configured with database-level SNS parameters .

> **Note**
> Several SNS configuration parameters have and fall back to equivalents for S3 . This lets you to share configurations between S3 and SNS. For example, if you set the values for AWSAuth but not for SNSAuth, Vertica automatically uses the AWSAuth credentials. For brevity, the procedures on this page will not use this fallback behavior and instead use the SNS configuration parameters.
>
> For details, see SNS parameters .

Minimally, to send DC data to SNS topics, you must configure and specify the following:

- An SNS notifier
- A Simple Notification Service (SNS) topic
- An AWS region (SNSRegion)
- An SNS endpoint (SNSEndpoint) (FIPS only)
- Credentials to authenticate to AWS
- Information about how to handle HTTPS

Creating an SNS notifier

To create an SNS notifier, use CREATE NOTIFIER , specifying sns as the ACTION.

SNS configuration

SNS topics and their subscribers should be configured with AWS. For details, see the AWS documentation:

- Creating an Amazon SNS topic
- Subscribing to an Amazon SNS topic

AWS region and endpoint

In most use cases, you only need to set the AWS region with the SNSRegion parameter; if the SNSEndpoint is set to an empty string (default) and the SNSRegion is set, Vertica automatically finds and uses the appropriate endpoint:

```
=> ALTER DATABASE DEFAULT SET SNSRegion='us-east-1';
=> ALTER DATABASE DEFAULT SET SNSEndpoint='';
```

If you want to specify an endpoint, its region must match the region specified in SNSRegion:

```
=> ALTER DATABASE DEFAULT SET SNSEndpoint='sns.us-east-1.amazonaws.com';
```

If you use FIPS, you should manually set SNSEndpoint to a FIPS-compliant endpoint :

```
=> ALTER DATABASE DEFAULT SET SNSEndpoint='sns-fips.us-east-1.amazonaws.com';
```

AWS credentials

AWS credentials can be set with SNSAuth, which takes an access key and secret access key in the following format:

```
access_key:secret_access_key
```

To set SNSAuth:

```
=> ALTER DATABASE DEFAULT SET SNSAuth='VNDDNVOPIUQF917O5PDB:+mcnVONVIbjOnf1ekNis7nm3mE83u9fjdwmlq36Z';
```

Handling HTTPS

The SNSEnableHttps parameter determines whether the SNS notifier uses TLS to secure the connection between Vertica and AWS. HTTPS is enabled

by default and can be manually enabled with:

```
=> ALTER DATABASE DEFAULT SET SNSEnableHttps=1;
```

If SNSEnableHttps is enabled, depending on your configuration, you might need to specify a custom set of CA bundles with SNSCAFile or SNSCAPath. Amazon root certificates are typically contained in the set of trusted CA certificates already, so you should not have to set these parameters in most environments:

```
=> ALTER DATABASE DEFAULT SET SNSCAFile='path/to/ca/bundle.pem'
=> ALTER DATABASE DEFAULT SET SNSCAPath='path/to/ca/bundles/'
```

HTTPS can be manually disabled with:

```
=> ALTER DATABASE DEFAULT SET SNSEnableHttps=0;
```

Examples

The following example creates an SNS topic, subscribes to it with an SQS queue, and then configures an SNS notifier for the DC component LoginFailures :

1. Create an SNS topic .
2. Create an SQS queue .
3. Subscribe the SQS queue to the SNS topic.
4. Set SNSAuth with your AWS credentials:
   ```
   => ALTER DATABASE DEFAULT SET SNSAuth='VNDDNVOPIUQF917O5PDB:+mcnVONVIbjOnf1ekNis7nm3mE83u9fjdwmlq36Z';
   ```
5. Set SNSRegion:
   ```
   => ALTER DATABASE DEFAULT SET SNSRegion='us-east-1'
   ```
6. Enable HTTPS:
   ```
   => ALTER DATABASE DEFAULT SET SNSEnableHttps=1;
   ```
7. Create an SNS notifier:
   ```
   => CREATE NOTIFIER v_sns_notifier ACTION 'sns' MAXPAYLOAD '256K' MAXMEMORYSIZE '10M' CHECK COMMITTED;
   ```
8. Verify that the SNS notifier, SNS topic, and SQS queue are properly configured:
   1. Manually send a message from the notifier to the SNS topic with NOTIFY :
      ```
      => SELECT NOTIFY('test message', 'v_sns_notifier', 'arn:aws:sns:us-east-1:123456789012:MyTopic')
      ```
   2. Poll the SQS queue for your message.
9. Attach the SNS notifier to the LoginFailures component with SET_DATA_COLLECTOR_NOTIFY_POLICY :
   ```
   => SELECT SET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures', 'v_sns_notifier', 'Login failed!', true)
   ```

To disable an SNS notifier:

```
=> SELECT SET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures', 'v_sns_notifier', 'Login failed!', false)
```

Configuring reporting for syslog

Syslog is a network-logging utility that issues, stores, and processes log messages. It is a useful way to get heterogeneous data into a single data repository.

To log events to syslog, enable event reporting for each individual event you want logged. Messages are logged, by default, to /var/log/messages .

Configuring event reporting to syslog consists of:

1. Enabling Vertica to trap events for syslog.
2. Defining which events Vertica traps for syslog.
   Vertica strongly suggests that you trap the Stale Checkpoint event.
3. Defining which syslog facility to use.

Enabling Vertica to trap events for syslog

To enable event trapping for syslog, issue the following SQL command:

```
=> ALTER DATABASE DEFAULT SET SyslogEnabled = 1;
```

To disable event trapping for syslog, issue the following SQL command:

```
=> ALTER DATABASE DEFAULT SET SyslogEnabled = 0;
```

Defining events to trap for syslog

To define events that generate a syslog entry, issue the following SQL command, one of the events described in the list below the command:

```
=> ALTER DATABASE DEFAULT SET SyslogEvents = 'events-list';
```

where *events-list* is a comma-delimited list of events, one or more of the following:

- Low Disk Space
- Read Only File System
- Loss Of K Safety
- Current Fault Tolerance at Critical Level
- Too Many ROS Containers
- Node State Change
- Recovery Failure
- Recovery Error
- Recovery Lock Error
- Recovery Projection Retrieval Error
- Refresh Error
- Refresh Lock Error
- Tuple Mover Error
- Timer Service Task Error
- Stale Checkpoint

The following example generates a syslog entry for low disk space and recovery failure:

```
=> ALTER DATABASE DEFAULT SET SyslogEvents = 'Low Disk Space, Recovery Failure';
```

Defining the SyslogFacility to use for reporting

The syslog mechanism allows for several different general classifications of logging messages, called facilities. Typically, all authentication-related messages are logged with the auth (or authpriv) facility. These messages are intended to be secure and hidden from unauthorized eyes. Normal operational messages are logged with the daemon facility, which is the collector that receives and optionally stores messages.

The SyslogFacility directive allows all logging messages to be directed to a different facility than the default. When the directive is used, *all* logging is done using the specified facility, both authentication (secure) and otherwise.

To define which SyslogFacility Vertica uses, issue the following SQL command:

```
=> ALTER DATABASE DEFAULT SET SyslogFacility = 'Facility_Name';
```

Where the facility-level argument <Facility_Name> is one of the following:

- auth
- authpriv (Linux only)
- cron
- uucp (UUCP subsystem)
- daemon
- ftp (Linux only)
- lpr (line printer subsystem)
- mail (mail system)
- news (network news subsystem)
- user (default system)
- local0 (local use 0)
- local1 (local use 1)
- local2 (local use 2)
- local3 (local use 3)
- local4 (local use 4)
- local5 (local use 5)
- local6 (local use 6)
- local7 (local use 7)

Trapping other event types

To trap events other than the ones listed above, create a syslog notifier and allow it to trap the desired events with SET_DATA_COLLECTOR_NOTIFY_POLICY .

Events monitored by this notifier type are not logged to MONITORING_EVENTS nor vertica.log .

The following example creates a notifier that writes a message to syslog when the Data collector (DC) component LoginFailures updates:

1. Enable syslog notifiers for the current database:

```
=> ALTER DATABASE DEFAULT SET SyslogEnabled = 1;
```

2. Create and enable a syslog notifier v_syslog_notifier :

```
=> CREATE NOTIFIER v_syslog_notifier ACTION 'syslog'
   ENABLE
   MAXMEMORYSIZE '10M'
   IDENTIFIED BY 'f8b0278a-3282-4e1a-9c86-e0f3f042a971'
   PARAMETERS 'eventSeverity = 5';
```

3. Configure the syslog notifier v_syslog_notifier for updates to the LoginFailures DC component with SET_DATA_COLLECTOR_NOTIFY_POLICY :

```
=> SELECT SET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures','v_syslog_notifier', 'Login failed!', true);
```

   This notifier writes the following message to syslog (default location: /var/log/messages) when a user fails to authenticate as the user Bob :

```
Apr 25 16:04:58
vertica_host_01
vertica:
   Event Posted:
      Event Code:21
      Event Id:0
      Event Severity: Notice [5]
      PostedTimestamp: 2022-04-25 16:04:58.083063
      ExpirationTimestamp: 2022-04-25 16:04:58.083063
      EventCodeDescription: Notifier
      ProblemDescription: (Login failed!)
   {
      "_db":"VMart",
      "_schema":"v_internal",
      "_table":"dc_login_failures",
      "_uuid":"f8b0278a-3282-4e1a-9c86-e0f3f042a971",
      "authentication_method":"Reject",
      "client_authentication_name":"default: Reject",
      "client_hostname":"::1",
      "client_label":"",
      "client_os_user_name":"dbadmin",
      "client_pid":523418,
      "client_version":"",
      "database_name":"dbadmin",
      "effective_protocol":"3.8",
      "node_name":"v_vmart_node0001",
      "reason":"REJECT",
      "requested_protocol":"3.8",
      "ssl_client_fingerprint":"",
      "ssl_client_subject":"",
      "time":"2022-04-25 16:04:58.082568-05",
      "user_name":"Bob"
   }#012
   DatabaseName: VMart
   Hostname: vertica_host_01
```

See also

Event reporting examples

Configuring reporting for SNMP

Configuring event reporting for SNMP consists of:

1. Configuring Vertica to enable event trapping for SNMP as described below.
2. Importing the Vertica Management Information Base (MIB) file into the SNMP monitoring device.
   The Vertica MIB file allows the SNMP trap receiver to understand the traps it receives from Vertica. This, in turn, allows you to configure the actions it takes when it receives traps.
   Vertica supports the SNMP V1 trap protocol, and it is located in /opt/vertica/sbin/VERTICA-MIB. See the documentation for your SNMP monitoring device for more information about importing MIB files.
3. Configuring the SNMP trap receiver to handle traps from Vertica.
   SNMP trap receiver configuration differs greatly from vendor to vendor. As such, the directions presented here for configuring the SNMP trap receiver to handle traps from Vertica are generic.
   Vertica traps are single, generic traps that contain several fields of identifying information. These fields equate to the event data described in Monitoring events . However, the format used for the field names differs slightly. Under SNMP, the field names contain no spaces. Also, field names are pre-pended with “vert”. For example, Event Severity becomes vertEventSeverity.
   When configuring your trap receiver, be sure to use the same hostname, port, and community string you used to configure event trapping in Vertica.
   Examples of network management providers:
   - Network Node Manager i
   - IBM Tivoli
   - AdventNet
   - Net-SNMP (Open Source)
   - Nagios (Open Source)
   - Open NMS (Open Source)

Configuring event trapping for SNMP

The following events are trapped by default when you configure Vertica to trap events for SNMP:

- Low Disk Space
- Read Only File System
- Loss of K Safety
- Current Fault Tolerance at Critical Level
- Too Many ROS Containers
- Node State Change
- Recovery Failure
- Stale Checkpoint
- CRC Mismatch

To configure Vertica to trap events for SNMP

1. Enable Vertica to trap events for SNMP.
2. Define where Vertica sends the traps.
3. Optionally redefine which SNMP events Vertica traps.

> **Note**
> After you complete steps 1 and 2 above, Vertica automatically traps the default SNMP events. Only perform step 3 if you want to redefine which SNMP events are trapped. Vertica recommends that you trap the Stale Checkpoint event even if you decide to reduce the number events Vertica traps for SNMP. The specific settings you define have no effect on traps sent to the log. All events are trapped to the log.

To enable event trapping for SNMP

Use the following SQL command:

```
=> ALTER DATABASE DEFAULT SET SnmpTrapsEnabled = 1;
```

To define where Vertica send traps

Use the following SQL command, where Host_name and port identify the computer where SNMP resides, and CommunityString acts like a password to control Vertica's access to the server:

```
=> ALTER DATABASE DEFAULT SET SnmpTrapDestinationsList = 'host_name port CommunityString';
```

For example:

```
=> ALTER DATABASE DEFAULT SET SnmpTrapDestinationsList = 'localhost 162 public';
```

You can also specify multiple destinations by specifying a list of destinations, separated by commas:

```
=> ALTER DATABASE DEFAULT SET SnmpTrapDestinationsList = 'host_name1 port1 CommunityString1, hostname2 port2 CommunityString2';
```

> **Note**
> : Setting multiple destinations sends any SNMP trap notification to all destinations listed.

To define which events Vertica traps

Use the following SQL command, where Event_Name is one of the events in the list below the command:

```
=> ALTER DATABASE DEFAULT SET SnmpTrapEvents = 'Event_Name1, Even_Name2';
```

- Low Disk Space
- Read Only File System
- Loss Of K Safety
- Current Fault Tolerance at Critical Level
- Too Many ROS Containers
- Node State Change
- Recovery Failure
- Recovery Error
- Recovery Lock Error
- Recovery Projection Retrieval Error
- Refresh Error
- Tuple Mover Error
- Stale Checkpoint
- CRC Mismatch

> **Note**
> The above values are case sensitive.

The following example specifies two event names:

```
=> ALTER DATABASE DEFAULT SET SnmpTrapEvents = 'Low Disk Space, Recovery Failure';
```

Verifying SNMP configuration

To create a set of test events that checks SNMP configuration:

1. Set up SNMP trap handlers to catch Vertica events.
2. Test your setup with the following command:

```
SELECT SNMP_TRAP_TEST();
   SNMP_TRAP_TEST
--------------------------
 Completed SNMP Trap Test
(1 row)
```

Event reporting examples

Vertica.log

The following example illustrates a Too Many ROS Containers event posted and cleared within vertica.log:

```
08/14/15 15:07:59 thr:nameless:0x45a08940 [INFO] Event Posted: Event Code:4 Event Id:0 Event Severity: Warning [4] PostedTimestamp:
2015-08-14 15:07:59.253729 ExpirationTimestamp: 2015-08-14 15:08:29.253729
EventCodeDescription: Too Many ROS Containers ProblemDescription:
Too many ROS containers exist on this node. DatabaseName: TESTDB
Hostname: fc6-1.example.com
08/14/15 15:08:54 thr:Ageout Events:0x2aaab0015e70 [INFO] Event Cleared:
Event Code:4 Event Id:0 Event Severity: Warning [4] PostedTimestamp:
2015-08-14 15:07:59.253729 ExpirationTimestamp: 2015-08-14 15:08:53.012669
EventCodeDescription: Too Many ROS Containers ProblemDescription:
Too many ROS containers exist on this node. DatabaseName: TESTDB
Hostname: fc6-1.example.com
```

SNMP

The following example illustrates a Too Many ROS Containers event posted to SNMP:

```
Version: 1, type: TRAPREQUESTEnterprise OID: .1.3.6.1.4.1.31207.2.0.1
Trap agent: 72.0.0.0
Generic trap: ENTERPRISESPECIFIC (6)
Specific trap: 0
.1.3.6.1.4.1.31207.1.1 ---> 4
.1.3.6.1.4.1.31207.1.2 ---> 0
.1.3.6.1.4.1.31207.1.3 ---> 2008-08-14 11:30:26.121292
.1.3.6.1.4.1.31207.1.4 ---> 4
.1.3.6.1.4.1.31207.1.5 ---> 1
.1.3.6.1.4.1.31207.1.6 ---> site01
.1.3.6.1.4.1.31207.1.7 ---> suse10-1
.1.3.6.1.4.1.31207.1.8 ---> Too many ROS containers exist on this node.
.1.3.6.1.4.1.31207.1.9 ---> QATESTDB
.1.3.6.1.4.1.31207.1.10 ---> Too Many ROS Containers
```

Syslog

The following example illustrates a Too Many ROS Containers event posted and cleared within syslog:

```
Aug 14 15:07:59 fc6-1 vertica: Event Posted: Event Code:4 Event Id:0 Event Severity: Warning [4] PostedTimestamp: 2015-08-14 15:07:59.253729
ExpirationTimestamp:
2015-08-14 15:08:29.253729 EventCodeDescription: Too Many ROS Containers ProblemDescription:
Too many ROS containers exist on this node. DatabaseName: TESTDB Hostname: fc6-1.example.com
Aug 14 15:08:54 fc6-1 vertica: Event Cleared: Event Code:4 Event Id:0 Event Severity:
Warning [4] PostedTimestamp: 2015-08-14 15:07:59.253729 ExpirationTimestamp:
2015-08-14 15:08:53.012669 EventCodeDescription: Too Many ROS Containers ProblemDescription:
Too many ROS containers exist on this node. DatabaseName: TESTDB Hostname: fc6-1.example.com
```

Using system tables

Vertica system tables provide information about system resources, background processes, workload, and performance—for example, load streams, query profiles, and tuple mover operations. Vertica collects and refreshes this information automatically.

You can query system tables using expressions, predicates, aggregates, analytics, subqueries, and joins. You can also save system table query results into a user table for future analysis. For example, the following query creates a table, mynode , selecting three node-related columns from the NODES system table:

```
=> CREATE TABLE mynode AS SELECT node_name, node_state, node_address FROM nodes;
CREATE TABLE
=> SELECT * FROM mynode;
   node_name     | node_state |  node_address
-----------------+------------+----------------
 v_vmart_node0001 | UP         | 192.168.223.11
(1 row)
```

> **Note**
> You cannot query system tables if the database cluster is in a recovering state. The database refuses connection requests and cannot be monitored. Vertica also does not support DDL and DML operations on system tables.

Where system tables reside

System tables are grouped into two schemas:

- V_CATALOG schema : Provides information about persistent objects in the catalog
- V_MONITOR schema : Provides information about transient system state

These schemas reside in the default search path. Unless you change the search path to exclude V_MONITOR or V_CATALOG or both, queries can specify a system table name that omits its schema.

You can query the SYSTEM_TABLES table for all Vertica system tables and their schemas. For example:

```
SELECT * FROM system_tables ORDER BY table_schema, table_name;
```

System table categories

Vertica system tables can be grouped into the following areas:

- System information
- System resources
- Background processes
- Workload and performance

Vertica reserves some memory to help monitor busy systems. Using simple system table queries makes it easier to troubleshoot issues. See also SYSQUERY .

Note

You can use external monitoring tools or scripts to query the system tables and act upon the information, as necessary. For example, when a host failure causes the K-safety level to fall below the desired level, the tool or script can notify the database administrator and/or appropriate IT personnel of the change, typically in the form of an e-mail.

Privileges

You can GRANT and REVOKE privileges on system tables, with the following restrictions:

- You cannot GRANT privileges on system tables to the SYSMONITOR or PSEUDOSUPERUSER roles.
- You cannot GRANT on system schemas.

Case-sensitive system table data

Some system table data might be stored in mixed case. For example, Vertica stores mixed-case identifier names the way you specify them in the CREATE statement, even though case is ignored when you reference them in queries. When these object names appear as data in the system tables, you'll encounter errors if you query them with an equality (=) operator because the case must exactly match the stored identifier. In particular, data in columns TABLE_SCHEMA and TABLE_NAME in system table TABLES are case sensitive.

If you don't know how the identifiers are stored, use the case-insensitive operator ILIKE . For example, given the following schema:

```
=> CREATE SCHEMA SS;
=> CREATE TABLE SS.TT (c1 int);
=> CREATE PROJECTION SS.TTP1 AS SELECT * FROM ss.tt UNSEGMENTED ALL NODES;
=> INSERT INTO ss.tt VALUES (1);
```

A query that uses the = operator returns 0 rows:

```
=> SELECT table_schema, table_name FROM v_catalog.tables WHERE table_schema ='ss';
table_schema | table_name
--------------+------------
(0 rows)
```

A query that uses case-insensitive ILIKE returns the expected results:

```
=> SELECT table_schema, table_name FROM v_catalog.tables WHERE table_schema ILIKE 'ss';
table_schema | table_name
--------------+------------
 SS          | TT
(1 row)
```

Examples

The following examples illustrate simple ways to use system tables in queries.

```
=> SELECT current_epoch, designed_fault_tolerance, current_fault_tolerance FROM SYSTEM;
 current_epoch | designed_fault_tolerance | current_fault_tolerance
---------------+--------------------------+-------------------------
           492 |                        1 |                       1
(1 row)

=> SELECT node_name, total_user_session_count, executed_query_count FROM query_metrics;
    node_name     | total_user_session_count | executed_query_count
------------------+--------------------------+----------------------
 v_vmart_node0001 |                      115 |                  353
 v_vmart_node0002 |                      114 |                   35
 v_vmart_node0003 |                      116 |                   34
(3 rows)

=> SELECT DISTINCT(schema_name), schema_owner FROM schemata;
 schema_name  | schema_owner
--------------+--------------
 v_catalog    | dbadmin
 v_txtindex   | dbadmin
 v_func       | dbadmin
 TOPSCHEMA    | dbadmin
 online_sales | dbadmin
 v_internal   | dbadmin
 v_monitor    | dbadmin
 structs      | dbadmin
 public       | dbadmin
 store        | dbadmin
(10 rows)
```

Data collector utility

The Data Collector collects and retains history of important system activities, and records essential performance and resource utilization counters.

Data Collector extends system table functionality with minimal overhead, performing the following tasks:

- Provides a framework for recording events.
- Propagates information to system tables.

You can query Data Collector information to obtain the past state of system tables and extract aggregate information. It can also help you:

- See what actions users have taken.
- Locate performance bottlenecks.
- Identify potential improvements to Vertica configuration.

> **Note**
> Data Collector does not collect data for nodes that are down, so no historical data is available for that node.

Data Collector works with Workload Analyzer , a tool that intelligently monitors the performance of SQL queries and workloads, and recommends tuning actions based on observations of the actual workload history.

Configuring and accessing data collector information

Data Collector retains the data it gathers according to configurable retention policies . Data Collector is on by default; you can disable it by setting set configuration parameter EnableDataCollector to 0, at the database and node levels, with ALTER DATABASE and ALTER NODE , respectively.

You can access metadata on collected data of all components through system table DATA_COLLECTOR . This table includes information about current collection policies on that component, and how much data is retained in memory and on disk.

Collected data is logged on disk in the DataCollector directory under the Vertica /catalog path. You can query logged data from component-specific Data Collector tables . You can also manage logged data with Vertica meta-functions; see Managing data collection logs for details.

In this section

- Configuring data retention policies
- Querying data collector tables
- Managing data collection logs

Configuring data retention policies

Data collector maintains retention policies for each Vertica component that it monitors—for example, TupleMoverEvents, or DepotEvictions. You can identify monitored components by querying system table DATA_COLLECTOR . For example, the following query returns partition activity components:

```
=> SELECT DISTINCT component FROM data_collector WHERE component ILIKE '%partition%';
     component
----------------------
 HiveCustomPartitions
 CopyPartitions
 MovePartitions
 SwapPartitions
(4 rows)
```

Each component has its own retention policy, which is comprised of several properties:

- MEMORY_BUFFER_SIZE_KB specifies in kilobytes the maximum amount of collected data that the Data Collector buffers in memory before moving it to disk.
- DISK_SIZE_KB specifies in kilobytes the maximum disk space allocated for this component's Data Collector table.
- INTERVAL_TIME is an INTERVAL data type that specifies how long data of a given component is retained in that component's Data Collector table.

Vertica sets default values on all properties, which you can modify with meta-functions SET_DATA_COLLECTOR_POLICY and SET_DATA_COLLECTOR_TIME_POLICY .

You can view retention policy settings by calling GET_DATA_COLLECTOR_POLICY . For example, the following statement returns the retention policy for the TupleMoverEvents component:

```
=> SELECT get_data_collector_policy('TupleMoverEvents');
                        get_data_collector_policy
-------------------------------------------------------------------------------
 1000KB kept in memory, 15000KB kept on disk. Time based retention disabled.
(1 row)
```

Setting retention memory and disk storage

Retention policy properties MEMORY_BUFFER_SIZE_KB and DISK_SIZE_KB combine to determine how much collected data is available at any given time. The two properties have the following dependencies: if MEMORY_BUFFER_SIZE_KB is set to 0, the Data Collector does not retain any data for this component either in memory or on disk; and if DISK_SIZE_KB is set to 0, then the Data Collector retains only as much component data as it can buffer, as set by MEMORY_BUFFER_SIZE_KB .

For example, the following statement changes memory and disk setting for component ResourceAcquisitions from its current setting of 1,000 KB memory and 10,000 KB disk space to 1500 KB and 25000 KB, respectively:

```
=> SELECT set_data_collector_policy('ResourceAcquisitions', '1500', '25000');
 set_data_collector_policy
---------------------------
 SET
(1 row)
```

You should consider setting MEMORY_BUFFER_SIZE_KB to a high value in the following cases:

- Unusually high levels of data collection. If MEMORY_BUFFER_SIZE_KB is set too low, the Data Collector might be unable to flush buffered data to disk fast enough to keep up with the activity level, which can lead to loss of in-memory data.
- Very large data collector records—for example, records with very long query strings. The Data Collector uses double-buffering, so it cannot retain in memory records that are more than 50 percent larger than MEMORY_BUFFER_SIZE_KB.

Setting time-based retention

By default, all collected data of a given component remain on disk and are accessible in the component's Data Collector table, up to the disk storage limit of that component's retention policy as set by its DISK_SIZE_KB property. You can call SET_DATA_COLLECTOR_POLICY to limit how long data is retained in a component's Data Collector table. In the following example, SET_DATA_COLLECTOR_POLICY is called on component TupleMoverEvents and sets its INTERVAL_TIME property to an interval of 30 minutes:

```
SELECT set_data_collector_policy('TupleMoverEvents ', '30 minutes'::interval);
set_data_collector_time_policy
--------------------------------
SET
(1 row)
```

After this call, the Data Collector table dc_tuple_mover_events only retains records of Tuple Mover activity that occurred in the last 30 minutes. Older Tuple Mover data are automatically dropped from this table. For example, after the previous call to SET_DATA_COLLECTOR_POLICY, querying dc_tuple_mover_events returns data of Tuple Mover activity that was collected over the last 30 minutes—in this case, since 07:58:21:

```
=> SELECT current_timestamp(0)  - '30 minutes'::interval AS '30 minutes ago';
   30 minutes ago
---------------------
 2020-08-13 07:58:21
(1 row)

=> SELECT time, node_name, session_id, user_name, transaction_id, operation FROM dc_tuple_mover_events WHERE node_name='v_vmart_node0001'
ORDER BY transaction_id;
            time             |    node_name     |           session_id            | user_name |  transaction_id   | operation
-------------------------------+------------------+---------------------------------+-----------+-------------------+-----------
 2020-08-13 08:16:54.360597-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807826 | Mergeout
 2020-08-13 08:16:54.397346-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807826 | Mergeout
 2020-08-13 08:16:54.424002-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807826 | Mergeout
 2020-08-13 08:16:54.425989-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807829 | Mergeout
 2020-08-13 08:16:54.456829-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807829 | Mergeout
 2020-08-13 08:16:54.485097-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807829 | Mergeout
 2020-08-13 08:19:45.8045-04   | v_vmart_node0001 | v_vmart_node0001-190508:0x37b08 | dbadmin   | 45035996273807855 | Mergeout
 2020-08-13 08:19:45.742-04    | v_vmart_node0001 | v_vmart_node0001-190508:0x37b08 | dbadmin   | 45035996273807855 | Mergeout
 2020-08-13 08:19:45.684764-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x37b08 | dbadmin   | 45035996273807855 | Mergeout
 2020-08-13 08:19:45.799796-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807865 | Mergeout
 2020-08-13 08:19:45.768856-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807865 | Mergeout
 2020-08-13 08:19:45.715424-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807865 | Mergeout
 2020-08-13 08:25:20.465604-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807890 | Mergeout
 2020-08-13 08:25:20.497266-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807890 | Mergeout
 2020-08-13 08:25:20.518839-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807890 | Mergeout
 2020-08-13 08:25:20.52099-04  | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807893 | Mergeout
 2020-08-13 08:25:20.549075-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807893 | Mergeout
 2020-08-13 08:25:20.569072-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807893 | Mergeout
(18 rows)
```

After 25 minutes elapse, 12 of these records age out of the 30 minute interval set for TupleMoverEvents., and are dropped from dc_tuple_mover_events :

```
=> SELECT current_timestamp(0)  - '30 minutes'::interval AS '30 minutes ago';
   30 minutes ago
---------------------
 2020-08-13 08:23:33
(1 row)


=> SELECT time, node_name, session_id, user_name, transaction_id, operation FROM dc_tuple_mover_events WHERE node_name='v_vmart_node0001'
ORDER BY transaction_id;
             time              |    node_name     |           session_id            | user_name |  transaction_id   | operation
-------------------------------+------------------+---------------------------------+-----------+-------------------+-----------
 2020-08-13 08:25:20.465604-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807890 | Mergeout
 2020-08-13 08:25:20.497266-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807890 | Mergeout
 2020-08-13 08:25:20.518839-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807890 | Mergeout
 2020-08-13 08:25:20.52099-04  | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807893 | Mergeout
 2020-08-13 08:25:20.549075-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807893 | Mergeout
 2020-08-13 08:25:20.569072-04 | v_vmart_node0001 | v_vmart_node0001-190508:0x375db | dbadmin   | 45035996273807893 | Mergeout
(6 rows)
```

> **Note**
> Setting a component policy's INTERVAL_TIME property has no effect on how much data storage the Data Collector retains on disk for that component. Maximum disk storage capacity is determined by the DISK_SIZE_KB property. Setting the INTERVAL_TIME property only affects how long data is retained by the component's Data Collector table.

The meta-function SET_DATA_COLLECTOR_TIME_POLICY also sets a retention policy's INTERVAL_TIME property. Unlike SET_DATA_COLLECTOR_POLICY, this meta-function only sets the INTERVAL_TIME property . It also differs in that you can use this meta-function to update INTERVAL_TIME on all components, by omitting the component argument. For example:

```
SELECT set_data_collector_time_policy('1 day'::interval);
set_data_collector_time_policy
--------------------------------
SET
(1 row)

=> SELECT DISTINCT component, INTERVAL_SET, INTERVAL_TIME FROM DATA_COLLECTOR WHERE component ILIKE '%partition%';
     component       | INTERVAL_SET | INTERVAL_TIME
---------------------+--------------+---------------
 HiveCustomPartitions | t            | 1
 MovePartitions       | t            | 1
 CopyPartitions       | t            | 1
 SwapPartitions       | t            | 1
(4 rows)
```

To clear the INTERVAL_TIME policy property, call SET_DATA_COLLECTOR_TIME_POLICY with a negative integer argument. For example:

```
=> SELECT set_data_collector_time_policy('-1');
 set_data_collector_time_policy
--------------------------------
 SET
(1 row)

=> SELECT DISTINCT component, INTERVAL_SET, INTERVAL_TIME FROM DATA_COLLECTOR WHERE component ILIKE '%partition%';
     component       | INTERVAL_SET | INTERVAL_TIME
---------------------+--------------+---------------
 MovePartitions       | f            | 0
 SwapPartitions       | f            | 0
 HiveCustomPartitions | f            | 0
 CopyPartitions       | f            | 0
(4 rows)
```

Note
Setting INTERVAL_TIME on a retention policy also sets its BOOLEAN property INTERVAL_SET.

Querying data collector tables

Caution
Data Collector tables (prefixed by dc_) are in the V_INTERNAL schema. If you use Data Collector tables in scripts or monitoring tools, be aware that any Vertica upgrade is liable to remove or change them without notice.

You can obtain component-specific data from Data Collector tables. The Data Collector compiles the component data from its log files in a table format that you can query with standard SQL queries. You can identify Data Collector table names for specific components through system table Data Collector. For example:

```
=> SELECT distinct component, table_name FROM data_collector where component ILIKE 'lock%';
  component   |    table_name
--------------+------------------
 LockRequests | dc_lock_requests
 LockReleases | dc_lock_releases
 LockAttempts | dc_lock_attempts
(3 rows)
```

You can then query the desired Data Collector tables—for example, check for lock delays in dc_lock_attempts :

```
=> SELECT * from dc_lock_attempts WHERE description != 'Granted immediately';
-[ RECORD 1 ]------+----------------------------
time               | 2020-08-17 00:14:07.187607-04
node_name          | v_vmart_node0001
session_id         | v_vmart_node0001-319647:0x1d
user_id            | 45035996273704962
user_name          | dbadmin
transaction_id     | 45035996273819050
object             | 0
object_name        | Global Catalog
mode               | X
promoted_mode      | X
scope              | TRANSACTION
start_time         | 2020-08-17 00:14:07.184663-04
timeout_in_seconds | 300
result             | granted
description        | Granted after waiting
```

Managing data collection logs

On startup, Vertica creates a DataCollector directory under the database catalog directory of each node. This directory contains one or more logs for individual components. For example:

```
[dbadmin@doch01 DataCollector]$ pwd
/home/dbadmin/VMart/v_vmart_node0001_catalog/DataCollector
[dbadmin@doch01 DataCollector]$ ls -1 -g Lock*
-rw------- 1 verticadba 2559879 Aug 17 00:14 LockAttempts_650572441057355.log
-rw------- 1 verticadba  614579 Aug 17 05:28 LockAttempts_650952885486175.log
-rw------- 1 verticadba 2559895 Aug 14 18:31 LockReleases_650306482037650.log
-rw------- 1 verticadba 1411127 Aug 17 05:28 LockReleases_650759468041873.log
```

The DataCollector directory also contains a pair of SQL template files for each component:

- CREATE_ *component* _TABLE.sql provides DDL for creating a table where you can load Data Collector logs for a given component—for example, LockAttempts:

```
[dbadmin@doch01 DataCollector]$ cat CREATE_LockAttempts_TABLE.sql
\set dcschema 'echo ${DCSCHEMA:-dc}'
CREATE TABLE :dcschema.dc_lock_attempts(
  "time" TIMESTAMP WITH TIME ZONE,
  "node_name" VARCHAR(128),
  "session_id" VARCHAR(128),
  "user_id" INTEGER,
  "user_name" VARCHAR(128),
  "transaction_id" INTEGER,
  "object" INTEGER,
  "object_name" VARCHAR(128),
  "mode" VARCHAR(128),
  "promoted_mode" VARCHAR(128),
  "scope" VARCHAR(128),
  "start_time" TIMESTAMP WITH TIME ZONE,
  "timeout_in_seconds" INTEGER,
  "result" VARCHAR(128),
  "description" VARCHAR(64000)
);
```

- COPY_ *component* _TABLE.sql contains SQL for loading (with COPY) the data log files into the table that the CREATE script creates. For example:

```
[dbadmin@doch01 DataCollector]$ cat COPY_LockAttempts_TABLE.sql
\set dcpath 'echo ${DCPATH:-$PWD}'
\set dcschema 'echo ${DCSCHEMA:-dc}'
\set logfiles ""':dcpath'/LockAttempts_*.log'"
COPY :dcschema.dc_lock_attempts(
  LockAttempts_start_filler FILLER VARCHAR(64) DELIMITER E'\n',
  "time_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "time" FORMAT '_internal' DELIMITER E'\n',
  "node_name_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "node_name" ESCAPE E'\001' DELIMITER E'\n',
  "session_id_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "session_id" ESCAPE E'\001' DELIMITER E'\n',
  "user_id_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "user_id" FORMAT 'd' DELIMITER E'\n',
  "user_name_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "user_name" ESCAPE E'\001' DELIMITER E'\n',
  "transaction_id_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "transaction_id" FORMAT 'd' DELIMITER E'\n',
  "object_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "object" FORMAT 'd' DELIMITER E'\n',
  "object_name_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "object_name" ESCAPE E'\001' DELIMITER E'\n',
  "mode_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "mode" ESCAPE E'\001' DELIMITER E'\n',
  "promoted_mode_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "promoted_mode" ESCAPE E'\001' DELIMITER E'\n',
  "scope_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "scope" ESCAPE E'\001' DELIMITER E'\n',
  "start_time_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "start_time" FORMAT '_internal' DELIMITER E'\n',
  "timeout_in_seconds_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "timeout_in_seconds" FORMAT 'd' DELIMITER E'\n',
  "result_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "result" ESCAPE E'\001' DELIMITER E'\n',
  "description_nfiller" FILLER VARCHAR(32) DELIMITER ':',
  "description" ESCAPE E'\001'
)  FROM :logfiles RECORD TERMINATOR E'\n.\n' DELIMITER E'\n';
```

Log management meta-functions

You can manage Data Collector logs with Vertica meta-functions FLUSH_DATA_COLLECTOR and CLEAR_DATA_COLLECTOR . Both functions can specify a single component, or execute on all components:

- FLUSH_DATA_COLLECTOR waits until memory logs are moved to disk and then flushes the Data Collector, synchronizing the log with disk storage. For example, the following statement executes on all components:

```
=> SELECT flush_data_collector();
 flush_data_collector
----------------------
 FLUSH
(1 row)
```

- CLEAR_DATA_COLLECTOR clears all memory and disk records from Data Collector tables and logs, and resets collection statistics in system table DATA_COLLECTOR . For example, the following statement executes on data collected for component ResourceAcquisitions:

```
=> SELECT clear_data_collector('ResourceAcquisitions');
 clear_data_collector
----------------------
 CLEAR
(1 row)
```

Monitoring partition reorganization

When you use ALTER TABLE ...REORGANIZE , the operation reorganizes the data in the background.

You can monitor details of the reorganization process by polling the following system tables:

- V_MONITOR.PARTITION_STATUS displays the fraction of each table that is partitioned correctly.
- V_MONITOR.PARTITION_REORGANIZE_ERRORS logs errors issued by the reorganize process.
- V_MONITOR.PARTITIONS displays NULL in the partition_key column for any ROS that was not reorganized.

> **Note**
> The corresponding foreground process to ALTER TABLE...REORGANIZE is PARTITION_TABLE .

Monitoring resource pools

You can use the following to find information about resource pools:

- RESOURCE_POOL_STATUS returns current data from resource pools—for example, current memory usage, resources requested and acquired by various requests, and state of the queues.
- RESOURCE_ACQUISITIONS displays all resources granted to the queries that are currently running.
- SHOW SESSION shows, along with other session-level parameters, the current session's resource pool .

You can also use the Management Console to obtain run-time data on resource pool usage .

> **Note**
> The Linux top command returns data on overall CPU usage and I/O wait time across the system. Because of file system caching, the resident memory size returned by top is not the best indicator of actual memory use or available reserves.

Viewing resource pool status

The following example queries RESOURCE_POOL_STATUS for memory size data:

```
=> SELECT pool_name poolName,
    node_name nodeName,
    max_query_memory_size_kb maxQueryMemSizeKb,
    max_memory_size_kb maxMemSizeKb,
    memory_size_actual_kb memSizeActualKb
    FROM resource_pool_status WHERE pool_name='ceo_pool';
 poolName |     nodeName     | maxQueryMemSizeKb | maxMemSizeKb | memSizeActualKb
----------+------------------+-------------------+--------------+-----------------
 ceo_pool | v_vmart_node0001 |          12179388 |     13532654 |         1843200
 ceo_pool | v_vmart_node0002 |          12191191 |     13545768 |         1843200
 ceo_pool | v_vmart_node0003 |          12191170 |     13545745 |         1843200
(3 rows)
```

Viewing query resource acquisitions

The following example displays all resources granted to the queries that are currently running. The information shown is stored in system table RESOURCE_ACQUISITIONS table. You can see that the query execution used 708504 KB of memory from the GENERAL pool.

```
=> SELECT pool_name, thread_count, open_file_handle_count, memory_inuse_kb,
   queue_entry_timestamp, acquisition_timestamp
   FROM V_MONITOR.RESOURCE_ACQUISITIONS WHERE node_name ILIKE '%node0001';

-[ RECORD 1 ]----------+-------------------------------
pool_name              | sysquery
thread_count           | 4
open_file_handle_count | 0
memory_inuse_kb        | 4103
queue_entry_timestamp  | 2013-12-05 07:07:08.815362-05
acquisition_timestamp  | 2013-12-05 07:07:08.815367-05
-[ RECORD 2 ]----------+-------------------------------
...
-[ RECORD 8 ]----------+-------------------------------
pool_name              | general
thread_count           | 12
open_file_handle_count | 18
memory_inuse_kb        | 708504
queue_entry_timestamp  | 2013-12-04 12:55:38.566614-05
acquisition_timestamp  | 2013-12-04 12:55:38.566623-05
-[ RECORD 9 ]----------+-------------------------------
...
```

You can determine how long a query waits in the queue before it can run. To do so, you obtain the difference between acquisition_timestamp and queue_entry_timestamp using a query as this example shows:

```
=> SELECT pool_name, queue_entry_timestamp, acquisition_timestamp,
    (acquisition_timestamp-queue_entry_timestamp) AS 'queue wait'
   FROM V_MONITOR.RESOURCE_ACQUISITIONS WHERE node_name ILIKE '%node0001';

-[ RECORD 1 ]---------+-------------------------------
pool_name             | sysquery
queue_entry_timestamp | 2013-12-05 07:07:08.815362-05
acquisition_timestamp | 2013-12-05 07:07:08.815367-05
queue wait            | 00:00:00.000005
-[ RECORD 2 ]---------+-------------------------------
pool_name             | sysquery
queue_entry_timestamp | 2013-12-05 07:07:14.714412-05
acquisition_timestamp | 2013-12-05 07:07:14.714417-05
queue wait            | 00:00:00.000005
-[ RECORD 3 ]---------+-------------------------------
pool_name             | sysquery
queue_entry_timestamp | 2013-12-05 07:09:57.238521-05
acquisition_timestamp | 2013-12-05 07:09:57.281708-05
queue wait            | 00:00:00.043187
-[ RECORD 4 ]---------+-------------------------------
...
```

Querying user-defined resource pools

The Boolean column IS_INTERNAL in system tables RESOURCE_POOLS and RESOURCE_POOL_STATUS lets you get data on user-defined resource pools only. For example:

```
SELECT name, subcluster_oid, subcluster_name, memorysize, maxmemorysize, priority, maxconcurrency
dbadmin->     FROM V_CATALOG.RESOURCE_POOLS where is_internal ='f';
   name       |  subcluster_oid   | subcluster_name | memorysize | maxmemorysize | priority | maxconcurrency
--------------+-------------------+-----------------+------------+---------------+----------+----------------
 load_pool    | 72947297254957395 | default         | 0%         |               |       10 |
 ceo_pool     | 63570532589529860 | c_subcluster    | 250M       |               |       10 |
 ad hoc_pool  |                 0 |                 | 200M       | 200M          |        0 |
 billing_pool | 45579723408647896 | ar_subcluster   | 0%         |               |        0 |              3
 web_pool     |                 0 | analytics_1     | 25M        |               |       10 |              5
 batch_pool   | 47479274633682648 | default         | 150M       | 150M          |        0 |             10
 dept1_pool   |                 0 |                 | 0%         |               |        5 |
 dept2_pool   |                 0 |                 | 0%         |               |        8 |
 dashboard    | 45035996273843504 | analytics_1     | 0%         |               |        0 |
(9 rows)
```

Monitoring recovery

When your Vertica database is recovering from a failure, it's important to monitor the recovery process. There are several ways to monitor database recovery:

In this section

- Viewing log files on each node
- Using system tables to monitor recovery
- Viewing cluster state and recovery status
- Monitoring cluster status after recovery

Viewing log files on each node

During database recovery, Vertica adds logging information to the vertica.log on each host. Each message is identified with a [Recover] string.

Use the tail command to monitor recovery progress by viewing the relevant status messages, as follows.

```
$ tail -f catalog-path/database-name/node-name_catalog/vertica.log
01/23/08 10:35:31 thr:Recover:0x2a98700970 [Recover] <INFO> Changing host v_vmart_node0001 startup state from INITIALIZING to RECOVERING
01/23/08 10:35:31 thr:CatchUp:0x1724b80 [Recover] <INFO> Recovering to specified epoch 0x120b6
01/23/08 10:35:31 thr:CatchUp:0x1724b80 [Recover] <INFO> Running 1 split queries
01/23/08 10:35:31 thr:CatchUp:0x1724b80 [Recover] <INFO> Running query: ALTER PROJECTION proj_tradesquotes_0 SPLIT v_vmart_node0001 FROM 73911;
```

Using system tables to monitor recovery

Use the following system tables to monitor recover:

- RECOVERY_STATUS
- PROJECTION_RECOVERIES

Specifically, the recovery_status system table includes information about the node that is recovering, the epoch being recovered, the current recovery phase, and running status:

```
=>select node_name, recover_epoch, recovery_phase, current_completed, is_running from recovery_status;
node_name           | recover_epoch | recovery_phase    | current_completed | is_running
--------------------+---------------+-------------------+-------------------+------------
 v_vmart_node0001   |               |                   | 0                 | f
 v_vmart_node0002   | 0             | historical pass 1 | 0                 | t
 v_vmart_node0003   | 1             | current           | 0                 | f
```

The projection_recoveries system table maintains history of projection recoveries. To check the recovery status, you can summarize the data for the recovering node, and run the same query several times to see if the counts change. Differing counts indicate that the recovery is working and in the process of recovering all missing data.

```
=> select node_name, status , progress from projection_recoveries;
node_name             | status     | progress
----------------------+------------+---------
v_vmart_node0001      | running    | 61
```

To see a single record from the projection_recoveries system table, add limit 1 to the query.

After a recovery has completed, Vertica continues to store information from the most recent recovery in these tables.

Viewing cluster state and recovery status

Use the admintools view_cluster tool from the command line to see the cluster state:

```
$ /opt/vertica/bin/admintools -t view_cluster
DB | Host | State
---------+--------------+------------
<data_base> | 112.17.31.10 | RECOVERING
<data_base> | 112.17.31.11 | UP
<data_base> | 112.17.31.12 | UP
<data_base> | 112.17.31.17 | UP
_____
```

Monitoring cluster status after recovery

When recovery has completed:

1. Launch Administration Tools.
2. From the Main Menu, select **View Database Cluster** State and click **OK.**
   The utility reports your node's status as UP .

> **Note**
> You can also monitor the state of your database nodes on the Management Console Overview page under the Database section, which tells you the number of nodes that are up, critical, recovering, or down. To get node-specific information, click Manage at the bottom of the page.

Clearing projection refresh history

System table PROJECTION_REFRESHES records information about refresh operations , successful and unsuccessful. PROJECTION_REFRESHES retains projection refresh data until one of the following events occurs:

- Another refresh operation starts on a given projection.
- CLEAR_PROJECTION_REFRESHES is called and clears data on all projections.
- The table's storage quota is exceeded.

To immediately purge this information, call CLEAR_PROJECTION_REFRESHES :

```
=> SELECT clear_projection_refreshes();
clear_projection_refreshes
---------------------------
 CLEAR
(1 row)
```

> **Note**
> PROJECTION_REFRESHES checks the Boolean column IS_EXECUTING in PROJECTION_REFRESHES to determine whether refresh operations are still running or are complete. The function only removes information for refresh operations that are complete.

See also

PROJECTION_REFRESHES

Monitoring Vertica using notifiers

A Vertica notifier is a push-based mechanism for sending messages from Vertica to endpoints like Apache Kafka or syslog. For example, you can configure a long-running script to send notifications at various stages and then at the completion of a task.

To use a notifier:

1. Use CREATE NOTIFIER to create one of the following notifier types:
   - Syslog
   - SNS
   - Kafka
2. Send a notification to the NOTIFIER endpoint with any of the following:
   - NOTIFY : Manually sends a message to the NOTIFIER endpoint.
   - SET_DATA_COLLECTOR_NOTIFY_POLICY : Creates a notification policy, which automatically sends a message to the NOTIFIER endpoint when a specified event occurs.

Backing up and restoring the database

Important
Inadequate security on backups can compromise overall database security. Be sure to secure backup locations and strictly limit access to backups only to users who already have permissions to access all database data.

Creating regular database backups is an important part of basic maintenance tasks. Vertica supplies a comprehensive utility, vbr , for this purpose. vbr lets you perform the following operations. Unless otherwise noted, operations are supported in both Enterprise Mode and Eon Mode:

- Back up a database.
- Back up specific objects (schemas or tables) in a database.
- Restore a database or individual objects from backup.
- Copy a database to another cluster. For example, to promote a test cluster to production (Enterprise Mode only).
- Replicate individual objects (schemas or tables) to another cluster.
- List available backups.

When you run vbr , you specify a configuration (.ini) file. In this file you specify all of the configuration parameters for the operation: what to back up, where to back it up, how many backups to keep, whether to encrypt transmissions, and much more. Vertica provides several Sample vbr configuration files that you can use as templates.

You can use vbr to restore a backup created by vbr . Typically, you use the same configuration file for both operations. Common use cases introduces the most common vbr operations.

When performing a backup, you can save your data to one of the following locations:

- Local directory on each node
- Remote file system
- Different Vertica cluster (effectively cloning your database)
- Cloud storage

You cannot back up an Enterprise Mode database and restore it in Eon Mode, or vice versa.

Supported cloud storage

Vertica supports backup and restore operations in the following cloud storage locations:

- Amazon Web Services (AWS) S3
- S3-compatible private cloud storage, such as Pure Storage or Minio
- Google Cloud Storage (GCS)
- Azure Blob Storage

If you are backing up an Eon Mode database, you must use a supported cloud storage location.

You cannot perform backup or restore operations between different cloud providers. For example, you cannot back up or restore from GCS to an S3 location.

Additional considerations for HDFS storage locations

If your database has any storage locations on HDFS, additional configuration is required to enable those storage locations for backup operations. See Requirements for backing up and restoring HDFS storage locations .

In this section

- Common use cases
- Sample vbr configuration files
- Eon Mode database requirements
- Requirements for backing up and restoring HDFS storage locations
- Setting up backup locations
- Creating backups
- Restoring backups
- Copying the database to another cluster
- Replicating objects to another database cluster
- Including and excluding objects
- Managing backups
- Troubleshooting backup and restore
- vbr reference
- vbr configuration file reference

Common use cases

You can use vbr to perform many tasks related to backup and restore. The vbr reference describes all of the tasks in detail. This section summarizes common use cases. For each of these cases, there are additional requirements not covered here. Be sure to read the linked topics for details.

This is not a complete list of Backup/Restore capabilities.

Routine backups in Enterprise Mode

A full backup stores a copy of your data in another location—ideally a location that is separated from your database location, such as on different hardware or in the cloud. You give the backup a name (the snapshot name), which allows you to have different backups and backup types without interference. In your configuration file, you can map database nodes to backup locations and set some other parameters.

Before your first backup, run the vbr init task .

Use the vbr backup task to perform a full backup. The External full backup/restore example provides a starting point for your configuration. For complete documentation of full backups, see Creating full backups .

Routine backups in Eon Mode

For the most part, backups in Eon Mode work the same way as backups in Enterprise Mode. Eon Mode has some additional requirements described in Eon Mode database requirements , and some configuration parameters are different for backups to cloud storage. You can back up or restore Eon Mode databases that run in the cloud or on-premises using a supported cloud storage location.

Use the vbr backup task to perform a full backup. The Backup/restore to cloud storage example provides a starting point for your configuration. For complete documentation of full backups, see Creating full backups .

Checkpoint backups: backing up before a major operation

It is a good idea to back up your database before performing destructive operations such as dropping tables, or before major operations such as upgrading Vertica to a new version.

You can perform a regular full backup for this purpose, but a faster way is to create a hard-link local backup. This kind of backup copies your catalog and links your data files to another location on the local file system on each node. (You can also do a hard-link backup of specific objects rather than the whole database.) A hard-link local backup does not provide the same protection as a backup stored externally. For example, it does not protect you from local system failures. However, for a backup that you expect to need only temporarily, a hard-link local backup is an expedient option. Do not use hard-link local backups as substitutes for regular backups to other nodes.

Hard-link backups use the same vbr backup task as other backups, but with a different configuration. The Full hard-link backup/restore example provides a starting point for your configuration. See Creating hard-link local backups for more information.

Restoring selected objects

Sometimes you need to restore specific objects, such as a table you dropped, rather than the entire database. You can restore individual tables or schemas from any backup that contains them, whether a full backup or an object backup.

Use the vbr restore task and the --restore-objects parameter to specify what to restore. Usually you use the same configuration file that you used to create the backup. See Restoring individual objects for more information.

Restoring an entire database

You can restore both Enterprise Mode and Eon Mode databases from complete backups. You cannot use restore to change the mode of your database. In Eon Mode, you can restore to the primary subcluster without regard to secondary subclusters.

Use the vbr restore task to restore a database. As when restoring selected objects, you usually use the same configuration file that you used to create the backup. See Restoring a database from a full backup and Restoring hard-link local backups for more information.

Copying a cluster

You might need to copy a database to another cluster of computers, such as when you are promoting a database from a staging environment to production. Copying a database to another cluster is essentially a simultaneous backup and restore operation. The data is backed up from the source database cluster and restored to the destination cluster in a single operation.

Use the vbr copycluster task to copy a cluster. The Database copy to an alternate cluster example provides a starting point for your configuration. See Copying the database to another cluster for more information.

Replicating selected objects to another database

You might want to replicate specific tables or schemas from one database to another. For example, you might do this to copy data from a production database to a test database to investigate a problem in isolation. Another example is when you complete a large data load in one database, replication to another database might be more efficient than repeating the load operation in the other database.

Use the vbr replicate task to replicate objects. You specify the objects to replicate in the configuration file. The Object replication to an alternate database example provides a starting point for your configuration. See Replicating objects to another database cluster for more information.

Sample vbr configuration files

The vbr utility uses configuration files to provide the information it needs to back up and restore a full or object-level backup or copy a cluster. No default configuration file exists. You must always specify a configuration file with the vbr command.

Vertica includes sample configuration files that you can copy, edit, and deploy for various vbr tasks. Vertica automatically installs these files at:

/opt/vertica/share/vbr/example_configs

In this section

- External full backup/restore
- Backup/restore to cloud storage
- Full hard-link backup/restore
- Full local backup/restore
- Object-level local backup/restore in Enterprise Mode
- Restore object from backup to an alternate cluster
- Object replication to an alternate database
- Database copy to an alternate cluster
- Password file

External full backup/restore

backup_restore_full_external.ini

An external (distributed) backup backs up each database node to a distinct backup host. Nodes are mapped to hosts in the [Mapping] section.

To restore, use the same configuration file that you used to create the backup.

```
; This sample vbr configuration file shows full or object backup and restore to a separate remote backup-host for each respective database host.
; Section headings are enclosed by square brackets.
; Comments have leading semicolons (;) or pound signs (#).
; An equal sign separates options and values.
; Specify arguments marked '!!Mandatory!!' explicitly.
; All commented parameters are set to their default value.

; ------------------------------------------- ;
;;; BASIC PARAMETERS ;;;
; ------------------------------------------- ;

[Mapping]
; !!Mandatory!! This section defines what host and directory will store the backup for each node.
; node_name = backup_host:backup_dir
; In this "parallel backup" configuration, each node backs up to a distinct external host.
; To backup all database nodes to a single external host, use that single hostname/IP address in each entry below.
v_exampledb_node0001 = 10.20.100.156:/home/dbadmin/backups
v_exampledb_node0002 = 10.20.100.157:/home/dbadmin/backups
v_exampledb_node0003 = 10.20.100.158:/home/dbadmin/backups
v_exampledb_node0004 = 10.20.100.159:/home/dbadmin/backups

[Misc]
; !!Recommended!! Snapshot name.  Object and full backups should always have different snapshot names.
; Backups with the same snapshotName form a time sequence limited by restorePointLimit.
; SnapshotName is used for naming archives in the backup directory, and for monitoring and troubleshooting.
; Valid characters: a-z A-Z 0-9 - _
; snapshotName = backup_snapshot

[Database]
; !!Recommended!! If you have more than one database defined on this Vertica cluster, use this parameter to specify which database to backup/restore.
; dbName = current_database

; If this parameter is True, vbr prompts the user for the database password every time.
; If False, specify the location of password config file in 'passwordFile' parameter in [Misc] section.
; dbPromptForPassword = True

; If true, vbr attempts to connect to the database using a local connection.
; dbUseLocalConnection = False

; ------------------------------------------- ;
;;; ADVANCED PARAMETERS ;;;
; ------------------------------------------- ;

[Misc]
; The temp directory location on all database hosts.
; The directory must be readable and writeable by the dbadmin, and must implement POSIX style fcntl lockf locking.
; tempDir = /tmp/vbr

; Specifies the number of historical backups to retain in addition to the most recent backup.
; 1 current + n historical backups
; restorePointLimit = 1

; Full path to the password configuration file
; Store this file in directory readable only by the dbadmin
; (no default)
```

```
; (no default)
; passwordFile = /path/to/vbr/pw.txt

; When enabled, Vertica confirms that the specified backup locations contain
; sufficient free space and inodes to allow a successful backup. If a backup
; location has insufficient resources, Vertica displays an error message explaining the shortage and
; cancels the backup. If Vertica cannot determine the amount of available space
; or number of inodes in the backupDir, it displays a warning and continues
; with the backup.
; enableFreeSpaceCheck = True

[Transmission]
; Specifies the default port number for the rsync protocol.
; port_rsync = 50000

; Total bandwidth limit for all backup connections in KBPS, 0 for unlimited. Vertica distributes
; this bandwidth evenly among the number of connections set in concurrency_backup.
; total_bwlimit_backup = 0

; The maximum number of backup TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_backup = 1

; The total bandwidth limit for all restore connections in KBPS, 0 for unlimited
; total_bwlimit_restore = 0

; The maximum number of restore TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_restore = 1

; The maximum number of delete TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_delete = 16

[Database]
; Vertica user name for vbr to connect to the database.
; This setting is rarely needed since dbUser is normally identical to the database administrator
; dbUser = current_username
```

Backup/restore to cloud storage

backup_restore_cloud_storage.ini

You can backup and restore Enterprise Mode and Eon Mode databases to a cloud storage location. You must back up Eon Mode databases to a supported cloud storage location. Configuration settings in the [CloudStorage] section are identical for both Enterprise Mode and Eon Mode.

There are one-time configurations that you must complete before your first backup to a new cloud storage location. See Additional considerations for cloud storage for more information.

Backups to on-premises cloud storage destinations require additional configuration for both Enterprise Mode and Eon databases. For details about the additional requirements, see Configuring cloud storage backups .

To restore, use the same configuration file that you used to create the backup. To restore selected objects rather than the entire database, specify the objects to restore on the vbr command line using --restore-objects .

```
; This sample vbr configuration file shows backup to Cloud Storage e.g AWS S3, GCS, HDFS or on-premises (e.g. Pure Storage)
; This can be used for Vertica databases in Enterprise or Eon mode.
; Section headings are enclosed by square brackets.
; Comments have leading semicolons (;) or pound signs (#).
; Option and values are separated by an equal sign.
; Only arguments marked as '!!Mandatory!!' must be specified explicitly.
```

```
; All commented parameters are set to their default value.

; ------------------------------------------- ;
;;; BASIC PARAMETERS ;;;
; ------------------------------------------- ;

[CloudStorage]
; This section replaces the [Mapping] section and is required to back up to cloud storage.

; !!Mandatory!! Backup location on Cloud or HDFS (no default).
cloud_storage_backup_path = gs://backup_bucket/database_backup_path/
; cloud_storage_backup_path = s3://backup_bucket/database_backup_path/
; cloud_storage_backup_path = webhdfs://backup_nameservice/database_backup_path/
; cloud_storage_backup_path = azb://backup_account/backup_container/

; !!Mandatory!! directory used to manage locking during a backup (no default).  If the directory is mounted on the initiator host, you
; should use "[]" instead of the local host name.  The file system must support POSIX fcntl flock.
cloud_storage_backup_file_system_path = []:/home/dbadmin/backup_locks_dir/

[Misc]
; !!Recommended!! Snapshot name
; Backups with the same snapshotName form a time sequence limited by restorePointLimit.
; SnapshotName is used for naming archives in the backup directory, and for monitoring and troubleshooting.
; Valid values: a-z A-Z 0-9 - _
; snapshotName = backup_snapshot

; Specifies how Vertica handles objects of the same name when restoring schema or table backups.
; objectRestoreMode = createOrReplace

; Specifies which tables and/or schemas to copy. For tables, the containing schema defaults to public.
; Note: 'objects' is incompatible with 'includeObjects' and 'excludeObjects'.
; (no default)
; objects = mytable, myschema, myothertable

; Specifies the set of objects to backup/restore; wildcards may be used.
; Note: 'includeObjects' is incompatible with 'objects'.
; includeObjects = public.mytable, customer*, s?

; Subtracts from the set of objects to backup/restore; wildcards may be used
; Note: 'excludeObjects' is incompatible with 'objects'.
; excludeObjects = public.*temp, etl.phase?

[Database]
; !!Recommended!! If you have more than one database defined on this Vertica cluster, use this parameter to specify which database to backup/restore.
; dbName = current_database

; If this parameter is True, vbr prompts the user for the database password every time.
; If False, specify the location of password config file in 'passwordFile' parameter in [Misc] section.
; dbPromptForPassword = True

; If true, vbr attempts to connect to the database using a local connection.
; dbUseLocalConnection = False

; ------------------------------------------- ;
;;; ADVANCED PARAMETERS ;;;
; ------------------------------------------- ;
[CloudStorage]
; Specifies encryption-at-rest on S3
; cloud_storage_encrypt_at_rest = sse
; cloud_storage_sse_kms_key_id = <key_id>
```

```
; Specifies SSL encrypted transfer.
; cloud_storage_encrypt_transport = True

; Specifies the number of threads for upload/download - backup
; cloud_storage_concurrency_backup = 10

; Specifies the number of threads for upload/download - restore
; cloud_storage_concurrency_restore = 10

; Specifies the number of threads for deleting objects from the backup location
; cloud_storage_concurrency_delete = 10

; Specifies the path to a custom SSL server certificate bundle
; cloud_storage_ca_bundle = /home/user/ssl_folder/ca_bundle.pem

[Misc]
; The temp directory location on all database hosts.
; The directory must be readable and writeable by the dbadmin, and must implement POSIX style fcntl lockf locking.
; tempDir = /tmp/vbr

; Specifies the number of historical backups to retain in addition to the most recent backup.
; 1 current + n historical backups
; restorePointLimit = 1

; Full path to the password configuration file
; Store this file in directory readable only by the dbadmin.
; (no default)
; passwordFile = /path/to/vbr/pw.txt

; Specifies the service name of the Vertica Kerberos principal. This only applies to HDFS.
; kerberos_service_name = vertica

; Specifies the realm (authentication domain) of the Vertica Kerberos principal. This only applies to HDFS.
; kerberos_realm = your_auth_domain

; Specifies the location of the keytab file which contains the credentials for the Vertica Kerberos principal. This only applies to HDFS.
; kerberos_keytab_file = /path/to/keytab_file

; Specifies the location of the Hadoop XML configuration files of the HDFS clusters. Only set this when your cluster is on HA. This only applies to HDFS.
; If you have multiple conf directories, please separate them with ':'.
; hadoop_conf_dir = /path/to/conf or /path/to/conf1:/path/to/conf2

[Database]
; Vertica user name for vbr to connect to the database.
; This setting is rarely needed since dbUser is normally identical to the database administrator
; dbUser = current_username
```

Full hard-link backup/restore

backup_restore_full_hardlink.ini

The following requirements apply to configuring hard-link local backups:

- Under the [Transmission] section, add the parameter hardLinkLocal :

  ```
  hardLinkLocal = True
  ```

- The backup directory must be in the same file system as the database data directory.
- Omit the encrypt parameter. If the configuration file sets both parameters encrypt and hardLinkLocal to true, then vbr issues a warning and ignores the encrypt parameter.

```
; This sample vbr configuration file shows backup and restore using hard-links to data files on each database host for that host's backup.
; Section headings are enclosed by square brackets.
; Comments have leading semicolons (;) or pound signs (#).
```

```
; An equal sign separates options and values.
; Specify arguments marked '!!Mandatory!!' explicitly.
; All commented parameters are set to their default value.

; ---------------------------------------------- ;
;;; BASIC PARAMETERS ;;;
; ---------------------------------------------- ;

[Mapping]
; For each database node there must be one [Mapping] entry to indicate the directory to store the backup.
; !!Mandatory!! Backup host name (no default) and Backup directory (no default).
; node_name = backup_host:backup_dir
; Must use [] for hardlink backups
v_exampledb_node0001 = []:/home/dbadmin/backups
v_exampledb_node0002 = []:/home/dbadmin/backups
v_exampledb_node0003 = []:/home/dbadmin/backups
v_exampledb_node0004 = []:/home/dbadmin/backups

[Misc]
; !!Recommended!! Snapshot name.  Object and full backups should always have different snapshot names.
; Backups with the same snapshotName form a time sequence limited by restorePointLimit.
; Valid characters: a-z A-Z 0-9 - _
; snapshotName = backup_snapshot

[Transmission]
; !!Mandatory!! Identifies the backup as a hardlink style backup.
hardLinkLocal = True
; If copyOnHardLinkFailure is True, when a hard-link local backup cannot create links the data is copied instead.
copyOnHardLinkFailure = False

; ---------------------------------------------- ;
;;; ADVANCED PARAMETERS ;;;
; ---------------------------------------------- ;

[Database]
; !!Recommended!! If you have more than one database defined on this Vertica cluster, use this parameter to specify which database to backup/restore.
; dbName = current_database

; If this parameter is True, vbr prompts the user for the database password every time.
; If False, specify the location of password config file in 'passwordFile' parameter in [Misc] section.
; dbPromptForPassword = True

[Misc]
; The temp directory location on all database hosts.
; The directory must be readable and writeable by the dbadmin, and must implement POSIX style fcntl lockf locking.
; tempDir = /tmp/vbr

; Full path to the password configuration file
; Store this file in directory readable only by the dbadmin.
; (no default)
; passwordFile =

; Specifies the number of historical backups to retain in addition to the most recent backup.
; 1 current + n historical backups
; restorePointLimit = 1

; When enabled, Vertica confirms that the specified backup locations contain
; sufficient free space and inodes to allow a successful backup. If a backup
; location has insufficient resources, Vertica displays an error message explaining the shortage and
; cancels the backup. If Vertica cannot determine the amount of available space
; or number of inodes in the backupDir, it displays a warning and continues
```

```
; with the backup.
; enableFreeSpaceCheck = True

[Database]
; Vertica user name for vbr to connect to the database.
; This setting is rarely needed since dbUser is normally identical to the database administrator.
; dbUser = current_username
```

Full local backup/restore

backup_restore_full_local.ini

```
; This is a sample vbr configuration file for backup and restore using a file system on each database host for that host's backup.
; Section headings are enclosed by square brackets.
; Comments have leading semicolons (;) or pound signs (#).
; An equal sign separates options and values.
; Specify arguments marked '!!Mandatory!!' explicitly.
; All commented parameters are set to their default value.

; ------------------------------------------- ;
;;; BASIC PARAMETERS ;;;
; ------------------------------------------- ;

[Mapping]
; !!Mandatory!! For each database node there must be one [Mapping] entry to indicate the directory to store the backup.
; node_name = backup_host:backup_dir
; [] indicates backup to localhost
v_exampledb_node0001 = []:/home/dbadmin/backups
v_exampledb_node0002 = []:/home/dbadmin/backups
v_exampledb_node0003 = []:/home/dbadmin/backups
v_exampledb_node0004 = []:/home/dbadmin/backups

[Misc]
; !!Recommended!! Snapshot name
; Backups with the same snapshotName form a time sequence limited by restorePointLimit.
; SnapshotName is used for naming archives in the backup directory, and for monitoring and troubleshooting.
; Valid values: a-z A-Z 0-9 - _
; snapshotName = backup_snapshot

[Database]
; !!Recommended!! If you have more than one database defined on this Vertica cluster, use this parameter to specify which database to backup/restore.
; dbName = current_database

; If this parameter is True, vbr prompts the user for the database password every time.
; If False, specify the location of password config file in 'passwordFile' parameter in [Misc] section.
; dbPromptForPassword = True

; ------------------------------------------- ;
;;; ADVANCED PARAMETERS ;;;
; ------------------------------------------- ;

[Misc]

; The temp directory location on all database hosts.
; The directory must be readable and writeable by the dbadmin, and must implement POSIX style fcntl lockf locking.
; tempDir = /tmp/vbr

; Specifies the number of historical backups to retain in addition to the most recent backup.
; 1 current + n historical backups
; restorePointLimit = 1

; Full path to the password configuration file
```

```
; Store this file in directory readable only by the dbadmin.
; (no default)
; passwordFile = /path/to/vbr/pw.txt

; When enabled, Vertica confirms that the specified backup locations contain
; sufficient free space and inodes to allow a successful backup. If a backup
; location has insufficient resources, Vertica displays an error message explaining the shortage and
; cancels the backup. If Vertica cannot determine the amount of available space
; or number of inodes in the backupDir, it displays a warning and continues
; with the backup.
; enableFreeSpaceCheck = True

[Transmission]
; The total bandwidth limit for all restore connections in KBPS, 0 for unlimited
; total_bwlimit_restore = 0

; The maximum number of restore TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_restore = 1

; Total bandwidth limit for all backup connections in KBPS, 0 for unlimited. Vertica distributes
; this bandwidth evenly among the number of connections set in concurrency_backup.
; total_bwlimit_backup = 0

; The maximum number of backup TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_backup = 1

; The maximum number of delete TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_delete = 16

[Database]
; Vertica user name for vbr to connect to the database.
; This setting is rarely needed since dbUser is normally identical to the database administrator
; dbUser = current_username
```

Object-level local backup/restore in Enterprise Mode

backup_restore_object_local.ini

An object backup backs up only the schemas or tables that are specified in the [Misc] section by the parameter objects, or parameters includeObjects and excludeObjects.

For an object restore, use the same configuration file that you used to create the backup, and specify the objects to restore with the vbr command-line parameter --restore-objects .

```
; This sample vbr configuration file shows object-level backup and restore
; using a file system on each database host for that host's backup.
; Section headings are enclosed by square brackets.
; Comments have leading semicolons (;) or pound signs (#).
; Option and values are separated by an equal sign.
; Only arguments marked as '!!Mandatory!!' must be specified explicitly.
; All commented parameters are set to their default value.

; ------------------------------------------- ;
;;; BASIC PARAMETERS ;;;
; ------------------------------------------- ;

[Mapping]
```

; There must be one [Mapping] section for all of the nodes in your database cluster.
; !!Mandatory!! Backup host name (no default) and Backup directory (no default)
; node_name = backup_host:backup_dir
; [] indicates backup to localhost
v_exampledb_node0001 = []:/home/dbadmin/backups
v_exampledb_node0002 = []:/home/dbadmin/backups
v_exampledb_node0003 = []:/home/dbadmin/backups
v_exampledb_node0004 = []:/home/dbadmin/backups

[Misc]
; !!Recommended!! Snapshot name. Object and full backups should always have different snapshot names.
; Backups with the same snapshotName form a time sequence limited by restorePointLimit.
; SnapshotName is used for naming archives in the backup directory, and for monitoring and troubleshooting.
; Valid values: a-z A-Z 0-9 - _
; snapshotName = backup_snapshot

; Specifies how Vertica handles objects of the same name when restoring schema or table backups.
; objectRestoreMode = createOrReplace

; Specifies which tables and/or schemas to copy. For tables, the containing schema defaults to public.
; Note: 'objects' is incompatible with 'includeObjects' and 'excludeObjects'.
; (no default)
objects = mytable, myschema, myothertable

; Specifies the set of objects to backup/restore; wildcards may be used.
; Note: 'includeObjects' is incompatible with 'objects'.
; includeObjects = public.mytable, customer*, s?

; Subtracts from the set of objects to backup/restore; wildcards may be used
; Note: 'excludeObjects' is incompatible with 'objects'.
; excludeObjects = public.*temp, etl.phase?

[Database]
; !!Recommended!! If you have more than one database defined on this Vertica cluster, use this parameter to specify which database to backup/restore.
; dbName = current_database

; If this parameter is True, vbr will prompt user for database password every time.
; If set to False, specify location of password config file in 'passwordFile' parameter in [Misc] section.
; dbPromptForPassword = True

; --- ;
;;; ADVANCED PARAMETERS ;;;
; --- ;

[Misc]
; The temp directory location on all database hosts.
; The directory must be readable and writeable by the dbadmin, and must implement POSIX style fcntl lockf locking.
; tempDir = /tmp/vbr

; Specifies the number of historical backups to retain in addition to the most recent backup.
; 1 current + n historical backups
; restorePointLimit = 1

; Full path to the password configuration file
; Store this file in directory readable only by the dbadmin.
; (no default)
; passwordFile = /path/to/vbr/pw.txt

; When enabled, Vertica confirms that the specified backup locations contain
; sufficient free space and inodes to allow a successful backup. If a backup
; location has insufficient resources, Vertica displays an error message explaining the shortage and

```
; cancels the backup. If Vertica cannot determine the amount of available space
; or number of inodes in the backupDir, it displays a warning and continues
; with the backup.
; enableFreeSpaceCheck = True

[Transmission]
; The total bandwidth limit for all restore connections in KBPS, 0 for unlimited
; total_bwlimit_restore = 0

; The maximum number of restore TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_restore = 1

; Total bandwidth limit for all backup connections in KBPS, 0 for unlimited. Vertica distributes
; this bandwidth evenly among the number of connections set in concurrency_backup.
; total_bwlimit_backup = 0

; The maximum number of backup TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_backup = 1

; The maximum number of delete TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_delete = 16

[Database]
; Vertica user name for vbr to connect to the database.
; This setting is rarely needed since dbUser is normally identical to the database administrator.
; dbUser = current_username
```

Restore object from backup to an alternate cluster

object_restore_to_other_cluster.ini

```
; This sample vbr configuration file shows object restore to another cluster from an existing full or object backup.
; To restore objects from an existing backup(object or full), you must use the "--restore-objects" vbr command line option.
; Section headings are enclosed by square brackets.
; Comments have leading semicolons (;) or pound signs (#).
; An equal sign separates options and values.
; Specify arguments marked '!!Mandatory!!' explicitly.
; All commented parameters are set to their default value.

; ------------------------------------------- ;
;;; BASIC PARAMETERS ;;;
; ------------------------------------------- ;

[Mapping]
; There must be one [Mapping] section for all of the nodes in your database cluster.
; !!Mandatory!! Backup host name (no default) and Backup directory (no default)
; node_name = backup_host:backup_dir
v_exampledb_node0001 = backup_host0001:/home/dbadmin/backups
v_exampledb_node0002 = backup_host0002:/home/dbadmin/backups
v_exampledb_node0003 = backup_host0003:/home/dbadmin/backups
v_exampledb_node0004 = backup_host0004:/home/dbadmin/backups

[NodeMapping]
; !!Recommended!! This section is required when performing an object restore from a full/object backup to a different cluster and node names are different
between source (backup) and destination (restoring) databases.
v_sourcedb_node0001 = v_exampledb_node0001
```

```
v_sourcedb_node0002 = v_exampledb_node0002
v_sourcedb_node0003 = v_exampledb_node0003
v_sourcedb_node0004 = v_exampledb_node0004

[Database]
; !!Recommended!! If you have more than one database defined on this Vertica cluster, use this parameter to specify which database to backup/restore.
; dbName = current_database

; If this parameter is True, vbr prompts the user for database password every time.
; If False, specify location of password config file in 'passwordFile' parameter in [Misc] section.
; dbPromptForPassword = True

; ------------------------------------------- ;
;;; ADVANCED PARAMETERS ;;;
; ------------------------------------------- ;

[Misc]
; !!Recommended!! Snapshot name.
; SnapshotName is useful for monitoring and troubleshooting.
; Valid characters: a-z A-Z 0-9 - _
; snapshotName = backup_snapshot

; Specifies how Vertica handles objects of the same name when restoring schema or table backups.  Options are coexist, createOrReplace or create.
; objectRestoreMode = createOrReplace

; The temp directory location on all database hosts.
; The directory must be readable and writeable by the dbadmin, and must implement POSIX style fcntl lockf locking.
; tempDir = /tmp/vbr

; Full path to the password configuration file.
; Store this file in a directory only readable by the dbadmin.
; (no default)
; passwordFile = /path/to/vbr/pw.txt

; When enabled, Vertica confirms that the specified backup locations contain
; sufficient free space and inodes to allow a successful backup. If a backup
; location has insufficient resources, Vertica displays an error message and
; cancels the backup. If Vertica cannot determine the amount of available space
; or number of inodes in the backupDir, it displays a warning and continues
; with the backup.
; enableFreeSpaceCheck = True

[Transmission]
; Sets options for transmitting the data when using backup hosts.

; Specifies the default port number for the rsync protocol.
; port_rsync = 50000

; The total bandwidth limit for all restore connections in KBPS, 0 for unlimited
; total_bwlimit_restore = 0

; The maximum number of backup TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_restore = 1

[Database]
; Vertica user name for vbr to connect to the database.
; This setting is rarely needed since dbUser is normally identical to the database administrator.
; dbUser = current_username
```

Object replication to an alternate database

replicate.ini

```
; This sample vbr configuration file shows the replicate vbr task.
; Section headings are enclosed by square brackets.
; Comments have leading semicolons (;) or pound signs (#).
; An equal sign separates options and values.
; Specify arguments marked '!!Mandatory!!' explicitly.
; All commented parameters are set to their default value.

; ------------------------------------------- ;
;;; BASIC PARAMETERS ;;;
; ------------------------------------------- ;

[Mapping]
; There must be one [Mapping] section for all of the nodes in your database cluster.
; !!Mandatory!! Target host name (no default)
; node_name = new_host
v_exampledb_node0001 = destination_host0001
v_exampledb_node0002 = destination_host0002
v_exampledb_node0003 = destination_host0003
v_exampledb_node0004 = destination_host0004

[Misc]
; !!Recommended!! Snapshot name.
; SnapshotName is useful for monitoring and troubleshooting.
; Valid characters: a-z A-Z 0-9 - _
; snapshotName = backup_snapshot



; Specifies which tables and/or schemas to copy.  For tables, the containing schema defaults to public.
; objects for replication. You must specify only one of either objects or includeObjects.
; Use comma-separated list for multiple objects
; (no default)
objects = mytable, myschema, myothertable

; Specifies the set of objects to replicate; wildcards may be used.
; Note: 'includeObjects' is incompatible with 'objects'.
; includeObjects = public.mytable, customer*, s?

; Subtracts from the set of objects to replicate; wildcards may be used
; Note: 'excludeObjects' is incompatible with 'objects'.
; excludeObjects = public.*temp, etl.phase?

; Specifies how Vertica handles objects of the same name when copying schema or tables.
; objectRestoreMode = createOrReplace

[Database]
; !!Recommended!! If you have more than one database defined on this Vertica cluster, use this parameter to specify which database to replicate.
; dbName = current_database

; If this parameter is True, vbr prompts the user for the database password every time.
; If False, specify the location of password config file in 'passwordFile' parameter in [Misc] section.
; dbPromptForPassword = True

; !!Mandatory!! These settings are all mandatory for replication. None of which have defaults.
dest_dbName = target_db
dest_dbUser = dbadmin
dest_dbPromptForPassword = True

; ------------------------------------------- ;
```

```
;;; ADVANCED PARAMETERS ;;;
; ------------------------------------------- ;

[Misc]
; The temp directory location on all database hosts.
; The directory must be readable and writeable by the dbadmin, and must implement POSIX style fcntl lockf locking.
; tempDir = /tmp/vbr

; Full path to the password configuration file containing database password credentials
; Store this file in directory readable only by the dbadmin.
; (no default)
; passwordFile = /path/to/vbr/pw.txt

; Specifies the service name of the Vertica Kerberos principal. This only applies to HDFS.
; kerberos_service_name = vertica

; Specifies the realm (authentication domain) of the Vertica Kerberos principal. This only applies to HDFS.
; kerberos_realm = your_auth_domain

; Specifies the location of the keytab file which contains the credentials for the Vertica Kerberos principal. This only applies to HDFS.
; kerberos_keytab_file = /path/to/keytab_file

; Specifies the location of the Hadoop XML configuration files of the HDFS clusters. Only set this when your cluster is on HA. This only applies to HDFS.
; If you have multiple conf directories, please separate them with ':'.
; hadoop_conf_dir = /path/to/conf or /path/to/conf1:/path/to/conf2

[Transmission]
; Specifies the default port number for the rsync protocol.
; port_rsync = 50000

; Total bandwidth limit for all backup connections in KBPS, 0 for unlimited. Vertica distributes
; this bandwidth evenly among the number of connections set in concurrency_backup.
; total_bwlimit_backup = 0

; The maximum number of replication TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_backup = 1

; The maximum number of restore TCP rsync connection threads per node.
; Results vary depending on environment, but values between 2 and 16 are sometimes quite helpful.
; concurrency_restore = 1

; The maximum number of delete TCP rsync connection threads per node.
; Results vary depending on environment, but values between 2 and 16 are sometimes quite helpful.
; concurrency_delete = 16

[Database]
; Vertica user name for vbr to connect to the database.
; This is very rarely be needed since dbUser is normally identical to the database administrator.
; dbUser = current_username
```

Database copy to an alternate cluster

copycluster.ini

```
; This sample vbr configuration file is configured for the copycluster vbr task.
; Copycluster supports full database copies only, not specific objects.
; Section headings are enclosed by square brackets.
; Comments have leading semicolons (;) or pound signs (#).
; An equal sign separates options and values.
; Specify arguments marked '!!Mandatory!!' explicitly.
```

```
; All commented parameters are set to their default value.

; ------------------------------------------- ;
;;; BASIC PARAMETERS ;;;
; ------------------------------------------- ;

[Mapping]
; For each node of the source database, there must be a [Mapping] entry specifying the corresponding hostname of the destination database node.
; !!Mandatory!!  node_name = new_host/ip  (no defaults)
v_exampledb_node0001 = destination_host1.example
v_exampledb_node0002 = destination_host2.example
v_exampledb_node0003 = destination_host3.example
v_exampledb_node0004 = destination_host4.example
; v_exampledb_node0001 = 10.0.90.17
; v_exampledb_node0002 = 10.0.90.18
; v_exampledb_node0003 = 10.0.90.19
; v_exampledb_node0004 = 10.0.90.20

[Database]
; !!Recommended!! If you have more than one database defined on this Vertica cluster, use this parameter to specify which database to copy.
; dbName = current_database

; If this parameter is True, vbr prompts the user for the database password every time.
; If False, specify the location of password config file in 'passwordFile' parameter in [Misc] section.
; dbPromptForPassword = True

; ------------------------------------------- ;
;;; ADVANCED PARAMETERS ;;;
; ------------------------------------------- ;

[Misc]
; !!Recommended!! Snapshot name.
; SnapshotName is used for monitoring and troubleshooting.
; Valid characters: a-z A-Z 0-9 - _
; snapshotName = backup_snapshot

; The temp directory location on all database hosts.
; The directory must be readable and writeable by the dbadmin, and must implement POSIX style fcntl lockf locking.
; tempDir = /tmp/vbr

; Full path to the password configuration file containing database password credentials
; Store this file in directory readable only by the dbadmin.
; (no default)
; passwordFile = /path/to/vbr/pw.txt

[Transmission]
; Specifies the default port number for the rsync protocol.
; port_rsync = 50000

; Total bandwidth limit for all copycluster connections in KBPS, 0 for unlimited. Vertica distributes
; this bandwidth evenly among the number of connections set in concurrency_backup.
; total_bwlimit_backup = 0

; The maximum number of backup TCP rsync connection threads per node.
; Optimum settings depend on your particular environment.
; For best performance, experiment with values between 2 and 16.
; concurrency_backup = 1

; The maximum number of restore TCP rsync connection threads per node.
; Results vary depending on environment, but values between 2 and 16 are sometimes quite helpful.
; concurrency_restore = 1
```

```
; The maximum number of delete TCP rsync connection threads per node.
; Results vary depending on environment, but values between 2 and 16 are sometimes quite helpful.
; concurrency_delete = 16

[Database]
; Vertica user name for vbr to connect to the database.
; This setting is rarely needed since dbUser is normally identical to the database administrator
; dbUser = current_username
```

Password file

password.ini

Unlike other configuration (.ini) files, the password configuration file must be referenced by another configuration file, through its passwordFile parameter.

```
; This is a sample password configuration file.
; Point to this file in the 'passwordFile' parameter of the [Misc] section.
; Section headings are enclosed by square brackets.
; Comments have leading semicolons (;) or pound signs (#).
; Option and values are separated by an equal sign.

[Passwords]
; The database administrator's password, and used if dbPromptForPassword is False.
; dbPassword=myDBsecret

; The password for the rsync user account.
; serviceAccessPass=myrsyncpw

; The password for the dest_dbuser Vertica account, for replication tasks only.
; dest_dbPassword=destDBsecret
```

Eon Mode database requirements

Eon Mode databases perform the same backup and restore operations as Enterprise Mode databases. Additional requirements pertain to Eon Mode because it uses a different architecture.

> **Note**
> These requirements are for cloud storage locations listed in Backing up and restoring the database , and on-premises with communal storage on HDFS.

Cloud storage requirements

Eon Mode databases must be backed up to supported cloud storage locations. The following [CloudStorage] configuration parameters must be set:

- cloud_storage_backup_path
- cloud_storage_backup_file_system_path

A backup path is valid for one database only. You cannot use the same path to store backups for multiple databases.

Eon Mode databases that use S3-compatible on-premises cloud storage can back up to Amazon Web Services (AWS) S3.

Cloud storage access

In addition to having access to the cloud storage bucket used for the database's communal storage, you must have access to the cloud storage backup location. Verify that the credential you use to access communal storage also has access to the backup location. For more information about configuring cloud storage access for Vertica, see Configuring cloud storage backups .

> **Note**
> While an AWS backup location can be in a different region, backup and restore operations across different S3 regions are incompatible with virtual private cloud (VPC) endpoints.

Eon on-premises and private cloud storage

If an Eon database runs on-premises, then communal storage is not on AWS but on another storage platform that uses the S3 or GS protocol. This means there can be two endpoints and two sets of credentials, depending on where you back up. This additional information is stored in environment variables , and not in vbr configuration parameters.

Backups of Eon Mode on-premises databases do not support AWS IAM profiles.

HDFS on-premises storage

To back up an Eon Mode database that uses HDFS on-premises storage, the communal storage and backup location must use the same HDFS credentials and domain. All vbr operations are supported, except copycluster .

Vertica supports Kerberos authentication, High Availability Name Node, and wire encryption for vbr operations. Vertica does not support at-rest encryption for Hadoop storage.

For details, see Configuring backups to and from HDFS .

Database restore requirements

When restoring a backup of an Eon Mode database, the restore database must satisfy the following requirements:

- Share the same name as the backup database.
- Have at least as many nodes as the primary subcluster(s) in the backup database.
- Have the same node names as the nodes of the backup database.
- Use the same catalog directory location as the backup database.
- Use the same port numbers as the backup database.
- For object restore, have the same shard subscriptions. If the shard subscriptions have changed, you cannot do object restores but can do a full restore. Shard subscriptions can change when you add or remove nodes or rebalance your cluster.

You can restore a full or object backup that was taken from a database with primary and secondary subclusters to the primary subclusters in the target database. The database can have only primary subclusters, or it can also have any number of secondary subclusters. Secondary subclusters do not need to match the backup database. The same is true for replicating a database; only the primary subclusters are required. The requirements are similar to those for Reviving an Eon Mode database cluster .

Use the [Mapping] section in the configuration file to specify the mappings for the primary subcluster.

Requirements for backing up and restoring HDFS storage locations

There are several considerations for backing up and restoring HDFS storage locations:

- The HDFS directory for the storage location must have snapshotting enabled. You can either directly configure this yourself or enable the database administrator's Hadoop account to do it for you automatically. See Hadoop configuration for backup and restore for more information.
- If the Hadoop cluster uses Kerberos, Vertica nodes must have access to certain Hadoop configuration files. See Configuring Kerberos below.
- To restore an HDFS storage location, your Vertica cluster must be able to run the Hadoop distcp command. See Configuring distcp on a Vertica Cluster below.
- HDFS storage locations do not support object-level backups. You must perform a full database backup to back up the data in your HDFS storage locations.
- Data in an HDFS storage location is backed up to HDFS. This backup guards against accidental deletion or corruption of data. It does not prevent data loss in the case of a catastrophic failure of the entire Hadoop cluster. To prevent data loss, you must have a backup and disaster recovery plan for your Hadoop cluster.
  Data stored on the Linux native file system is still backed up to the location you specify in the backup configuration file. It and the data in HDFS storage locations are handled separately by the vbr backup script.

Configuring Kerberos

If HDFS uses Kerberos, then to back up your HDFS storage locations you must take the following additional steps:

1. Grant Hadoop superuser privileges to the Kerberos principals for each Vertica node.
2. Copy Hadoop configuration files to your database nodes as explained in Accessing Hadoop Configuration Files . Vertica needs access to core-site.xml , hdfs-site.xml , and yarn-site.xml for backup and restore. If your Vertica nodes are co-located on HDFS nodes, these files are already present.
3. Set the HadoopConfDir parameter to the location of the directory containing these files. The value can be a path, if the files are in multiple directories. For example:

```
=> ALTER DATABASE exampledb SET HadoopConfDir = '/etc/hadoop/conf:/etc/hadoop/test';
```

All three configuration files must be present on this path on every database node.

If your Vertica nodes are co-located on HDFS nodes and you are using Kerberos, you must also change some Hadoop configuration parameters. These changes are needed in order for restoring from backups to work. In yarn-site.xml on every Vertica node, set the following parameters:

Parameter	Value
yarn.resourcemanager.proxy-user-privileges.enabled	true
yarn.resourcemanager.proxyusers.*.groups	•
yarn.resourcemanager.proxyusers.*.hosts	•
yarn.resourcemanager.proxyusers.*.users	•
yarn.timeline-service.http-authentication.proxyusers.*.groups	•
yarn.timeline-service.http-authentication.proxyusers.*.hosts	•
yarn.timeline-service.http-authentication.proxyusers.*.users	•

No changes are needed on HDFS nodes that are not also Vertica nodes.

Configuring distcp on a Vertica cluster

Your Vertica cluster must be able to run the Hadoop distcp command to restore a backup of an HDFS storage location. The easiest way to enable your cluster to run this command is to install several Hadoop packages on each node. These packages must be from the same distribution and version of Hadoop that is running on your Hadoop cluster.

The steps you need to take depend on:

- The distribution and version of Hadoop running on the Hadoop cluster containing your HDFS storage location.
- The distribution of Linux running on your Vertica cluster.

> **Note**
> Installing the Hadoop packages necessary to run distcp does not turn your Vertica database into a Hadoop cluster. This process installs just enough of the Hadoop support files on your cluster to run the distcp command. There is no additional overhead placed on the Vertica cluster, aside from a small amount of additional disk space consumed by the Hadoop support files.

Configuration overview

The steps for configuring your Vertica cluster to restore backups for HDFS storage location are:

1. If necessary, install and configure a Java runtime on the hosts in the Vertica cluster.
2. Find the location of your Hadoop distribution's package repository.
3. Add the Hadoop distribution's package repository to the Linux package manager on all hosts in your cluster.
4. Install the necessary Hadoop packages on your Vertica hosts.
5. Set two configuration parameters in your Vertica database related to Java and Hadoop.
6. Confirm that the Hadoop distcp command runs on your Vertica hosts.

The following sections describe these steps in greater detail.

Installing a Java runtime

Your Vertica cluster must have a Java Virtual Machine (JVM) installed to run the Hadoop distcp command. It already has a JVM installed if you have configured it to:

- Execute user-defined extensions developed in Java. See Developing user-defined extensions (UDxs) for more information.
- Access Hadoop data using the HCatalog Connector. See Using the HCatalog Connector for more information.

If your Vertica database has a JVM installed, verify that your Hadoop distribution supports it. See your Hadoop distribution's documentation to determine which JVMs it supports.

If the JVM installed on your Vertica cluster is not supported by your Hadoop distribution you must uninstall it. Then you must install a JVM that is supported by both Vertica and your Hadoop distribution. See Vertica SDKs for a list of the JVMs compatible with Vertica.

If your Vertica cluster does not have a JVM (or its existing JVM is incompatible with your Hadoop distribution), follow the instructions in Installing the Java runtime on your Vertica cluster .

Finding your Hadoop distribution's package repository

Many Hadoop distributions have their own installation system, such as Cloudera Manager or Ambari. However, they also support manual installation using native Linux packages such as RPM and .deb files. These package files are maintained in a repository. You can configure your Vertica hosts to access this repository to download and install Hadoop packages.

Consult your Hadoop distribution's documentation to find the location of its Linux package repository. This information is often located in the portion of the documentation covering manual installation techniques.

Each Hadoop distribution maintains separate repositories for each of the major Linux package management systems. Find the specific repository for the Linux distribution running your Vertica cluster. Be sure that the package repository that you select matches the version used by your Hadoop cluster.

Configuring Vertica nodes to access the Hadoop Distribution's package repository

Configure the nodes in your Vertica cluster so they can access your Hadoop distribution's package repository. Your Hadoop distribution's documentation should explain how to add the repositories to your Linux platform. If the documentation does not explain how to add the repository to your packaging system, refer to your Linux distribution's documentation.

The steps you need to take depend on the package management system your Linux platform uses. Usually, the process involves:

- Downloading a configuration file.
- Adding the configuration file to the package management system's configuration directory.
- For Debian-based Linux distributions, adding the Hadoop repository encryption key to the root account keyring.
- Updating the package management system's index to have it discover new packages.

You must add the Hadoop repository to all hosts in your Vertica cluster.

Installing the required Hadoop packages

After configuring the repository, you are ready to install the Hadoop packages. The packages you need to install are:

- hadoop
- hadoop-hdfs
- hadoop-client

The names of the packages are usually the same across all Hadoop and Linux distributions. These packages often have additional dependencies. Always accept any additional packages that the Linux package manager asks to install.

To install these packages, use the package manager command for your Linux distribution. The package manager command you need to use depends on your Linux distribution:

- On Red Hat and CentOS, the package manager command is yum .
- On Debian and Ubuntu, the package manager command is apt-get .
- On SUSE the package manager command is zypper .

Consult your Linux distribution's documentation for instructions on installing packages.

Setting configuration parameters

You must set two Hadoop configuration parameters to enable Vertica to restore HDFS data:

- JavaBinaryForUDx is the path to the Java executable. You may have already set this value to use Java UDxs or the HCatalog Connector. You can find the path for the default Java executable from the Bash command shell using the command:
  ```
  $ which java
  ```
- HadoopHome is the directory that contains bin/hadoop (the bin directory containing the Hadoop executable file). The default value for this parameter is /usr . The default value is correct if your Hadoop executable is located at /usr/bin/hadoop .

The following example shows how to set and then review the values of these parameters:

```
=> ALTER DATABASE DEFAULT SET PARAMETER JavaBinaryForUDx = '/usr/bin/java';
=> SELECT current_value FROM configuration_parameters WHERE parameter_name = 'JavaBinaryForUDx';
 current_value
---------------
 /usr/bin/java
(1 row)
=> ALTER DATABASE DEFAULT SET HadoopHome = '/usr';
=> SELECT current_value FROM configuration_parameters WHERE parameter_name = 'HadoopHome';
 current_value
---------------
 /usr
(1 row)
```

You can also set the following parameters:

- HadoopFSReadRetryTimeout and HadoopFSWriteRetryTimeout specify how long to wait before failing. The default value for each is 180 seconds. If you are confident that your file system will fail more quickly, you can improve performance by lowering these values.
- HadoopFSReplication specifies the number of replicas HDFS makes. By default, the Hadoop client chooses this; Vertica uses the same value for all nodes.

  > **Caution**
  > Do not change this setting unless directed otherwise by Vertica support.

- HadoopFSBlockSizeBytes is the block size to write to HDFS; larger files are divided into blocks of this size. The default is 64MB.

Confirming that distcp runs

After the packages are installed on all hosts in your cluster, your database should be able to run the Hadoop distcp command. To test it:

1. Log into any host in your cluster as the database superuser .
2. At the Bash shell, enter the command:
   ```
   $ hadoop distcp
   ```
3. The command should print a message similar to the following:

```
usage: distcp OPTIONS [source_path...] <target_path>
          OPTIONS
-async                Should distcp execution be blocking
-atomic               Commit all changes or none
-bandwidth <arg>      Specify bandwidth per map in MB
-delete               Delete from target, files missing in source
-f <arg>              List of files that need to be copied
-filelimit <arg>      (Deprecated!) Limit number of files copied to <= n
-i                    Ignore failures during copy
-log <arg>            Folder on DFS where distcp execution logs are
                      saved
-m <arg>              Max number of concurrent maps to use for copy
-mapredSslConf <arg>  Configuration for ssl config file, to use with
                      hftps://
-overwrite            Choose to overwrite target files unconditionally,
                      even if they exist.
-p <arg>              preserve status (rbugpc)(replication, block-size,
                      user, group, permission, checksum-type)
-sizelimit <arg>      (Deprecated!) Limit number of files copied to <= n
                      bytes
-skipcrccheck         Whether to skip CRC checks between source and
                      target paths.
-strategy <arg>       Copy strategy to use. Default is dividing work
                      based on file sizes
-tmp <arg>            Intermediate work path to be used for atomic
                      commit
-update               Update target, copying only missingfiles or
                      directories
```

4. Repeat these steps on the other hosts in your database to verify that all of the hosts can run distcp .

Troubleshooting

If you cannot run the distcp command, try the following steps:

- If Bash cannot find the hadoop command, you may need to manually add Hadoop's bin directory to the system search path. An alternative is to create a symbolic link in an existing directory in the search path (such as /usr/bin) to the hadoop binary.
- Ensure the version of Java installed on your Vertica cluster is compatible with your Hadoop distribution.
- Review the Linux package installation tool's logs for errors. In some cases, packages may not be fully installed, or may not have been downloaded due to network issues.
- Ensure that the database administrator account has permission to execute the hadoop command. You might need to add the account to a specific group in order to allow it to run the necessary commands.

Setting up backup locations

Important

Inadequate security on backups can compromise overall database security. Be sure to secure backup locations and strictly limit access to backups only to users who already have permissions to access all database data.

Full and object-level backups reside on *backup hosts* , the computer systems on which backups and archives are stored. On the backup hosts, Vertica saves backups in a specific *backup location* (directory).

You must set up your backup hosts before you can create backups.

The storage format type at your backup locations must support fcntl lockf (POSIX) file locking.

In this section

- Configuring backup hosts and connections
- Configuring hard-link local backup hosts
- Configuring cloud storage backups
- Additional considerations for cloud storage
- Configuring backups to and from HDFS

Configuring backup hosts and connections

You use vbr to back up your database to one or more hosts (known as *backup hosts*) that can be outside of your database cluster.

You can use one or more backup hosts or a single cloud storage bucket to back up your database. Use the vbr configuration file to specify which backup host each node in your cluster should use.

Before you back up to hosts outside of the local cluster, configure the target backup locations to work with vbr . The backup hosts you use must:

- Have sufficient backup disk space .
- Be accessible from your database cluster through SSH.
- Have passwordless SSH access for the Database Administrator account.
- Have either the Vertica rpm or Python 3.7 and rsync 3.0.5 or later installed.
- If you are using a stateful firewall, configure your tcp_keepalive_time and tcp_keepalive_intvl sysctl settings to use values less than your firewall timeout value.

Configuring TCP forwarding on database hosts

vbr depends on TCP forwarding to forward connections from database hosts to backup hosts. For copycluster and replication tasks, you must enable TCP forwarding on both sets of hosts. SSH connections to backup hosts do not require SSH forwarding.

If it is not already set by default, set AllowTcpForwarding = Yes in /etc/ssh/sshd_config and then send a SIGHUP signal to sshd on each host. See the Linux sshd documentation for more information.

If TCP forwarding is not enabled, tasks requiring it fail with the following message: "Errors connecting to remote hosts: Check SSH settings, and that the same Vertica version is installed on all nodes."

On a single-node cluster, vbr uses a random high-number port to create a local ssh tunnel. This fails if PermitOpen is set to restrict the port. Comment out the PermitOpen line in sshd_config.

Creating configuration files for backup hosts

Create separate configuration files for full or object-level backups, using distinct names for each configuration file. Also, use the same node, backup host, and directory location pairs. Specify different backup directory locations for each database.

> **Note**
> For optimal network performance when creating a backup, Vertica recommends that you give each node in the cluster its own dedicated backup host.

Preparing backup host directories

Before vbr can back up a database, you must prepare the target backup directory. Run vbr with a task type of init to create the necessary manifests for the backup process. You need to perform the init process only once. After that, Vertica maintains the manifests automatically.

Estimating backup host disk requirements

Wherever you plan to save data backups, consider the disk requirements for historical backups at your site. Also, if you use more than one archive, multiple archives potentially require more disk space. Vertica recommends that each backup host have space for at least twice the database node footprint size. Follow this recommendation regardless of the specifics of your site's backup schedule and retention requirements.

To estimate the database size, use the used_bytes column of the storage_containers system table as in the following example:

```
=> SELECT SUM(used_bytes) FROM storage_containers WHERE node_name='v_mydb_node0001';
total_size
------------
 302135743
(1 row)
```

Making backup hosts accessible

You must verify that any firewalls between the source database nodes and the target backup hosts allow connections for SSH and rsync on port 50000.

The backup hosts must be running identical versions of rsync and Python as those supplied in the Vertica installation package.

Setting up passwordless SSH access

For vbr to access a backup host, the database superuser must meet two requirements:

- Have an account on each backup host, with write permissions to the backup directory.
- Have passwordless SSH access from each database cluster host to the corresponding backup host.

How you fulfill these requirements depends on your platform and infrastructure.

SSH access among the backup hosts and access from the backup host to the database node is not necessary.

If your site does not use a centralized login system (such as LDAP), you can usually add a user with the useradd command or through a GUI administration tool. See the documentation for your Linux distribution for details.

If your platform supports it, you can enable passwordless SSH logins using the ssh-copy-id command to copy a database administrator's SSH identity file to the backup location from one of your database nodes. For example, to copy the SSH identity file from a node to a backup host named backup01 :

```
$ ssh-copy-id -i dbadmin@backup01|
Password:
```

Try logging into the machine with "ssh dbadmin@backup01" . Then, check the contents of the ~/.ssh/authorized_keysfile to verify that you have not added extra keys that you did not intend to include.

```
$ ssh backup01
Last login: Mon May 23 11:44:23 2011 from host01
```

Repeat the steps to copy a database administrator's SSH identity to all backup hosts you use to back up your database.

After copying a database administrator's SSH identity, you should be able to log in to the backup host from any of the nodes in the cluster without being prompted for a password.

Increasing the SSH maximum connection settings for a backup host

If your configuration requires backing up multiple nodes to one backup host (n:1), increase the number of concurrent SSH connections to the SSH daemon (sshd). By default, the number of concurrent SSH connections on each host is 10 , as set in the sshd_config file with the MaxStartups keyword. The MaxStartups value for each backup host should be greater than the total number of hosts being backed up to this backup host. For more information on configuring MaxStartups , refer to the man page for that parameter .

See also

- vbr configuration file reference
- Enable secure shell (SSH) logins

Configuring hard-link local backup hosts

When specifying the backupHost parameter for your hard-link local configuration files, use the database host names (or IP addresses) as known to admintools. Do not use the node names. Host names (or IP addresses) are what you used when setting up the cluster. Do not use localhost for the backupHost parameter.

Listing host names

To query node names and host names:

```
=> SELECT node_name, host_name FROM node_resources;
   node_name     |   host_name
-----------------+----------------
 v_vmart_node0001 | 192.168.223.11
 v_vmart_node0002 | 192.168.223.22
 v_vmart_node0003 | 192.168.223.33
(3 rows)
```

Because you are creating a local backup, use square brackets [ ] to map the host to the local host. For more information, refer to [mapping] .

```
[Mapping]
v_vmart_node0001 = []:/home/dbadmin/data/backups
v_vmart_node0002 = []:/home/dbadmin/data/backups
v_vmart_node0003 = []:/home/dbadmin/data/backups
```

Configuring cloud storage backups

Backing up an Enterprise Mode or Eon Mode database to a supported cloud storage location requires that you add parameters to the backup

configuration file. You can create these backups from the local cluster or from your cloud provider's virtual servers. Additional cloud storage configuration is required to configure authentication and encryption.

Configuration file requirements

To back up any Eon Mode or Enterprise Mode cluster to a cloud storage destination, the backup configuration file must include a [CloudStorage] section. Vertica provides a sample cloud storage configuration file that you can copy and edit.

Environment variable requirements

Environment variables securely pass credentials for backup locations. Eon and Enterprise Mode databases require environment variables in the following backup scenarios:

- Vertica on Google Cloud Platform (GCP) to Google Cloud Storage (GCS).
  For backups to GCS, you must have a hash-based message authentication code (HMAC) key that contains an access ID and a secret. See Eon Mode on GCP prerequisites for instructions on how to create your HMAC key.
- On-premises databases to any of the following storage locations:
  - Amazon Web Services (AWS)
  - Any S3-compatible storage
  - Azure Blob Storage (Enterprise Mode only)

  On-premises database backups require you to pass your credentials with environment variables. You cannot use other methods of credentialing with cross-endpoint backups.
- Any Azure user environment that does not manage resources with Azure managed identities.

The vbr log captures when you sent an environment variable. For security purposes, the value that the environment variable represents is not logged. For details about checking vbr logs, see Troubleshooting backup and restore .

Enterprise Mode and Eon Mode

All Enterprise Mode and Eon Mode databases require the following environment variables:

Environment Variable	Description
VBR_BACKUP_STORAGE_ACCESS_KEY_ID	Credentials for the backup location.
VBR_BACKUP_STORAGE_SECRET_ACCESS_KEY	Credentials for the backup location.
VBR_BACKUP_STORAGE_ENDPOINT_URL	The endpoint for the on-premises S3 backup location, includes the scheme HTTP or HTTPS. Important Do not set this variable for backup locations on AWS or GCS.

Eon Mode only

Eon Mode databases require the following environment variables:

Environment Variable	Description
VBR_COMMUNAL_STORAGE_ACCESS_KEY_ID	Credentials for the communal storage location.
VBR_COMMUNAL_STORAGE_SECRET_ACCESS_KEY	Credentials for the communal storage location.
VBR_COMMUNAL_STORAGE_ENDPOINT_URL	The endpoint for the communal storage, includes the scheme HTTP or HTTPS. Important Do not set this variable for backup locations on GCS.

Azure Blob Storage only

If the user environment does not manage resources with Azure-managed identities, you must provide credentials with environment variables. If you set environment variables in an environment that uses Azure-managed identities, credentials set with environment variables take precedence over Azure-managed identity credentials.

You can back up and restore between two separate Azure accounts. Cross-account operations require a credential configuration JSON object and an endpoint configuration JSON object for each account. Each environment variable accepts a collection of one or more comma-separated JSON objects.

Cross-account and cross-region backup and restore operations might result in decreased performance. For details about performance and cost, see the Azure documentation .

The Azure Blob Storage environment variables are described in the following table:

Environment Variable	Description
VbrCredentialConfig	Credentials for the backup location. Each JSON object requires values for the following keys: • accountName : Name of the storage account. • blobEndpoint : Host address and optional port for the endpoint to use as the backup location. • accountKey : Access key for the account. • sharedAccessSignature : A token that provides access to the backup endpoint.
VbrEndpointConfig	The endpoint for the backup location. To backup and restore between two separate Azure accounts, provide each set of endpoint information as a JSON object. Each JSON object requires values for the following keys: • accountName : Name of the storage account. • blobEndpoint : Host address and optional port for the endpoint to use as the backup location. • protocol : HTTPS (default) or HTTP. • isMultiAccountEndpoint : Boolean (by default false), indicates whether blobEndpoint supports multiple accounts

The following commands export the Azure Blob Storage environment variables to the current shell session:

```
$ export VbrCredentialConfig=[{"accountName": "account1","blobEndpoint": "host[:port]","accountKey": "account-key1","sharedAccessSignature": "sas-token1"}]
$ export VbrEndpointConfig=[{"accountName": "account1", "blobEndpoint": "host[:port]", "protocol": "http"}]
```

Additional considerations for cloud storage

If you are backing up to a supported cloud storage location, you need to do some additional one-time configuration. You must also take additional steps if the cluster you are backing up is running on instances in the cloud. For Amazon Web Services (AWS), you might choose to encrypt your backups, which requires additional steps.

By default, bucket access is restricted to the communal storage bucket. For one-time operations with other buckets like backing up and restoring the database, use the appropriate credentials. See Google Cloud Storage parameters and S3 parameters for additional information.

Configuring cloud storage for backups

As with any storage location, you must initialize a cloud storage location with the vbr task init .

Because cloud storage does not support file locking, Vertica uses either your local file system or the cloud storage file system to handle file locks during a backup. You identify this location by setting the cloud_storage_backup_file_system_path parameter in your vbr configuration file. During a backup, Vertica creates a locked identity file on your local or cloud instance, and a duplicate file in your cloud storage backup location. If the files match, Vertica proceeds with the backup, releasing the lock when the backup is complete. As long as the files remain identical, you can use the cloud storage location for backup and restore tasks.

Reinitializing cloud backup storage

If the files in your locking location become out of sync with the files in your backup location, backup and restore tasks fail with an error message. You can resolve locking inconsistencies by rerunning the init task qualified by --cloud-force-init :

```
$ /opt/vertica/bin/vbr --task init --cloud-force-init -c filename.ini
```

> **Note**
> If a backup fails, confirm that your Vertica cluster has permission to access your cloud storage location.

Configuring authentication for Google Cloud Storage

If you are backing up to Google Cloud Storage (GCS) from a Google Cloud Platform-based cluster, you must provide authentication to the GCS communal storage location. Set the environment variables as detailed in Configuring cloud storage backups to authenticate to GCS storage.

See Eon Mode on GCP prerequisites for additional authentication information, including how to create your hash-based message authentication code (HMAC) key.

Configuring EC2 authentication for Amazon S3

If you are backing up to S3 from an EC2-based cluster, you must provide authentication to your S3 host. Regardless of the authentication type you choose, your credentials do not leave your EC2 cluster. Vertica supports the following authentication types:

- AWS credential file
- Environment variables
- IAM role

AWS credential file - You can manually create a configuration file on your EC2 initiator host at ~/.aws/credentials.

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY
aws_secret_access_key = YOUR_SECRET_KEY
```

For more information on credential files, refer to Amazon Web Services documentation .

Environment variables - Amazon Web Services provides the following environment variables:

- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY

Use these variables on your initiator to provide authentication to your S3 host. When your session ends, AWS deletes these variables. For more information, refer to the AWS documentation .

IAM role - Create an AWS IAM role and grant that role permission to access your EC2 cluster and S3 resources. This method is recommended for managing long-term access. For more information, refer to Amazon Web Services documentation .

Encrypting backups on Amazon S3

Backups made to Amazon S3 can be encrypted using native server-side S3 encryption capability. For more information on Amazon S3 encryption, refer to Amazon documentation .

> **Note**
> Vertica supports server-side encryption only. Client-side encryption is not supported.

Vertica supports the following forms of S3 encryption:

- Server-Side Encryption with Amazon S3-Managed Keys (SSE-S3)
  - Encrypts backups with AES-256
  - Amazon manages encryption keys
- Server-Side Encryption with AWS KMS-Managed Keys (SSE-KMS)
  - Encrypts backups with AES-256
  - Requires an encryption key from Amazon Key Management Service
  - Your S3 bucket must be from the same region as your encryption key
  - Allows auditing of user activity

When you enable encryption of your backups, Vertica encrypts backups as it creates them. If you enable encryption after creating an initial backup, only increments added after you enabled encryption are encrypted. To ensure that your backup is entirely encrypted, create new backups after enabling encryption.

To enable encryption, add the following settings to your configuration file:

- cloud_storage_encrypt_transport: Encrypts your backups during transmission. You must enable this parameter if you are using SSE-KMS encryption.
- cloud_storage_encrypt_at_rest: Enables encryption of your backups. If you enable encryption and do not provide a KMS key, Vertica uses SSE-S3 encryption.
- cloud_storage_sse_kms_key_id: If you are using KMS encryption, use this parameter to provide your key ID.

See [CloudStorage] for more information on these settings.

The following example shows a typical configuration for KMS encryption of backups.

```
[CloudStorage]
cloud_storage_encrypt_transport = True
cloud_storage_encrypt_at_rest = sse
cloud_storage_sse_kms_key_id = 6785f412-1234-4321-8888-6a774ba2aaaa
```

Configuring backups to and from HDFS

Eon Mode only

To back up an Eon Mode database that uses HDFS on-premises storage, the communal storage and backup location must use the same HDFS credentials and domain. All vbr operations are supported, except copycluster.

Vertica supports Kerberos authentication, High Availability Name Node, and TLS (wire encryption) for vbr operations.

Creating a cloud storage configuration file

To back up Eon Mode on-premises with communal storage on HDFS, you must provide a backup configuration file . In the [CloudStorage] section, provide the cloud_storage_backup_path and cloud_storage_backup_file_system_path values.

If you use Kerberos authentication or High Availability NameNode with your Hadoop cluster, the vbr utility requires access to the same values set in the bootstrapping file that you created during the database install . Include these values in the [misc] section of the backup file.

The following table maps the vbr configuration option to its associated bootstrap file parameter:

vbr Configuration Option	**Bootstrap File Parameter**
kerberos_service_name	KerberosServiceName
kerberos_realm	KerberosRealm
kerberos_keytab_file	KerberosKeytabFile
hadoop_conf_dir	HadoopConfDir

For example, if KerberosServiceName is set to *principal-name* in the bootstrap file, set kerberos_service_name to *principal-name* in the [Misc] section of your configuration file.

Encryption between communal storage and backup locations

Vertica supports vbr operations using wire encryption between your communal storage and backup locations. Use the cloud_storage_encrypt_transport parameter in the [CloudStorage] section of your backup configuration file to configure encryption.

To enable encryption:

- Set cloud_storage_encrypt_transport to true .
- Use the swebhdfs:// protocol for cloud_storage_backup_path.

If you do not use encryption:

- Set cloud_storage_encrypt_transport to false .
- Use the webhdfs:// protocol for cloud_storage_backup_path.

Vertica does not support at-rest encryption for Hadoop storage.

Creating backups

Important

Inadequate security on backups can compromise overall database security. Be sure to secure backup locations and strictly limit access to backups only to users who already have permissions to access all database data.

You should perform full backups of your database regularly. You should also perform a full backup under the following circumstances:

Before...

- Upgrading Vertica to another release.
- Dropping a partition.
- Adding, removing, or replacing nodes in the database cluster.

After...

- Loading a large volume of data.
- Adding, removing, or replacing nodes in the database cluster.
- Recovering a cluster from a crash.

If...

- The epoch of the latest backup predates the current ancient history mark.

Ideally, schedule ongoing backups to back up your data. You can run the Vertica vbr from a cron job or other task scheduler.

You can also back up selected objects. Use object backups to supplement full backups, not to replace them. Backup types are described in Types of backups .

Running vbr does not affect active database applications. vbr supports creating backups while concurrently running applications that execute DML statements, including COPY, INSERT, UPDATE, DELETE, and SELECT.

Backup locations and contents

Full and object-level backups reside on *backup hosts* , the computer systems on which backups and archives are stored.

Vertica saves backups in a specific *backup location* , the directory on a backup host. This location can contain multiple backups, both full and object-level, including associated archives. The backups are also compatible, allowing you to restore any objects from a full database backup. Backup locations for Eon Mode databases must be on S3.

> **Note**
> Vertica does not recommend concurrent backups. If you must run multiple backups concurrently, use separate backup and temp directories for each. Having separate backup directories detracts from the advantage of sharing data among historical backups.

Before beginning a backup, you must prepare your backup locations using the vbr init task , as in the following example:

```
$ vbr -t init -c full_backup.ini
```

For more information about backup locations, see Setting up backup locations .

Backups contain all committed data for the backed-up objects as of the start time of the backup. Backups do not contain uncommitted data or data committed during the backup. Backups do not delay mergeout or load activity.

Backing up HDFS storage locations

If your Vertica cluster uses HDFS storage locations, you must do some additional configuration before you can perform backups. See Requirements for backing up and restoring HDFS storage locations .

HDFS storage locations support only full backup and restore. You cannot perform object backup or restore on a cluster that uses HDFS storage locations.

Impact of backups on Vertica nodes

While a backup is taking place, the backup process can consume additional storage. The amount of space consumed depends on the size of your catalog and any objects that you drop during the backup. The backup process releases this storage when the backup is complete.

Best practices for creating backups

When creating backup configuration files:

- Create separate configuration files to create full and object-level backups.
- Use a unique snapshot name in each configuration file.
- Use the same backup host directory location for both kinds of backups:
  - Because the backups share disk space, they are compatible when performing a restore.
  - Each cluster node must also use the same directory location on its designated backup host.
- For best network performance, use one backup host per cluster node.

- Use one directory on each backup node to store successive backups.
- For future reference, append the major Vertica version number to the configuration file name (mybackup *9x*).

The selected objects of a backup can include one or more schemas or tables, or a combination of both. For example, you can include schema S1 and tables T1 and T2 in an object-level backup. Multiple backups can be combined into a single backup. A schema-level backup can be integrated with a database backup (and a table backup integrated with a schema-level backup, and so on).

In this section

- Types of backups
- Creating full backups
- Creating object-level backups
- Creating hard-link local backups
- Incremental or repeated backups

Types of backups

vbr supports the following kinds of backups:

- Full backups
- Object-level backups
- Hard-link local backups

The vbr configuration file includes the snapshotName parameter. Use different snapshot names for different types of backups, including different combinations of objects in object-level backups. Backups with the same snapshot name form a time sequence limited by restorePointLimit ,. Avoid giving all backups the same snapshot name; otherwise, they eventually interfere with each other.

Full backups

A full backup is a complete copy of the database catalog, its schemas, tables, and other objects. This type of backup provides a consistent image of the database at the time the backup occurred. You can use a full backup for disaster recovery to restore a damaged or incomplete database. You can also restore individual objects from a full backup .

When a full backup already exists, vbr performs incremental backups, whose scope is confined to data that is new or changed since the last full backup occurred. You can specify the number of historical backups to keep.

Archives contain a collection of same-name backups. Each archive can have a different retention policy. For example, TBak might be the name of an object-level backup of table T . If you create a daily backup each week, the seven backups of a given week become part of the TBak archive. Keeping a backup archive lets you revert back to any one of the saved backups.

Object-level backups

An object-level backup consists of one or more schemas or tables or a group of such objects. The conglomerate parts of the object-level backup do not contain the entire database. When an object-level backup exists, you can restore all of its contents or individual objects .

> **Note**
> Object-level backups are not supported for Enterprise Mode databases that use a Hadoop File System (HDFS) storage location.

Object-level backups contain the following object types:

Object Type	Description
Selected objects	Objects you choose to be part of an object-level backup. For example, if you specify tables T1 and T2 to include in an object-level backup, they are the selected objects.
Dependent objects	Objects that must be included as part of an object-level backup, due to dependencies. Suppose you want to create an object-level backup that includes a table with a foreign key. To do so, table constraints require that you include the primary key table, and vbr enforces this requirement. Projections anchored on a table in the selected objects are also dependent objects.
Principal objects	The objects on which both selected and dependent objects depend are called *principal objects* . For example, each table and projection has an owner, and each is a *principal object* .

Hard-link local backups

Valid only for Enterprise Mode, hard-link local backups are saved directly on the database nodes, and can be performed on the entire database or specific objects. Typically you use this kind of backup temporarily before performing a disruptive operation. Do not rely on this kind of backup for long-term use; it cannot protect you from node failures because data and backups are on the same nodes.

A checkpoint backup is a hard-link local backup that comprises a complete copy of the database catalog, and a set of hard file links to corresponding data files. You must save a hard-link local backup on the same file system that is used by the catalog and database files.

Creating full backups

Before you create a database backup, verify the following:

- You have prepared your backup directory with the vbr init task :
  ```
  $ vbr -t init -c full_backup.ini
  ```
- Your database is running. It is unnecessary for all nodes to be up in a K-safe database. However, any nodes that are DOWN are not backed up.
- All of the backup hosts are up and available.
- The backup host (either on the database cluster or elsewhere) has sufficient disk space to store the backups.
- The user account of the user who starts vbr has write access to the target directories on the host backup location. This user can be dbadmin or another assigned role. However, you cannot run vbr as root.
- Each backup has a unique file name.
- If you want to keep earlier backups, restorePointLimit is set to a number greater than 1 in the configuration file.
- If you are backing up an Eon Mode database, you have met the Eon Mode database requirements .

Run vbr from a terminal. Use the database administrator account from an initiator node in your database cluster. The command requires only the --task backup and --config-file arguments (or their short forms, -t and -c).

If your configuration file does not contain the database administrator password, vbr prompts you to enter the password. It does not display what you type.

vbr requires no further interaction after you invoke it.

The following example shows a full backup:

```
$ vbr -t backup -c full_backup.ini
Starting backup of database VTDB.
Participating nodes: v_vmart_node0001, v_vmart_node0002, v_vmart_node0003, v_vmart_node0004.
Snapshotting database.
Snapshot complete.
Approximate bytes to copy: 2315056043 of 2356089422 total.
[==================================================] 100%
Copying backup metadata.
Finalizing backup.
Backup complete!
```

By default, no output is displayed, other than the progress bar. To include additional progress information, use the --debug option, with a value of 1, 2, or 3.

Creating object-level backups

Use object-level backups to back up individual schemas or tables. Object-level backups are especially useful for multi-tenanted database sites. For example, an international airport could use a multi-tenanted database to represent different airlines in its schemas. Then, tables could maintain various types of information for the airline, including ARRIVALS, DEPARTURES, and PASSENGER information. With such an organization, creating object-level backups of the specific schemas would let you restore by airline tenant, or any other important data segment.

To create one or more object-level backups, create a configuration file specifying the backup location, the object-level backup name, and a list of objects to include (one or more schemas and tables). You can use the includeObjects and excludeObjects parameters together with wildcards to specify the objects of interest. For more information about specifying the objects to include, see Including and excluding objects .

For more information about configuration files for full or object-level backups, see Sample vbr configuration files and vbr configuration file reference .

While not required, Vertica recommends that you first create a full backup before creating any object-level backups.

> **Note**

Apache Kafka uses internal configuration settings to maintain the integrity of your data. When backing up your Kafka data, Vertica recommends that you perform a full database backup rather than an object-level backup.

Performing the backup

Before you can create a backup, you must prepare your backup directory with the vbr -init task . You must also create a configuration file specifying which objects to back up.

Run vbr from a terminal using the database administrator account from a node in your database cluster. You cannot run vbr as root.

You can create an object-level backup as in the following example.

```
$ vbr --task backup --config-file objectbak.ini
Preparing...
Found Database port:  5433
Copying...
[==================================================] 100%
All child processes terminated successfully.
Committing changes on all backup sites...
backup done!
```

Naming conventions

Give each object-level backup configuration file a distinct and descriptive name. For instance, at an airport terminal, schema-based backup configuration files use a naming convention with an airline prefix, followed by further description, such as:

AIR1_daily_arrivals_backup

AIR2_hourly_arrivals_backup

AIR2_hourly_departures_backup

AIR3_daily_departures_backup

When database and object-level backups exist, you can recover the backup of your choice.

Caution
Do not change object names in an object-level configuration file if a backup already exists. Doing so overwrites the original configuration file, and you cannot restore it from the earlier backup. Instead, create a different configuration file.

Understanding object-level backup contents

Object-level backups comprise only the elements necessary to restore the schema or table, including the selected, dependent, and principal objects. An object-level backup includes the following contents:

- Storage: Data files belonging to any specified objects
- Metadata: Including the cluster topology, timestamp, epoch, AHM, and so on
- Catalog snippet: Persistent catalog objects serialized into the principal and dependent objects

Some of the elements that AIR2 comprises, for instance, are its parent schema, tables, named sequences, primary key and foreign key constraints, and so on. To create such a backup, vbr saves the objects directly associated with the table. It also saves any dependencies, such as foreign key (FK) tables, and creates an object map from which to restore the backup.

Note
Because the data in local temp tables persists only within a session, local temporary tables are excluded when you create an object-level backup. For global temporary tables, vbr stores the table's definition.

Making changes after an object-level backup

Be aware how changes made after an object-level backup affect subsequent backups. Suppose you create an object-level backup and later drop schemas and tables from the database. In this case, the objects you dropped are also dropped from subsequent backups. If you do not save an archive of the object backup, such objects could be lost permanently.

Changing a table name after creating a table backup does not persist after restoring the backup. Suppose that, after creating a backup, you drop a user who owns any selected or dependent objects in that backup. In this case, restoring the backup re-creates the object and assigns ownership to the user performing the restore. If the owner of a restored object still exists, that user retains ownership of the restored object.

To restore a dropped table from a backup:

1. Rename the newly created table from t1 to t2.
2. Restore the backup containing t1.
3. Restore t1. Tables t1 and t2 now coexist.

For information on how Vertica handles object overwrites, refer to the objectRestoreMode parameter in [misc] .

K-safety can increase after an object backup. Restoration of a backup fails if *both* of the following conditions occur:

- An increase in K-safety occurs.
- Any table in the backup has insufficient projections.

Changing principal and dependent objects

If you create a backup and then drop a principal object, restoring the backup restores that principal object. If the owner of the restored object has also been dropped, Vertica assigns the restored object to the current dbadmin.

You can specify how Vertica handles object overwrites in the vbr configuration file. For more information, refer to the objectRestoreMode parameter in [misc] .

IDENTITY sequences are dependent objects because they cannot exist without their tables. An object-level backup includes such objects, along with the tables on which they depend.

Named sequences are not dependent objects because they exist autonomously. A named sequence remains after you drop the table in which the sequence is used. In this case, the named sequence is a principal object. Thus, you must back up the named sequence with the table. Then you can regenerate it, if it does not already exist when you restore the table. If the sequence does exist, vbr uses it, unmodified. Sequence values could repeat, if you restore the full database and then restore a table backup to a newer epoch.

Considering constraint references

When database objects are related through constraints, you must back them up together. For example, a schema with tables whose constraints reference only tables in the same schema can be backed up. However, a schema containing a table with an FK/PK constraint on a table in another schema cannot. To back up the second table, you must include the other schema in the list of selected objects.

Configuration files for object-level backups

vbr automatically associates configurations with different backup names but uses the same backup location.

Always create a cluster-wide configuration file and one or more object-level configuration files pointing to the same backup location. Storage between backups is shared, preventing multiple copies of the same data. For object-level backups, using the same backup location causes vbr to encounter fewer OID conflict prevention techniques. Avoiding OID conflict prevention results in fewer problems when restoring the backup.

When using cluster and object configuration files with the same backup location, vbr includes additional provisions to ensure that the object-level backups can be used following a full cluster restore. One approach to restoring a full cluster is to use a full database backup to bootstrap the cluster. After the cluster is operational again, you can restore the most recent object-level backups for schemas and tables.

Attempting to restore a full database using an object-level configuration file fails, resulting in this error:

```
VMart=> /tmp/vbr --config-file=Table2.ini -t restore
Preparing...
Invalid metadata file. Cannot restore.
restore failed!
```

See Restoring all objects from an object-level backup for more information.

Backup epochs

Each backup includes the epoch to which its contents can be restored. When vbr restores data, Vertica updates to the current epoch.

vbr attempts to create an object-level backup five times before an error occurs and the backup fails.

Creating hard-link local backups

You can use the hardLinkLocal option to create a full or object-level backup with hard file links on a local database host.

Creating hard-link local backups can provide the following advantages over a remote host backup:

- **Speed** : A hard-link local backup is significantly faster than a remote host backup. When backing up, vbr does not copy files if the backup directory exists on the same file system as the database directory.
- **Reduced network activities** : The hard-link local backup minimizes network load because it does not require rsync to copy files to a remote backup host.
- **Less disk space** : The backup includes a copy of the catalog and hard file links. Therefore, the local backup uses significantly less disk space than a backup with copies of database data files. However, a hard-link local backup saves a full copy of the catalog each time you run vbr . Thus, the disk size increases with the catalog size over time.

Hard-link local backups can help you during experimental designs and development cycles. Database designers and developers can create hard-link local object backups of schemas and tables on a regular schedule during design and development phases. If any new developments are unsuccessful, developers can restore one or more objects from the backup.

Planning hard-link local backups

If you plan to use hard-link local backups as a standard site procedure, design your database and hardware configuration appropriately. Consider storing all of the data files on one file system per node. Such a configuration has the advantage of being set up automatically for hard-link local backups.

Specifying backup directory locations

The backupDir parameter of the configuration file specifies the location of the top-level backup directory. Hard-link local backups require that the backup directory be located on the same Linux file system as the database data. The Linux operating system cannot create hard file links to another file system.

Do not create the hard-link local backup directory in a database data storage location. For example, as a best practice, the database data directory should not be at the top level of the file system, as it is in the following example:

```
/home/dbadmin/data/VMart/v_vmart_node0001
```

Instead, Vertica recommends adding another subdirectory for data above the database level, such as in this example:

```
/home/dbadmin/data/dbdata/VMart/v_vmart_node0001
```

You can then create the hard-link local backups subdirectory as a peer of the data directory you just created, such as in this example:

```
/home/dbadmin/data/backups
/home/dbadmin/data/dbdata
```

When you specify the hard-link backup location, be sure to avoid these common errors when adding the hardLinkLocal=True parameter to the configuration file:

If ...	Then...	Solution
You specify a backup directory on a *different* node	vbr issues an error message and aborts the backup.	Change the configuration file to include a backup directory on the same host and file system as the database files. Then, run vbr again.
You specify a backup location on the same node, but a backup destination directory on a *different* file system from the database and catalog files.	vbr issues a warning message and performs the backup by copying (not linking) the files from one file system to the other.	No action required, but copying consumes more disk space and takes longer than linking.

Creating the backup

Before creating a full hard-link local database backup of an Enterprise Mode database, verify the following:

- Your database is running. All nodes need not be up in a K-safe database for vbr to run. However, be aware that any nodes that are DOWN are not backed up.
- The user account that starts vbr (dbadmin or other) has write access to the target backup directories.

Hard-link backups are not supported in Eon Mode.

When you create a full or object-level hard link local backup, that backup contains the following:

Backup	Catalog	Database files
Full backup	Full copy	Hard file links to all database files
Object-level backup	Full copy	Hard file links for all objects listed in the configuration file, and any of their dependent objects

Run the vbr script from a terminal using the database administrator account from a node in your database cluster. You cannot run vbr as root.

Hard-link backups use the same vbr arguments as other backups. Configuring a backup as a hard-link backup is done entirely in the configuration file. The following example shows the syntax:

```
$ vbr --task backup --config fullbak.ini
```

Creating hard-link local backups for external media storage

You can use hard-link local backups as a staging mechanism to back up to tape or other forms of storage media. The following steps present a simplified approach to saving, and then restoring, hard-link local backups from tape storage:

1. Create a configuration file by copying an existing one or one of the samples described in Sample vbr configuration files .
2. Edit the configuration file (localbak.ini in this example) to include the hardLinkLocal=True parameter in the [Transmission] section.
3. Run vbr with the configuration file:
   ```
   $ vbr --task backup --config-file localbak.ini
   ```
4. Copy the hard-link local backup directory with a separate process (not vbr) to tape or other external media.
5. If the database becomes corrupted, transfer the backup files from tape to their original backup directory and restore as explained in Restoring hard-link local backups .

> **Note**
> Vertica recommends that you preserve the directory containing the hard-link backup after copying it to other media. If you delete the directory and later copy the files back from external media, the copied files will no longer be links. Instead, they will use as much disk space as if you had done a full (not hard-link) backup.

Restoring hard-link local backups requires some additional (manual) steps. Do not use them as a substitute for regular full backups (Creating full backups).

Hard-link local backups and disaster recovery

Hard-link local backups are only as reliable as the disk on which they are stored. If the local disk becomes corrupt, so does the hard-link local backup. In this case, you are unable to restore the database from the hard-link local backup because it is also corrupt.

All sites should maintain full backups externally for disaster recovery because hard-link local backups do not actually copy any database files.

Incremental or repeated backups

As a best practice, Vertica recommends that you take frequent backups if database contents diverge in significant ways. Always take backups after any event that significantly modifies the database, such as performing a rebalance. Mixing many backups with significant differences can weaken data K-safety. For example, taking backups both before and after a rebalance is not a recommended practice in cases where the backups are all part of one archive.

Each time you back up your database with the same configuration file, vbr creates an additional backup and might remove the oldest backup. The backup operation copies new storage containers, which can include:

- Data that existed the last time you performed a database backup
- New and changed data since the last full backup

Use the restorePointLimit parameter in the configuration file to increase the number of stored backups. If a backup task would cause this limit to be exceeded, vbr deletes the oldest backup after a successful backup.

When you run a backup task, vbr first creates the new backup in the specified location, which might temporarily exceed the limit. It then checks whether the number of backups exceeds the value of restorePointLimit , and, if necessary, deletes the oldest backups until only restorePointLimit remain. If the requested backup fails or is interrupted, vbr does not delete any backups.

When you restore a database, you can choose to restore from any retained backup rather than the most recent, so raise the limit if you expect to need access to older backups.

Restoring backups

You can use the vbr restore task to restore your full database or selected objects from backups created by vbr . Typically you use the same configuration file for both operations. The minimal restore command is:

```
$ vbr --task restore --config-file config-file
```

You must log in using the database administrator's account (not root).

For full restores, the database must be DOWN. For object restores, the database must be UP.

Usually you restore to the cluster that you backed up, but you can also restore to an alternate cluster if the original one is no longer available.

Restoring must be done on the same architecture as the backup from which you are restoring. You cannot back up an Enterprise Mode database and restore it in Eon Mode or vice versa.

You can perform restore tasks on Permanent node types. You cannot restore data on Ephemeral, Execute, or Standby nodes. To restore or replicate to these nodes, you must first change the destination node type to PERMANENT. For more information, refer to Setting node type .

Restoring objects to a higher Vertica version

Vertica supports restoration to a database that is no more than one minor version higher than the current database version. For example, you can restore objects from a 12.0.x database to a 12.1.x database.

If restored objects require a UDx library that is not present in the later-version database, Vertica displays the following error:

```
ERROR 2858:  Could not find function definition
```

You can resolve this issue by installing compatible libraries in the target database.

Restoring HDFS storage locations

If your Vertica cluster uses HDFS storage locations, you must do some additional configuration before you can restore. See Requirements for backing up and restoring HDFS storage locations .

HDFS storage locations support only full backup and restore. You cannot perform object backup or restore on a cluster that uses HDFS storage locations.

In this section

- Restoring a database from a full backup
- Restoring a database to an alternate cluster
- Restoring all objects from an object-level backup
- Restoring individual objects
- Restoring objects to an alternate cluster
- Restoring hard-link local backups
- Ownership of restored objects

Restoring a database from a full backup

You can restore a full database backup to the database that was backed up, or to an alternate cluster with the same architecture. One reason to restore to an alternate cluster is to set up a test cluster to investigate a problem in your production cluster.

To restore a full database backup, you must verify that:

- Database is DOWN. You cannot restore a full backup when the database is running.
- All backup hosts are available.
- Backup directory exists and contains backups of the data to restore.
- Cluster to which you are restoring the backup has:
  - Same number of nodes as used to create the backup (Enterprise Mode), or at least as many nodes as the primary subclustes (Eon Mode)
  - Same architecture as the one used to create the backup
  - Identical node names
- Target database already exists on the cluster where you are restoring data.
  - Database can be completely empty, without any data or schema.
  - Database name must match the name in the backup

    - All node names in the database must match the names of the nodes in the configuration file.
- The user performing the restore is the database administrator.
- If you are restoring an Eon Mode database, you have met the Eon Mode database requirements .

You can use only a full database backup to restore a complete database. If you have saved multiple backup archives, you can restore from either the last backup or a specific archive.

Restoring from a full database backup injects the OIDs from each backup into the restored catalog of the full database backup. The catalog also receives all archives. Additionally, the OID generator and current epoch are set to the current epoch.

You can also restore a full backup to a different database than the one you backed up. See Restoring a database to an alternate cluster .

Important
When you restore an Eon Mode database to another database, the restore operation copies the source database's communal storage. The original communal storage is unaffected.

Restoring the most recent backup

Usually, when a node or cluster is DOWN, you want to return the cluster to its most-recent state. Doing so requires restoring a full database backup. You can restore any full database backup from the archive by identifying the name in the configuration file.

To restore from the most recent backup, use the vbr restore task with the configuration file. If your password configuration file does not contain the database superuser password, vbr prompts you to enter it.

The following example shows how you can use the db.ini configuration file for restoration:

```
> vbr --task restore --config-file db.ini
Copying...
1871652633 out of 1871652633, 100%
All child processes terminated successfully.
restore done!
```

Restoring an archive

If you saved multiple backups, you can specify an archive to restore. To list the archives that exist to choose one to restore, use the vbr --listbackup task, with a specific configuration file. See Viewing backups .

To restore from an archive, add the --archive parameter to the command line. The value is the *date_timestamp* suffix of the directory name that identifies the archive to restore. For example:

```
$ vbr --task restore --config-file fullbak.ini --archive=20121111_205841
```

The --archive parameter identifies the archive created on 11-11-2012 (_archive20121111), at time 205841 (20:58:41). You need specify only the _archive suffix, because the configuration file identifies the backup name of the subdirectory, and the OID identifier indicates the backup is an archive.

Restore failures in Eon Mode

When a restore operation fails, vbr can leave extra files in the communal storage location. If you use communal storage in the cloud, those extra files cost you money. To remove them, restart the database and call CLEAN_COMMUNAL_STORAGE with an argument of true.

Restoring a database to an alternate cluster

Vertica supports restoring a full backup to an alternate cluster.

Requirements

The process is similar to the process for Restoring a database from a full backup , with the following additional requirements.

The destination database must:

- Be DOWN.
- Share the same name as the source database.
- Have the same number of nodes as the source database.
- Have the same names as the source nodes.
- Use the same catalog directory location as the source database.
- Use the same port numbers as the source database.

Procedure

1. Copy the vbr configuration file that you used to create the backup to any node on the destination cluster.
2. If you are using a stored password , copy the password configuration file to the same location as the vbr configuration file.
3. From the destination node, issue a vbr restore command, such as:

```
$ vbr -t restore -c full.ini
```

4. After the restore has completed, start the restored database .

Restoring all objects from an object-level backup

To restore everything in an object-level backup to the database from which it was taken, use the vbr restore task with the configuration file you used to create the backup, as in the following example:

```
$ vbr --task restore --config-file MySchema.ini
Copying...
1871652633 out of 1871652633, 100%
All child processes terminated successfully.
restore done!
```

The database must be UP.

You can specify how Vertica reacts to duplicate objects by setting the objectRestoreMode parameter in the configuration file.

Object-level backup and restore are not supported for HDFS storage locations.

Restoring objects to a changed cluster

Unlike restoring from a full database backup, vbr supports restoring object-level backups after adding nodes to the cluster. Any nodes that were not in the cluster when you created the object-level backup do not participate in the restore. You can rebalance your cluster after the restore to distribute data among the new nodes.

You cannot restore an object-level backup after removing nodes, altering node names, or changing IP addresses. Trying to restore an object-level backup after such changes causes vbr to fail and display this message:

```
Preparing...
Topology changed after backup; cannot restore.
restore failed!
```

Projection epoch after restore

All object-level backup and restore events are treated as DDL events. If a table does not participate in an object-level backup, possibly because a node is down, restoring the backup affects the projection in the following ways:

- Its epoch is reset to 0.
- It must recover any data that it does not have by comparing epochs and other recovery procedures.

Catalog locks during restore

As with other databases, Vertica transactions follow strict locking protocols to maintain data integrity.

When restoring an object-level backup into a cluster that is UP, vbr begins by copying data and managing storage containers. If necessary, vbr splits the containers. This process does not require any database locks.

After completing data-copying tasks, vbr first requires a table object lock (O-lock) and then a global catalog lock (GCLX).

In some circumstances, other database operations, such as DML statements, are in progress when the process attempts to get an O-lock on the table. In such cases, vbr is blocked from progress until the DML statement completes and releases the lock. After securing an O-lock first, and then a GCLX lock, vbr blocks other operations that require a lock on the same table.

While vbr holds its locks, concurrent table modifications are blocked. Database system operations, such as the Tuple Mover (TM) transferring data from memory to disk, are canceled to permit the object-level restore to complete.

Catalog restore events

Each object-level backup includes a section of the database catalog, or a *snippet* . A snippet contains the selected objects, their dependent objects, and principal objects. A catalog snippet is similar in structure to the database catalog but consists of a subset representing the object information. Objects being restored can be read from the catalog snippet and used to update both global and local catalogs.

Each object from a restored backup is updated in the catalog. If the object no longer exists, vbr drops the object from the catalog. Any dependent objects that are not in the backup are also dropped from the catalog.

vbr uses existing dependency verification methods to check the catalog and adds a restore event to the catalog for each restored table. That event also includes the epoch at which the event occurred. If a node misses the restore table event, it recovers projections anchored on the given table.

Reverting object DDL changes

If you restore the database to an epoch that precedes changes to an object's DDL, the restore operation reverts the object to its earlier definition. For example, if you change a table column's data type from CHAR(8) to CHAR(16) in epoch 10, and then restore the database from epoch 5, the column reverts to CHAR(8) data type.

Restoring objects to a higher Vertica version

Vertica supports restoration to a database that is no more than one minor version higher than the current database version. For example, you can restore objects from a 12.0.x database to a 12.1.x database.

If restored objects require a UDx library that is not present in the later-version database, Vertica displays the following error:

```
ERROR 2858:  Could not find function definition
```

You can resolve this issue by installing compatible libraries in the target database.

Catalog size limitations

Object-level restores can fail if your catalog size is greater than five percent of the total memory available in the node performing the restore. In this situation, Vertica recommends restoring individual objects from the backup. For more information, refer to Restoring individual objects .

See also

- Failure recovery
- Transactions

Restoring individual objects

You can use vbr to restore individual tables and schemas from a full or object-level backup: qualify the restore task with --restore-objects , and specify the objects to restore as a comma-delimited list:

```
$ vbr --task restore --config-file=filename --restore-objects='objectname[,...]' [--archive=archive-id]
```

The following requirements and restrictions apply:

- The database must be running, and nodes must be UP.
- Tables must include their schema names.
- Do not embed spaces before or after comma delimiters of the --restore-objects list; otherwise, vbr interprets the space as part of the object name.
- Object-level restore is not supported for HDFS storage locations. To restore an HDFS storage location you must do a full restore.

If the schema has a disk quota and restoring the table would exceed the quota, the operation fails.

By default, --restore-objects restores the specified objects from the most recent backup. You can restore from an earlier backup with the --archive parameter.

The following example uses the db.ini configuration file, which includes the database administrator's password:

```
> vbr --task restore --config-file=db.ini --restore-objects=salesschema,public.sales_table,public.customer_info
Preparing...
Found Database port:  5433
Copying...
[==================================================] 100%
All child processes terminated successfully.
All extract object child processes terminated successfully.
Copying...
[==================================================] 100%
All child processes terminated successfully.
restore done!
```

Object dependencies

When you restore an object, Vertica does not always restore dependent objects. For example, if you restore a schema containing views, Vertica does not automatically restore the tables of those views. One exception applies: if database tables are linked through foreign keys, you must restore them together, unless drop_foreign_constraints is set in the vbr configuration file to true.

Note
You must also set objectRestoreMode to coexist , otherwise Vertica ignores `drop_foreign_constraints` .

Duplicate objects

You can specify how restore operations handle duplicate objects by configuring objectRestoreMode . By default, it is set to createOrReplace , so if a duplicate object exists, the restore operation overwrites it with the archived version.

Eon Mode considerations

Restoring objects to an Eon Mode database can leave unneeded files in cloud storage. These files have no effect on database performance or data integrity. However, they can incur extra cloud storage expenses. To remove these files, restart the database and call CLEAN_COMMUNAL_STORAGE with an argument of true.

See also

- Monitoring recovery
- Viewing backups
- Restoring a database from a full backup
- Restoring all objects from an object-level backup
- Ownership of restored objects
- Including and excluding objects

Restoring objects to an alternate cluster

You can use the restore task to copy objects from one database to another. You might do this to "promote" tables from a development environment to a production environment, for example. All restrictions described in Restoring individual objects apply when restoring to an alternate cluster.

To restore to an alternate database, you must make changes to a copy of the configuration file that was used to create the backup. The changes are in the [Mapping] and [NodeMapping] sections. Essentially, you create a configuration file for the restore operation that looks to vbr like a backup of the target database, but it actually describes the backup from the source database. See Restore object from backup to an alternate cluster for an example configuration file.

The following example uses two databases, named source and target. The source database contains a table named sales. The following source_snapshot.ini configuration file is used to back up the source database:

```
[Misc]
snapshotName = source_snapshot
restorePointLimit = 2
objectRestoreMode = createOrReplace

[Database]
dbName = source
dbUser = dbadmin
dbPromptForPassword = True

[Transmission]

[Mapping]
v_source_node0001 = 192.168.50.168:/home/dbadmin/backups/
```

The target_snapshot.ini file starts as a copy of source_snapshot.ini. Because the [Mapping] section describes the database that vbr operates on, we must change the node names to point to the target nodes. We must also add the [NodeMapping] section and change the database name:

```
[Misc]
snapshotName = source_snapshot
restorePointLimit = 2
objectRestoreMode = createOrReplace

[Database]
dbName = target
dbUser = dbadmin
dbPromptForPassword = True

[Transmission]

[Mapping]
v_target_node0001 = 192.168.50.151:/home/dbadmin/backups/
[NodeMapping]
v_source_node0001 = v_target_node0001
```

As far as vbr is concerned, we are restoring objects from a backup of the target database. In reality, we are restoring from the source database.

The following command restores the sales table from the source backup into the target database:

```
$ vbr --task restore --config-file target_snapshot.ini --restore-objects sales
Starting object restore of database target.
Participating nodes: v_target_node0001.
Objects to restore: sales.
Enter vertica password:
Restoring from restore point: source_snapshot_20160204_191920
Loading snapshot catalog from backup.
Extracting objects from catalog.
Syncing data from backup to cluster nodes.
[==================================================] 100%
Finalizing restore.
Restore complete!
```

Restoring hard-link local backups

You restore from hard-link local backups the same way that you restore from full backups, using the restore task. If you used hard-link local backups to back up to external media, you need to take some additional steps.

Transferring backups to and from remote storage

When a full hard-link local backup exists, you can transfer the backup to other storage media, such as tape or a locally-mounted NFS directory. Transferring hard-link local backups to other storage media may copy the data files associated with the hard file links.

You can use a different directory when you return the backup files to the hard-link local backup host. However, you must also change the backupDir parameter value in the configuration file before restoring the backup.

Complete the following steps to restore hard-link local backups from external media:

1. If the original backup directory no longer exists on one or more local backup host nodes, re-create the directory.

   The directory structure into which you restore hard-link backup files must be identical to what existed when the backup was created. For example, if you created hard-link local backups at the following backup directory, you can then re-create that directory structure:

   ```
   /home/dbadmin/backups/localbak
   ```

2. Copy the backup files to their original backup directory, as specified for each node in the configuration file. For more information, refer to [Mapping] .
3. Restore the backup, using one of three options:
   1. To restore the latest version of the backup, move the backup files to the following directory:

      ```
      /home/dbadmin/backups/localbak/node_name/snapshotname
      ```

   2. To restore a different backup version, move the backup files to this directory:

      ```
      /home/dbadmin/backups/localbak/node_name/snapshotname_archivedate_timestamp
      ```

4. When the backup files are returned to their original backup directory, use the original configuration file to invoke vbr . Verify that the configuration

file specifies hardLinkLocal = true . Then restore the backup as follows:

```
$ vbr --task restore --config-file localbak.ini
```

Ownership of restored objects

For a full restore, objects have the owners that they had in the backed-up database.

When performing an object restore, Vertica inserts data into existing database objects. By default, the restore does not affect the ownership, storage policies, or permissions of the restored objects. However, if the restored object does not already exist, Vertica re-creates it. In this situation, the restored object is owned by the user performing the restore. Vertica does not restore dependent grants, roles, or client authentications with restored objects.

If the storage policies of a restored object are not valid, vbr applies the default storage policy. Restored storage policies can become invalid due to HDFS storage locations, table incompatibility, and unavailable min-max values at restore time.

Sometimes, Vertica encounters a catalog object that it does not need to restore. When this situation occurs, Vertica generates a warning message for that object and the restore continues.

Examples

Suppose you have a full backup, including Schema1, owned by the user Alice. Schema1 contains Table1, owned by Bob, who eventually passes ownership to Chris. The user dbadmin performs the restore. The following scenarios might occur that affect ownership of these objects.

Scenario 1:

Schema1.Table1 has been dropped at some point since the backup was created. When dbadmin performs the restore, Vertica re-creates Schema1.Table1. As the user performing the restore, dbadmin takes ownership of Schema1.Table1. Because Schema1 still exists, Alice retains ownership of the schema.

Scenario 2:

Schema1 is dropped, along with all contained objects. When dbadmin performs the restore, Vertica re-creates the schema and all contained objects. dbadmin takes ownership of Schema1 and Schema1.Table1.

Scenario 3:

Schema1 and Schema1.Table1 both exist in the current database. When dbadmin rolls back to an earlier backup, the ownership of the objects remains unchanged. Alice owns Schema1, and Bob owns Schema1.Table1.

Scenario 4:

Schema1.Table1 exists and dbadmin wants to roll back to an earlier version. In the time since the backup was made, ownership of Schema1.Table1 has changed to Chris. When dbadmin restores Schema1.Table1, Alice remains owner of Schema1 and Chris remains owner of Schema1.Table1. The restore does not revert ownership of Schema1.Table1 from Chris to Bob.

Copying the database to another cluster

Important

Inadequate security on backups can compromise overall database security. Be sure to secure backup locations and strictly limit access to backups only to users who already have permissions to access all database data.

The vbr task copycluster combines two other vbr tasks— backup and restore —as a single operation, enabling you to back up an entire data from one Enterprise Mode database cluster and then restore it on another. This can facilitate routine operations, such as copying a database between development and production environments.

> **Caution**
> copycluster overwrites all existing data in the destination database. To preserve that data, back up the destination database before launching the copycluster task.

Restrictions

copycluster is invalid with Eon databases. It is also incompatible with HDFS storage locations; Vertica does not transfer data to a remote HDFS cluster as it does for a Linux cluster.

Prerequisites

copycluster requires that the target and source database clusters be identical in the following respects:

- Vertica hotfix version—for example, 12.0.1-1
- Number of nodes and node names, as shown in the system table NODES:

```
=> SELECT node_name FROM nodes;
 node_name
------------------
 v_vmart_node0001
 v_vmart_node0002
 v_vmart_node0003
(3 rows)
```

- Database name
- Vertica catalog, data, and temp directory paths as shown in the system table DISK_STORAGE:

```
=> SELECT node_name,storage_path,storage_usage FROM disk_storage;
   node_name     |                    storage_path                    | storage_usage
-----------------+----------------------------------------------------+---------------
 v_vmart_node0001 | /home/dbadmin/VMart/v_vmart_node0001_catalog/Catalog | CATALOG
 v_vmart_node0001 | /home/dbadmin/VMart/v_vmart_node0001_data           | DATA,TEMP
 v_vmart_node0001 | /home/dbadmin/verticadb                             | DEPOT
 v_vmart_node0002 | /home/dbadmin/VMart/v_vmart_node0002_catalog/Catalog | CATALOG
...
```

> **Note**
> Directory paths for the catalog, data, and temp storage are the same on all nodes.

- Database administrator accounts

The following requirements also apply:

- The target cluster has adequate disk space for copycluster to complete.
- The source cluster's database administrator must be able to log in to all target cluster nodes through SSH without a password.

> **Note**
> Passwordless access *within* the cluster is not the same as passwordless access *between* clusters. The SSH ID of the administrator account on the source cluster and the target cluster are likely not the same. You must configure each host in the target cluster to accept the SSH authentication of the source cluster.

Copycluster procedure

1. Create a configuration file for the copycluster operation. The Vertica installation includes a sample configuration file:

   ```
   /opt/vertica/share/vbr/example_configs/copycluster.ini
   ```

   For each node in the source database, create a [Mapping] entry that specifies the host name of each destination database node. Unlike other vbr tasks such as restore and backup , mappings for copycluster only require the destination host name. copycluster always stores backup data in the catalog and data directories of the destination database.
   The following example configures vbr to copy the vmart database from its three-node v_vmart cluster to the test-host cluster:

```
[Misc]
snapshotName = CopyVmart
tempDir = /tmp/vbr

[Database]
dbName = vmart
dbUser = dbadmin
dbPassword = password
dbPromptForPassword = False

[Transmission]
encrypt = False
port_rsync = 50000

[Mapping]
; backupDir is not used for cluster copy
v_vmart_node0001= test-host01
v_vmart_node0002= test-host02
v_vmart_node0003= test-host03
```

2. Stop the target cluster.
3. As database administrator, invoke the vbr task copycluster from a source database node:

```
$ vbr -t copycluster -c copycluster.ini
Starting copy of database VMART.
Participating nodes: vmart_node0001, vmart_node0002, vmart_node0003, vmart_node0004.
Enter vertica password:
Snapshotting database.
Snapshot complete.
Determining what data to copy.
[==================================================] 100%
Approximate bytes to copy: 987394852 of 987394852 total.
Syncing data to destination cluster.
[==================================================] 100%
Reinitializing destination catalog.
Copycluster complete!
```

Important
If the copycluster task is interrupted, the destination cluster retains data files that already transferred. If you retry the operation, Vertica does not resend these files.

Replicating objects to another database cluster

The vbr task replicate supports replication of tables and schemas from one database cluster to another. You might consider replication for the following reasons:

- Copy tables and schemas between test, staging, and production clusters.Replicate certain objects immediately after an important change, such as a large table data load, instead of waiting until the next scheduled backup.

In both cases, replicating objects is generally more efficient than exporting and importing them. The first replication of an object replicates the entire object. Subsequent replications copy only data that has changed since the last replication. Vertica replicates data as of the current epoch on the target database. Used with a cron job, you can replicate key objects to create a backup database.

Replicate versus copycluster

replicate only supports tables and schemas. In situations where the target database is down, or you plan to replicate the entire database, Vertica recommends that you use the copycluster task to copy the database to another cluster. Thereafter, you can use replicate to update individual tables and schema.

Replication procedure

To replicate objects to another database, perform these actions from the source database:

1. Verify replication requirements .

2. Identify the objects to replicate and target database in the vbr configuration file.
3. Replicate objects .

Verify replication requirements

The following requirements apply to the source and target databases and their respective clusters:

- All nodes in both databases are UP, else DOWN nodes are handled as described below .
- Versions of the two databases must be compatible. Vertica supports object replication to a target database up to one minor version higher than the current database version. For example, you can replicate objects from a 12.0.x database to a 12.1.x database.
- The same Linux user is associated with the dbadmin account of both databases.
- The source cluster database administrator can log on to all target nodes through SSH without a password.

> **Note**
> The SSH ID of the administrator account on the source cluster and the target cluster are likely not the same. You must configure each host in the target cluster to accept the SSH authentication of the source cluster.

- Enterprise Mode: The following requirements apply:
  - Both databases have the same number of nodes.
  - Clusters of both databases have the same number of fault groups, where corresponding fault groups in each cluster have the same number of nodes.
- Eon Mode: The primary subclusters of both databases have the same node subscriptions. Primary subclusters of the target database have as many nodes (or more) as primary subclusters of the source database.

Edit vbr configuration file

> **Tip**
> As a best practice, create a separate configuration file for each replication task.

Edit the vbr configuration file to use for the replicate task as follows:

1. In the [misc] section, set the objects parameter to the objects to be replicated:

```
; Identify the objects that you want to replicate
objects = schema.objectName
```

2. In the [misc] section, set the snapshotName parameter to a unique snapshot identifier. Multiple replicate tasks can run concurrently with each other and with backup tasks, but only if their snapshot names are different.

```
snapshotName = name
```

3. In the [database] section, set the following parameters:

```
; parameters used to replicate objects between databases
dest_dbName =
dest_dbUser =
dest_dbPromptForPassword =
```

   If you use a stored password, be sure to configure the dest_dbPassword parameter in your password configuration file .

4. In the [mapping] section, map source nodes to target hosts:

```
[Mapping]
v_source_node0001 = targethost01
v_source_node0002 = targethost02
v_source_node0003 = targethost03
```

Replicate objects

Run vbr with the replicate task:

```
vbr -t replicate -c configfile.ini
```

The replicate task can run concurrently with backup and other replicate tasks in either direction, provided all tasks have unique snapshot names. replicate cannot run concurrently with other vbr tasks.

Handling DOWN nodes

You can replicate objects if some nodes are down in either the source or target database, provided the nodes are visible on the network.

The effect of DOWN nodes on a replication task depends on whether they are present in the source or target database.

Location	Effect on replication
DOWN source nodes	Vertica can replicate objects from a source database containing DOWN nodes. If nodes in the source database are DOWN, set the corresponding nodes in the target database to DOWN as well.
DOWN target nodes	Vertica can replicate objects when the target database has DOWN nodes. If nodes in the target database are DOWN, exclude the corresponding source database nodes using the --nodes parameter on the vbr command line.

Monitoring object replication

You can monitor object replication in the following ways:

- View vbr logs on the source database
- Check database logs on the source and target databases
- Query REMOTE_REPLICATION_STATUS on the source database

Including and excluding objects

You specify objects to include in backup, restore, and replicate operations with the vbr configuration and command-line parameters includeObject s and --include-objects , respectively. You can optionally modify the set of included objects with the vbr configuration and command line parameters excludeObjects and --exclude-objects , respectively. Both parameters support wildcard expressions to include and exclude groups of objects.

For example, you might back up all tables in the schema store , and then exclude from the backup the table store.orders and all tables in the same schema whose name includes the string account :

```
vbr --task=backup --config-file=db.ini --include-objects 'store.*' --exclude-objects 'store.orders,store.*account*'
```

Wildcard characters

Character	Description
?	Matches any single character. Case-insensitive.
*	Matches 0 or more characters. Case-insensitive.
\	Escapes the next character. To include a literal ? or * in your table or schema name, use the \ character immediately before the escaped character. To escape the \ character itself, use a double \.
"	Escapes the . character. To include a literal . in your table or schema name, wrap the character in double quotation marks.

Matching schemas

Any string pattern without a period (.) character represents a schema. For example, the following includeObjects list can match any schema name that starts with the string customer , and any two-character schema name that starts with the letter s :

```
includeObjects = customer*,s?
```

When a vbr operation specifies a schema that is unqualified by table references, the operation includes all tables of that schema. In this case, you cannot exclude individual tables from the same schema. For example, the following vbr.ini entries are invalid:

```
; invalid:
includeObjects = VMart
excludeObjects = VMart.?table?
```

You can exclude tables from an included schema by identifying the schema with the pattern *schemaname* .*. In this case, the pattern explicitly specifies to include all tables in that schema with the wildcard *. In the following example, the include-objects parameter includes all tables in the VMart schema, and then excludes specific tables—specifically, the table VMart.sales and all VMart tables that include the string account :

```
--include-objects 'VMart.*'
--exclude-objects 'VMart.sales,VMart.*account*'
```

Matching tables

Any pattern that includes a period (.) represents a table. For example, in a configuration file, the following includeObjects list matches the table name sales.newclients , and any two-character table name in the same schema:

```
includeObjects = sales.newclients,sales.??
```

You can also match all schemas and tables in a database or backup by using the pattern *.*. For example, you can restore all tables and schemas in a backup using this command:

```
--include-objects '*.*'
```

Because a vbr parameter is evaluated on the command line, you must enclose the wildcards in single quote marks to prevent Linux from misinterpreting them.

Testing wildcard patterns

You can test the results of any pattern by using the --dry-run parameter with a backup or restore command. Commands that include --dry-run do not affect your database. Instead, vbr displays the result of the command without executing it. For more information on --dry-run , refer to the vbr reference .

Using wildcards with backups

You can identify objects to include in your object backup tasks using the includeObjects and excludeObjects parameters in your configuration file. A typical configuration file might include the following content:

```
[Misc]
snapshotName = dbobjects
restorePointLimit = 1
enableFreeSpaceCheck = True
includeObjects = VMart.*,online_sales.*
excludeObjects = *.*temp*
```

In this example, the backup would include all tables from the VMart and online_sales schemas, while excluding any table containing the string 'temp' in its name belonging to any schema.

After it evaluates included objects, vbr evaluates excluded objects and removes excluded objects from the included set. For example, if you included schema1.table1 and then excluded schema1.table1, that object would be excluded. If no other objects were included in the task, the task would fail. The same is true for wildcards. If an exclusion pattern removes all included objects, the task fails.

Using wildcards with restore

You can identify objects to include in your restore tasks using the --include-objects and --exclude-objects parameters.

> **Note**
> Take extra care when using wildcard patterns to restore database objects. Depending on your object restore mode settings, restored objects can overwrite existing objects. Test the impact of a wildcard restore with the --dry-run vbr parameter before performing the actual task.

As with backups, vbr evaluates excluded objects after it evaluates included objects and removes excluded objects from the included set. If no objects remain, the task fails.

A typical restore command might include this content. (Line wrapped in the documentation for readability, but this is one command.)

```
$ vbr -t restore -c verticaconfig --include-objects 'customers.*,sales??'
   --exclude-objects 'customers.199?,customers.200?'
```

This example includes the schema customers, minus any tables with names matching 199 and 200 plus one character, as well as all any schema matching 'sales' plus two characters.

Another typical restore command might include this content.

```
$ vbr -t restore -c replicateconfig --include-objects '*.transactions,flights.*'
   --exclude-objects 'flights.DTW*,flights.LAS*,flights.LAX*'
```

This example includes any table named transactions, regardless of schema, and any tables beginning with DTW, LAS, or LAX belonging to the schema flights. Although these three-letter airport codes are capitalized in the example, vbr is case-insensitive.

Managing backups

Important
Inadequate security on backups can compromise overall database security. Be sure to secure backup locations and strictly limit access to backups only to users who already have permissions to access all database data.

vbr provides several tasks related to managing backups: listing them, checking their integrity, selectively deleting them, and more. In addition, vbr has parameters to allow you to restrict its use of system resources.

In this section

- Viewing backups
- Checking backup integrity
- Repairing backups
- Removing backups
- Estimating log file disk requirements
- Allocating resources

Viewing backups

You can view backups in three ways:

- vbr listbackup task: List backups on the local or remote backup host.
- DATABASE_BACKUPS system table: Query for historical information about backups.
- vbr log file : Check the status of a backup. The log file resides on the node where you ran vbr , in the directory specified by the vbr configuration parameter tempDir , by default set to /tmp/vbr .

vbr listbackup

The vbr task listbackup returns a list of all backups on backup hosts, whether local or remote. If unqualified by task options , listbackup returns the list to standard output in columnar format.

The following example lists two full backups of a three-node cluster, where each node is mapped to the same backup host, bkhost . Backups are listed in reverse chronological order:

```
$ vbr -t listbackup -c fullbackup.ini
backup                        backup_type  epoch  objects  include_patterns  exclude_patterns  nodes(hosts)
version          file_system_type
backup_snapshot_20220912_131918  full         3915                                    v_vmart_node0001(10.20.100.247),
v_vmart_node0002(10.20.100.248), v_vmart_node0003(10.20.100.249)  v12.0.2-20220911  [Linux]
backup_snapshot_20220909_122300  full         3910                                    v_vmart_node0001(10.20.100.247),
v_vmart_node0002(10.20.100.248), v_vmart_node0003(10.20.100.249)  v12.0.2-20220911  [Linux]
```

The following table contains information about output columns that are returned from a vbr listbackup task:

Column	Description

backup	Identifies a backup by concatenating the configured snapshot name with the backup timestamp: *snapshot-name _ YYYYMMDD _ HHMMSS* For example, the following identifier identifies a backup generated by the configuration file that sets snapshotName to monthlyBackup on April 14 2022, at 13:44:52. monthlyBackup_20220414_134452 Use the timestamp portion of this identifier— 20220414_134452 —to specify the archived backup you wish to restore.
backup_type	Type of backup, full or object.
epoch	Epoch when the backup was created.
objects	Objects that were backed up, blank if a full backup.
include_patterns	Wildcard patterns included in object backup tasks using the includeObjects parameter in your configuration file, blank for full backups.
exclude_patterns	Wildcard patterns included in your object backup tasks using the excludeObjects parameter in your configuration file, blank for full backups.
nodes (hosts)	(Enterprise Mode only) Names of database nodes and hosts that received the backup.
version	Version of Vertica used to create the backup.
file_system_type	Storage location file system of the Vertica hosts that comprise this backup—for example, Linux or GCS.
communal_storage	(Eon Mode only) Communal storage location for the backup.

Important
If you try to list backups on a local cluster with no database, the backup configuration node-host mappings must provide full paths. If the configuration maps to local hosts using the [] shortcut , the listbackup task fails.

Listbackup options

You can qualify the listbackup task with one or more options:

```
vbr --task listbackup [--list-all] [--json] [--list-output-file filepath] --config-file filepath
```

Option	Description
--list-all	Generate a list of all snapshots stored on the hosts and paths listed in the specified configuration file.
--json	Use JSON delimited format.
--list-output-file	Redirect output to the specified file.

The following example qualifies the listbackup task with the --list-all option. The output shows three nightly backups from nodes vmart_1 , vmart_2 , and v_mart3 , which the configuration file nightly.ini maps to their respective hosts doca01 , doca02 , and doca03 . The listbackup output shows that these locations contain not only object backups that were generated with nightly.ini , but also full backups created with a second configuration file, weekly.ini , which maps to the same nodes and host:

```
$ vbr --task listbackup --list-all --config-file /home/dbadmin/nightly.ini

backup                 backup_type  epoch  objects     include_patterns  exclude_patterns  nodes(hosts)                                  version  file_system_type
weekly_20220508_183249   full        1720                                  vmart_1(doca01), vmart_2(doca02), vmart_3(doca03) v11.0.1  [Linux]
weekly_20220501_182816   full        1403                                  vmart_1(doca01), vmart_2(doca02), vmart_3(doca03) v11.0.1  [Linux]
weekly_20220424_192754   full        1109                                  vmart_1(doca01), vmart_2(doca02), vmart_3(doca03) v11.0.1  [Linux]
nightly_20220507_183034  object      1705   sales_schema                   vmart_1(doca01), vmart_2(doca02), vmart_3(doca03) v11.0.1  [Linux]
nightly_20220506_181808  object      1692   sales_schema                   vmart_1(doca01), vmart_2(doca02), vmart_3(doca03) v11.0.1  [Linux]
nightly_20220505_193906  object      1632   sales_schema                   vmart_1(doca01), vmart_2(doca02), vmart_3(doca03) v11.0.1  [Linux]
```

Query backup history

You can query the system table DATABASE_BACKUPS to get historical information about backups. The objects column lists which objects were included in object-level backups.

Important

Do not use the backup_timestamp value to restore an archive . Instead, use the values provided by vbr listbackup task.

```
=> SELECT * FROM v_monitor.database_backups;
-[ RECORD 1 ]----+------------------------------
backup_timestamp | 2013-05-10 14:41:12.673381-04
node_name        | v_vmart_node0003
snapshot_name    | schemabak
backup_epoch     | 174
node_count       | 3
file_system_type | [Linux]
objects          | public, store, online_sales
-[ RECORD 2 ]----+------------------------------
backup_timestamp | 2013-05-13 11:17:30.913176-04
node_name        | v_vmart_node0003
snapshot_name    | kantibak
backup_epoch     | 175
node_count       | 3
file_system_type | [Linux]
objects          |
-[ RECORD 13 ]---+------------------------------
backup_timestamp | 2013-05-16 07:02:23.721657-04
node_name        | v_vmart_node0003
snapshot_name    | objectbak
backup_epoch     | 180
node_count       | 3
file_system_type | [Linux]
objects          | test, test2
-[ RECORD 14 ]---+------------------------------
backup_timestamp | 2013-05-16 07:19:44.952884-04
node_name        | v_vmart_node0003
snapshot_name    | table1bak
backup_epoch     | 180
node_count       | 3
file_system_type | [Linux]
objects          | test
-[ RECORD 15 ]---+------------------------------
backup_timestamp | 2013-05-16 07:20:18.585076-04
node_name        | v_vmart_node0003
snapshot_name    | table2bak
backup_epoch     | 180
node_count       | 3
file_system_type | [Linux]
objects          | test2
```

Checking backup integrity

Vertica can confirm the integrity of your backup files and the manifest that identifies them. By default, backup integrity checks output their results to the command line.

Quick check

The quick-check task gathers all backup metadata from the backup location specified in the configuration file and compares that metadata to the backup manifest. A quick check does not verify the objects themselves. Instead, this task outputs an exceptions list of any discrepancies between objects in the backup location and objects listed in the backup manifest.

Use the following format to perform quick check task:

```
$ vbr -t quick-check -c configfile.ini
```

For example:

```
$ vbr -t quick-check -c backupconfig.ini
```

Full check

The full-check task verifies all objects listed in the backup manifest against filesystem metadata. A full check includes the same steps as a quick check. You can include the optional --report-file parameter to output results to a delimited JSON file. This task outputs an exceptions list that identifies the following inconsistencies:

- Incomplete restore points
- Damaged restore points
- Missing backup files
- Unreferenced files

Use the following template to perform a full check task:

```
$ vbr -t full-check -c configfile.ini --report-file=path/filename
```

For example:

```
$ vbr -t full-check -c backupconfig.ini --report-file=logging/fullintegritycheck.json
```

Repairing backups

Vertica can reconstruct backup manifests and remove unneeded backup objects.

Quick repair

The quick-repair task rebuilds the backup manifest, based on the manifests contained in the backup location.

Use the following template to perform a quick repair task:

```
$ vbr -t quick-repair -c configfile.ini
```

Garbage collection

The collect-garbage task rebuilds your backup manifest and deletes any backup objects that do not appear in the manifest. You can include the optional --report-file parameter to output results to a delimited JSON file.

Use the following template to perform a garbage collection task:

```
$ vbr -t collect-garbage -c configfile.ini --report-file=path/filename
```

Removing backups

You can remove existing backups and restore points using vbr . When you use the remove task, vbr updates the manifests affected by the removal and maintains their integrity. If the backup archive contains multiple restore points, removing one does not affect the others. When you remove the last restore point, vbr removes the backup entirely.

> **Note**
> Vertica does not support removing backups through the file system.

Use the following template to perform a remove task:

```
$ vbr -t remove -c configfile.ini --archive timestamp
```

You can remove multiple restore points using the archive parameter. To obtain the timestamp for a particular restore point, use the listbackup task .

- To remove multiple restore points, use a comma separator:
  ```
  --archive="`*`restore-point1`*`,`*`restore-point2`*`"
  ```
- To remove an inclusive range of restore points, use a colon:
  ```
  --archive="`*`restore-point1`*`:`*`restore-point2`*`"
  ```
- To remove all restore points, specify an archive value of all :
  ```
  --archive all
  ```

The following example shows how you can remove a restore point from an existing backup:

```
$ vbr -t remove -c backup.ini --archive  20160414_134452
Removing restore points: 20160414_134452
Remove complete!
```

Estimating log file disk requirements

One of the vbr configuration parameters is tempDir . This parameter specifies the database host location where vbr writes its log files and some other temp files (of negligible size). The default location is the /tmp/vbr directory on each database host. You can change the default location by specifying a different path in the configuration file.

The temporary storage directory also contains local log files describing the progress, throughput, and any errors encountered for each node. Each time you run vbr , the script creates a separate log file, each named with a timestamp. When using default settings, the log file typically uses about 4KB of space per node per backup.

The vbr log files are not removed automatically, so you must delete older log files manually, as necessary.

Allocating resources

By default, vbr allows a single rsync connection (for Linux file systems), 10 concurrent threads (for cloud storage connections), and unlimited bandwidth for any backup or restore operation. You can change these values in your configuration file. See vbr configuration file reference for details about these parameters.

Connections

You might want to increase the number of concurrent connections. If you have many Vertica files, more connections can provide a significant performance boost as each connection increases the number of concurrent file transfers.

For more information, refer to the following parameters in [transmission] :

- total_bwlimit_backup
- total_bwlimit_restore
- concurrency_backup
- concurrency_restore

and the following parameters in [CloudStorage] :

- cloud_storage_concurrency_backup
- cloud_storage_concurrency_restore

Bandwidth limits

You can limit network bandwidth use through the total_bwlimit_backup and total_bwlimit_restore data transmission parameters. For more information, refer to [transmission] .

Troubleshooting backup and restore

These tips can help you avoid issues related to backup and restore with Vertica and to troubleshoot any problems that occur.

Check vbr log

The vbr log is separate from the Vertica log. Its location is set by the vbr configuration parameter tempDir , by default /tmp/vbr .

If the log has no explanation for an error or unexpected results, try increasing the logging level with the vbr option --debug :

```
vbr -t backup -c config-file --debug debug-level
```

where *debug-level* is an integer between 0 (default) and 3 (verbose), inclusive. As you increase the logging level, the file size of the log increases. For example:

```
$ vbr -t backup -c full_backup.ini --debug 3
```

> **Note**
> Scrutinize reports do not include vbr logs.

Check status of backup nodes

Backups fail if you run out of disk space on the backup hosts or if vbr cannot reach them all. Check that you have sufficient space on each backup host and that you can reach each host via ssh.

Sometimes vbr leaves rsync processes running on the database or backup nodes. These processes can interfere with new ones. If you get an rsync error in the console, look for runaway processes and kill them.

Common errors

Object replication fails

If you do not exclude the DOWN node, replication fails with the following error:

```
Error connecting to a destination database node on the host <hostname> : <error>  ...
```

Confirm that you excluded all DOWN nodes from the object replication operation.

Error restoring an archive

You might see an error like the following when restoring an archive:

```
$ vbr --task restore --archive prd_db_20190131_183111 --config-file /home/dbadmin/backup.ini
IOError: [Errno 2] No such file or directory: '/tmp/vbr/vbr_20190131_183111_s0rpYR/prd_db.info'
```

The problem is that the archive name is not in the correct format. Specify only the date/timestamp suffix of the directory name that identifies the archive to restore, as described in Restoring an Archive . For example:

```
$ vbr --task restore --archive 20190131_183111 --config-file /home/dbadmin/backup.ini
```

Backup or restore fails when using an HDFS storage location

When performing a backup of a cluster that includes HDFS storage locations, you might see an error like the following:

```
ERROR 5127:  Unable to create snapshot No such file /usr/bin/hadoop:
check the HadoopHome configuration parameter
```

This error is caused by the backup script not being able to back up the HDFS storage locations. You must configure Vertica and Hadoop to enable the backup script to back up these locations. See Requirements for backing up and restoring HDFS storage locations .

Object-level backup and restore are not supported with HDFS storage locations. You must use full backup and restore.

Could not connect to endpoint URL

(Eon Mode) When performing a cross-endpoint operation, you can see a connection error if you failed to specify the endpoint URL for your communal storage (VBR_COMMUNAL_STORAGE_ENDPOINT_URL). When the endpoint is missing but you specify credentials for communal storage, vbr tries to use those credentials to access AWS. This access fails, because those credentials are for your on-premises storage, not AWS. When performing cross-endpoint operations, check that all environment variables described in Cross-Endpoint Backups in Eon Mode are set correctly.

vbr reference

vbr can back up and restore the full database, or specific schemas and tables. It also supports a number of other backup-related tasks—for example, list the history of all backups.

vbr is located in the Vertica binary directory—typically, /opt/vertica/bin/vbr .

Syntax

```
vbr { --help | -h }
 | { --task | -t } task  { --config-file | -c } configfile [ option[...] ]
```

Global options

The following options apply to all vbr tasks. For additional options, see Task-Specific Options .

Option	Description
--help | -h	Display a brief vbr usage guide.
{--task | -t} *task*	The vbr task to execute, one of the following: • backup : create a full or object-level backup • collect-garbage : rebuild the backup manifest and delete any unreferenced objects in the backup location • copycluster : copy the database to another cluster (Enterprise Mode only, invalid for HDFS) • full-check : verify all objects in the backup manifest and report missing or unreferenced objects • init : prepare a new backup location • listbackup : show available backups • quick-check : confirm that all backed-up objects are in the backup manifest and report discrepancies between objects in the backup location and objects listed in the backup manifest • quick-repair : build a replacement backup manifest based on storage locations and objects • remove : remove specified restore points • replicate : copy objects from one cluster to another • restore : restore a full or object-level backup **Note** In general, tasks cannot run concurrently, with one exception: multiple replicate tasks can run concurrently with each other, and with backup .
{--config-file | -c} *path*	File path of the configuration file to use for the given task.
--debug *level*	Level of debug messaging to the vbr log, an integer from 0 to 3 inclusive, where 0 (default) turns off debug messaging, and 3 is the most verbose level of messaging.
--nodes *nodeslist*	(Enterprise Mode only) Comma-delimited list of nodes on which to perform a vbr task. Listed nodes must match names in the Mapping section of the configuration file. Use this option to exclude DOWN nodes from a task, so vbr does not return with an error. **Caution** If you use --nodes with a backup task, be sure that the nodes list includes all UP nodes; omitting any UP node can cause data loss in that backup.
--showconfig	Displays the configuration values used to perform a specific task, displayed in raw JSON format before vbr starts task execution: vbr -t *task* -c *configfile* --showconfig --showconfig can also show settings for a given configuration file: vbr -c *configfile* --showconfig

Task-specific options

Some vbr tasks support additional options, described in the sections that follow.

The following vbr tasks have no task-specific options:

- copycluster
- quick-check
- quick-repair

Backup

Create a full database or object-level backup, depending on configuration file settings.

Option	Description
--dry-run	Perform a test run to evaluate impact of the backup operation—for example, its size and potential overhead.

Collect-garbage

Rebuild the backup manifest and delete any unreferenced objects in the backup location.

Option	Description
--report-file	Output results to a delimited JSON file.

Full-check

Produce a full backup integrity check that verifies all objects in the backup manifest against file system metadata, and then outputs missing and unreferenced objects.

Option	Description
--report-file	Output results to a delimited JSON file.

Init

Create a backup directory or prepare an existing one for use, and create backup manifests. This task must precede the first vbr backup operation.

Option	Description
-- cloud-force-init	Qualifies the --task init command to force the init task to succeed on S3 or GS storage targets when an identity/lock file mismatch occurs.
--report-file	Output results to a delimited JSON file.

Listbackup

Displays backups associated with the specified configuration file. Use this task to get archive (restore point) identifiers for restore and remove tasks.

Option	Description
-- list-all	List all backups stored on the hosts and paths in the configuration file.
--list-output-file *filename*	Redirect output to the specified file.
--json	Use JSON delimited format.

Remove

Remove the backup restore points specified by the --archive option.

Option	Description

--archive	Restore points to remove, one of the following: • *timestamp* : A single restore point to remove. • *timestamp* : *timestamp* : A range of contiguous restore points to remove. • all : Remove all restore points. You obtain timestamp identifiers for the target restore points with the listbackup task. For details, see vbr listbackup .

Replicate

Copy objects from one cluster to an alternate cluster. This task can run concurrently with backup and other replicate tasks.

Option	Description
--archive	Timestamp of the backup restore point to replicate, obtained from the listbackup task.
--dry-run	Perform a test run to evaluate impact of the replicate operation—for example, its size and potential overhead.

Restore

Restore a full or object-level database backup.

Option	Description
--archive	Timestamp of the backup to restore, obtained from the listbackup task. If omitted, vbr restores the latest backup of the specified configuration.
--restore-objects	Comma-delimited list of objects—tables and schemas—to restore from a given backup.
--include-objects	Comma-delimited list of database objects or patterns of objects to include from a full or object-level backup.
--exclude-objects	Comma-delimited list of database objects or patterns of objects to exclude from the set specified by --include-objects . This option can only be used together with --include-objects .
--dry-run	Perform a test run to evaluate impact of the restore operation—for example, its size and potential overhead.

Note
The --restore-objects option and the --include-objects / exclude-objects options are mutually exclusive. You can use --include-objects to specify a set of objects and combine it with --exclude-objects to remove objects from the set.

Interrupting vbr

To cancel a backup, use Ctrl+C or send a SIGINT to the vbr Python process. vbr stops the backup process after it completes copying the data. Canceling a vbr backup with Ctrl+C closes the session immediately.

The files generated by an interrupted backup process remain in the target backup location directory. The next backup process picks up where the interrupted process left off.

Backup operations are atomic, so interrupting a backup operation does not affect the previous backup. The latest backup replaces the previous backup only after all other backup steps are complete.

Caution
restore or copycluster operations overwrite the database catalog directory. Interrupting either of these processes leaves the database unusable until you restart the process and allow it to finish.

See also

- vbr configuration file reference
- Sample vbr configuration files

vbr configuration file reference

vbr configuration files divide backup settings into sections, under section-specific headings such as [Database] and [CloudStorage] , which contain database access and cloud storage location settings, respectively. Sections can appear in any order and can be repeated—for example, multiple [Database] sections.

Important
Section headings are case-sensitive.

In this section

- [CloudStorage]
- [database]
- [mapping]
- [misc]
- [NodeMapping]
- [transmission]
- Password configuration file

[CloudStorage]

Eon Mode only

Important
The [CloudStorage] section replaces the now-deprecated [S3] section of earlier releases. Likewise, cloud storage-specific configuration variables replace the equivalent S3 configuration variables.

Do not include [S3] and [CloudStorage] sections in the same configuration file; otherwise, vbr will use [S3] configuration settings and ignore [CloudStorage] settings, which can yield unexpected results.

Sets options for storing backup data on in a supported cloud storage location.

> **Note**
> The [CloudStorage] and [Mapping] configuration sections are mutually exclusive. If you include both, the backup fails with this error message:
>
> ```
> Config has conflicting sections (Mapping, CloudStorage), specify only one of them.
> ```

Parameter	Description
cloud_storage_backup_file_system_path	Specifies the host and path that you are using to handle file locking during the backup process. vbr must be able to create a passwordless ssh connection to the location that you specify here. To use a local NFS file system, specify a value of: cloud_storage_backup_file_system_path = []: *path* To use a host, specify a value of: cloud_storage_backup_file_system_path = [*host-name*]: *path*

cloud_storage_backup_path	Specifies the backup location. For S3-compatible or cloud locations, provide the bucket name and backup path. For HDFS locations, provide the appropriate protocol and backup path. When you back up to cloud storage, all nodes back up to same cloud storage bucket. You must create the backup location in the cloud storage before performing a backup. The following example specifies the backup path for S3 storage: cloud_storage_backup_path = s3:// *backup-bucket* / *database-backup-path* / When you back up to an HDFS location, use the swebhdfs protocol if you use wire encryption. Use the webhdfs protocol if you do not use wire encryption. The following example uses encryption: cloud_storage_backup_path = swebhdfs:// *backup-nameservice* / *database-backup-path* /
cloud_storage_ca_bundle	Specifies the path to an SSL server certificate bundle. **Note** The key (*pem) file must be on the same path on all nodes of the database cluster. For example: cloud_storage_ca_bundle = / *home* / *user* / *ssl-folder* / *ca-bundle*
cloud_storage_concurrency_backup	The maximum number of concurrent backup threads for backup to cloud storage. For very large data volumes (greater than 10TB), you might need to reduce this value to avoid vbr failures. **Default:** 10
cloud_storage_concurrency_delete	The maximum number of concurrent delete threads for deleting files from cloud storage. If the vbr configuration file contains a [CloudStorage] section, this value is set to 10 by default. **Default:** 10
cloud_storage_concurrency_restore	The maximum number of concurrent restore threads for restoring from cloud storage. For very large data volumes (>10TB), you might need to reduce this value to avoid vbr failures. **Default:** 10
cloud_storage_encrypt_at_rest	S3 storage only. To enable at-rest encryption of your backups to S3, specify a value of sse . For more information, see Encrypting Backups on Amazon S3 . This value takes the following form: cloud_storage_encrypt_at_rest = sse
cloud_storage_encrypt_transport	Boolean, if set to true uses SSL encryption to encrypt data moving between your Vertica cluster and your cloud storage instance. You must set this parameter to true if backing up or restoring from: • Amazon EC2 cluster • Google Cloud Storage (GCS) • Eon Mode on-premises database with communal storage on HDFS, to use wire encryption. **Default:** true

cloud_storage_sse_kms_key_id	S3 storage only. If you use Amazon Key Management Security, use this parameter to provide your key ID. If you enable encryption and do not include this parameter, vbr uses SSE-S3 encryption. This value takes the following form: cloud_storage_sse_kms_key_id = *key-id*

[database]

Sets options for accessing the database.

Parameter	Description
dbName	Name of the database to back up. If you do not supply a database name, vbr selects the current database to back up. OpenText recommends that you provide a database name.
dbPromptForPassword	Boolean, specifies whether vbr prompts for a password. If set to false (no prompt at runtime), then the dbPassword parameter in the password configuration file must provide the password; otherwise, vbr prompts for one at runtime. As a best practice, set dbPromptForPassword to false if dbUseLocalConnection is set to true. **Default:** true
dbUser	Identifies the Vertica user that performs vbr operations on the database operations. In the case of replicate tasks, this user is the source database user. You must be logged on as the database administrator to back up the database. The user password can be stored in the dbPassword parameter of the password configuration file ; otherwise, vbr prompts for one at runtime. **Default:** Current user name

dbUseLocalConnection	Boolean, specifies whether vbr accesses the target database over a local connection with the user's Vertica password. If dbUseLocalConnection is enabled, vbr can operate on a local database without the user password being set in the vbr configuration. vbr ignores the passwordFile parameter and any settings in the password configuration file , including dbPassword .
	
	If dbUseLocalConnection is enabled, then an authentication method must be granted to vbr users—typically a dbadmin—where method type is set to trust, and access is set to local:
	
	`=> CREATE AUTHENTICATION h1 method 'trust' local;`
	`=> GRANT AUTHENTICATION h1 to dbadmin;`
	
	Default: false

Set destination database parameters only if replicating objects on alternate clusters:

Parameter	Description
dest_dbName	Name of the destination database.
dest_dbPromptForPassword	Boolean, controls whether vbr prompts for the destination database password. If set to false (no prompt at runtime), then dest_dbPassword parameter in the password configuration file must provide the password; otherwise, vbr prompts for one at runtime.
dest_dbUser	Vertica user name in the destination database to use for loading replicated data. This user must have superuser privileges.

[mapping]

Enterprise Mode only

Specifies all database nodes to include in an Enterprise Mode database backup. This section also specifies the backup host and directory of each node. If objects are replicated to an alternative database, the [Mapping] section maps target database nodes to the corresponding source database backup locations.

> **Note**
> [CloudStorage] and [Mapping] configuration sections are mutually exclusive. If you include both, the backup fails.

Unlike other configuration file sections, the [Mapping] section does not use named parameters. Instead, it contains entries of the following format:

```
dbNode = backupHost:backupDir
```

Element	Description
dbNode	Name of the database node as recognized by Vertica. This value is not the node's host name; rather, it is the name Vertica uses internally to identify the node, typically in this format: v_ *dbname* _node000 *int* To find database node names in your cluster, query the node_name column of the NODES system table.

backupHost	The target host name or IP address on which to store this node's backup. *backupHost* is different from *dbNode* . The copycluster command uses this value to identify the target database node host name.
	IPv6 addresses must be enclosed by square brackets [] . For example:
	`v_backup_restore_node0001 = [fdfb:dbfa:0:2000::112]:/backupdir/backup_restore.2021-06-01T16:17:57`
	`v_backup_restore_node0002 = [fdfb:dbfa:0:2000::113]:/backupdir/backup_restore.2021-06-01T16:17:57`
	`v_backup_restore_node0003 = [fdfb:dbfa:0:2000::114]:/backupdir/backup_restore.2021-06-01T16:17:57`
	Important
	Although supported, backups to an NFS host might perform poorly, particularly on networks shared with rsync operations.

backupDir	The full path to the directory on the backup host or node where the backup will be stored. The following requirements apply this directory: • Already exists when you run vbr with --task backup • Writable by the user account used to run vbr . • Unique to the database you are backing up. Multiple databases cannot share the same backup directory. • File system at this location supports fcntl lockf file locking.

For example:

```
[Mapping]
v_sec_node0001 = pri_bsrv01:/archive/backup
v_sec_node0002 = pri_bsrv02:/archive/backup
v_sec_node0003 = pri_bsrv03:/archive/backup
```

Mapping to the local host

vbr does not support using localhost to specify a backup host. To back up a database node to its own disk, specify the host name with empty square brackets. For example:

```
[Mapping]
NodeName = []:/backup/path
```

Mapping to the same database

The following example shows a [Mapping] section that specifies a single node to back up: v_vmart_node0001 . The node is assigned to backup host srv01 and backup directory /home/dbadmin/backups . Although a single-node cluster is backed up, and the backup host and the database node are the same system, they are specified differently.

Specify the backup host and directory using a colon (:) as a separator:

```
[Mapping]
v_vmart_node0001 = srv01:/home/dbadmin/backups
```

Mapping to an alternative database

> **Note**
> Replicating objects to an alternative database requires the vbr configuration file to include a [NodeMapping] section. This section points source nodes to their target database nodes.

To restore an alternative database, add mapping information as follows:

```
[Mapping]
targetNode = backupHost:backupDir
```

For example:

```
[Mapping]
v_sec_node0001 = pri_bsrv01:/archive/backup
v_sec_node0002 = pri_bsrv02:/archive/backup
v_sec_node0003 = pri_bsrv03:/archive/backup
```

[misc]

Configures basic backup settings.

Parameter	Description
passwordFile	Path name of the password configuration file , ignored if dbUseLocalConnection (under [Database] is set to true.
restorePointLimit	Number of earlier backups to retain with the most recent backup. If set to 1 (the default), Vertica maintains two backups: the latest backup and the one before it. **Note** vbr saves multiple backups to the same location, which are shared through hard links. In such cases, the listbackup task displays the common backup prefix with unique time and date suffixes: my_archive20111111_205841 **Default:** 1

snapshotName	Base name of the backup used in the directory tree structure that vbr creates for each node, containing up to 240 characters limited to the following: • a–z • A–Z • 0–9 • Hyphen (-) • Underscore (_) Each iteration in this series (up to restorePointLimit) consists of snapshotName and the backup timestamp. Each series of backups should have a unique and descriptive snapshot name. Full and object-level backups cannot share names. For most vbr tasks, snapshotName serves as a useful identifier in diagnostics and system tables. For object restore and replication tasks, snapshotName is used to build schema names in coexist mode operations. **Default:** snapshotName
tempDir	Absolute path to a temporary storage area on the cluster nodes. This path must be the same on all database cluster nodes. vbr uses this directory as temporary storage for log files, lock files, and other bookkeeping information while it copies files from the source cluster node to the destination backup location. vbr also writes backup logs to this location. The file system at this location must support fcntl lockf (POSIX) file locking. **Caution** Do not use the same location as your database's data or catalog directory. Unexpected files and directories in your data or catalog location can cause errors during database startup or restore. **Default:** /tmp/vbr
drop_foreign_constraints	Boolean, if set to true, all foreign key constraints are unconditionally dropped during object-level restore. You can then restore database objects independent of their foreign key dependencies. Important You must set objectRestoreMode to coexist, or Vertica ignores this setting. **Default:** false
enableFreeSpaceCheck	Boolean, if set to true (default) or omitted, vbr confirms that the specified backup locations contain sufficient free space to allow a successful backup. If a backup location has insufficient resources, vbr displays an error message and cancels the backup. If vbr cannot determine the amount of available space or number of nodes in the backup directory , it displays a warning and continues with the backup. **Default:** true
excludeObjects	Database objects and wildcard patterns to exclude from the set specified by includeObjects. Unicode characters are case-sensitive; others are not. This parameter can be set only if includeObjects is also set.

hadoop_conf_dir	(Eon Mode on HDFS with high availability (HA) nodes only) Directory path containing the XML configuration files copied from Hadoop. If the vbr operation includes more than one HA HDFS cluster, use a colon-separated list to provide the directory paths to the XML configuration files for each HA HDFS cluster. For example: hadoop_conf_dir = *path* / *to* / *xml-config-hahdfs1* : *path* / *to* / *xml-config-hahdfs2* This value must match the HadoopConfDir value set in the bootstrapping file created during installation .
includeObjects	Database objects and wildcard patterns to include with a backup task. You can use this parameter together with excludeObjects . Unicode characters are case-sensitive; others are not. Important The includeObjects and objects parameters are mutually exclusive.
kerberos_keytab_file	(Eon Mode on HDFS only) Location of the keytab file that contains credentials for the Vertica Kerberos principal. This value must match the KerberosKeytabFile value set in the bootstrapping file created during installation .
kerberos_realm	(Eon Mode on HDFS only) Realm portion of the Vertica Kerberos principal. This value must match the KerberosRealm value set in the bootstrapping file created during installation .
kerberos_service_name	(Eon Mode on HDFS only) Service name portion of the Vertica Kerberos principal. This value must match the KerberosServiceName value set in the bootstrapping file created during installation . **Default:** vertica
objectRestoreMode	Specifies how vbr handles objects of the same name when restoring schema or table backups, one of the following: • createOrReplace : vbr creates any objects that do not exist. If an object does exist, vbr overwrites it with the version from the archive. • create : vbr creates any objects that do not exist and does not replace existing objects. If an object being restored does exist, the restore fails. • coexist : vbr creates the restored version of each object with a name formatted as follows: *backup _ timestamp _ objectname* This approach allows existing and restored objects to exist simultaneously. If the appended information pushes the schema name past the maximum length of 128 characters, Vertica truncates the name. You can perform a reverse lookup of the original schema name by querying the system table TRUNCATED_SCHEMATA . In all modes, vbr restores data with the current epoch. Object restore mode settings do not apply to backups and full restores. **Default:** createOrReplace

objects	For an object-level backup or object replication, object (schema or table) names to include. To specify more than one object, enter multiple names in a comma-delimited list. If you specify no objects, vbr creates a full backup. Important This parameter cannot be used together with the parameters includeObjects and excludeObjects . You specify objects as follows: • Specify table names in the form *schema . objectname* . For example, to make backups of the table customers from the schema finance , enter: finance.customers If a public table and a schema have the same name, vbr backs up only the schema. Use the *schema . objectname* convention to avoid confusion. • Object names can include UTF-8 alphanumeric characters. Object names cannot include escape characters, single- (') or double-quote (") characters. • Specify non-alphanumeric characters with a backslash () followed by a hex value. For instance, if the table name is my table (my followed by a space character, then table), enter the object name as follows: objects=my\20table • If an object name includes a period, enclose the name with double quotes. **Tip** To identify objects with wildcards , use the includeObjects / excludeObjects parameters.

[NodeMapping]

vbr uses the node mapping section exclusively to restore objects from a backup of one database to a different database. Be sure to update the [Mapping] section of your configuration file to point your target database nodes to their source backup locations. The target database must have at least as many UP nodes as the source database.

Use the following format to specify node mapping: source_node = target_node For example, you can use the following mapping to restore content from one 4-node database to an alternate 4-node database.

```
[NodeMapping]
v_sourcedb_node0001 = v_targetdb_node0001
v_sourcedb_node0002 = v_targetdb_node0002
v_sourcedb_node0003 = v_targetdb_node0003
v_sourcedb_node0004 = v_targetdb_node0004
```

See Restoring a database to an alternate cluster for a complete example.

[transmission]

Sets options for transmitting data when using backup hosts.

Parameter	**Description**
concurrency_backup	Maximum number of backup TCP rsync connection threads per node. To improve local and remote backup, replication, and copy cluster performance, you can increase the number of threads available to perform backups. Increasing the number of threads allocates more CPU resources to the backup task and can, for remote backups, increase the amount of bandwidth used. The optimal value for this setting depends greatly on your specific configuration and requirements. Values higher than 16 produce no additional benefit. **Default:** 1

concurrency_delete	Maximum number of delete TCP rsync connections per node. To improve local and remote restore, replication, and copycluster performance, increase the number of threads available to delete files. Increasing the number of threads allocates more CPU resources to the delete task and can increase the amount of bandwidth used for deletes on remote backups. The optimal value for this setting depends on your specific configuration and requirements. **Default:** 16
concurrency_restore	Maximum number of restore TCP rsync connections per node. To improve local and remote restore, replication, and copycluster performance, increase the number of threads available to perform restores. Increasing the number of threads allocates more CPU resources to the restore task and can increase the amount of bandwidth used for restores of remote backups. The optimal value for this setting depends greatly on your specific configuration and requirements. Values higher than 16 produce no additional benefit. **Default:** 1
copyOnHardLinkFailure	If a hard-link local backup cannot create links, copy the data instead. Copying takes longer than linking, so the default behavior is to return an error if links cannot be created on any node. **Default:** false
encrypt	Boolean, specifies whether transmitted data is encrypted while it is copied to the target backup location. Set this parameter to true only if performing a backup over an untrusted network—for example, backing up to a remote host across the Internet. Important Encrypting data transmission causes significant processing overhead and slows transfer. One of the processor cores of each database node is consumed during the encryption process. Use this option only if you are concerned about the security of the network used when transmitting backup data. Omit this parameter from the configuration file for hard-link local backups. If you set both encrypt and hardLinkLocal to true in the same configuration file, vbr issues a warning and ignores encrypt. **Default:** false
hardLinkLocal	Boolean, specifies whether to create a full- or object-level backup using hard file links on the local file system, rather than copying database files to a remote backup host. Add this configuration parameter manually to the Transaction section of the configuration file. For details on usage, see Full Hardlink Backup/Restore . **Default:** false
port_rsync	Default port number for the rsync protocol. Change this value if the default rsync port is in use on your cluster, or you need rsync to use another port to avoid a firewall restriction. **Default:** 50000
serviceAccessUser	User name used for simple authentication of rsync connections. This user is neither a Linux nor Vertica user name, but rather an arbitrary identifier used by the rsync protocol. If you omit setting this parameter, rsync runs without authentication, which can create a potential security risk. If you choose to save the password, store it in the password configuration file .

total_bwlimit_backup	Total bandwidth limit in KBps for backup connections. Vertica distributes this bandwidth evenly among the number of connections set in concurrency_backup. The default value of 0 allows unlimited bandwidth. The total network load allowed by this value is the number of nodes multiplied by the value of this parameter. For example, a three node cluster and a total_bwlimit_backup value of 100 would allow 300Kbytes/sec of network traffic. **Default:** 0
total_bwlimit_restore	Total bandwidth limit in KBps for restore connections. distributes this bandwidth evenly among the number of connections set in concurrency_restore. The default value of 0 allows unlimited bandwidth. The total network load allowed by this value is the number of nodes multiplied by the value of this parameter. For example, a three node cluster and a total_bwlimit_restore value of 100 would allow 300Kbytes/sec of network traffic. **Default:** 0

Password configuration file

For improved security, store passwords in a password configuration file and then restrict read access to that file. Set the passwordFile parameter in your vbr configuration file to this file.

[passwords] password settings

All password configuration parameters are inside the file's [Passwords] section.

Parameter	Description
dbPassword	Database administrator's Vertica password, used if the dbPromptForPassword parameter is false. This parameter is ignored if dbUseLocalConnection is set to true.
dest_dbPassword	Password for the dest_dbuser Vertica account, for replication tasks only.
serviceAccessPass	Password for the rsync user account.

Examples

See Password file .

Failure recovery

Hardware or software issues can force nodes in your cluster to fail. In this case, the node or nodes leave the database. You must recover these failed nodes before they can rejoin the cluster and resume normal operation.

Node failure's impact on the database

Having failed nodes in your database affects how your database operates. If you have an Enterprise Mode database with K-safety 0, the loss of any node causes the database to shut down. Eon Mode databases usually do not have a K-safety of 0 (see Data integrity and high availability in an Eon Mode database).

In a database in either mode with K-safety of 1 or greater, your database continues to run normally after losing a node. However, its performance is affected:

- In Enterprise Mode, another node fills in for a down node, using its copy of the down node's data. This node must perform up to twice the amount of work it usually does. Operations such as queries will take longer because the rest of the cluster waits for the node to finish.
- In Eon Mode, another node fills in for the down node. Nodes in Eon Mode databases do not maintain buddy projections like nodes in Enterprise Mode databases. The node filling in for the down node retrieves the down node's data from communal storage to process queries. It does not store that data in the depot. Having to retrieve all of the data from communal storage slows down the processing of the query, in addition to the node having to perform more work. The performance impact of the down node is usually limited to the subcluster that contains it.

Because of these performance impacts, you should recover the failed nodes as soon as possible.

If too many database nodes fail, your database loses the ability to maintain K-safety or quorum . In an Eon Mode database, loss of primary nodes can also result in loss of primary shard coverage . In any of these cases, your database stops normal operations to prevent data corruption. How it responds to the loss of K-safety or quorum depends on its mode:

- In Enterprise Mode, the database shuts down because it does not have access to all of its data.
- In Eon Mode, the database continues running in read-only mode. Operations that change the global catalog such as inserting data or altering table schemas fail. However, queries can run on any subcluster that still has shard coverage. See Read-Only Mode .

To return your database to normal operation, you must restore the failed nodes and recover the database.

Recovery scenarios

Vertica begins the database recovery process when you restart failed nodes or the database. The mode of recovery for a K-safe database depends on the type of failure:

- One or more nodes in the database failed, but the database continued to operate normally. See Recovery of Failed Nodes .
- A database administrator shut down the database cleanly after losing one or more nodes. See Recovery After Clean Shutdown .
- An Eon Mode database went into read-only mode after quorum or primary shard coverage loss. See Recovery of a Read-Only Eon Mode Database .
- An Enterprise Mode database shut down uncleanly due to loss of quorum or K-safety, or a database in either mode shut down due to site-wide failures. See Recovery After Unclean Shutdown .

In the first three cases, nodes automatically rejoin the database after you resolve their failure; in the fourth case (unclean shutdown), you must manually intervene to recover the database. The following sections discuss these cases in greater detail.

If a recovery causes a table or schema to exceed its disk quota, the recovery proceeds anyway. You must then either reduce disk usage or increase the quota before you can perform other operations that consume disk space. For more information, see Disk quotas .

Recovery of failed nodes

One or more nodes in your database have failed. However, the database maintained quorum and K-safety so it continued running without interruption.

Recover the down nodes by restarting the Vertica process on them using:

- The admintools Restart Vertica on Host option.
- The start or restart buttons on the **Manage > Subcluster** page in the Management Console. See Starting, stopping, and restarting nodes in MC .

While restarted nodes recover their data from other nodes, their status is set to RECOVERING . Except for a short period at the end, the recovery phase has no effect on database transaction processing. After recovery is complete, the restarted nodes status changes to UP .

Recovery after clean shutdown

An administrator shut down the database cleanly after the loss of nodes. To recover:

1. Resolve any hardware or system problems that caused the node's host to go down.
2. Restart the database. See Starting the database .

On restart, all nodes whose status was UP before the shutdown resume a status of UP . If the database contained one or more failed nodes on shutdown and they are now available, they begin the recovery process as described in the previous section.

Recovery of a read-only Eon Mode database

A database in Eon Mode has lost enough primary nodes to cause it to go into read-only mode. To return the database to normal operation, restart the failed nodes. See Recover from Read-Only Mode .

Recovery after unclean shutdown

In an unclean shutdown, Vertica was not able to complete a normal shutdown process. Reasons for unclean shutdown include:

- A critical node in an Enterprise Mode database failed, leaving part of the database's data unavailable. The database immediately shuts down to prevent potential data corruption.
- A site-wide event such as a power failure caused all nodes to reboot.
- Vertica processes on the nodes exited due to a software or hardware failure.

Unclean shutdown can put the database in an inconsistent state—for example, Vertica might have been in the middle of writing data to disk at the time of failure, and this process was left incomplete. When you restart the database, Vertica determines that normal startup is not possible and uses the Last Good Epoch to determine when data was last consistent on all nodes. Vertica prompts you to accept recovery with the suggested epoch. If you accept, the database recovers and all data changes after the Last Good Epoch are lost. If you do not accept, the database does not start.

Instead of accepting the recommended epoch, you can recover from a backup . You can also choose an epoch that precedes the Last Good Epoch, through the Administration Tools Advanced Menu option Roll Back Database to Last Good Epoch. This is useful in special situations—for example the failure occurs during a batch of loads, where it is easier to restart the entire batch, even though some of the work must be repeated. In most cases, you should accept the recommended epoch.

Epochs and node recovery

The checkpoint epochs (CPEs) for both the source and target projections are updated as ROS containers are moved. The start and end epochs of all storage containers, such as ROS containers, are modified to the commit epoch. When this occurs, the epochs of all columns without an actual data file rewrite advance the CPE to the commit epoch of MOVE_PARTITIONS_TO_TABLE . If any nodes are down during the partition move operation, they detect that there is storage to recover. On rejoining the cluster, the restarted nodes recover from other nodes with the correct epoch.

See Epochs for additional information about how Vertica uses epochs.

Manual recovery notes

- You can manually recover a database where up to K nodes are offline—for example, they were physically removed for repair or were not reachable at the time of recovery. When the missing nodes are restored, they recover and rejoin the cluster as described in Recovery Scenarios .
- You can manually recover a database if the nodes to be restarted can supply all partition segments, even if more than K nodes remain down at startup. In this case, all data is available from the remaining cluster nodes, so the database can successfully start.
- The default setting for the HistoryRetentionTime configuration parameter is 0, so Vertica only keeps historical data when nodes are down. This setting prevents use of the Administration tools Roll Back Database to Last Good Epoch option because the AHM remains close to the current epoch and a rollback is not permitted to an epoch that precedes the AHM. If you rely on the Roll Back option to remove recently loaded data, consider setting a day-wide window to remove loaded data. For example:

  ```
  => ALTER DATABASE DEFAULT SET HistoryRetentionTime = 86400;
  ```

  For more information, see Epoch management parameters .
- When a node is down and manual recovery is required, it can take a full minute or longer for Vertica processes to time out while the system tries to form a cluster. Wait approximately one minute until the system returns the manual recovery prompt. Do not press CTRL-C during database startup.

In this section

- Restarting Vertica on a host
- Restarting the database
- Recovering the cluster from a backup
- Phases of a recovery
- Epochs
- Best practices for disaster recovery
- Recovery by table

Restarting Vertica on a host

When one node in a running database cluster fails, or if any files from the catalog or data directories are lost from any one of the nodes, you can check the status of failed nodes using either the Administration Tools or the Management Console.

Restarting Vertica on a host using the administration tools

1. Run Administration tools .
2. From the Main Menu, select **Restart Vertica on Host** and click **OK** .
3. Select the database host you want to recover and click **OK** .

> **Note**
> You might see additional nodes in the list, which are used internally by the Administration Tools. You can safely ignore these nodes.

4. Verify recovery state by selecting **View Database Cluster State** from the **Main Menu** .

After the database is fully recovered, you can check the status at any time by selecting **View Database Cluster State** from the Administration Tools **Main Menu** .

Restarting Vertica on a host using the Management Console

1. Connect to a cluster node (or the host on which MC is installed).
2. Open a browser and connect to MC as an MC administrator.
3. On the MC **Home** page, double-click the running database under the **Recent Databases** section.
4. Within the **Overview** page, look at the node status under the Database sub-section and see if all nodes are up. The status will indicate how many nodes are up, critical, down, recovering, or other.
5. If a node is down, click **Manage** at the bottom of the page and inspect the graph. A failed node will appear in red.
6. Click the failed node to select it and in the Node List, click the **Start node** button.

Restarting the database

If you lose the Vertica process on more than one node (for example, due to power loss), or if the servers are shut down without properly shutting down the Vertica database first, the database cluster indicates that it did not shut down gracefully the next time you start it.

The database automatically detects when the cluster was last in a consistent state and then shuts down, at which point an administrator can restart it.

From the Main Menu in the Administration tools :

1. Verify that the database has been stopped by clicking **Stop Database** .
   A message displays: No databases owned by < *dbadmin* > are running
2. Start the database by selecting **Start Database** from the Main Menu.
3. Select the database you want to restart and click **OK** .
   If you are starting the database after an unclean shutdown, messages display, which indicate that the startup failed. Press **RETURN** to continue with the recovery process.
   An epoch represents committed changes to the data stored in a database between two specific points in time. When starting the database, Vertica searches for last good epoch .
4. Upon determining the last good epoch, you are prompted to verify that you want to start the database from the good epoch date. Select **Yes** to continue with the recovery.

   Caution
   If you do not want to start from the last good epoch, you may instead restore the data from a backup and attempt to restart the database. For this to be useful, the backup must be more current than the last good epoch.

   Vertica continues to initialize and recover all data prior to the last good epoch.
   If recovery takes more than a minute, you are prompted to answer <Yes> or <No> to "Do you want to continue waiting?"
   When all the nodes' status have changed to RECOVERING or UP, selecting <No> lets you exit this screen and monitor progress via the Administration Tools Main Menu. Selecting <Yes> continues to display the database recovery window.

   Note
   Be sure to reload any data that was added after the last good epoch date to which you have recovered.

Recovering the cluster from a backup

To recover a cluster from a backup, refer to the following topics:

- Backing up and restoring the database
- Restoring a database from a full backup

Phases of a recovery

The phases of a Vertica recovery are the same regardless of whether you are recovering by table or node. In the case of a recovery by table , tables become individually available as they complete the final phase. In the case of a recovery by node, the database objects only become available after the entire node completes recovery.

When you perform a recovery in Vertica, each recovered table goes through the following phases:

Order	Phase	Description	Lock Type
1	Historical	Vertica copies any historical data it may have missed while in a state of DOWN or INITIALIZING.	none
2	Historical Dirty	Vertica recovers any DML transactions that committed after the node or table began recovery.	none
3	Current Replay Delete	Vertica replays any delete transactions that took place during the recovery.	T-lock
4	Aggregate Projections	Vertica recovers any aggregate projections.	T-lock

After a table completes the last phase, Vertica considers it fully recovered. At this point, the table can participate in DDL and DML operations.

Epochs

An epoch represents a cutoff point of historical data within the database. The timestamp of all commits within a given epoch are equal to or less than the epoch's timestamp. Understanding epochs is useful when you need to perform the following operations:

- Database recovery : Vertica uses epochs to determine the last time data was consistent across all nodes in a database cluster.
- Execute historical queries : A SELECT statement that includes an AT *epoch* clause only returns data that was committed on or before the specified epoch.
- Purge deleted data : Deleted data is not removed from physical storage until it is purged from the database. You can purge deleted data from the database only if it precedes the ancient history marker (AHM) epoch.

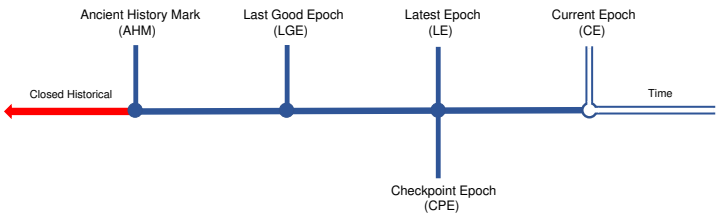
Vertica has one open epoch and any number of closed epochs, depending on your system configuration. New and updated data is written into the open epoch, and each closed epoch represents a previous commit to your database. When data is committed with a DML operation (INSERT, UPDATE, MERGE, COPY, or DELETE), Vertica writes the data, closes the open epoch, and opens a new epoch. Each row committed to the database is associated with the epoch in which it was written.

The EPOCHS system table contains information about each available closed epoch. The epoch_close_time column stores the date and time of the commit. The epoch_number column stores the corresponding epoch number:

```
=> SELECT * FROM EPOCHS;
     epoch_close_time       | epoch_number
-------------------------------+--------------
 2020-07-27 14:29:49.687106-04 |          91
 2020-07-28 12:51:53.291795-04 |          92
(2 rows)
```

Epoch milestones

As an epoch progresses through its life cycle, it reaches milestones that Vertica uses it to perform a variety of operations and maintain the state of the database. The following image generally depicts these milestones within the epoch life cycle:



Vertica defines each milestone as follows:

- Current epoch (CE): The current, open epoch that you are presently writing data to.
- Latest epoch (LE): The most recently closed epoch.
- Checkpoint epoch: Enterprise Mode only. A node-level epoch that is the latest epoch in which data is consistent across all projections on that node.
- Last good epoch (LGE): The minimum checkpoint epoch in which data is consistent across all nodes.
- Ancient history mark (AHM): The oldest epoch that contains data that is accessible by historical queries.

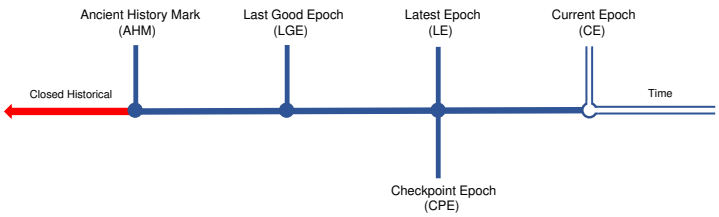
See Epoch life cycle for detailed information about each stage.

In this section

- Epoch life cycle
- Managing epochs
- Configuring epochs

Epoch life cycle

The epoch life cycle consists of a sequence of milestones that enable you to perform a variety of operations and manage the state of your database.



> **Note**
> Depending on your configuration, a single epoch can represent the latest epoch, last good epoch, checkpoint epoch, and ancient history mark.

Vertica provides epoch management parameters and functions so that you can retrieve and adjust epoch values. Additionally, see Configuring epochs for recommendations on how to set epochs for specific use cases.

Current epoch (CE)

The open epoch that contains all uncommitted changes that you are presently writing to the database. The current epoch is stored in the SYSTEM system table:

```
=> SELECT CURRENT_EPOCH FROM SYSTEM;
 CURRENT_EPOCH
---------------
         71
(1 row)
```

The following example demonstrates how the current epoch advances when you commit data:

1. Query the SYSTEM systems table to return the current epoch:
   ```
   => SELECT CURRENT_EPOCH FROM SYSTEM;
    CURRENT_EPOCH
   ---------------
            71
   (1 row)
   ```
   The current epoch is open, which means it is the epoch that you are presently writing data to.
2. Insert a row into the orders table:
   ```
   => INSERT INTO orders VALUES ('123456789', 323426, 'custacct@example.com');
    OUTPUT
   --------
         1
   (1 row)
   ```
   Each row of data has an implicit epoch column that stores that row's commit epoch. The row that you just inserted into the table was not committed, so the epoch column is blank:
   ```
   => SELECT epoch, orderkey, custkey, email_addrs FROM orders;
    epoch | orderkey  | custkey |      email_addrs
   -------+-----------+---------+----------------------
          | 123456789 |  323426 | custacct@example.com
   (1 row)
   ```
3. Commit the data, then query the table again. The committed data is associated with epoch 71 , the current epoch that was previously returned from the SYSTEM systems table:
   ```
   => COMMIT;
   COMMIT
   => SELECT epoch, orderkey, custkey, email_addrs FROM orders;
    epoch | orderkey  | custkey |      email_addrs
   -------+-----------+---------+----------------------
      71 | 123456789 |  323426 | custacct@example.com
   (1 row)
   ```
4. Query the SYSTEMS table again to return the current epoch. The current epoch is 1 integer higher:
   ```
   => SELECT CURRENT_EPOCH FROM SYSTEM;
    CURRENT_EPOCH
   ---------------
            72
   (1 row)
   ```

Latest epoch (LE)

The most recently closed epoch. The current epoch becomes the latest epoch after a commit operation.

The LE is the most recent epoch stored in the EPOCHS system table:

```
=> SELECT * FROM EPOCHS;
      epoch_close_time       | epoch_number
-----------------------------+--------------
 2020-07-27 14:29:49.687106-04 |          91
 2020-07-28 12:51:53.291795-04 |          92
(2 rows)
```

Checkpoint epoch (CPE)

Valid in Enterprise Mode only. Each node has a checkpoint epoch, which is the most recent epoch in which the data on that node is consistent across all projections. When the database runs optimally, the checkpoint epoch is equal to the LE, which is always one epoch older than the current epoch.

The checkpoint epoch is used during node failure and recovery. When a single node fails, that node attempts to rebuild data beyond its checkpoint epoch from other nodes. If the failed node cannot recover data using any of those epochs, then the failed node recovers data using the checkpoint epoch.

Use PROJECTION_CHECKPOINT_EPOCHS to query information about the checkpoint epochs. The following query returns information about the checkpoint epoch on nodes that store the orders projection:

```
=> SELECT checkpoint_epoch, node_name, projection_name, is_up_to_date, would_recover, is_behind_ahm
    FROM PROJECTION_CHECKPOINT_EPOCHS WHERE projection_name ILIKE 'orders_b%';
 checkpoint_epoch |    node_name     | projection_name | is_up_to_date | would_recover | is_behind_ahm
------------------+------------------+-----------------+---------------+---------------+---------------
               92 | v_vmart_node0001 | orders_b1       | t             | f             | f
               92 | v_vmart_node0001 | orders_b0       | t             | f             | f
               92 | v_vmart_node0003 | orders_b1       | t             | f             | f
               92 | v_vmart_node0003 | orders_b0       | t             | f             | f
               92 | v_vmart_node0002 | orders_b0       | t             | f             | f
               92 | v_vmart_node0002 | orders_b1       | t             | f             | f
(6 rows)
```

This query confirms that the database epochs are advancing correctly. The would_recover column displays an f when the last good epoch (LGE) is equal to the CPE because Vertica gives precedence to the LGE for recovery when possible. The is_behind_ahm column shows whether the checkpoint epoch is behind the AHM. Any data in an epoch that precedes the ancient history mark (AHM) is unrecoverable in case of a database or node failure.

Last good epoch (LGE)

The minimum checkpoint epoch in which data is consistent across all nodes in the cluster. Each node has an LGE, and Vertica evaluates the LGE for each node to determine the cluster LGE. The cluster's LGE is stored in the SYSTEM system table:

```
=> SELECT LAST_GOOD_EPOCH FROM SYSTEM;
 LAST_GOOD_EPOCH
-----------------
              70
(1 row)
```

You can retrieve the LGE for each node by querying the expected recovery epoch:

```
=> SELECT GET_EXPECTED_RECOVERY_EPOCH();
INFO 4544:  Recovery Epoch Computation:
Node Dependencies:
011 - cnt: 21
101 - cnt: 21
110 - cnt: 21
111 - cnt: 9

001 - name: v_vmart_node0001
010 - name: v_vmart_node0002
100 - name: v_vmart_node0003
Nodes certainly in the cluster:
       Node 0(v_vmart_node0001), epoch 70
       Node 1(v_vmart_node0002), epoch 70
Filling more nodes to satisfy node dependencies:
Data dependencies fulfilled, remaining nodes LGEs don't matter:
       Node 2(v_vmart_node0003), epoch 70
--
 GET_EXPECTED_RECOVERY_EPOCH
-----------------------------
                          70
(1 row)
```

Because the LGE is a snapshot of all of the most recent data on the disk, it is used to recover from database failure. Administration Tools uses the LGE to manually reset the database . If you are recovering from database failure after an unclean shutdown, Vertica prompts you to accept recovery using the LGE during restart.

Ancient history mark (AHM)

The oldest epoch that contains data that is accessible by historical queries . The AHM is stored in the SYSTEM system table:

```
=> SELECT AHM_EPOCH FROM SYSTEM;
 AHM_EPOCH
-----------
       70
(1 row)
```

Epochs that precede the AHM are unavailable for historical queries. The following example returns the AHM, and then returns an error when executing a historical query that precedes the AHM:

```
=> SELECT GET_AHM_EPOCH();
 GET_AHM_EPOCH
---------------
          93
(1 row)

=> AT EPOCH 92 SELECT * FROM orders;
ERROR 3183:  Epoch number out of range
HINT:  Epochs prior to [93] do not exist. Epochs [94] and later have not yet closed
```

The AHM advances according to your HistoryRetentionTime , HistoryRetentionEpochs , and AdvanceAHMInterval parameter settings . By default, the AHM advances every 180 seconds until it is equal with the LGE. This helps reduce the number of epochs saved to the epoch map , which reduces the catalog size. The AHM cannot advance beyond the LGE.

The AHM serves as the cutoff epoch for purging data from physical disk. As the AHM advances, the Tuple Mover mergeout process purges any deleted data that belongs to an epoch that precedes the AHM. See Purging deleted data for details about automated or manual purges.

Managing epochs

Epochs are stored in the epoch map, a catalog object that contains a list of closed epochs beginning at ancient history mark (AHM) epoch and ending at the latest epoch (LE). As the epoch map increases in size, the catalog uses more memory. Additionally, the AHM is used to determine what data is purged from disk. It is important to monitor database epochs to verify that they are advancing correctly to optimize database performance.

Monitoring epochs

When Vertica is running properly using the default Vertica settings, the ancient history mark, last good epoch (LGE), and checkpoint epoch (CPE, Enterprise Mode only) are equal to the latest epoch, or 1 less than the current epoch. This maintains control on the size of the epoch map and catalog by making sure that disk space is not used storing data that is eligible for purging. The SYSTEM system table stores the current epoch, last good epoch, and ancient history mark:

```
=> SELECT CURRENT_EPOCH, LAST_GOOD_EPOCH, AHM_EPOCH FROM SYSTEM;
 CURRENT_EPOCH | LAST_GOOD_EPOCH | AHM_EPOCH
---------------+-----------------+-----------
            88 |              87 |        87
(1 row)
```

Vertica provides GET_AHM_EPOCH , GET_AHM_TIME , GET_CURRENT_EPOCH , and GET_LAST_GOOD_EPOCH to retrieve these epochs individually.

In Enterprise Mode, you can query the checkpoint epoch using the PROJECTION_CHECKPOINT_EPOCHS table to return the checkpoint epoch for each node in your cluster. The following query returns the CPE for any node that stores the orders projection:

```
=> SELECT checkpoint_epoch, node_name, projection_name
    FROM PROJECTION_CHECKPOINT_EPOCHS WHERE projection_name ILIKE 'orders_b%';
 checkpoint_epoch |    node_name     | projection_name
------------------+------------------+-----------------
               87 | v_vmart_node0001 | orders_b1
               87 | v_vmart_node0001 | orders_b0
               87 | v_vmart_node0003 | orders_b1
               87 | v_vmart_node0003 | orders_b0
               87 | v_vmart_node0002 | orders_b0
               87 | v_vmart_node0002 | orders_b1
(6 rows)
```

Troubleshooting the ancient history mark

A properly functioning AHM is critical in determining how well your database utilizes disk space and executes queries. When you commit a DELETE or UPDATE (a combination of DELETE and INSERT) operation, the data is not deleted from disk immediately. Instead, Vertica marks the data for deletion so that you can retrieve it with historical queries. Deleted data takes up space on disk and impacts query performance because Vertica must read the deleted data during non-historical queries.

Epochs advance as you commit data, and any data that is marked for deletion is automatically purged by the Tuple Mover mergeout process when its epoch advances past the AHM. You can create an automated purge policy or manually purge any deleted data that was committed in an epoch that precedes the AHM. See Setting a purge policy for additional information.

By default, the AHM advances every 180 seconds until it is equal to the LGE. Monitor the SYSTEM system table to ensure that the AHM is advancing according properly:

```
=> SELECT CURRENT_EPOCH, LAST_GOOD_EPOCH, AHM_EPOCH FROM SYSTEM;
 CURRENT_EPOCH | LAST_GOOD_EPOCH | AHM_EPOCH
---------------+-----------------+-----------
            94 |              93 |        86
(1 row)
```

If you notice that the AHM is not advancing correctly, it might be due to one or more of the following:

- Your database contains unrefreshed projections. This occurs when you create a projection for a table that already contains data. See Refreshing projections for details on how to refresh projections.
- A node is DOWN. When a node is DOWN, the AHM cannot advance. See Restarting Vertica on a host for information on how to resolve this issue.

  > **Caution**
  > You can use the MAKE_AHM_NOW , SET_AHM_EPOCH , or SET_AHM_TIME epoch management functions to manually set the AHM to a specific epoch. If the selected epoch is later than the DOWN node's LGE, the node must recover from scratch upon restart.

- Confirm that the AHMBackupManagement epoch parameter is set to 0 . If this parameter is set to 1, the AHM does not advance beyond the most recent full backup:

```
=> SELECT node_name, parameter_name, current_value FROM CONFIGURATION_PARAMETERS WHERE
parameter_name='AHMBackupManagement';
 node_name |   parameter_name    | current_value
-----------+---------------------+---------------
 ALL       | AHMBackupManagement | 0
(1 row)
```

Configuring epochs

Epoch configuration impacts how your database recovers from failure, handles historical data, and purges data from disk. Vertica provides epoch management parameters for system-wide epoch configuration. Epoch management functions enable you to make ad hoc adjustments to epoch values.

Important
Epoch configuration has a significant impact on how your database functions. Make sure that you understand how epochs work before you save any configurations.

Historical query and recovery precision

When you execute a historical query, Vertica returns an epoch within the amount of time specified by the EpochMapInterval configuration parameter. For example, when you execute a historical query using the AT TIME *time* epoch clause, Vertica returns an epoch within the parameter setting. By default, EpochMapInterval is set to 180 seconds. You must set EpochMapInterval to a value greater than or equal to the AdvanceAHMInterval parameter:

```
=> SELECT node_name, parameter_name, current_value FROM CONFIGURATION_PARAMETERS
    WHERE parameter_name='EpochMapInterval' OR parameter_name='AdvanceAHMInterval';
 node_name |   parameter_name   | current_value
-----------+--------------------+---------------
 ALL       | EpochMapInterval   | 180
 ALL       | AdvanceAHMInterval | 180
(2 rows)
```

During failure recovery, Vertica uses the EpochMapInterval setting to determine which epoch is reported as the last good epoch (LGE).

History retention and purge workflows

Vertica recommends that you configure your epoch parameters to create a purge policy that determines when deleted data is purged from disk. If you use historical queries often, then you need to find a balance between saving deleted historical data and purging it from disk. An aggressive purge policy increases disk utilization and improves query performance, but also limits your recovery options and narrows the window of data available for historical queries.

There are two strategies to creating a purge policy:

- Set HistoryRetentionTime to specify how long deleted data is saved (in seconds) as an historical reference.
- Set HistoryRetentionEpochs to specify the number of historical epochs to save.

See Setting a purge policy for details about configuring each workflow.

Setting HistoryRetentionTime is the preferred method for creating a purge policy. By default, Vertica sets this value to 0 , so the AHM is 1 less than the current epoch when the database is running properly. You cannot execute historical queries on epochs that precede the AHM, so you might want to adjust this setting to save more data between the present time and the AHM. Another reason to adjust this parameter is if you use the Roll Back Database to Last Good Epoch option for manual roll backs. For example, the following command sets HistoryRetentionTime to 1 day (in seconds) to provide a wider range of epoch roll back options:

```
=> ALTER DATABASE vmart SET HistoryRetentionTime = 86400;
```

Vertica checks the status of your retention settings using the AdvanceAHMInterval setting and advances the AHM as necessary. After the AHM advances, any deleted data in an epoch that precedes the AHM is purged automatically by the Tuple Mover mergeout process .

If you want to disable any purge policy and preserve all historical data, set both HistoryRetentionTime and HistoryRetentionEpochs to -1 :

```
=> ALTER DABABASE vmart SET HistoryRetentionTime = -1;
=> ALTER DATABASE vmart SET HistoryRetentionEpochs = -1;
```

If you do not set a purge policy, you can use epoch management functions to adjust the AHM to manually purge deleted data as needed. Manual purges are useful if you need to update or delete data uploaded by mistake. See Manually purging data for details.

Best practices for disaster recovery

To protect your database from site failures caused by catastrophic disasters, maintain an off-site replica of your database to provide a standby. In case of disaster, you can switch database users over to the standby database. The amount of data loss between a disaster and fail over to the offsite replica depends on how frequently you save a full database backup.

The solution to employ for disaster recover depends upon two factors that you must determine for your application:

- **Recovery point objective (RPO)** : How much data loss can your organization tolerate upon a disaster recovery?
- **Recovery time objective (RTO):** How quickly do you need to recover the database following a disaster?

Depending on your RPO and RTO, Vertica recommends choosing from the following solutions:

1. **Dual-load:** During each load process for the database, simultaneously load a second database. You can achieve this easily with off-the-shelf ETL software.
2. **Periodic Incremental Backups** : Use the procedure described in Copying the database to another cluster to periodically copy the data to the target database. Remember that the script copies only files that have changed.
3. **Replication solutions provided by Storage Vendors** : Although some users have had success with SAN storage, the number of vendors and possible configurations prevent Vertica from providing support for SANs.

The following table summarizes the RPO, RTO, and the pros and cons of each approach:

	Dual Load	**Periodic Incremental**	**Storage Replication**
RPO	Up to the minute data	Up to the last backup	Recover to the minute
RTO	Available at all times	Available except when backup in progress	Available at all times
Pros	• Standby database can have different configuration • Can use the standby database for queries	• Built-in scripts • High performance due to compressed file transfers	Transparent to the database
Cons	• Possibly incur additional ETL licenses • Requires application logic to handle errors	Need identical standby system	• More expensive • Media corruptions are also replicated

Recovery by table

Vertica supports node recovery on a per-table basis. Unlike node-based recovery, recovering by table makes tables available as they recover, before the node itself is completely restored. You can prioritize your most important tables so they become available as soon as possible. Recovered tables support all DDL and DML operations.

To enhance recovery speed, Vertica recovers multiple tables in parallel. The maximum number of tables recoverable at one time is set by the MAXCONCURRENCY parameter in the RECOVERY resource pool .

After a node has fully recovered, it enables full Vertica functionality.

In this section

- Prioritizing table recovery
- Viewing table recovery status

Prioritizing table recovery

You can specify the order in which Vertica recovers tables. This feature ensures that your most critical tables become available as soon as possible. To specify the recovery order of your tables, assign an integer priority value. Tables with higher priority values recover first. For example, a table with a priority of 1000 is recovered before a table with a value of 500. Table priorities have the maximum value of a 64-bit integer.

If you do not assign a priority, or if multiple tables have the same priority, Vertica restores tables by OID order. Assign a priority with a query such as this:

```
=> SELECT set_table_recover_priority('avro_basic', '1000');
     set_table_recover_priority
---------------------------------------
 Table recovery priority has been set.
(1 row)
```

View assigned priorities with a query using this form: SELECT table_name,recover_priority FROM v_catalog.tables; The next example shows prioritized tables from the VMart sample database. In this case, the table with the highest recovery priorities are listed first (DESC). The shipping_dimension table has the highest priority and will be recovered first. (Example has hard Returns for display purposes.)

```
=> SELECT table_name AS Name, recover_priority from v_catalog.tables WHERE recover_priority > 1
   ORDER BY recover_priority DESC;
       Name         | recover_priority
--------------------+------------------
 shipping_dimension |            60000
 warehouse_dimension |           50000
 employee_dimension |            40000
 vendor_dimension   |            30000
 date_dimension     |            20000
 promotion_dimension |           10000
 iris2              |             9999
 product_dimension  |               10
 customer_dimension |               10
(9 rows)
```

Viewing table recovery status

View general information about a recovery querying the V_MONITOR. TABLE_RECOVERY_STATUS table. You can also view detailed information about the status of the recovery the table being restored by querying the V_MONITOR. TABLE_RECOVERIES table.

Collecting database statistics

The Vertica cost-based query optimizer relies on data statistics to produce query plans. If statistics are incomplete or out-of-date, the optimizer is liable to use a sub-optimal plan to execute a query.

When you query a table, the Vertica optimizer checks for statistics as follows:

1. If the table is partitioned, the optimizer checks whether the partitions required by this query have recently been analyzed. If so, it retrieves those statistics and uses them to facilitate query planning.
2. Otherwise, the optimizer uses table-level statistics, if available.
3. If no valid partition- or table-level statistics are available, the optimizer assumes uniform distribution of data values and equal storage usage for all projections.

Statistics management functions

Vertica provides two functions that generate up-to-date statistics on table data: ANALYZE_STATISTICS and ANALYZE_STATISTICS_PARTITION collect table-level and partition-level statistics, respectively. After computing statistics, the functions store them in the database catalog.

Both functions perform the following operations:

- Collect statistics using historical queries (at epoch latest) without any locks.
- Perform fast data sampling, which expedites analysis of relatively small tables with a large number of columns.
- Recognize deleted data instead of ignoring delete markers.

Vertica also provides several functions that help you management database statistics—for example, to export and import statistics, validate statistics, and drop statistics.

After you collect the desired statistics, you can run Workload Analyzer to retrieve hints about under-performing queries and their root causes, and obtain tuning recommendations.

In this section

- Collecting table statistics
- Collecting partition statistics
- Analyzing row counts
- Canceling statistics collection

- Getting data on table statistics
- Best practices for statistics collection

Collecting table statistics

ANALYZE_STATISTICS collects and aggregates data samples and storage information from all nodes that store projections of the target tables.

You can set the scope of the collection at several levels:

- Database
- Table
- Table columns

ANALYZE_STATISTICS can also control the size of the data sample that it collects.

Analyze all database tables

If ANALYZE_STATISTICS specifies no table, it collects statistics for all database tables and their projections. For example:

```
=> SELECT ANALYZE_STATISTICS ('');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

Analyze a single table

You can compute statistics on a single table as follows:

```
=> SELECT ANALYZE_STATISTICS ('public.store_orders_fact');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

When you query system table PROJECTION_COLUMNS , it confirms that statistics have been collected on all table columns for all projections of store_orders_fact :

```
=> SELECT projection_name, statistics_type, table_column_name,statistics_updated_timestamp
    FROM projection_columns WHERE projection_name ilike 'store_orders_fact%' AND table_schema='public';
  projection_name    | statistics_type | table_column_name | statistics_updated_timestamp
---------------------+-----------------+-------------------+-------------------------------
 store_orders_fact_b0 | FULL            | product_key       | 2019-04-04 18:06:55.747329-04
 store_orders_fact_b0 | FULL            | product_version   | 2019-04-04 18:06:55.747329-04
 store_orders_fact_b0 | FULL            | store_key         | 2019-04-04 18:06:55.747329-04
 store_orders_fact_b0 | FULL            | vendor_key        | 2019-04-04 18:06:55.747329-04
 store_orders_fact_b0 | FULL            | employee_key      | 2019-04-04 18:06:55.747329-04
 store_orders_fact_b0 | FULL            | order_number      | 2019-04-04 18:06:55.747329-04
 store_orders_fact_b0 | FULL            | date_ordered      | 2019-04-04 18:06:55.747329-04
 store_orders_fact_b0 | FULL            | date_shipped      | 2019-04-04 18:06:55.747329-04
 store_orders_fact_b0 | FULL            | quantity_ordered  | 2019-04-04 18:06:55.747329-04
 store_orders_fact_b0 | FULL            | shipper_name      | 2019-04-04 18:06:55.747329-04
 store_orders_fact_b1 | FULL            | product_key       | 2019-04-04 18:06:55.747329-04
 store_orders_fact_b1 | FULL            | product_version   | 2019-04-04 18:06:55.747329-04
...
(20 rows)
```

Analyze table columns

Within a table, you can narrow scope of analysis to a subset of its columns. Doing so can save significant processing overhead for big tables that contain many columns. It is especially useful if you frequently query these tables on specific columns.

Important
If you collect statistics on specific columns, be sure to include all columns that you are likely to query. If a query includes other columns in that table, the query optimizer regards the statistics as incomplete for that query and ignores them in its plan.

For example, instead of collecting statistics on all columns in store_orders_fact , you can select only those columns that are frequently queried: product_key , product_version , order_number , and quantity_shipped :

```
=> SELECT DROP_STATISTICS('public.store_orders_fact');
=> SELECT ANALYZE_STATISTICS ('public.store_orders_fact', 'product_key, product_version, order_number, quantity_ordered');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

If you query PROJECTION_COLUMNS again, it returns the following results:

```
=> SELECT projection_name, statistics_type, table_column_name,statistics_updated_timestamp
    FROM projection_columns WHERE projection_name ilike 'store_orders_fact%' AND table_schema='public';
   projection_name    | statistics_type | table_column_name | statistics_updated_timestamp
----------------------+-----------------+-------------------+-------------------------------
 store_orders_fact_b0 | FULL            | product_key       | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b0 | FULL            | product_version   | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b0 | ROWCOUNT        | store_key         | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b0 | ROWCOUNT        | vendor_key        | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b0 | ROWCOUNT        | employee_key      | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b0 | FULL            | order_number      | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b0 | ROWCOUNT        | date_ordered      | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b0 | ROWCOUNT        | date_shipped      | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b0 | FULL            | quantity_ordered  | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b0 | ROWCOUNT        | shipper_name      | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b1 | FULL            | product_key       | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b1 | FULL            | product_version   | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b1 | ROWCOUNT        | store_key         | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b1 | ROWCOUNT        | vendor_key        | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b1 | ROWCOUNT        | employee_key      | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b1 | FULL            | order_number      | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b1 | ROWCOUNT        | date_ordered      | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b1 | ROWCOUNT        | date_shipped      | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b1 | FULL            | quantity_ordered  | 2019-04-04 18:09:40.05452-04
 store_orders_fact_b1 | ROWCOUNT        | shipper_name      | 2019-04-04 18:09:40.05452-04
(20 rows)
```

In this case, columns statistics_type is set to FULL only for those columns on which you ran ANALYZE_STATISTICS. The remaining table columns are set to ROWCOUNT , indicating that only row statistics were collected for them.

> **Note**
> ANALYZE_STATISTICS always invokes ANALYZE_ROW_COUNT on all table columns, even if ANALYZE_STATISTICS specifies a subset of those columns.

Data collection percentage

By default, Vertica collects a fixed 10-percent sample of statistical data from disk. Specifying a percentage of data to read from disk gives you more control over deciding between sample accuracy and speed.

The percentage of data you collect affects collection time and accuracy:

- A smaller percentage is faster but returns a smaller data sample, which might compromise histogram accuracy.
- A larger percentage reads more data off disk. Data collection is slower, but *a larger data sample enables greater histogram accuracy.*

For example:

Collect data on all projections for shipping_dimension from 20 percent of the disk:

```
=> SELECT ANALYZE_STATISTICS ('shipping_dimension', 20);
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

Collect data from the entire disk by setting the percent parameter to 100:

```
=> SELECT ANALYZE_STATISTICS ('shipping_dimension', 'shipping_key', 100);
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

Sampling size

ANALYZE_STATISTICS constructs a column histogram from a set of rows that it randomly selects from all collected data. Regardless of the percentage setting, the function always creates a statistical sample that contains up to (approximately) the smaller of:

- 2^{17} (131,072) rows
- Number of rows that fit in 1 GB of memory

If a column has fewer rows than the maximum sample size, ANALYZE_STATISTICS reads all rows from disk and analyzes the entire column.

> **Note**
> The data collected in a sample range does not indicate how data should be distributed.

The following table shows how ANALYZE_STATISTICS, when set to different percentages, obtains a statistical sample from a given column:

Number of column rows	%	Number of rows read	Number of sampled rows
<= max-sample-size	20	All	All
400K	10	*max-sample-size*	*max-sample-size*
4000K	10	400K	*max-sample-size*

> **Note**
> When a column specified for ANALYZE_STATISTICS is first in a projection's sort order, the function reads all data from disk to avoid a biased sample.

Collecting partition statistics

ANALYZE_STATISTICS_PARTITION collects and aggregates data samples and storage information for a range of partitions in the specified table. Vertica writes the collected statistics to the database catalog.

For example, the following table stores sales data and is partitioned by order dates:

```
CREATE TABLE public.store_orders_fact
(
    product_key int,
    product_version int,
    store_key int,
    vendor_key int,
    employee_key int,
    order_number int,
    date_ordered date NOT NULL,
    date_shipped date NOT NULL,
    quantity_ordered int,
    shipper_name varchar(32)
);

ALTER TABLE public.store_orders_fact PARTITION BY date_ordered::DATE GROUP BY CALENDAR_HIERARCHY_DAY(date_ordered::DATE, 2, 2)
REORGANIZE;
ALTER TABLE public.store_orders_fact ADD CONSTRAINT fk_store_orders_product FOREIGN KEY (product_key, product_version) references
public.product_dimension (product_key, product_version);
ALTER TABLE public.store_orders_fact ADD CONSTRAINT fk_store_orders_vendor FOREIGN KEY (vendor_key) references public.vendor_dimension
(vendor_key);
ALTER TABLE public.store_orders_fact ADD CONSTRAINT fk_store_orders_employee FOREIGN KEY (employee_key) references
public.employee_dimension (employee_key);
```

At the end of each business day you might call ANALYZE_STATISTICS_PARTITION and collect statistics on all data of the latest (today's) partition:

```
=> SELECT ANALYZE_STATISTICS_PARTITION('public.store_orders_fact', CURRENT_DATE::VARCHAR(10), CURRENT_DATE::VARCHAR(10));
 ANALYZE_STATISTICS_PARTITION
------------------------------
                            0
(1 row)
```

The function produces a set of fresh statistics for the most recent partition in store.store_sales_fact . If you query this table each morning on yesterday's sales, the optimizer uses these statistics to generate an optimized query plan:

```
=> EXPLAIN SELECT COUNT(*) FROM public.store_orders_fact WHERE date_ordered = CURRENT_DATE-1;

                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 QUERY PLAN DESCRIPTION:
 ------------------------------

 EXPLAIN SELECT COUNT(*) FROM public.store_orders_fact WHERE date_ordered = CURRENT_DATE-1;

 Access Path:
 +-GROUPBY NOTHING [Cost: 2, Rows: 1] (PATH ID: 1)
 |  Aggregates: count(*)
 |  Execute on: All Nodes
 | +---> STORAGE ACCESS for store_orders_fact [Cost: 1, Rows: 222(PARTITION-LEVEL STATISTICS)] (PATH ID: 2)
 | |      Projection: public.store_orders_fact_v1_b1
 | |      Filter: (store_orders_fact.date_ordered = '2019-04-01'::date)
 | |      Execute on: All Nodes
```

Narrowing the collection scope

Like ANALYZE_STATISTICS , ANALYZE_STATISTICS_PARTITION lets you narrow the scope of analysis to a subset of a table's columns . You can also control the size of the data sample that it collects. For details on these options, see Collecting table statistics .

Collecting statistics on multiple partition ranges

If you specify multiple partitions, they must be continuous. Different collections of statistics can overlap. For example, the following table t1 is partitioned on column c1 :

```
=> SELECT export_tables('','t1');
                                    export_tables
-------------------------------------------------------------------------------------------------------
CREATE TABLE public.t1
(
    a int,
    b int,
    c1 int NOT NULL
)
PARTITION BY (t1.c1);


=> SELECT * FROM t1 ORDER BY c1;
 a  | b  | c1
----+----+----
  1 |  2 |  3
  4 |  5 |  6
  7 |  8 |  9
 10 | 11 | 12
(4 rows)
```

Given this dataset, you can call ANALYZE_STATISTICS_PARTITION on t1 twice. The successive calls collect statistics for two overlapping ranges of partition keys, 3 through 9 and 6 through 12:

```
=> SELECT drop_statistics_partition('t1', '', '');
 drop_statistics_partition
---------------------------
                         0
(1 row)

=> SELECT analyze_statistics_partition('t1', '3', '9');
 analyze_statistics_partition
------------------------------
                            0
(1 row)

=> SELECT analyze_statistics_partition('t1', '6', '12');
 analyze_statistics_partition
------------------------------
                            0
(1 row)

=> SELECT table_name, min_partition_key, max_partition_key, row_count FROM table_statistics WHERE table_name = 't1';
 table_name | min_partition_key | max_partition_key | row_count
------------+-------------------+-------------------+-----------
 t1         | 3                 | 9                 |         3
 t1         | 6                 | 12                |         3
(2 rows)
```

If two statistics collections overlap, Vertica stores only the most recent statistics for each partition range. Thus, given the previous example, Vertica uses only statistics from the second collection for partition keys 6 through 9.

Statistics that are collected for a given range of partition keys always supersede statistics that were previously collected for a subset of that range. For example, given a call to ANALYZE_STATISTICS_PARTITION that specifies partition keys 3 through 12, the collected statistics are a superset of the two sets of statistics collected earlier, so it supersedes both:

```
=> SELECT analyze_statistics_partition('t1', '3', '12');
 analyze_statistics_partition
------------------------------
                            0
(1 row)

=> SELECT table_name, min_partition_key, max_partition_key, row_count FROM table_statistics WHERE table_name = 't1';
 table_name | min_partition_key | max_partition_key | row_count
------------+-------------------+-------------------+-----------
 t1         | 3                 | 12                |         4
(1 row)
```

Finally, ANALYZE_STATISTICS_PARTITION collects statistics on partition keys 3 through 6. This collection is a subset of the previous collection, so Vertica retains both sets and uses the latest statistics from each:

```
=> SELECT analyze_statistics_partition('t1', '3', '6');
 analyze_statistics_partition
------------------------------
                            0
(1 row)

=> SELECT table_name, min_partition_key, max_partition_key, row_count FROM table_statistics WHERE table_name = 't1';
 table_name | min_partition_key | max_partition_key | row_count
------------+-------------------+-------------------+-----------
 t1         | 3                 | 12                |         4
 t1         | 3                 | 6                 |         2
(2 rows)
```

Supported date/time functions

ANALYZE_STATISTICS_PARTITION can collect partition-level statistics on tables where the partition expression specifies one of the following date/time functions:

- DATE
- DATE_PART
- DATE_TRUNC
- DAY
- DAYOFMONTH
- DAYOFYEAR
- DAYS
- EXTRACT
- HOUR
- MINUTE
- MONTH
- QUARTER
- WEEK
- WEEK_ISO
- YEAR
- YEAR_ISO

Requirements and restrictions

The following requirements and restrictions apply to ANALYZE_STATISTICS_PARTITION:

- The table must be partitioned and cannot contain unpartitioned data.
- The table partition expression must specify a single column. The following expressions are supported:
  - Expressions that specify only the column—that is, partition on all column values. For example:
    ```
    PARTITION BY ship_date GROUP BY CALENDAR_HIERARCHY_DAY(ship_date, 2, 2)
    ```
  - If the column is a DATE or TIMESTAMP/TIMESTAMPTZ , the partition expression can specify a supported date/time function that returns that column or any portion of it, such as month or year. For example, the following partition expression specifies to partition on the year portion of column order_date :

```
PARTITION BY YEAR(order_date)
```

- Expressions that perform addition or subtraction on the column. For example:

```
PARTITION BY YEAR(order_date) -1
```

- The table partition expression cannot coerce the specified column to another data type.
- Vertica collects no statistics from the following projections:
  - Live aggregate and Top-K projections
  - Projections that are defined to include an SQL function within an expression

Analyzing row counts

Vertica lets you obtain row counts for projections and for external tables, through ANALYZE_ROW_COUNT and ANALYZE_EXTERNAL_ROW_COUNT, respectively.

Projection row count

ANALYZE_ROW_COUNT is a lightweight operation that collects a minimal set of statistics and aggregate row counts for a projection, and saves it in the database catalog. In many cases, this data satisifes many optimizer requirements for producing optimal query plans. This operation is invoked on the following occasions:

- At the time intervals specified by configuration parameter AnalyzeRowCountInterval —by default, once a day.
- During loads. Vertica updates the catalog with the current aggregate row count data for a given table when the percentage of difference between the last-recorded aggregate projection row count and current row count exceeds the setting in configuration parameter ARCCommitPercentage.
- On calls to meta-functions ANALYZE_STATISTICS and ANALYZE_STATISTICS_PARTITION.

You can explicitly invoke ANALYZE_ROW_COUNT through calls to DO_TM_TASK. For example:

```
=> SELECT DO_TM_TASK('analyze_row_count', 'store_orders_fact_b0');
                                  do_tm_task
-------------------------------------------------------------------------------------------------
 Task: row count analyze
(Table: public.store_orders_fact) (Projection: public.store_orders_fact_b0)

(1 row)
```

You can change the intervals when Vertica regularly collects row-level statistics by setting configuration parameter AnalyzeRowCountInterval. For example, you can change the collection interval to 1 hour (3600 seconds):

```
=> ALTER DATABASE DEFAULT SET AnalyzeRowCountInterval = 3600;
ALTER DATABASE
```

External table row count

ANALYZE_EXTERNAL_ROW_COUNT calculates the exact number of rows in an external table. The optimizer uses this count to optimize for queries that access external tables. This is especially useful when an external table participates in a join. This function enables the optimizer to identify the smaller table to use as the inner input to the join, and facilitate better query performance.

The following query calculates the exact number of rows in the external table loader_rejects :

```
=> SELECT ANALYZE_EXTERNAL_ROW_COUNT('loader_rejects');
 ANALYZE_EXTERNAL_ROW_COUNT
----------------------------
                          0
```

Canceling statistics collection

To cancel statistics collection mid analysis, execute CTRL-C on vsql or call the INTERRUPT_STATEMENT() function.

If you want to remove statistics for the specified table or type, call the DROP_STATISTICS() function.

> **Caution**
> After you drop statistics, it can be time consuming to regenerate them.

Getting data on table statistics

Vertica provides information about statistics for a given table and its columns and partitions in two ways:

- The query optimizer notifies you about the availability of statistics to process a given query.
- System table PROJECTION_COLUMNS shows what types of statistics are available for the table columns, and when they were last updated.

Query evaluation

During predicate selectivity estimation, the query optimizer can identify when histograms are not available or are out of date. If the value in the predicate is outside the histogram's maximum range, the statistics are stale. If no histograms are available, then no statistics are available to the plan.

When the optimizer detects stale or no statistics, such as when it encounters a column predicate for which it has no histogram, the optimizer performs the following actions:

- Displays and logs a message that you should run ANALYZE_STATISTICS .
- Annotates EXPLAIN -generated query plans with a statistics entry.
- Ignores stale statistics when it generates a query plan. The optimizer uses other considerations to create a query plan, such as FK-PK constraints.

For example, the following query plan fragment shows no statistics (histograms unavailable):

```
| | +-- Outer -> STORAGE ACCESS for fact [Cost: 604, Rows: 10K (NO STATISTICS)]
```

The following query plan fragment shows that the predicate falls outside the histogram range:

```
| | +-- Outer -> STORAGE ACCESS for fact [Cost: 35, Rows: 1 (PREDICATE VALUE OUT-OF-RANGE)]
```

Statistics data in PROJECTION_COLUMNS

Two columns in system table PROJECTION_COLUMNS show the status of each table column's statistics, as follows:

- STATISTICS_TYPE returns the type of statistics that are available for this column, one of the following: NONE , ROWCOUNT , or FULL .
- STATISTICS_UPDATED_TIMESTAMP returns the last time statistics were collected for this column.

For example, the following sample schema defines a table named trades, which groups the highly-correlated columns bid and ask and stores the stock column separately:

```
=> CREATE TABLE trades (stock CHAR(5), bid INT, ask INT);
=> CREATE PROJECTION trades_p (
    stock ENCODING RLE, GROUPED(bid ENCODING DELTAVAL, ask))
    AS (SELECT * FROM trades) ORDER BY stock, bid;
=> INSERT INTO trades VALUES('acme', 10, 20);
=> COMMIT;
```

Query the PROJECTION_COLUMNS table for table trades :

```
=> SELECT table_name AS table, projection_name AS projection, table_column_name AS column, statistics_type, statistics_updated_timestamp AS
last_updated
    FROM projection_columns WHERE table_name = 'trades';
 table  | projection  | column | statistics_type | last_updated
--------+-------------+--------+-----------------+--------------
 trades | trades_p_b0 | stock  | NONE            |
 trades | trades_p_b0 | bid    | NONE            |
 trades | trades_p_b0 | ask    | NONE            |
 trades | trades_p_b1 | stock  | NONE            |
 trades | trades_p_b1 | bid    | NONE            |
 trades | trades_p_b1 | ask    | NONE            |
(6 rows)
```

The statistics_type column returns NONE for all columns in the trades table, while statistics_updated_timestamp is empty because statistics have not yet been collected on this table.

Now, run ANALYZE_STATISTICS on the stock column:

```
=> SELECT ANALYZE_STATISTICS ('public.trades', 'stock');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

Now, when you query PROJECTION_COLUMNS , it returns the following results:

```
=> SELECT table_name AS table, projection_name AS projection, table_column_name AS column, statistics_type, statistics_updated_timestamp AS
last_updated
    FROM projection_columns WHERE table_name = 'trades';
 table  | projection  | column | statistics_type |         last_updated
--------+-------------+--------+-----------------+-------------------------------
 trades | trades_p_b0 | stock  | FULL            | 2019-04-03 12:00:12.231564-04
 trades | trades_p_b0 | bid    | ROWCOUNT        | 2019-04-03 12:00:12.231564-04
 trades | trades_p_b0 | ask    | ROWCOUNT        | 2019-04-03 12:00:12.231564-04
 trades | trades_p_b1 | stock  | FULL            | 2019-04-03 12:00:12.231564-04
 trades | trades_p_b1 | bid    | ROWCOUNT        | 2019-04-03 12:00:12.231564-04
 trades | trades_p_b1 | ask    | ROWCOUNT        | 2019-04-03 12:00:12.231564-04
(6 rows)
```

This time, the query results contain several changes:

statistics_type	• Set to FULL for the stock column, confirming that full statistics were run on this column. • Set to ROWCOUNT for the bid and ask columns, confirming that ANALYZE_STATISTICS always invokes ANALYZE_ROW_COUNT on all table columns, even if ANALYZE_STATISTICS specifies a subset of those columns.
statistics_updated_timestamp	Set to the same timestamp for all columns, confirming that statistics (either full or row count) were updated on all.

Best practices for statistics collection

You should call ANALYZE_STATISTICS or ANALYZE_STATISTICS_PARTITION when one or more of following conditions are true:

- Data is bulk loaded for the first time.
- A new projection is refreshed.
- The number of rows changes significantly.
- A new column is added to the table.
- Column minimum/maximum values change significantly.
- New primary key values with referential integrity constraints are added . The primary key and foreign key tables should be re-analyzed.
- Table size notably changes relative to other tables it is joined to—for example, a table that was 50 times larger than another table is now only five times larger.
- A notable deviation in data distribution necessitates recalculating histograms—for example, an event causes abnormally high levels of trading for a particular stock.
- The database is inactive for an extended period of time.

Overhead considerations

Running ANALYZE_STATISTICS is an efficient but potentially long-running operation. You can run it concurrently with queries and loads in a production environment. However, the function can incur considerable overhead on system resources (CPU and memory), at the expense of queries and load operations. To minimize overhead, consider calling ANALYZE_STATISTICS_PARTITIONS on those partitions that are subject to significant activity—typically, the most recently loaded partitions, including the table's active partition. You can further narrow the scope of both functions by specifying a subset of the table columns—generally, those that are queried most often.

Related tools

You can diagnose and resolve many statistics-related issues by calling ANALYZE_WORKLOAD , which returns tuning recommendations. If you update statistics and find that a query still performs poorly, run it through the Database Designer and choose incremental as the design type.

Using diagnostic tools

In this section

- Determining your version of Vertica

- Collecting diagnostics: scrutinize command
- Exporting a catalog
- Exporting profiling data

Determining your version of Vertica

To determine which version of Vertica is installed on a host, log in to that host and type:

```
$ rpm -qa | grep vertica
```

The command returns the name of the installed package, which contains the version and build numbers. The following example indicates that both Vertica 9.3.x and Management Console 9.3.x are running on the targeted host:

```
$ rpm -qa | grep vertica
vertica-9.3.0-0
vertica-console-9.3.0-0.x86_64
```

When you are logged in to your Vertica Analytic Database database, you can also run a query for the version only, by running the following command:

```
=> SELECT version();
                version
-----------------------------------------
 Vertica Analytic Database v9.3.0-0
```

Collecting diagnostics: scrutinize command

The diagnostics tool scrutinize collects a broad range of information from a Vertica cluster. It also supports a range of options that let you control the amount and type of data that is collected. Collected data can include but is not limited to:

- Host diagnostics and configuration data
- Run-time state (number of nodes up or down)
- Log files from the installation process, the database, and the administration tools (such as, vertica.log, dbLog, /opt/vertica/log/adminTools.log)
- Error messages
- Database design
- System table information, such as system, resources, workload, and performance
- Catalog metadata, such as system configuration parameters
- Backup information

Requirements

scrutinize requires that a cluster be configured to support the Administration Tools utility. If Administration Tools cannot run on the initiating host, then scrutinize cannot run on that host.

In this section

- Running scrutinize
- Informational options
- Redirecting scrutinize output
- Scrutinize security
- Data collection scope
- Uploading scrutinize results
- Troubleshooting scrutinize

Running scrutinize

You can run scrutinize with the following command:

```
$ /opt/vertica/bin/scrutinize
```

Unqualified, scrutinize collects a wide range of information from all cluster nodes. It stores the results in a .tar file (VerticaScrutinize. *NumericID* .tar), with minimal effect on database performance. scrutinize output can help diagnose most issues and yet reduces upload size by omitting fine-grained profiling data.

> **Note**
> scrutinize is designed to collect information for troubleshooting your database and cluster. Depending on your system configuration, logs

generated from running scrutinize might contain proprietary information. If you are concerned with sharing proprietary information, please remove it from the .tar file before you send it to Vertica Customer Support for assistance.

Command options

scrutinize options support the following tasks:

- Obtain version information about scrutinize and Vertica, and online help.
- Redirect output .
- Access a password-protected database .
- Control the scope of data collection .
- Upload results to Vertica Customer Support.

Privileges

In order for scrutinize to collect data from all system tables, you must have superuser or SYSMONITOR privileges; otherwise, scrutinize collects data only from the system tables that you have privileges to access. If you run scrutinize as root when the dbadmin user exists, Vertica returns an error.

Disk space requirements

scrutinize requires temporary disk space where it can collect data before posting the final compressed (.tar) output. How much space depends on variables such as the size of the Vertica log and extracted system tables, as well as user-specified options that limit the scope of information collected. Before scrutinize runs, it verifies that the temporary directory contains at least 1 GB of space; however, the actual amount needed can be much higher.

You can redirect scrutinize output to another directory. For details, see Redirecting scrutinize output .

Database specification

If multiple databases are defined on the cluster and more than one is active, or none is active, you must run scrutinize with one of the following options:

```
$ /opt/vertica/bin/scrutinize {--database=`*`database `*`| -d `*`database`*`}
```

If you omit this option when these conditions are true, scrutinize returns with an error.

Informational options

scrutinize supports two informational options that cannot be combined with any other options:

--version

Obtains the version number of the Vertica server and the scrutinize version number, and then exits. For example:

```
$ scrutinize --version
Scrutinize Version 12.0.2-20221107
```

--help -h

Lists all scrutinize options to the console, and then exits:

```
$ scrutinize -h
Usage: scrutinize [options]

Options:
 --version             show program's version number and exit
 -h, --help            show this help message and exit
 -X LIST, --exclude-tasks=LIST
                       Skip tasks of a particular type. Provide a comma-
                       separated lists of types to skip. Types are case-
                       sensitive. Possible types are: Command, File,
                       VerticaLog, DC, SystemTable, CatalogObject, Query,
                       UdxLog, KafkaLog, MemoryReportLog, all.
 -v, --vsql-off        Does -X Query,SystemTable and skips vsql checks.
                       Useful if vertica is running, but slow to respond.
 -s, --local_diags     Gather diagnostics for local machine only
 -d DB, --database=DB  Only report on database <DB>
 -n HOST_LIST, --hosts=HOST_LIST
                       Gather diagnostics for these hosts only. Host list
                       must be a comma-separated list. Ex. host1,host2,host3
                       or 'host1, host2, host3'
 -m MESSAGE, --message=MESSAGE
                       Reason for gathering diagnostics
 -o OUTPUT_DIR, --output_dir=OUTPUT_DIR
                       redirect output to somewhere other than the current
                       directory
 -U USERNAME, --user=USERNAME
                       Specify DB user
 -P PASSWORD, --password=PASSWORD
                       Specify DB user password
 -W, --prompt-password
                       Force Scrutinize to prompt for DB user password
 ...
```

Redirecting scrutinize output

By default, scrutinize uses the temporary directory /opt/vertica/tmp execution to compile output while it executes. On completing its collection, it saves the collection to a tar file to the current directory. You can redirect scrutinize output with two options:

--tmpdir= ***path***

Directs temporary output to the specified path, where the following requirements apply to *path* :

- The directory must have at least 1 GB of free space.
- You must have write permission to it.

--output_dir= ***path***

-o ***path***

Saves scrutinize results to a tar file in *path* . For example:

```
$ scrutinize --output_dir="/my_diagnostics/"
```

Scrutinize security

scrutinize can specify user names and passwords as follows:

--user= ***username***

-U ***username***

Specifies the dbadmin user name. By default, scrutinize uses the user name of the invoking user.

--password= ***password*** **-P** ***password***

Sets the database password as an argument to the scrutinize command. Use this option if the administrator account (default dbadmin) has password authentication. If you omit this option on a password-protected database, scrutinize returns a warning, unless the environment variable VSQL_PASSWORD is set.

Passwords with special characters must be enclosed with single quotes. For example:

```
$ scrutinize -P '@passWord**'
$ scrutinize --password='$password1*'
```

`-prompt-password` `-W`

Specifies to prompt users for their database password before scrutinize begins to collect data.

Data collection scope

scrutinize options let you control the scope of the data collection. You can specify the scope of the data collection according to the following criteria:

- Amount of data , including its level of granularity
- Specific nodes
- Types of data to include and exclude

You can use these options singly or in combination, to achieve the desired level of granularity.

Amount of collected data

Several options let you limit how much data scrutinize collects:

--by-second

Collect data every second. This is the highest level of granularity when collecting from Data Collector tables.

--by-minute= *boolean-value*

Collect data every minute (if the value is true) or every hour (if the value is false).

--get-files *file-list*

Collect the specified additional files, including globs, where *file-list* is a semicolon-delimited list of files.

--include_gzlogs= *num-files*

-z *num-files*

Include *num-files* rotated log files (vertica.log*.gz) in the scrutinize output, where *num-files* can be one of the following:

- An integer specifies the number of rotated log files to collect.
- all specifies to collect all rotated log files.

By default, scrutinize includes three rotated log files.

--log-limit= *limit*

-l *limit*

How much data to collect from Vertica logs, in gigabytes, starting from the most recent log entry. By default, scrutinize collects 1 GB of log data.

Node-specific collection

By default, scrutinize collects data from all cluster nodes. You can specify that scrutinize collect from individual nodes in two ways:

--local_diags -s

Collect diagnostics only from the host on which scrutinize was invoked. To collect data from multiple nodes in the cluster, use the --hosts option.

--hosts= *host-list* -n *host-list*

Collect diagnostics only from the hosts specified in *host-list* , a comma-separated list of IP addresses or host names.

For example:

```
$ scrutinize --hosts=127.0.0.1,host_3,host_1
```

Types of data to include

scrutinize provides several options that let you specify the type of data to collect:

--debug

Collects debug information for the log.

--diag-dump

Limits the collection to database design, system tables, and Data Collector tables. Use this option to collect data to analyze system performance.

--diagnostics

Limits the collection to log file data and output from commands that are run against Vertica and its host system. Use this option to collect data to evaluate unexpected behavior in your Vertica system.

--include-ros-info

Includes ROS related information from system tables.

--no-active-queries `--with-active-queries`

Excludes diagnostic information from system tables and Data Collector tables about currently running queries. By default, scrutinize collects this information (--with-active-queries).

--tasks= *tasks* -T *tasks*

Gathers diagnostics on one or more tasks, as specified in a file or JSON list. This option is typically used together with --exclude .

> **Note**
> Use this option only in consultation with Vertica Customer Support

--type= *type* -t *type*

Type of diagnostics collection to perform, one of the following:

- profiling : Gather profiling data.
- context : Gather summary information.

--with-active-queries

The default setting, includes diagnostic information from system tables and Data Collector tables about currently running queries. To omit this data, use --no-active-queries .

Types of data to exclude

scrutinize options also let you specify the types of data to exclude from its collection:

--exclude= *tasks* -X *tasks*

Excludes one or more types of tasks from the diagnostics collection, where *tasks* is a comma-separated list of the tasks to exclude:

- all : All default tasks
- DC : Data Collector tables
- File : Log files from the installation process, the database, and Administration Tools, such as vertica .log , dbLog , and adminTools.log
- VerticaLog : Vertica logs
- CatalogObject : Vertica catalog metadata, such as system configuration parameters
- SystemTable : Vertica system tables that contain information about system, resources, workload, and performance
- Query : Vertica meta-functions that use vsql to connect to the database, such as EXPORT_CATALOG()
- Command : Operating system information, such as the length of time that a node has been up

> **Note**
> This option is typically used only in consultation with your Vertica Customer Support contact.

--no-active-queries

Omits diagnostic information from system tables and Data Collector tables about currently running queries. By default, scrutinize always collects active query information (--with-active-queries).

--vsql-off -v

Excludes Query and SystemTable tasks, which are used to connect to the database. This option can help you deal with problems that occur during an upgrade, and is typically used in the following cases:

- Vertica is running but is slow to respond.
- You haven't yet created a database but need help troubleshooting other cluster issues.

Uploading scrutinize results

scrutinize provides several options for uploading data to Vertica customer support.

Upload packaging

When you use an upload option, scrutinize does not bundle all output in a single tar file. Instead, each node posts its output directly to the specified URL as follows:

1. Uploads a smaller, context file, enabling Customer Support to review high-level information.
2. On completion of scrutinize execution, uploads the complete diagnostics collection.

Upload prerequisites

Before you run scrutinize with an upload option:

- Install the cURL program in the path for the database administrator user who is running scrutinize .
- Verify each node in the cluster can make an HTTP or FTP connection directly to the Internet.

Upload options

> **Note**
> Two options upload scrutinize output to a Vertica support-provided URL or FTP address: --auth-upload and --url . Each option authenticates the upload differently, as noted below.

--auth-upload= ***url***

-A ***url***

Uses your Vertica license to authenticate with the Vertica server, by uploading your customer name. Customer Support uses this information to verify your identity on receiving your uploaded file. This option requires a valid Vertica license.

--url ***=url***

-u ***url***

Requires *url* to include a user name and password that is supplied by Vertica Customer Support.

--message= ***message***

-m ***message***

Includes a message with the scrutinize output, where *message* is a text string, a path to a text file, or PROMPT to open an input stream in which to compose a message. scrutinize reads input until you type a period (.) on a new line. This closes the input stream, and **scrutinize** writes the message to the collected output.

The message is written in the output directory in reason.txt . If no message is specified, scrutinize generates the default message Unknown reason for collection . Messages typically include the following information:

- Reason for gathering/submitting diagnostics.
- Support-supplied case number and other issue-specific information, to help Vertica Customer Support identify your case and analyze the problem.

Examples

The --auth-upload option uses your Vertica to identify yourself:

```
$ scrutinize -U username -P 'password' --auth-upload="support-provided-url"
```

The --url option includes the FTP username and password, supplied by support, in the URL:

```
$ scrutinize -U username -P 'password' --url='ftp://username/password@customers.vertica.com/'
```

You can supply a message as a text string or in a text file:

```
$ scrutinize --message="re: case number #ABC-12345"
$ scrutinize --message="/path/to/msg.txt"
```

Alternatively, you can open an input stream and type a message:

```
$ scrutinize --message=PROMPT
Enter reason for collecting diagnostics; end with '.' on a line by itself:
Query performance degradation noticed around 9AM EST on Saturday
.
Vertica Scrutinize Report
-----------------------------
Result Dir:             /home/dbadmin/VerticaScrutinize.20131126083311
...
```

Troubleshooting scrutinize

The troubleshooting advice in this section can help you resolve common issues that you might encounter when using scrutinize .

Collection time is too slow

To speed up collection time, omit system tables when running an instance of scrutinize . Be aware that collecting from fewer nodes does not necessarily speed up the collection process.

Output size is too large

Output size depends on system table size and vertica log size.

To create a smaller scrutinize output, omit some system tables or truncate the vertica log. For more information, see Narrowing the Scope of scrutinize Data Collection.

System tables not collected on databases with password

Running scrutinize on a password-protected database might require you to supply a user name and password:

```
$ scrutinize -U username -P 'password'
```

Exporting a catalog

When you export a catalog you can quickly move a catalog to another cluster. Exporting a catalog transfers schemas, tables, constraints, projections, and views. System tables are not exported.

Exporting catalogs can also be useful for support purposes.

See the EXPORT_CATALOG function for details.

Exporting profiling data

The diagnostics audit script gathers system table contents, design, and planning objects from a running database and exports the data into a file named ./diag_dump_<timestamp>.tar.gz , where <timestamp> denotes when you ran the script.

If you run the script without parameters, you will be prompted for a database password.

Syntax

```
/opt/vertica/scripts/collect_diag_dump.sh [ -U value ] [ -w value ] [ -c ]
```

Arguments

-U *value*

User name, typically the database administrator account, dbadmin.

-w *value*

Database password.

-c

Include a compression analysis, resulting in a longer script execution time.

Example

The following command runs the audit script with all arguments:

```
$ /opt/vertica/scripts/collect_diag_dump.sh -U dbadmin -w password -c
```

Profiling database performance

You can profile database operations to evaluate performance. Profiling can deliver information such as the following:

- How much memory and how many threads each operator is allocated.
- How data flows through each operator at different points in time during query execution.
- Whether a query is network bound.

Profiling data can help provide valuable input into database design considerations, such as how best to segment and sort projections, or facilitate better distribution of data processing across the cluster.

For example, profiling can show data skew, where some nodes process more data than others. The rows produced counter in system table EXECUTION_ENGINE_PROFILES shows how many rows were processed by each operator. Comparing rows produced across all nodes for a given operator can reveal whether a data skew problem exists.

The topics in this section focus on obtaining profile data with vsql statements. You can also view profiling data in the Management Console .

In this section

- Enabling profiling
- Profiling single statements
- Labeling statements
- Real-time profiling
- Profiling query resource consumption
- Profiling query plans
- Sample views for counter information

Enabling profiling

You can enable profiling at three scopes:

- Globally across all database sessions
- The current session
- Individual SQL statements

Vertica meta-function SHOW_PROFILING_CONFIG shows whether profiling is enabled at global and session scopes. In the following example, the function shows that profiling is disabled across all categories for the current session, and enabled globally across all categories:

```
=> SELECT SHOW_PROFILING_CONFIG();
SHOW_PROFILING_CONFIG
-----------------------------------------
 Session Profiling: Session off, Global on
 EE Profiling:      Session off, Global on
 Query Profiling:   Session off, Global on
(1 row)
```

Global profiling

When global profiling is enabled or disabled for a given category, that setting persists across all database sessions. You set global profiling with ALTER DATABASE , as follows:

```
ALTER DATABASE db-spec SET profiling-category = {0 | 1}
```

profiling-category specifies a profiling category with one of the following arguments:

Argument	Data profiled
GlobalQueryProfiling	Query-specific information, such as query string and duration of execution, divided between two system tables: • QUERY_PLAN_PROFILES : Real-time status for each query plan path. • QUERY_PROFILES : Query information.
GlobalSessionProfiling	General information about query execution on each node during the current session, stored in system table SESSION_PROFILES .
GlobalEEProfiling	Execution engine data, saved in system tables QUERY_CONSUMPTION and EXECUTION_ENGINE_PROFILES .

For example, the following statement globally enables query profiling on the current (DEFAULT) database:

```
=> ALTER DATABASE DEFAULT SET GlobalQueryProfiling = 1;
```

Session profiling

Session profiling can be enabled for the current session, and persists until you explicitly disable profiling, or the session ends. You set session profiling with the following Vertica meta-functions:

- ENABLE_PROFILING (*profiling-type*)
- DISABLE_PROFILING (*profiling-type*)

profiling-type specifies type of profiling data to enable or disable with one of the following arguments:

Argument	Data profiled

query	Query-specific information, such as query string and duration of execution, divided between two system tables: • QUERY_PLAN_PROFILES : Real-time status for each query plan path. • QUERY_PROFILES : Query information.
session	General information about query execution on each node during the current session, stored in system table SESSION_PROFILES .
ee	Execution engine data, saved in system tables QUERY_CONSUMPTION and EXECUTION_ENGINE_PROFILES .

For example, the following statement enables session-scoped profiling for the execution run of each query:

```
=> SELECT ENABLE_PROFILING('ee');
  ENABLE_PROFILING
----------------------
 EE Profiling Enabled
(1 row)
```

Statement profiling

You can enable profiling for individual SQL statements by prefixing them with the keyword PROFILE . You can profile a SELECT statement, or any DML statement such as INSERT , UPDATE , COPY , and MERGE . For detailed information, see Profiling single statements .

Precedence of profiling scopes

Vertica checks session and query profiling at the following scopes in descending order of precedence:

1. Statement profiling (highest)
2. Sesssion profiling (ignored if global profiling is enabled)
3. Global profiling (lowest)

Regardless of query and session profiling settings, Vertica always saves a minimum amount of profiling data in the pertinent system tables: QUERY_PROFILES , QUERY_PLAN_PROFILES , and SESSION_PROFILES .

For execution engine profiling, Vertica first checks the setting of configuration parameter SaveDCEEProfileThresholdUS . If the query runs longer than the specified threshold (by default, 60 seconds), Vertica gathers execution engine data for that query and saves it to system tables QUERY_CONSUMPTION and EXECUTION_ENGINE_PROFILES . Vertica uses profiling settings of other scopes (statement, session, global) only if the query's duration is below the threshold.

Important

To disable or minimize execution engine profiling:

- Set SaveDCEEProfileThresholdUS to a very high value, up to its maximum value of 2147483647 (2^{31} -1, or ~35.79 minutes).
- Disable profiling at session and global scopes.

Profiling single statements

To profile a single statement, prefix it with PROFILE . You can profile a query (SELECT) statement, or any DML statement such as INSERT , UPDATE , COPY , and MERGE . The statement returns with a profile summary:

- Profile identifiers transaction_id and statement_id
- Initiator memory for the query
- Total memory required

For example:

```
=> PROFILE SELECT customer_name, annual_income FROM public.customer_dimension
   WHERE (customer_gender, annual_income) IN (SELECT customer_gender, MAX(annual_income)
   FROM public.customer_dimension GROUP BY customer_gender);NOTICE 4788:  Statement is being profiled
HINT:  Select * from v_monitor.execution_engine_profiles where transaction_id=45035996274760535 and statement_id=1;
NOTICE 3557:  Initiator memory for query: [on pool general: 2783428 KB, minimum: 2312914 KB]
NOTICE 5077:  Total memory required by query: [2783428 KB]
  customer_name   | annual_income
------------------+---------------
 James M. McNulty |        999979
 Emily G. Vogel   |        999998
(2 rows)
```

You can use the profile identifiers transaction_id and statement_id to obtain detailed profile information for this query from system tables EXECUTION_ENGINE_PROFILES and QUERY_PLAN_PROFILES . You can also use these identifiers to obtain resource consumption data from system table QUERY_CONSUMPTION .

For example:

```
=> SELECT path_id, path_line::VARCHAR(68), running_time FROM v_monitor.query_plan_profiles
    WHERE transaction_id=45035996274760535 AND statement_id=1 ORDER BY path_id, path_line_index;
 path_id |                          path_line                          |  running_time
---------+-------------------------------------------------------------+-----------------
       1 | +-JOIN HASH [Semi] [Cost: 631, Rows: 25K (NO STATISTICS)] (PATH ID:  | 00:00:00.052478
       1 | |  Join Cond: (customer_dimension.customer_gender = VAL(2)) AND (cus |
       1 | |  Materialize at Output: customer_dimension.customer_name           |
       1 | |  Execute on: All Nodes                                             |
       2 | | +-- Outer -> STORAGE ACCESS for customer_dimension [Cost: 30, Rows | 00:00:00.051598
       2 | | |      Projection: public.customer_dimension_b0                    |
       2 | | |      Materialize: customer_dimension.customer_gender, customer_d |
       2 | | |      Execute on: All Nodes                                       |
       2 | | |      Runtime Filters: (SIP1(HashJoin): customer_dimension.custom |
       4 | | | +---> GROUPBY HASH (GLOBAL RESEGMENT GROUPS) (LOCAL RESEGMENT GR | 00:00:00.050566
       4 | | | |      Aggregates: max(customer_dimension.annual_income)         |
       4 | | | |      Group By: customer_dimension.customer_gender              |
       4 | | | |      Execute on: All Nodes                                     |
       5 | | | | +---> STORAGE ACCESS for customer_dimension [Cost: 30, Rows: 5 | 00:00:00.09234
       5 | | | | |      Projection: public.customer_dimension_b0                |
       5 | | | | |      Materialize: customer_dimension.customer_gender, custom |
       5 | | | | |      Execute on: All Nodes                                   |
(17 rows)
```

Labeling statements

To quickly identify queries and other operations for profiling and debugging purposes, include the LABEL hint.

LABEL hints are valid in the following statements:

- COPY
- DELETE
- EXPORT statements:
  - EXPORT TO DELIMITED
  - EXPORT TO ORC
  - EXPORT TO PARQUET
  - EXPORT TO VERTICA
- INSERT
- MERGE
- SELECT
- UPDATE
- UNION : Valid in the UNION's first SELECT statement. Vertica ignores labels in subsequent SELECT statements.

For example:

```
SELECT /*+label(myselectquery)*/ COUNT(*) FROM t;
INSERT /*+label(myinsertquery)*/ INTO t VALUES(1);
```

After you add a label to one or more statements, query the QUERY_PROFILES system table to see which queries ran with your supplied labels. The QUERY_PROFILES system table IDENTIFIER column returns the user-defined label that you previously assigned to a statement. You can also obtain other query-specific data that can be useful for querying other system tables, such as transaction IDs.

For example:

```
=> SELECT identifier, query FROM query_profiles;
    identifier | query
---------------+-------------------------------------------------------------
 myselectquery | SELECT /*+label(myselectquery)*/ COUNT(*) FROM t;
 myinsertquery | INSERT /*+label(myinsertquery)*/ INTO t VALUES(1);
 myupdatequery | UPDATE /*+label(myupdatequery)*/ t SET a = 2 WHERE a = 1;
 mydeletequery | DELETE /*+label(mydeletequery)*/ FROM t WHERE a = 1;
               | SELECT identifier, query from query_profiles;
(5 rows)
```

Real-time profiling

You can monitor long-running queries while they execute by querying system table EXECUTION_ENGINE_PROFILES . This table contains available profiling counters for internal operations and user statements. You can use the Linux watch command to query this table at frequent intervals.

Queries for real-time profiling data require a transaction ID. If the transaction executes multiple statements, the query also requires a statement ID to identify the desired statement. If you profile individual queries , the query returns with the statement's transaction and statement IDs. You can also obtain transaction and statement IDs from the SYSTEM_SESSIONS system table.

Profiling counters

The EXECUTION_ENGINE_PROFILES system table contains available profiling counters for internal operations and user statements. Real-time profiling counters are available for all statements while they execute, including internal operations such as mergeout , recovery , and refresh . Unless you explicitly enable profiling using the keyword PROFILE on a specific SQL statement, or generally enable profiling for the database and/or the current session, profiling counters are unavailable after the statement completes.

Useful counters include:

- Execution time (µs)
- Rows produced
- Total merge phases
- Completed merge phases
- Current size of temp files (bytes)

You can view all available counters by querying EXECUTION_ENGINE_PROFILES :

```
=> SELECT DISTINCT(counter_name) FROM EXECUTION_ENGINE_PROFILES;
```

To monitor the profiling counters, you can run a command like the following using a retrieved transaction ID (a000000000027):

```
=> SELECT * FROM execution_engine_profiles
  WHERE TO_HEX(transaction_id)='a000000000027'
  AND counter_name = 'execution time (us)'
  ORDER BY node_name, counter_value DESC;
```

The following example finds operators with the largest execution time on each node:

```
=> SELECT node_name, operator_name, counter_value execution_time_us FROM V_MONITOR.EXECUTION_ENGINE_PROFILES WHERE
counter_name='execution time (us)' LIMIT 1 OVER(PARTITION BY node_name ORDER BY counter_value DESC);
    node_name     | operator_name | execution_time_us
------------------+---------------+-------------------
 v_vmart_node0001 | Join          |            131906
 v_vmart_node0002 | Join          |            227778
 v_vmart_node0003 | NetworkSend   |            524080
(3 rows)
```

Linux watch command

You can use the Linux watch command to monitor long-running queries at frequent intervals. Common use cases include:

- Observing executing operators within a query plan on each Vertica cluster node.
- Monitoring workloads that might be unbalanced among cluster nodes—for example, some nodes become idle while others are active. Such imbalances might be caused by data skews or by hardware issues.

In the following example, watch queries operators with the largest execution time on each node. The command specifies to re-execute the query each second:

```
watch -n 1 -d "vsql VMart -c\"SELECT node_name, operator_name, counter_value execution_time_us
FROM v_monitor.execution_engine_profiles WHERE counter_name='execution time (us)'
LIMIT 1 OVER(PARTITION BY node_name ORDER BY counter_value DESC);

Every 1.0s: vsql VMart -c"SELECT node_name, operator_name, counter_value execution_time_us FROM v_monitor.execu...  Thu Jan 21 15:00:44 2016

    node_name     | operator_name | execution_time_us
------------------+---------------+-------------------
 v_vmart_node0001 | Root          |            110266
 v_vmart_node0002 | UnionAll      |             38932
 v_vmart_node0003 | Scan          |             22058
(3 rows)
```

Profiling query resource consumption

Vertica collects data on resource usage of all queries—including those that fail—and summarizes this data in system table QUERY_CONSUMPTION . This data includes the following information about each query:

- Wall clock duration
- CPU cycles consumed
- Memory reserved and allocated
- Network bytes sent and received
- Disk bytes read and written
- Bytes spilled
- Threads allocated
- Rows output to client
- Rows read and written

You can obtain information about individual queries through their transaction and statement IDs. Columns TRANSACTION_ID and STATEMENT_ID provide a unique key to each query statement.

> **Note**
> One exception applies: a query with multiple plans has a record for each plan.

For example, the following query is profiled:

```
=> PROFILE SELECT pd.category_description AS 'Category', SUM(sf.sales_quantity*sf.sales_dollar_amount) AS 'Total Sales'
    FROM store.store_sales_fact sf
    JOIN public.product_dimension pd ON pd.product_version=sf.product_version AND pd.product_key=sf.product_key
    GROUP BY pd.category_description;
NOTICE 4788:  Statement is being profiled
HINT:  Select * from v_monitor.execution_engine_profiles where transaction_id=45035996274751822 and statement_id=1;
NOTICE 3557:  Initiator memory for query: [on pool general: 256160 KB, minimum: 256160 KB]
NOTICE 5077:  Total memory required by query: [256160 KB]
          Category          | Total Sales
----------------------------------+-------------
 Non-food                         |  1147919813
 Misc                             |  1158328131
 Medical                          |  1155853990
 Food                             |  4038220327
(4 rows)
```

You can use the transaction and statement IDs that Vertica returns to get profiling data from QUERY_CONSUMPTION —for example, the total number of bytes sent over the network for a given query:

```
=> SELECT NETWORK_BYTES_SENT FROM query_consumption WHERE transaction_id=45035996274751822 AND statement_id=1;
 NETWORK_BYTES_SENT
--------------------
             757745
(1 row)
```

> **Note**
> QUERY_CONSUMPTION saves data from all queries, whether explicitly profiled or not.

QUERY_CONSUMPTION versus EXECUTION _ENGINE_PROFILES

QUERY_CONSUMPTION includes data that it rolls up from counters in EXECUTION_ENGINE_PROFILES . In the previous example, NETWORK_BYTES_SENT rolls up data that is accessible through multiple counters in EXECUTION_ENGINE_PROFILES . The equivalent query on EXECUTION_ENGINE_PROFILES looks like this:

```
=> SELECT operator_name, counter_name, counter_tag, SUM(counter_value) FROM execution_engine_profiles
    WHERE transaction_id=45035996274751822 AND statement_id=1 AND counter_name='bytes sent'
    GROUP BY ROLLUP (operator_name, counter_name, counter_tag) ORDER BY 1,2,3, GROUPING_ID();
 operator_name | counter_name |         counter_tag          |  SUM
---------------+--------------+------------------------------+--------
 NetworkSend   | bytes sent   | Net id 1000 - v_vmart_node0001 | 252471
 NetworkSend   | bytes sent   | Net id 1000 - v_vmart_node0002 | 251076
 NetworkSend   | bytes sent   | Net id 1000 - v_vmart_node0003 | 253717
 NetworkSend   | bytes sent   | Net id 1001 - v_vmart_node0001 |    192
 NetworkSend   | bytes sent   | Net id 1001 - v_vmart_node0002 |    192
 NetworkSend   | bytes sent   | Net id 1001 - v_vmart_node0003 |      0
 NetworkSend   | bytes sent   | Net id 1002 - v_vmart_node0001 |     97
 NetworkSend   | bytes sent   |                              | 757745
 NetworkSend   |              |                            | 757745
               |              |                          | 757745
(10 rows)
```

QUERY_CONSUMPTION and EXECUTION_ENGINE_PROFILES also differ as follows:

- QUERY_CONSUMPTION saves data from all queries, no matter their duration or whether they are explicitly profiled. It also includes data on unsuccessful queries.
- EXECUTION_ENGINE_PROFILES only includes data from queries whose length of execution exceeds a set threshold, or that you explicitly profile. It also excludes data of unsuccessful queries.

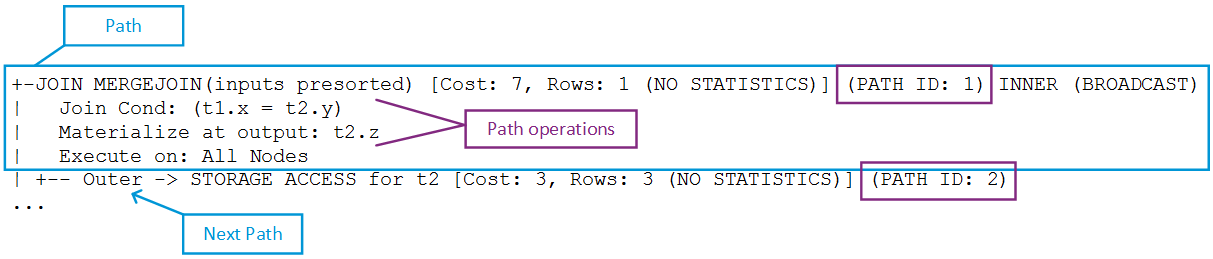
Profiling query plans

To monitor real-time flow of data through a query plan and its individual paths , query the following system tables:

EXECUTION_ENGINE_PROFILES and QUERY_PLAN_PROFILES . These tables provides data on how Vertica executed a query plan and its individual paths :

- EXECUTION_ENGINE_PROFILES summarizes query execution runs.
- QUERY_PLAN_PROFILES shows the real-time flow of data, and the time and resources consumed for each query plan path.

Each query plan path has a unique ID, as shown in the following EXPLAIN output fragment.



Both tables provide path-specific data. For example, QUERY_PLAN_PROFILES provides high-level data for each path, which includes:

- Length of a query operation execution
- How much memory that path's operation consumed
- Size of data sent/received over the network

For example, you might observe that a GROUP BY HASH operation executed in 0.2 seconds using 100MB of memory.

Requirements

Real-time profiling minimally requires the ID of the transaction to monitor. If the transaction includes multiple statements, you also need the statement ID. You can get statement and transaction IDs by issuing PROFILE on the query to profile. You can then use these identifiers to query system tables EXECUTION_ENGINE_PROFILES and QUERY_PLAN_PROFILES .

For more information, see Profiling single statements .

In this section

- Getting query plan status for small queries
- Getting query plan status for large queries
- Improving readability of QUERY_PLAN_PROFILES output
- Managing query profile data
- Analyzing suboptimal query plans

Getting query plan status for small queries

Real-time profiling counters, stored in system table EXECUTION_ENGINE_PROFILES , are available for all currently executing statements, including internal operations, such as a mergeout .

Profiling counters are available after query execution completes, if any one of the following conditions is true:

- The query was run via the PROFILE command
- Systemwide profiling is enabled by Vertica meta-function ENABLE_PROFILING .
- The query ran more than two seconds.

Profiling counters are saved in system table EXECUTION_ENGINE_PROFILES until the storage quota is exceeded.

For example:

1. Profile the query to get transaction_id and statement_id from from EXECUTION_ENGINE_PROFILES . For example:

```
=> PROFILE SELECT * FROM t1 JOIN t2 ON t1.x = t2.y;
NOTICE 4788:  Statement is being profiled
HINT:  Select * from v_monitor.execution_engine_profiles where transaction_id=45035996273955065 and statement_id=4;
NOTICE 3557:  Initiator memory for query: [on pool general: 248544 KB, minimum: 248544 KB]
NOTICE 5077:  Total memory required by query: [248544 KB]
 x | y |   z
---+---+-------
 3 | 3 | three
(1 row)
```

2. Query system table QUERY_PLAN_PROFILES .

> **Note**
> For best results, sort on columns transaction_id , statement_id , path_id , and path_line_index .

```
=> SELECT ... FROM query_plan_profiles
   WHERE transaction_id=45035996273955065 and statement_id=4;
   ORDER BY transaction_id, statement_id, path_id, path_line_index;
```

Getting query plan status for large queries

Real-time profiling is designed to monitor large (long-running) queries. Take the following steps to monitor plans for large queries:

1. Get the statement and transaction IDs for the query plan you want to profile by querying system table CURRENT_SESSION :

```
=> SELECT transaction_id, statement_id from current_session;
 transaction_id   | statement_id
------------------+--------------
 45035996273955001 |            4
(1 row)
```

2. Run the query:

```
=> SELECT * FROM t1 JOIN t2 ON x=y JOIN ext on y=z;
```

3. Query system table QUERY_PLAN_PROFILES , and sort on the transaction_id, statement_id, path_id, and path_line_index columns.

```
=> SELECT ... FROM query_plan_profiles WHERE transaction_id=45035996273955001 and statement_id=4
  ORDER BY transaction_id, statement_id, path_id, path_line_index;
```

You can also use the Linux watch command to monitor long-running queries (see Real-time profiling).

Example

The following series of commands creates a table for a long-running query and then queries system table QUERY_PLAN_PROFILES :

1. Create table longq :

```
=> CREATE TABLE longq(x int);
CREATE TABLE
=> COPY longq FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1
>> 2
>> 3
>> 4
>> 5
>> 6
>> 7
>> 8
>> 9
>> 10
>> \.
=> INSERT INTO longq SELECT f1.x+f2.x+f3.x+f4.x+f5.x+f6.x+f7.x
     FROM longq f1
     CROSS JOIN longq f2
     CROSS JOIN longq f3
     CROSS JOIN longq f4
     CROSS JOIN longq f5
     CROSS JOIN longq f6
     CROSS JOIN longq f7;
 OUTPUT
----------
 10000000
(1 row)
=> COMMIT;
COMMIT
```

2. Suppress query output on the terminal window by using the vsql \o command:

```
=> \o /home/dbadmin/longQprof
```

3. Query the new table:

```
=> SELECT * FROM longq;
```

4. Get the transaction and statement IDs:

```
=> SELECT transaction_id, statement_id from current_session;
  transaction_id   | statement_id
-------------------+--------------
 45035996273955021 |            4
(1 row)
```

5. Turn off the \o command so Vertica continues to save query plan information to the file you specified. Alternatively, leave it on and examine the file after you query system table QUERY_PLAN_PROFILES .

```
=> \o
```

6. Query system table QUERY_PLAN_PROFILES :

```
=> SELECT
    transaction_id,
    statement_id,
    path_id,
    path_line_index,
    is_executing,
    running_time,
    path_line
  FROM query_plan_profiles
  WHERE transaction_id=45035996273955021 AND statement_id=4
  ORDER BY transaction_id, statement_id, path_id, path_line_index;
```

Improving readability of QUERY_PLAN_PROFILES output

Output from the QUERY_PLAN_PROFILES table can be very wide because of the path_line column. To facilitate readability, query QUERY_PLAN_PROFILES using one or more of the following options:

- Sort output by transaction_id , statement_id , path_id , and path_line_index :

```
=> SELECT ... FROM query_plan_profiles
   WHERE ...
  ORDER BY transaction_id, statement_id, path_id, path_line_index;
```

- Use column aliases to decrease column width:

```
=> SELECT statement_id AS sid, path_id AS id, path_line_index AS order,
   is_started AS start, is_completed AS end, is_executing AS exe,
   running_time AS run, memory_allocated_bytes AS mem,
   read_from_disk_bytes AS read, received_bytes AS rec,
   sent_bytes AS sent, FROM query_plan_profiles
   WHERE transaction_id=45035996273910558 AND statement_id=3
   ORDER BY transaction_id, statement_id, path_id, path_line_index;
```

- Use the vsql \o command to redirect EXPLAIN output to a file:

```
=> \o /home/dbadmin/long-queries
=> EXPLAIN SELECT * FROM customer_dimension;
=> \o
```

Managing query profile data

Vertica retains data for queries until the storage quota for the table is exceeded, when it automatically purges the oldest queries to make room for new ones. You can also clear profiled data by calling one of the following functions:

- CLEAR_PROFILING clears profiled data from memory. For example, the following command clears profiling for general query-run information, such as the query strings used and the duration of queries.

```
=> SELECT CLEAR_PROFILING('query');
```

- CLEAR_DATA_COLLECTOR clears all memory and disk records on the Data Collector tables and functions and resets collection statistics in system table DATA_COLLECTOR .
- FLUSH_DATA_COLLECTOR waits until memory logs are moved to disk and then flushes the Data Collector, synchronizing the DataCollector log with the disk storage.

Configuring data retention policies

Vertica retains the historical data it gathers as specified by the configured retention policies.

Analyzing suboptimal query plans

If profiling uncovers a suboptimal query, invoking one of the following functions might help:

- ANALYZE_WORKLOAD analyzes system information held in system tables and provides tuning recommendations that are based on a combination of statistics, system and data collector events, and database-table-projection design.
- ANALYZE_STATISTICS collects and aggregates data samples and storage information from all nodes that store projections associated with the specified table or column.

You can also run your query through the Database Designer. See Incremental Design .

Sample views for counter information

The EXECUTION_ENGINE_PROFILES table contains the data for each profiling counter as a row within the table. For example, the execution time (us) counter is in one row, and the rows produced counter is in a second row. Since there are many different profiling counters, many rows of profiling data exist for each operator. Some sample views are installed by default to simplify the viewing of profiling counters.

Running scripts to create the sample views

The following script creates the v_demo schema and places the views in that schema.

```
/opt/vertica/scripts/demo_eeprof_view.sql
```

Viewing counter values using the sample views

There is one view for each of the profiling counters to simplify viewing of a single counter value. For example, to view the execution time for all operators, issue the following command from the database:

```
=> SELECT * FROM v_demo.eeprof_execution_time_us;
```

To view all counter values available for all profiled queries:

```
=> SELECT * FROM v_demo.eeprof_counters;
```

To select all distinct operators available for all profiled queries:

```
=> SELECT * FROM v_demo.eeprof_operators;
```

Combining sample views

These views can be combined:

```
=> SELECT * FROM v_demo.eeprof_execution_time_us
   NATURAL LEFT OUTER JOIN v_demo.eeprof_rows_produced;
```

To view the execution time and rows produced for a specific transaction and statement_id ranked by execution time on each node:

```
=> SELECT * FROM v_demo.eeprof_execution_time_us_rank
   WHERE transaction_id=45035996273709699
   AND statement_id=1
   ORDER BY transaction_id, statement_id, node_name, rk;
```

To view the top five operators by execution time on each node:

```
=> SELECT * FROM v_demo.eeprof_execution_time_us_rank
   WHERE transaction_id=45035996273709699
   AND statement_id=1 AND rk<=5
   ORDER BY transaction_id, statement_id, node_name, rk;
```

About locale

Locale specifies the user's language, country, and any special variant preferences, such as collation. Vertica uses locale to determine the behavior of certain string functions. Locale also determines the collation for various SQL commands that require ordering and comparison, such as aggregate GROUP BY and ORDER BY clauses, joins, and the analytic ORDER BY clause.

The default locale for a Vertica database is en_US@collation=binary (English US). You can define a new default locale that is used for all sessions on the database. You can also override the locale for individual sessions. However, projections are always collated using the default en_US@collation=binary collation, regardless of the session collation. Any locale-specific collation is applied at query time.

If you set the locale to null, Vertica sets the locale to en_US_POSIX . You can set the locale back to the default locale and collation by issuing the vsql meta-command \locale . For example:

Note

```
=> set locale to '';
INFO 2567:  Canonical locale: 'en_US_POSIX'
Standard collation: 'LEN'
English (United States, Computer)
SET
=> \locale en_US@collation=binary;
INFO 2567:  Canonical locale: 'en_US'
Standard collation: 'LEN_KBINARY'
English (United States)
=>  \locale
en_US@collation-binary;
```

You can set locale through ODBC , JDBC , and ADO.net .

Vertica locale specifications follow a subset of the Unicode LDML standard as implemented by the ICU library .

In this section

- Locale handling in Vertica
- Specifying locale: long form
- Specifying locale: short form
- Supported locales
- Locale and UTF-8 support
- Locale-aware string functions

Locale handling in Vertica

The following sections describes how Vertica handles locale.

Session locale

Locale is session-scoped and applies only to queries executed in that session. You cannot specify locale for individual queries. When you start a session it obtains its locale from the configuration parameter DefaultSessionLocale .

Query restrictions

The following restrictions apply when queries are run with locale other than the default en_US@collation=binary :

- When one or more of the left-side NOT IN columns is CHAR or VARCHAR , multi-column NOT IN subqueries are not supported . For example:

```
=> CREATE TABLE test (x VARCHAR(10), y INT);
=> SELECT ... FROM test WHERE (x,y) NOT IN (SELECT ...);
  ERROR: Multi-expression NOT IN subquery is not supported because a left
  hand expression could be NULL
```

> **Note**
> Even if columns test.x and test.y have a NOT NULL constraint, an error occurs.

- If the outer query contains a GROUP BY clause on a CHAR or VARCHAR column, correlated HAVING clause subqueries are not supported. In the following example, the GROUP BY x in the outer query causes the error:

```
=> DROP TABLE test CASCADE;
=> CREATE TABLE test (x VARCHAR(10));
=> SELECT COUNT(*) FROM test t GROUP BY x HAVING x
    IN (SELECT x FROM test WHERE t.x||'a' = test.x||'a' );
  ERROR: subquery uses ungrouped column "t.x" from outer query
```

- Subqueries that use analytic functions in the HAVING clause are not supported. For example:

```
=> DROP TABLE test CASCADE;
=> CREATE TABLE test (x VARCHAR(10));
=> SELECT MAX(x)OVER(PARTITION BY 1 ORDER BY 1) FROM test
    GROUP BY x HAVING x IN (SELECT MAX(x) FROM test);
  ERROR: Analytics query with having clause expression that involves
  aggregates and subquery is not supported
```

Collation and projections

Projection data is sorted according to the default en_US@collation=binary collation. Thus, regardless of the session setting, issuing the following command creates a projection sorted by col1 according to the binary collation:

```
=> CREATE PROJECTION p1 AS SELECT * FROM table1 ORDER BY col1;
```

In such cases, straße and strasse are not stored near each other on disk.

Sorting by binary collation also means that sort optimizations do not work in locales other than binary. Vertica returns the following warning if you create tables or projections in a non-binary locale:

```
WARNING:  Projections are always created and persisted in the default
Vertica locale. The current locale is de_DE
```

Non-binary locale input handling

When the locale is non-binary, Vertica uses the COLLATION function to transform input to a binary string that sorts in the proper order.

This transformation increases the number of bytes required for the input according to this formula:

```
result_column_width = input_octet_width * CollationExpansion + 4
```

The default value of configuration parameter CollationExpansion is 5.

Character data type handling

- CHAR fields are displayed as fixed length, including any trailing spaces. When CHAR fields are processed internally, they are first stripped of trailing spaces. For VARCHAR fields, trailing spaces are usually treated as significant characters; however, trailing spaces are ignored when sorting or comparing either type of character string field using a non-binary locale.
- The maximum length parameter for VARCHAR and CHAR data type refers to the number of octets (bytes) that can be stored in that field and not number of characters. When using multi-byte UTF-8 characters, the fields must be sized to accommodate from 1 to 4 bytes per character, depending on the data.

Specifying locale: long form

Vertica supports long forms that specify the collation keyword. Vertica extends long-form processing to accept collation arguments.

Syntax

```
[language][_script][_country][_variant][@collation-spec]
```

> **Note**
> The following syntax options apply:
>
> - Locale specification strings are case insensitive. For example, en_us and EN_US , are equivalent.
> - You can substitute underscores with hyphens. For example: [-script]

Parameters

language

A two- or three-letter lowercase code for a particular language. For example, Spanish is es English is en and French is fr . The two-letter language code uses the ISO-639 standard.

_ *script*

An optional four-letter script code that follows the language code. If specified, it should be a valid script code as listed on the Unicode ISO 15924 Registry.

_ *country*

A specific language convention within a generic language for a specific country or region. For example, French is spoken in many countries, but the currencies are different in each country. To allow for these differences among specific geographical, political, or cultural regions, locales are specified by two-letter, uppercase codes. For example, FR represents France and CA represents Canada. The two letter country code uses the ISO-3166 standard.

_ *variant*

Differences may also appear in language conventions used within the same country. For example, the Euro currency is used in several European countries while the individual country's currency is still in circulation. To handle variations inside a language and country pair, add a third code, the variant code. The variant code is arbitrary and completely application-specific. ICU adds _EURO to its locale designations for locales that support the Euro currency. Variants can have any number of underscored key words. For example, EURO_WIN is a variant for the Euro currency on a Windows computer.

Another use of the variant code is to designate the Collation (sorting order) of a locale. For instance, the es__TRADITIONAL locale uses the traditional sorting order which is different from the default modern sorting of Spanish.

@ *collation-spec*

Vertica only supports the keyword collation , as follows:

```
@collation=collation-type[;arg]...
```

Collation can specify one or more semicolon-delimited arguments , described below.
collation-type is set to one of the following values:

- big5han : Pinyin ordering for Latin, big5 charset ordering for CJK characters (used in Chinese).

- dict : For a dictionary-style ordering (such as in Sinhala).
- direct : Hindi variant.
- gb2312/gb2312han : Pinyin ordering for Latin, gb2312han charset ordering for CJK characters (used in Chinese).
- phonebook : For a phonebook-style ordering (such as in German).
- pinyin : Pinyin ordering for Latin and for CJK characters; that is, an ordering for CJK characters based on a character-by-character transliteration into a pinyin (used in Chinese).
- reformed : Reformed collation (such as in Swedish).
- standard : The default ordering for each language. For root it is [UCA] order; for each other locale it is the same as UCA (Unicode Collation Algorithm) ordering except for appropriate modifications to certain characters for that language. The following are additional choices for certain locales; they have effect only in certain locales.
- stroke : Pinyin ordering for Latin, stroke order for CJK characters (used in Chinese) not supported.
- traditional : For a traditional-style ordering (such as in Spanish).
- unihan : Pinyin ordering for Latin, Unihan radical-stroke ordering for CJK characters (used in Chinese) not supported.
- binary : Vertica default, providing UTF-8 octet ordering.

Notes:

- Collations might default to root, the ICU default collation.
- Invalid values of the collation keyword and its synonyms do not cause an error. For example, the following does not generate an error. It simply ignores the invalid value:

```
=> \locale en_GB@collation=xyz
INFO 2567:  Canonical locale: 'en_GB@collation=xyz'
Standard collation: 'LEN'
English (United Kingdom, collation=xyz)
```

For more about collation options, see Unicode Locale Data Markup Language (LDML) .

Collation arguments

collation can specify one or more of the following arguments :

Parameter	Short form	Description
colstrength	S	Sets the default strength for comparison. This feature is locale dependent. Set colstrength to one of the following: • 1 | primary : Ignores case and accents. Only primary differences are used during comparison—for example, a versus z . • 2 | secondary : Ignores case. Only secondary and above differences are considered for comparison—for example, different accented forms of the same base letter such as a versus \u00E4 . • 3 | tertiary (default): Only tertiary differences and higher are considered for comparison. Tertiary comparisons are typically used to evaluate case differences—for example, Z versus z . • 4 | quarternary : For example, used with Hiragana.
colAlternate	A	Sets alternate handling for variable weights, as described in UCA, one of the following: • non-ignorable | N | D • shifted | S

colBackwards	F	For Latin with accents, this parameter determines which accents are sorted. It sets the comparison for the second level to be backwards. **Note** colBackwards is automatically set for French accents. Set colBackwards to one of the following: • on | O : The normal UCA algorithm is used. • off | X: All strings that are in Fast C or D normalization form (FCD) sort correctly, but others do not necessarily sort correctly. Set to off if the strings to be compared are in FCD.
colNormalization	N	Set to one of the following: • on | O : The normal UCA algorithm is used. • off | X : All strings that are in Fast C or D normalization form (FCD) sort correctly, but others won't necessarily sort correctly. It should only be set off if the strings to be compared are in FCD.
colCaseLevel	E	Set to one of the following: • on | O : A level consisting only of case characteristics is inserted in front of tertiary level. To ignore accents but take cases into account, set strength to primary and case level to on. • off | X : This level is omitted.
colCaseFirst	C	Set to one of the following: • upper | U : Upper case sorts before lower case. • lower | L : Lower case sorts before upper case. This is useful for locales that have already supported ordering but require different order of cases. It affects case and tertiary levels. • off | short : Tertiary weights unaffected
colHiraganaQuarternary	H	Controls special treatment of Hiragana code points on quaternary level, one of the following: • on | O : Hiragana codepoints get lower values than all the other non-variable code points. The strength must be greater or equal than quaternary for this attribute to take effect. • off | X : Hiragana letters are treated normally.
colNumeric	D	If set to on, any sequence of Decimal Digits (General_Category = Nd in the [UCD]) is sorted at a primary level with its numeric value. For example, A-21 < A-123 .
variableTop	B	Sets the default value for the variable top. All code points with primary weights less than or equal to the variable top will be considered variable, and are affected by the alternate handling. For example, the following command sets variableTop to be HYPHEN (u2010) => \locale en_US@colalternate=shifted;variabletop=u2010

Locale processing notes

- Incorrect locale strings are accepted if the prefix can be resolved to a known locale version.
  For example, the following works because the language can be resolved:

```
=> \locale en_XX
INFO 2567:  Canonical locale: 'en_XX'
Standard collation: 'LEN'
English (XX)
```

The following does not work because the language cannot be resolved:

```
=> \locale xx_XX
xx_XX: invalid locale identifier
```

- POSIX-type locales such as en_US.UTF-8 work to some extent in that the encoding part "UTF-8" is ignored.
- Vertica uses the icu4c-4_2_1 library to support basic locale/collation processing with some extensions. This does not currently meet current standards for locale processing (https://tools.ietf.org/html/rfc5646).

Examples

Specify German locale as used in Germany (de), with phonebook -style collation:

```
=> \locale de_DE@collation=phonebook
INFO 2567:  Canonical locale: 'de_DE@collation=phonebook'
Standard collation: 'KPHONEBOOK_LDE'
German (Germany, collation=Phonebook Sort Order)
Deutsch (Deutschland, Sortierung=Telefonbuch-Sortierregeln)
```

Specify German locale as used in Germany (de), with phonebook -style collation and strength set to secondary:

```
=> \locale de_DE@collation=phonebook;colStrength=secondary
INFO 2567:  Canonical locale: 'de_DE@collation=phonebook'
Standard collation: 'KPHONEBOOK_LDE_S2'
German (Germany, collation=Phonebook Sort Order)
Deutsch (Deutschland, Sortierung=Telefonbuch-Sortierregeln)
```

Specifying locale: short form

Vertica accepts locales in short form. You can use the short form to specify the locale and keyname pair/value names.

To determine the short form for a locale, type in the long form and view the last line of INFO, as follows:

```
\locale frINFO:  Locale: 'fr'
INFO:    French
INFO:    franÃ§ais
INFO:  Short form: 'LFR'
```

Examples

Specify en (English) locale:

```
\locale LENINFO:  Locale: 'en'
INFO:  English
INFO:  Short form: 'LEN'
```

Specify German locale as used in Germany (de), with phonebook -style collation:

```
\locale LDE_KPHONEBOOKINFO:  Locale: 'de@collation=phonebook'
INFO:  German (collation=Phonebook Sort Order)
INFO:  Deutsch (Sortierung=Telefonbuch-Sortierregeln)
INFO:  Short form: 'KPHONEBOOK_LDE'
```

Specify German locale as used in Germany (de), with phonebook -style collation:

```
\locale LDE_KPHONEBOOK_S2INFO:  Locale: 'de@collation=phonebook'
INFO:  German (collation=Phonebook Sort Order)
INFO:  Deutsch (Sortierung=Telefonbuch-Sortierregeln)
INFO:  Short form: 'KPHONEBOOK_LDE_S2'
```

Supported locales

The following are the supported locale strings for Vertica. Each locale can optionally have a list of key/value pairs (see Specifying locale: long form).

Locale Name	Language or Variant	Region

af	Afrikaans	
af_NA	Afrikaans	Namibian Afrikaans
af_ZA	Afrikaans	South Africa
am	Ethiopic	
am_ET	Ethiopic	Ethiopia
ar	Arabic	
ar_AE	Arabic	United Arab Emirates
ar_BH	Arabic	Bahrain
ar_DZ	Arabic	Algeria
ar_EG	Arabic	Egypt
ar_IQ	Arabic	Iraq
ar_JO	Arabic	Jordan
ar_KW	Arabic	Kuwait
ar_LB	Arabic	Lebanon
ar_LY	Arabic	Libya
ar_MA	Arabic	Morocco
ar_OM	Arabic	Oman
ar_QA	Arabic	Qatar
ar_SA	Arabic	Saudi Arabia
ar_SD	Arabic	Sudan
ar_SY	Arabic	Syria
ar_TN	Arabic	Tunisia
ar_YE	Arabic	Yemen
as	Assamese	
as_IN	Assamese	India
az	Azerbaijani	
az_Cyrl	Azerbaijani	Cyrillic
az_Cyrl_AZ	Azerbaijani	Azerbaijan Cyrillic
az_Latn	Azerbaijani	Latin
az_Latn_AZ	Azerbaijani	Azerbaijan Latin

be	Belarusian	
be_BY	Belarusian	Belarus
bg	Bulgarian	
bg_BG	Bulgarian	Bulgaria
bn	Bengali	
bn_BD	Bengali	Bangladesh
bn_IN	Bengali	India
bo	Tibetan	
bo_CN	Tibetan	PR China
bo_IN	Tibetan	India
ca	Catalan	
ca_ES	Catalan	Spain
cs	Czech	
cs_CZ	Czech	Czech Republic
cy	Welsh	
cy_GB	Welsh	United Kingdom
da	Danish	
da_DK	Danish	Denmark
de	German	
de_AT	German	Austria
de_BE	German	Belgium
de_CH	German	Switzerland
de_DE	German	Germany
de_LI	German	Liechtenstein
de_LU	German	Luxembourg
el	Greek	
el_CY	Greek	Cyprus
el_GR	Greek	Greece
en	English	
en_AU	English	Australia

en_BE	English	Belgium
en_BW	English	Botswana
en_BZ	English	Belize
en_CA	English	Canada
en_GB	English	United Kingdom
en_HK	English	Hong Kong S.A.R. of China
en_IE	English	Ireland
en_IN	English	India
en_JM	English	Jamaica
en_MH	English	Marshall Islands
en_MT	English	Malta
en_NA	English	Namibia
en_NZ	English	New Zealand
en_PH	English	Philippines
en_PK	English	Pakistan
en_SG	English	Singapore
en_TT	English	Trinidad and Tobago
en_US	English	United States
en_US_POSIX	English	United States Posix
en_VI	English	U.S. Virgin Islands
en_ZA	English	Zimbabwe or South Africa
en_ZW	English	Zimbabwe
eo	Esperanto	
es	Spanish	
es_AR	Spanish	Argentina
es_BO	Spanish	Bolivia
es_CL	Spanish	Chile
es_CO	Spanish	Columbia
es_CR	Spanish	Costa Rica
es_DO	Spanish	Dominican Republic

es_EC	Spanish	Ecuador
es_ES	Spanish	Spain
es_GT	Spanish	Guatemala
es_HN	Spanish	Honduras
es_MX	Spanish	Mexico
es_NI	Spanish	Nicaragua
es_PA	Spanish	Panama
es_PE	Spanish	Peru
es_PR	Spanish	Puerto Rico
es_PY	Spanish	Paraguay
es_SV	Spanish	El Salvador
es_US	Spanish	United States
es_UY	Spanish	Uruguay
es_VE	Spanish	Venezuela
et	Estonian	
et_EE	Estonian	Estonia
eu	Basque	Spain
eu_ES	Basque	Spain
fa	Persian	
fa_AF	Persian	Afghanistan
fa_IR	Persian	Iran
fi	Finnish	
fi_FI	Finnish	Finland
fo	Faroese	
fo_FO	Faroese	Faroe Islands
fr	French	
fr_BE	French	Belgium
fr_CA	French	Canada
fr_CH	French	Switzerland
fr_FR	French	France

fr_LU	French	Luxembourg
fr_MC	French	Monaco
fr_SN	French	Senegal
ga	Gaelic	
ga_IE	Gaelic	Ireland
gl	Gallegan	
gl_ES	Gallegan	Spain
gsw	German	
gsw_CH	German	Switzerland
gu	Gujurati	
gu_IN	Gujurati	India
gv	Manx	
gv_GB	Manx	United Kingdom
ha	Hausa	
ha_Latn	Hausa	Latin
ha_Latn_GH	Hausa	Ghana (Latin)
ha_Latn_NE	Hausa	Niger (Latin)
ha_Latn_NG	Hausa	Nigeria (Latin)
haw	Hawaiian	Hawaiian
haw_US	Hawaiian	United States
he	Hebrew	
he_IL	Hebrew	Israel
hi	Hindi	
hi_IN	Hindi	India
hr	Croation	
hr_HR	Croation	Croatia
hu	Hungarian	
hu_HU	Hungarian	Hungary
hy	Armenian	
hy_AM	Armenian	Armenia

hy_AM_REVISED	Armenian	Revised Armenia
id	Indonesian	
id_ID	Indonesian	Indonesia
ii	Sichuan	
ii_CN	Sichuan	Yi
is	Icelandic	
is_IS	Icelandic	Iceland
it	Italian	
it_CH	Italian	Switzerland
it_IT	Italian	Italy
ja	Japanese	
ja_JP	Japanese	Japan
ka	Georgian	
ka_GE	Georgian	Georgia
kk	Kazakh	
kk_Cyrl	Kazakh	Cyrillic
kk_Cyrl_KZ	Kazakh	Kazakhstan (Cyrillic)
kl	Kalaallisut	
kl_GL	Kalaallisut	Greenland
km	Khmer	
km_KH	Khmer	Cambodia
kn	Kannada	
kn-IN	Kannada	India
ko	Korean	
ko_KR	Korean	Korea
kok	Konkani	
kok_IN	Konkani	India
kw	Cornish	
kw_GB	Cornish	United Kingdom
lt	Lithuanian	

lt_LT	Lithuanian	Lithuania
lv	Latvian	
lv_LV	Latvian	Latvia
mk	Macedonian	
mk_MK	Macedonian	Macedonia
ml	Malayalam	
ml_IN	Malayalam	India
mr	Marathi	
mr_IN	Marathi	India
ms	Malay	
ms_BN	Malay	Brunei
ms_MY	Malay	Malaysia
mt	Maltese	
mt_MT	Maltese	Malta
nb	Norwegian Bokml	
nb_NO	Norwegian Bokml	Norway
ne	Nepali	
ne_IN	Nepali	India
ne_NP	Nepali	Nepal
nl	Dutch	
nl_BE	Dutch	Belgium
nl_NL	Dutch	Netherlands
nn	Norwegian nynorsk	
nn_NO	Norwegian nynorsk	Norway
om	Oromo	
om_ET	Oromo	Ethiopia
om_KE	Oromo	Kenya
or	Oriya	
or_IN	Oriya	India
pa	Punjabi	

pa_Arab	Punjabi	Arabic
pa_Arab_PK	Punjabi	Pakistan (Arabic)
pa_Guru	Punjabi	Gurmukhi
pa_Guru_IN	Punjabi	India (Gurmukhi)
pl	Polish	
pl_PL	Polish	Poland
ps	Pashto	
ps_AF	Pashto	Afghanistan
pt	Portuguese	
pt_BR	Portuguese	Brazil
pt_PT	Portuguese	Portugal
ro	Romanian	
ro_MD	Romanian	Moldavia
ro_RO	Romanian	Romania
ru	Russian	
ru_RU	Russian	Russia
ru_UA	Russian	Ukraine
si	Sinhala	
si_LK	Sinhala	Sri Lanka
sk	Slovak	
sk_SK	Slovak	Slovakia
sl	Slovenian	
sl_SL	Slovenian	Slovenia
so	Somali	
so_DJ	Somali	Djibouti
so_ET	Somali	Ethiopia
so_KE	Somali	Kenya
so_SO	Somali	Somalia
sq	Albanian	
sq_AL	Albanian	Albania

sr	Serbian	
sr_Cyrl	Serbian	Cyrillic
sr_Cyrl_BA	Serbian	Bosnia and Herzegovina (Cyrillic)
sr_Cyrl_ME	Serbian	Montenegro (Cyrillic)
sr_Cyrl_RS	Serbian	Serbia (Cyrillic)
sr_Latn	Serbian	Latin
sr_Latn_BA	Serbian	Bosnia and Herzegovina (Latin)
sr_Latn_ME	Serbian	Montenegro (Latin)
sr_Latn_RS	Serbian	Serbia (Latin)
sv	Swedish	
sv_FI	Swedish	Finland
sv_SE	Swedish	Sweden
sw	Swahili	
sw_KE	Swahili	Kenya
sw_TZ	Swahili	Tanzania
ta	Tamil	
ta_IN	Tamil	India
te	Telugu	
te_IN	Telugu	India
th	Thai	
th_TH	Thai	Thailand
ti	Tigrinya	
ti_ER	Tigrinya	Eritrea
ti_ET	Tigrinya	Ethiopia
tr	Turkish	
tr_TR	Turkish	Turkey
uk	Ukrainian	
uk_UA	Ukrainian	Ukraine
ur	Urdu	
ur_IN	Urdu	India

ur_PK	Urdu	Pakistan
uz	Uzbek	
uz_Arab	Uzbek	Arabic
uz_Arab_AF	Uzbek	Afghanistan (Arabic)
uz_Cryl	Uzbek	Cyrillic
uz_Cryl_UZ	Uzbek	Uzbekistan (Cyrillic)
uz_Latin	Uzbek	Latin
us_Latin_UZ		Uzbekistan (Latin)
vi	Vietnamese	
vi_VN	Vietnamese	Vietnam
zh	Chinese	
zh_Hans	Chinese	Simplified Han
zh_Hans_CN	Chinese	China (Simplified Han)
zh_Hans_HK	Chinese	Hong Kong SAR China (Simplified Han)
zh_Hans_MO	Chinese	Macao SAR China (Simplified Han)
zh_Hans_SG	Chinese	Singapore (Simplified Han)
zh_Hant	Chinese	Traditional Han
zh_Hant_HK	Chinese	Hong Kong SAR China (Traditional Han)
zh_Hant_MO	Chinese	Macao SAR China (Traditional Han)
zh_Hant_TW	Chinese	Taiwan (Traditional Han)
zu	Zulu	
zu_ZA	Zulu	South Africa

Locale and UTF-8 support

Vertica supports Unicode Transformation Format-8, or UTF8, where 8 equals 8-bit. UTF-8 is a variable-length character encoding for Unicode created by Ken Thompson and Rob Pike. UTF-8 can represent any universal character in the Unicode standard. Initial encoding of byte codes and character assignments for UTF-8 coincides with ASCII. Thus, UTF8 requires little or no change for software that handles ASCII but preserves other values.

Vertica database servers expect to receive all data in UTF-8, and Vertica outputs all data in UTF-8. The ODBC API operates on data in UCS-2 on Windows systems, and normally UTF-8 on Linux systems. JDBC and ADO.NET APIs operate on data in UTF-16. Client drivers automatically convert data to and from UTF-8 when sending to and receiving data from Vertica using API calls. The drivers do not transform data loaded by executing a COPY or COPY LOCAL statement.

UTF-8 string functions

The following string functions treat VARCHAR arguments as UTF-8 strings (when USING OCTETS is not specified) regardless of locale setting.

String function	**Description**

LOWER	Returns a VARCHAR value containing the argument converted to lowercase letters.
UPPER	Returns a VARCHAR value containing the argument converted to uppercase letters.
INITCAP	Capitalizes first letter of each alphanumeric word and puts the rest in lowercase.
INSTR	Searches string for substring and returns an integer indicating the position of the character in string that is the first character of this occurrence.
SPLIT_PART	Splits string on the delimiter and returns the location of the beginning of the given field (counting from one).
POSITION	Returns an integer value representing the character location of a specified substring with a string (counting from one).
STRPOS	Returns an integer value representing the character location of a specified substring within a string (counting from one).

Locale-aware string functions

Vertica provides string functions to support internationalization. Unless otherwise specified, these string functions can optionally specify whether VARCHAR arguments should be interpreted as octet (byte) sequences, or as (locale-aware) sequences of characters. Specify this information by adding the parameter USING OCTETS and USING CHARACTERS (default) to the function.

The following table lists all string functions that are locale-aware:

String function	**Description**
BTRIM	Removes the longest string consisting only of specified characters from the start and end of a string.
CHARACTER_LENGTH	Returns an integer value representing the number of characters or octets in a string.
GREATEST	Returns the largest value in a list of expressions.
GREATESTB	Returns its greatest argument, using binary ordering, not UTF-8 character ordering.
INITCAP	Capitalizes first letter of each alphanumeric word and puts the rest in lowercase.
INSTR	Searches string for substring and returns an integer indicating the position of the character in string that is the first character of this occurrence.
LEAST	Returns the smallest value in a list of expressions.
LEASTB	Returns its least argument, using binary ordering, not UTF-8 character ordering.
LEFT	Returns the specified characters from the left side of a string.
LENGTH	Takes one argument as an input and returns returns an integer value representing the number of characters in a string.
LTRIM	Returns a VARCHAR value representing a string with leading blanks removed from the left side (beginning).
OVERLAY	Returns a VARCHAR value representing a string having had a substring replaced by another string.
OVERLAYB	Returns an octet value representing a string having had a substring replaced by another string.
REPLACE	replaces all occurrences of characters in a string with another set of characters.

RIGHT	Returns the *length* right-most characters of string.
SUBSTR	Returns a VARCHAR value representing a substring of a specified string.
SUBSTRB	Returns a byte value representing a substring of a specified string.
SUBSTRING	Given a value, a position, and an optional length, returns a value representing a substring of the specified string at the given position.
TRANSLATE	Replaces individual characters in *string_to_replace* with other characters.
UPPER	Returns a VARCHAR value containing the argument converted to uppercase letters.

Appendix: creating native binary format files

Using COPY to load data with the NATIVE parser requires that the input data files conform to the requirements described in this appendix. All NATIVE files must contain:

- File signature
- Column size definitions
- Rows of data

The subsection Loading a NATIVE file into a table: example describes an example of loading data with the NATIVE parser.

> **Note**
> You cannot mix Binary and ASCII source files in the same COPY statement.

In this section

- File signature
- Column definitions
- Row data
- Loading a NATIVE file into a table: example

File signature

The first part of a NATIVE binary file consists of a file signature. The contents of the signature are fixed, and listed in the following table.

Byte Offset	0	1	2	3	4	5	6	7	8	9	10
Hex Value	4E	41	54	49	56	45	0A	FF	0D	0A	00
Text Literals	N	A	T	I	V	E	E'\n'	E'\317'	E'\r'	E'\n'	E'\000'

The signature ensures that the file has neither been corrupted by a non-8-bit file transfer, nor stripped of carriage returns, linefeeds, or null values. If the signature is intact, Vertica determines that the file has not been corrupted.

Column definitions

Following the file signature, the file must define the widths of each column in the file as follows.

Byte Offset	Length (bytes)	Description	Comments
11	4	Header area length	32-bit integer in little-endian format that contains the length in bytes of remaining in the header, not including itself. This is the number of bytes from the end of this value to the start of the row data.

15	2	NATIVE file version	16-bit integer in little-endian format containing the version number of the NATIVE file format. The only valid value is currently 1. Future changes to the format could be assigned different version numbers to maintain backward compatibility.
17	1	Filler	Always 0.
18	2	Number of columns	16-bit integer in little-endian format that contains the number of columns in each row in the file.
20+	4 bytes for each column of data in the table	Column widths	Array of 32-bit integers in little-endian format that define the width of each column in the row. Variable-width columns have a value of -1 (0xFF 0xFF 0xFF 0xFF).

> **Note**
> All integers in NATIVE files are in **little-endian format** (least significant byte first).

The width of each column is determined by the data type it contains. The following table explains the column width needed for each data type, along with the data encoding.

Data Type	Length (bytes)	Column Content
INTEGER	1, 2, 4, 8	8-, 16-, 32-, and 64-bit integers are supported. All multi-byte values are stored in little-endian format. **Note:** All values for a column must be the width you specify here. If you set the length of an INTEGER column to be 4 bytes, then all of the values you supply for that column must be 32-bit integers.
BOOLEAN	1	0 for false, 1 for true.
FLOAT	8	Encoded in IEEE-754 format.
CHAR	User-specified	• Strings shorter than the specified length must be right-padded with spaces (E'\040'). • Strings are not null-terminated. • Character encoding is UTF-8. • UTF-8 strings can contain multi-byte characters. Therefore, number of characters in the string may not equal the number of bytes.
VARCHAR	4-byte integer (length) + data	The column width for a VARCHAR column is always -1 to signal that it contains variable-length data. • Each VARCHAR column value starts with a 32-bit integer that contains the number of bytes in the string. • The string must not be null-terminated. • Character encoding must be UTF-8. • Remember that UTF-8 strings can contain multi-byte characters. Therefore, number of characters in the string may not equal the number of bytes.
DATE	8	64-bit integer in little-endian format containing the Julian day since Jan 01 2000 (J2451545)
TIME	8	64-bit integer in little-endian format containing the number of microseconds since midnight in the UTC time zone.

TIMETZ	8	64-bit value where • Upper 40 bits contain the number of microseconds since midnight. • Lower 24 bits contain time zone as the UTC offset in microseconds calculated as follows: Time zone is logically from -24hrs to +24hrs from UTC. Instead it is represented here as a number between 0hrs to 48hrs. Therefore, 24hrs should be added to the actual time zone to calculate it. Each portion is stored in little-endian format (5 bytes followed by 3 bytes).
TIMESTAMP	8	64-bit integer in little-endian format containing the number of microseconds since Julian day: Jan 01 2000 00:00:00.
TIMESTAMPTZ	8	A 64-bit integer in little-endian format containing the number of microseconds since Julian day: Jan 01 2000 00:00:00 in the UTC timezone.
INTERVAL	8	64-bit integer in little-endian format containing the number of microseconds in the interval.
BINARY	User-specified	Similar to CHAR. The length should be specified in the file header in the *Field Lengths* entry for the field. The field in the record must contain *length* number of bytes. If the value is smaller than the specified length, the remainder should be filled with nulls (E'\000').
VARBINARY	4-byte integer + data	Stored just like VARCHAR but data is interpreted as bytes rather than UTF-8 characters.
NUMERIC	(precision, scale) (precision ¸ 19 + 1) ´ 8 rounded up	A constant-length data type. Length is determined by the precision, assuming that a 64-bit unsigned integer can store roughly 19 decimal digits. The data consists of a sequence of 64-bit integers, each stored in little-endian format, with the most significant integer first. Data in the integers is stored in base 2^{64}. 2's complement is used for negative numbers. If there is a scale, then the numeric is stored as numeric ´ 10^{scale}; that is, all real numbers are stored as integers, ignoring the decimal point. It is required that the scale matches that of the target column in the dataanchor table. Another option is to use FILLER columns to coerce the numeric to the scale of the target column.

Row data

Following the file header is a sequence of records that contain the data for each row of data. Each record starts with a header:

Length (bytes)	**Description**	**Comments**
4	Row length	A 32-bit integer in little-endian format containing the length of the row's data in bytes. It includes the size of data only, not the header. **Note:** The number of bytes in each row can vary not only because of variable-length data, but also because columns containing NULL values do not have any data in the row. If column 3 has a NULL value, then column 4's data immediately follows the end of column 2's data. See the next
Number of columns ¸ 8 rounded up (`CEILING(NumFields / ( sizeof(uint8) * 8) );`)	Null value bit field	A series of bytes whose bits indicate whether a column contains a NULL. The most significant bit of the first byte indicates whether the first column in this row contains a NULL, the next most significant bit indicates whether the next column contains a NULL, and so on. If a bit is 1 (true) then the column contains a NULL, and there is no value for the column in the data for the row.

Following the record header is the column values for the row. There is no separator characters for these values. Their location in the row of data is calculated based on where the previous column's data ended. Most data types have a fixed width, so their location is easy to determine. Variable-width values (such as VARCHAR and VARBINARY) start with a count of the number of bytes the value contains.

See the table in the previous section for details on how each data type's value is stored in the row's data.

Loading a NATIVE file into a table: example

The example below demonstrates creating a table and loading a NATIVE file that contains a single row of data. The table contains all possible data types.

```
=> CREATE TABLE allTypes (INTCOL INTEGER,
                          FLOATCOL FLOAT,
                          CHARCOL CHAR(10),
                          VARCHARCOL VARCHAR,
                          BOOLCOL BOOLEAN,
                          DATECOL DATE,
                          TIMESTAMPCOL TIMESTAMP,
                          TIMESTAMPTZCOL TIMESTAMPTZ,
                          TIMECOL TIME,
                          TIMETZCOL TIMETZ,
                          VARBINCOL VARBINARY,
                          BINCOL BINARY,
                          NUMCOL NUMERIC(38,0),
                          INTERVALCOL INTERVAL
                         );
=> COPY allTypes FROM '/home/dbadmin/allTypes.bin' NATIVE;
=> \pset expanded
Expanded display is on.
=> SELECT * from allTypes;
-[ RECORD 1 ]--+-----------------------
INTCOL         | 1
FLOATCOL       | -1.11
CHARCOL        | one
VARCHARCOL     | ONE
BOOLCOL        | t
DATECOL        | 1999-01-08
TIMESTAMPCOL   | 1999-02-23 03:11:52.35
TIMESTAMPTZCOL | 1999-01-08 07:04:37-05
TIMECOL        | 07:09:23
TIMETZCOL      | 15:12:34-04
VARBINCOL      | \253\315
BINCOL         | \253
NUMCOL         | 1234532
INTERVALCOL    | 03:03:03
```

The content of the allTypes.bin file appears below as a raw hex dump:

```
4E 41 54 49 56 45 0A FF 0D 0A 00 3D 00 00 00 01 00 00 0E 00
08 00 00 00 08 00 00 00 0A 00 00 00 FF FF FF FF 01 00 00 00
08 00 00 00 08 00 00 00 08 00 00 00 08 00 00 00 08 00 00 00
FF FF FF FF 03 00 00 00 18 00 00 00 08 00 00 00 73 00 00 00
00 00 01 00 00 00 00 00 00 00 C3 F5 28 5C 8F C2 F1 BF 6F 6E
65 20 20 20 20 20 20 20 03 00 00 00 4F 4E 45 01 9A FE FF FF
FF FF FF FF 30 85 B3 4F 7E E7 FF FF 40 1F 3E 64 E8 E3 FF FF
C0 2E 98 FF 05 00 00 00 D0 97 01 80 F0 79 F0 10 02 00 00 00
AB CD AB CD 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 64 D6 12 00 00 00 00 00 C0 47 A3 8E 02 00 00 00
```

The following table breaks this file down into each of its components, and describes the values it contains.

Hex Values	Description	Value
4E 41 54 49 56 45 0A FF 0D 0A 00	Signature	NATIVE\n\317\r\n\000
3D 00 00 00	Header area length	61 bytes

01 00	Native file format version	Version 1
00	Filler value	0
0E 00	Number of columns	14 columns
08 00 00 00	Width of column 1 (INTEGER)	8 bytes
08 00 00 00	Width of column 2 (FLOAT)	8 bytes
0A 00 00 00	Width of column 3 (CHAR(10))	10 bytes
FF FF FF FF	Width of column 4 (VARCHAR)	-1 (variable width column)
01 00 00 00	Width of column 5 (BOOLEAN)	1 bytes
08 00 00 00	Width of column 6 (DATE)	8 bytes
08 00 00 00	Width of column 7 (TIMESTAMP)	8 bytes
08 00 00 00	Width of column 8 (TIMESTAMPTZ)	8 bytes
08 00 00 00	Width of column 9 (TIME)	8 bytes
08 00 00 00	Width of column 10 (TIMETZ)	8 bytes
FF FF FF FF	Width of column 11 (VARBINARY)	-1 (variable width column)
03 00 00 00	Width of column 12 (BINARY)	3 bytes
18 00 00 00	Width of column 13 (NUMERIC)	24 bytes. The size is calculated by dividing 38 (the precision specified for the numeric column) by 19 (the number of digits each 64-bit chunk can represent) and adding 1. 38 ¸ 19 + 1 = 3. then multiply by eight to get the number of bytes needed. 3 ´ 8 = 24 bytes.
08 00 00 00	Width of column 14 (INTERVAL). last portion of the header section.	8 bytes
73 00 00 00	Number of bytes of data for the first row. this is the start of the first row of data.	115 bytes
00 00	Bit field for the null values contained in the first row of data	The row contains no null values.
01 00 00 00 00 00 00 00	Value for 64-bit INTEGER column	1

C3 F5 28 5C 8F C2 F1 BF	Value for the FLOAT column	-1.11
6F 6E 65 20 20 20 20 20 20 20	Value for the CHAR(10) column	"one " (padded With 7 spaces to fill the full 10 characters for the column)
03 00 00 00	The number of bytes in the following VARCHAR value.	3 bytes
4F 4E 45	The value for the VARCHAR column	" ONE "
01	The value for the BOOLEAN column	True
9A FE FF FF FF FF FF FF	The value for the DATE column	1999-01-08
30 85 B3 4F 7E E7 FF FF	The value for the TIMESTAMP column	1999-02-23 03:11:52.35
40 1F 3E 64 E8 E3 FF FF	The value for the TIMESTAMPTZ column	1999-01-08 07:04:37-05
C0 2E 98 FF 05 00 00 00	The value for the TIME column	07:09:23
D0 97 01 80 F0 79 F0 10	The value for the TIMETZ column	15:12:34-05
02 00 00 00	The number of bytes in the following VARBINARY value	2 bytes
AB CD	The value for the VARBINARY column	Binary data (\253\315 as octal values)
AB CD	The value for the BINARY column	Binary data (\253\315 as octal values)
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 64 D6 12 00 00 00 00 00	The value for the NUMERIC column	1234532
C0 47 A3 8E 02 00 00 00	The value for the INTERVAL column	03:03:03

Security and authentication

Vertica provides tools and features that allow you to ensure your system is secure as well as to prevent unauthorized users from accessing sensitive information.

Client authentication establish the identity of the requesting client and determines whether that client is authorized to connect to the Vertica server.

In this section

- Client authentication

- Internode TLS
- TLS protocol
- LDAP link service
- Connector framework service
- Federal information processing standard
- Database auditing
- System table restriction and access

Client authentication

Authentication records and their associated methods define what credentials a user/client application must provide to access the database. For example, the hash authentication method requires users to provide a password, while the oauth authentication method requires users to provide an access token.

Client authentication overview

Vertica uses the following procedure to authenticate users:

1. If a client attempts to authenticate as the dbadmin from a local connection (that is, on the same node as the database):
   - If the dbadmin does not have a password, Vertica authenticates the client with the trust method.
   - If the dbadmin has a password, Vertica authenticates the client with the hash method.
2. If a client attempts to authenticate as a database user that does not have a password, and the only authentication records defined are the defaults, then Vertica authenticates the client with the trust method. For details, see Implicit authentication.
3. If a client specifies the credentials for a particular authentication method, Vertica filters for granted authentication records that use that method, skipping higher priority authentication records (except TRUST, which is not skippable). However, if the client sends credentials that correspond with an authentication record that they do not have, Vertica uses the record with the highest priority.
4. If a client attempts to authenticate as a user that has a password and an authentication record, then Vertica attempts to authenticate the client with that record. If more than one authentication record exists for the user or role, Vertica chooses the one with the highest priority.

   > **Note**
   > Typically, if no custom authentication records were defined with CREATE AUTHENTICATION, the default authentication records take effect (unless they were deliberately dropped by the dbadmin), and clients are authenticated with the password method.

5. If the client fails to authenticate with the chosen authentication method and authentication fallthrough is enabled, Vertica attempts to authenticate the client with the authentication with the next highest priority. Otherwise, the client is rejected.
6. Otherwise, no authentication records exist and the default authentication records have been dropped; no users (other than the dbadmin from a local connection) can access the database.

Authentication management

Users with the DBADMIN role can perform the following authentication tasks:

- Create authentication records.
- Drop an authentication record from the database.
- Define parameters required by the following authentication methods:
  - LDAP
  - Ident
  - Kerberos
  - OAuth
- Grant (assign) or revoke an authentication record to a user.
- Use ALTER AUTHENTICATION to:
  - Enable/disable authentication methods.
  - Define a default authentication method to be used if a user has not been assigned a specific authentication method. To assign this as a default authentication method, use GRANT (authentication) to grant it to the PUBLIC role.
  - Change authentication record priority.
  - Enable fallthrough authentication.

In this section

- Default authentication records
- Configuring client authentication
- Fallthrough authentication
- Authentication filtering

- Implicit authentication
- Dbadmin authentication access
- Creating authentication records
- Modifying authentication records
- Authentication record priority
- Viewing information about client authentication records
- Enabling and disabling authentication methods
- Granting and revoking authentication methods
- Hiding database usernames
- Hash authentication
- Ident authentication
- Kerberos authentication
- LDAP authentication
- OAuth 2.0 authentication
- TLS authentication

Default authentication records

Vertica automatically creates the following default authentication records and grants them to the public role. These have the lowest possible priority (-1), so user-created authentication records take priority over these default records:

```
=> SELECT auth_name,is_auth_enabled,auth_host_type,auth_method,auth_priority,is_fallthrough_enabled FROM client_auth;
        auth_name         | is_auth_enabled | auth_host_type | auth_method | auth_priority | is_fallthrough_enabled
--------------------------+-----------------+----------------+-------------+---------------+------------------------
 default_hash_network_ipv4 | True            | HOST           | PASSWORD    |            -1 | False
 default_hash_network_ipv6 | True            | HOST           | PASSWORD    |            -1 | False
 default_hash_local        | True            | LOCAL          | PASSWORD    |            -1 | False
(3 rows)
```

> **Note**
> Do not use the password method for your custom authentication records. If you want to use password-based authentication for your custom authentication records, use hash instead.

If no authentication records are defined (and the default authentication records are dropped), only the dbadmin and users without passwords can access the database. For details, see Implicit authentication .

Configuring client authentication

You can restrict how database users can connect with authentication records. The Vertica database uses client authentication to establish the identity of the connecting client and determines whether that client is authorized to connect to the Vertica database using the supplied credentials and from their host address.

Basic authentication configuration workflow

In general, the workflow for configuring client authentication is as follows:

1. Create an authentication record. Authentication records are enabled automatically after creation. For details, see Creating authentication records .
2. Grant the authentication to a user or role.

To view existing authentication records, query the system table CLIENT_AUTH .

IPv4 and IPv6 for client authentication

Vertica supports clients using either the IPv4 or the IPv6 protocol to connect to the database server. Internal communication between database servers must consistently use one address family (IPv4 or IPv6). The client, however, can connect to the database from either type of IP address.

If the client will be connecting from either IPv4 or IPv6, you must create two authentication methods, one for each address. Any authentication method that uses HOST authentication requires an IP address.

For example, the first statement allows users to connect from any IPv4 address. The second statement allows users to connect from any IPv6 address:

```
=> CREATE AUTHENTICATION <name> METHOD 'gss' HOST '0.0.0.0/0'; --IPv4
=> CREATE AUTHENTICATION <name> METHOD 'gss' HOST '::/0'; --IPv6
```

If you are using a literal IPv6 address in a URL, you must enclose the IPv6 address in square brackets as shown in the following examples:

```
=> ALTER AUTHENTICATION Ldap SET host='ldap://[1dfa:2bfa:3:45:5:6:7:877]';
=> ALTER AUTHENTICATION Ldap SET host='ldap://[fdfb:dbfa:0:65::177]';
=> ALTER AUTHENTICATION Ldap SET host='ldap://[fdfb::177]';
=> ALTER AUTHENTICATION Ldap SET host='ldap://[::1]';
=> ALTER AUTHENTICATION Ldap SET host='ldap://[1dfa:2bfa:3:45:5:6:7:877]:5678';
```

If you are working with a multi-node cluster, any IP/netmask settings in (HOST, HOST TLS, HOST NO TLS) must match all nodes in the cluster. This setup allows the database owner to authenticate with and administer every node in the cluster. For example, specifying 10.10.0.8/30 allows a CIDR address range of 10.10.0.8-10.10.0.11.

For detailed information about IPv6 addresses, see RFC 1924 and RFC 2732 .

Supported client authentication methods

Vertica supports the following client authentication methods:

- trust : Users can authenticate with a valid username (that is, without a password).
- reject : Rejects the connection attempt.
- hash : Users must provide a valid username and password. For details, see Hash authentication .
- gss : Authorizes clients that connect to Vertica with an MIT Kerberos implementation. The Key Distribution Center (KDC) must support Kerberos 5 using the GSS-API. Non-MIT Kerberos implementations must use the GSS-API. For details, see Kerberos authentication .
- ident : Authenticates the client against a username on an Ident server. For details, see Ident authentication .
- ldap : Authenticates a client and their username and password with an LDAP or Active Directory server. For details, see LDAP authentication .
- tls : Authenticates clients that provide a certificate with a Common Name (CN) that specifies a valid database username. Vertica must be configured for mutual mode TLS to use this method. For details, see TLS authentication
- oauth : Authenticates a client with an access token. For details, see OAuth 2.0 authentication .

Local and host authentication

You can define a client authentication method as:

- Local: Local connection to the database.
- Host: Remote connection to the database from different hosts, each with their own IPv4 or IPv6 address and host parameters. For more information see IPv4 and IPv6 for Client Authentication above.

Some authentication methods can only be designated as local or host, as listed in this table:

Authentication Method	**Local?**	**Host?**
gss (Kerberos)	No	Yes
ident	Yes	No
ldap	Yes	Yes
hash	Yes	Yes
reject	Yes	Yes
trust	Yes	Yes
tls	No	Yes
oauth	Yes	Yes

Authentication for chained users and roles

Vertica supports creating chained users and roles, where you can grant ROLE2 privileges to ROLE1. All users in ROLE1 use the same authentication assigned to ROLE2. For example:

```
=> CREATE USER user1;
=> CREATE ROLE role1;
=> CREATE ROLE role2;
=> CREATE AUTHENTICATION h1 METHOD 'hash' LOCAL;
=> GRANT AUTHENTICATION h1 to role2;
=> GRANT role2 to role1;
=> GRANT role1 to user1;
```

The user and role chain in the example above can be illustrated as follows:

auth1 -> role2 -> role1 -> user1

In this example, since role2 privileges are granted to role1 you only need to grant authentication to role2 to also enable it for role1.

Fallthrough authentication

Normally, if a user fails to authenticate with the first chosen authentication record, the user is rejected. However, certain authentication methods can be specified to instead, upon failure, "fall through" to an authentication record with a lower priority . This can be useful, for example, for using multiple LDAP search attributes .

Authentication fallthrough is disabled by default for new records.

Another way to bypass method priority or implement fallthrough authentication is to implement it on the client instead. For details, see Authentication filtering .

Authentication method compatibility

The following table shows each authentication method's compatibility with fallthrough authentication. A value of "yes" indicates that the method in the column can fall through and validate the row method it falls through to. A value of "no" indicates that you cannot fall through to it and authentication ends.

A value of "-" (dash) indicates that while fallthrough works, it has no meaningful effect. For example, a hash authentication record can fall through to another hash authentication record, but authentication will always fail because you cannot fall through to a correct password:

Column falls through to row	ldap	hash	tls	ident	oauth	gss	trust	reject
ldap	yes	yes	yes	yes	no	no	no	no
hash	yes	-	yes	yes	no	no	no	no
tls	yes	yes	-	yes	no	no	no	no
ident	yes	yes	yes	yes	no	no	no	no
oauth	no	no	no	no	no	no	no	no
gss	no	no	no	no	no	no	no	no
trust	yes	yes	yes	yes	no	no	no	no
reject	yes	yes	yes	yes	no	no	no	no

For example, suppose the user Bob was granted the following authentication record with fallthrough enabled:

```
=> CREATE AUTHENTICATION v_tls_auth METHOD 'tls' HOST TLS '0.0.0.0/0' FALLTHROUGH;
```

In addition, Bob has the public role, and therefore has the default authentication records :

```
=> SELECT auth_name,is_auth_enabled,auth_host_type,auth_method,auth_priority,is_fallthrough_enabled FROM client_auth;
        auth_name         | is_auth_enabled | auth_host_type | auth_method | auth_priority | is_fallthrough_enabled
--------------------------+-----------------+----------------+-------------+---------------+------------------------
 default_hash_network_ipv4 | True           | HOST           | PASSWORD    |            -1 | False
 default_hash_network_ipv6 | True           | HOST           | PASSWORD    |            -1 | False
 default_hash_local        | True           | LOCAL          | PASSWORD    |            -1 | False
(3 rows)
```

If Bob , connecting from a remote address, fails to authenticate with v_tls_auth , Vertica attempts to authenticate him with the next (in order of priority) authentication record, default_hash_network_ipv4 .

Logging authentication failures

Login failures are only logged if the user is rejected after all granted authentication records have failed. Only the final failure is logged.

For example, suppose a user is granted both tls and password authentication records, and the tls record falls through to the password record.

If the user fails to authenticate with both the tls record and the password record, then only the password failure is logged in LOGIN_FAILURES .

However, if the user fails to authenticate with the tls record, but succeeds with the password record, then no failure is logged in LOGIN_FAILURES because the user was not rejected at the end of the record chain.

Examples

To enable fallthrough on a new authentication record, use CREATE AUTHENTICATION :

```
=> CREATE AUTHENTICATION v_tls_auth METHOD 'tls' HOST TLS '0.0.0.0/0' FALLTHROUGH;
```

To toggle fallthrough on an existing authentication record, use ALTER AUTHENTICATION :

```
=> ALTER AUTHENTICATION v_tls_auth NO FALLTHROUGH; -- disable
=> ALTER AUTHENTICATION v_tls_auth FALLTHROUGH; -- enable
```

Authentication filtering

You can filter for and authenticate with authentication records that use a particular method by specifying the credentials for that method. This process skips any higher-priority , non- trust authentication records.

If the selected record fails and fallthrough is enabled, Vertica uses to the record with the next highest priority .

If you provide the credentials for an authentication method for which you have not been granted , Vertica instead attempts to authenticate you with the highest priority authentication record.

The following table shows the credentials the client provides and the requested authentication method in descending filter priority. If you attempt to provide more than one type of credential, your client selects and sends the one with the higher priority:

Credentials sent by client	**Requested authentication method**
OAuthRefreshToken or OAuthAccessToken	oauth
Non-default KerberosServiceName or KerberosHostname	gss
Username, Password	ldap , hash , ident , trust , tls

For example, if a client provides a username and password for user Alice, and the Alice has oauth , ldap , and hash records, then Vertica attempts to authenticate the client with ldap or hash , whichever has the higher priority .

Similarly, if Alice instead provides an OAuthAccessToken, username, and password, then Vertica attempts to authenticate her with the oauth record because it has a higher filter priority.

However, if Alice provides a KerberosHostname, Vertica attempts to authenticate her with the highest priority authentication record between her oauth , ldap , and hash records.

Client-side fallthrough authentication

A workaround for the incompatibility of certain authentication methods with fallthrough authentication is to use authentication filtering to manually fall through to the proper authentication record by detecting the initial authentication failure, and then making a new connection attempt with a different set of credentials.

For example, oauth and hash cannot fall through to each other, so if a user is granted authentication records with these methods, they can only authenticate with the higher priority record. As a workaround, you can detect the authentication failure with oauth and then filter for hash in another connection attempt. Doing this programmatically depends on the client. In JDBC, you can catch SQLInvalidAuthorizationSpecException and then try to reconnect in the exception handler. In ODBC, you can check the return value of SQLDriverConnect() :

```
// error checking omitted for brevity
   SQLHENV hdlEnv;
   SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
   SQLHDBC hdlDbc;
   SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
            (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
   SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
   SQLRETURN ret = SQLDriverConnect(hdlDbc, NULL, (SQLCHAR *)("DSN=VerticaDSN;OAuthAccessToken=" + accessToken + ";OAuthRefreshToken="
+ refreshToken + ";OAuthJsonConfig=" + jsonConfig).c_str(), SQL_NTS, NULL, 0, NULL, false);

   if (!SQL_SUCCEEDED(ret))
   {
      std::cout << "Could not connect to database with OAuth, retrying with hash authentication" << std::endl;
      reportError <SQLHDBC>(SQL_HANDLE_DBC, hdlDbc);
      SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
      const char* dsnName = "ExampleDB";
      const char* userID = "Alice";
      const char* passwd = "mypassword";
      SQLConnect(hdlDbc, (SQLCHAR*)dsnName,SQL_NTS,(SQLCHAR*)userID,SQL_NTS,(SQLCHAR*)passwd, SQL_NTS);
   }
```

Examples

In the following example, the user Bob is granted two authentication records: one that uses Kerberos authentication (gss) and another that uses password authentication (hash):

```
=> SELECT auth_name, auth_host_type, auth_method FROM client_auth;
      auth_name     | auth_host_type | auth_method
--------------------+----------------+-------------
 v_hash             | HOST           | HASH
 v_kerberos         | HOST           | GSS
(2 rows)
```

1. Bob attempts to authenticate with Kerberos, but fails (the exact error message varies):
   ```
   $ vsql -h vertica_db.example.com -K vcluster.example.com -U Bob
   vsql: GSSAPI continuation error: Unspecified GSS failure.  Minor code may provide more information
   GSSAPI continuation error: No Kerberos credentials available (default cache: KEYRING:persistent:0)
   ```
2. Bob notices the failure. The steps involved in detecting this programmatically depend on your client.
3. An authentication record that uses gss cannot fall through to one that uses hash , so the only way for Bob to authenticate to the database is to filter for the hash by providing the username and password:
   ```
   => vsql -h vertica_db.example.com -U Bob -w 'mypassword'
   ```

To implement client-side fallthrough authentication, check the initial connection/authentication attempt for errors and then create a new connection in the error handler. For example, the user Alice is granted oauth and hash authentication records:

```
=> SELECT auth_name, auth_host_type, auth_method FROM client_auth;
      auth_name     | auth_host_type | auth_method
--------------------+----------------+-------------
 v_oauth            | HOST           | OAUTH
 v_hash             | HOST           | HASH
(2 rows)
```

Implicit authentication

Vertica has implicit (that is, not reflected in the CLIENT_AUTH system table) authentication records reserved for the dbadmin and users without passwords. These records cannot be dropped .

For the dbadmin user:

- The trust method is used if all of the following are true:
  - The dbadmin does not have a password.
  - The connection is local (that is, from a Vertica node).
- The hash method is used if all of the following are true:
  - The dbadmin has a password.
  - The connection is local (that is, from a Vertica node).

For a non-dbadmin user, the trust method is used if all of the following are true:

- The user does not have a password.
- No custom (that is, non-default) authentication methods are enabled.

Dbadmin authentication access

The dbadmin user must have access to the database at all times. Vertica automatically ensures that a client can authenticate as the dbadmin from a LOCAL connection.

If you need to authenticate as the dbadmin from a remote connection, the dbadmin must have a password. You can use the following methods:

- Use fallthrough authentication.
- Create a custom, dbadmin-specific authentication method.

Authenticating from a local connection

You can always implicitly authenticate as the dbadmin from a local connection. These dbadmin-specific authentication records are implicit, so they are not listed in the CLIENT_AUTH system table, and cannot be dropped .

If the dbadmin user does not have a password, then Vertica authenticates them with the trust method. Otherwise, Vertica authenticates them with the password method.

In this example, the dbadmin did not have a password and connected to Vertica from a local connection:

```
=> SELECT authentication_method, client_authentication_name FROM vs_sessions;
 authentication_method | client_authentication_name
-----------------------+----------------------------
 ImpTrust              | default: Implicit Trust
```

Authenticating from a remote connection

Fallthrough authentication

Vertica automatically creates the following authentication records and grants them to the public role (for details, see Client authentication):

```
=> SELECT auth_name,is_auth_enabled,auth_host_type,auth_method,auth_priority,is_fallthrough_enabled FROM client_auth;
        auth_name         | is_auth_enabled | auth_host_type | auth_method | auth_priority | is_fallthrough_enabled
--------------------------+-----------------+----------------+-------------+---------------+------------------------
 default_hash_network_ipv4 | True            | HOST           | PASSWORD    |            -1 | False
 default_hash_network_ipv6 | True            | HOST           | PASSWORD    |            -1 | False
 default_hash_local        | True            | LOCAL          | PASSWORD    |            -1 | False
(3 rows)
```

These default authentication records ensure that all users with the public role (which includes dbadmin) have access to the database, provided that any custom authentication records are set to fall through (disabled by default) to the default records.

For example, the following ldap authentication enables fallthrough, so if the LDAP server is down, users can still authenticate with password authentication (as defined by the default records).

```
=> CREATE AUTHENTICATION ldap1 METHOD 'ldap' LOCAL FALLTHROUGH;
=> ALTER AUTHENTICATION ldap1 SET host='ldap://localhost:5389',
   binddn='cn=Manager,dc=example,dc=com',
   bind_password='password',
   basedn='ou=dev,dc=example,dc=com',
   search_attribute='cn';
```

Custom authentication records

A dbadmin-specific authentication record should:

- Use the hash authentication method (so authentication is not dependent on some external service).
- Have a high priority (e.g. 10,000) so it supersedes all other authentication records.

The following example creates an authentication record v_dbadmin_hash and grants it to the dbadmin user. The hash method indicates that the dbadmin must provide a password when logging in. The HOST '0.0.0.0/0' access method indicates that the dbadmin can connect remotely from any IPv4 address:

```
=> CREATE AUTHENTICATION v_dbadmin_hash METHOD 'hash' HOST '0.0.0.0/0';
=> ALTER AUTHENTICATION v_dbadmin_hash PRIORITY 10000;
=> GRANT AUTHENTICATION v_dbadmin_hash TO dbadmin;
```

If you want to authenticate as the dbadmin from a local connection, but want to use an authentication record with the HOST access method, specify the --host option with the hostname or IP address of the database:

```
$ vsql database_name user --host hostname_or_ip;
```

Creating authentication records

You can manage client authentication records with vsql . You must be connected to the database as a supseruser.

Important

You cannot modify client authentication records using the Administration Tools. The Administration Tools interface allows you to modify the contents of the vertica.conf file. However, Vertica ignores any client authentication information stored in that file.

1. Create an authentication records, specifying:
   - The name of the authentication record.
   - The authentication method, one of the following:
     - trust : Users can authenticate with a valid username (that is, without a password).
     - reject : Rejects the connection attempt.
     - hash : Users must provide a valid username and password. For details, see Hash authentication .
     - gss : Authorizes clients that connect to Vertica with an MIT Kerberos implementation. The Key Distribution Center (KDC) must support Kerberos 5 using the GSS-API. Non-MIT Kerberos implementations must use the GSS-API. For details, see Kerberos authentication .
     - ident : Authenticates the client against a username on an Ident server. For details, see Ident authentication .
     - ldap : Authenticates a client and their username and password with an LDAP or Active Directory server. For details, see LDAP authentication .
     - tls : Authenticates clients that provide a certificate with a Common Name (CN) that specifies a valid database username. Vertica must be configured for mutual mode TLS to use this method. For details, see TLS authentication
     - oauth : Authenticates a client with an access token. For details, see OAuth 2.0 authentication .
   - The access method, one of the following, which specify the allowed connection type:
     - LOCAL: Authenticates users or applications that attempt to connect from the same node that the database is running on.
     - HOST: Authentications users or applications that attempt to connect from a node that has a different IPv4 or IPv6 address than the database. You can use TLS or NO TLS to specify an encrypted or plaintext connection, respectively.
   - Whether to enable Fallthrough authentication (disabled by default).
2. Grant the authentication record to a user or role.

Examples

The following examples show how to create authentication records.

Create authentication method localpwd to authenticate users who are trying to log in from a local host using a password:

```
=> CREATE AUTHENTICATION localpwd METHOD 'hash' LOCAL;
```

Create authentication method v_ldap that uses LDAP over TLS to authenticate users logging in from the host with the IPv4 address 10.0.0.0/23:

```
=> CREATE AUTHENTICATION v_ldap METHOD 'ldap' HOST TLS '10.0.0.0/23';
```

Create authentication method v_kerberos to authenticate users who are trying to connect from any host in the networks 2001:0db8:0001:12 *xx* :

```
=> CREATE AUTHENTICATION v_kerberos METHOD 'gss' HOST '2001:db8:1::1200/56';
```

The following authentication record v_oauth authenticates users from any IP address by contacting the identity provider to validate the OAuth token (rather than a username and password) and uses the following parameters:

- validate_type : The method used to validate the OAuth token. This should be set to IDP (default) to validate the OAuth token by contacting the identity provider.
- client_id : The client in the identity provider.
- client_secret : The client secret generated by the identity provider.
- discovery_url : Also known as the OpenID Provider Configuration Document , Vertica uses this endpoint to retrieve information about the identity provider's configuration and other endpoints (Keycloak only).
- introspect_url : Used by Vertica to introspect (validate) access tokens. You must specify the introspect_url if you do not specify the discovery_url and are not using JSON Web Token validation.

If discovery_url and introspect_url are both set, discovery_url takes precedence. The following example sets both for demonstration purposes; in general, you should prefer to set the discovery_url :

```
=> CREATE AUTHENTICATION v_oauth METHOD 'oauth' HOST '0.0.0.0/0';
=> ALTER AUTHENTICATION v_oauth SET validate_type = 'IDP';
=> ALTER AUTHENTICATION v_oauth SET client_id = 'vertica';
=> ALTER AUTHENTICATION v_oauth SET client_secret = 'client_secret';
=> ALTER AUTHENTICATION v_oauth SET discovery_url = 'https://203.0.113.1:8443/realms/myrealm/.well-known/openid-configuration';
=> ALTER AUTHENTICATION v_oauth SET introspect_url = 'https://203.0.113.1:8443/realms/myrealm/protocol/openid-connect/token/introspect';
```

Alternatively, if your identity provider supports the OpenID Connect protocol and your client is public, Vertica can use JWT validation, where Vertica validates OAuth tokens by verifying that it was signed by the identity provider's private key.

While Vertica does not contact the identity provider for JWT validation, the client does for token refresh if either the discovery or token endpoints are set in oauthjsonconfig.

JWT validation requires the following parameters:

- validate_type : The validation method, IDP by default. Setting this to JWT enables JWT validation.
- jwt_rsa_public_key : In PEM format, the public key used to sign the client's OAuth token. Vertica uses this to validate the OAuth token. If your identity provider does not natively provide PEM-formatted public keys, you must convert them to PEM format. For example, keys retrieved from an Okta endpoint are in JWK format and must be converted.
- jwt_issuer : The issuer of the OAuth token. This value is set by the identify provider.
- jwt_user_mapping : The name of the Vertica user.

You can also specify the following parameters to define a whitelist based on fields of the OAuth token:

- jwt_accepted_audience_list : Optional, a comma-delimited list of values to accept from the client JWT's aud field. If set, tokens must include in aud one of the accepted audiences to authenticate.
- jwt_accepted_scope_list : Optional, a comma-delimited list of values to accept from the client JWT's scope field. If set, tokens must include in scope at least one of the accepted scopes to authenticate.

The following authentication record v_oauth_jwt authenticates users from any IP address by verifying that the client's OAuth token was signed by the identity provider's private key. It also requires the user to provide the proper values in the token's aud and scope fields:

```
=> CREATE AUTHENTICATION v_oauth_jwt METHOD 'oauth' HOST '0.0.0.0/0';
=> ALTER AUTHENTICATION v_oauth_jwt SET validate_type = 'JWT';
=> ALTER AUTHENTICATION v_oauth_jwt SET jwt_rsa_public_key =
'MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAkjXjd6F8PyKkQtY5q2cJ9jNT9y+PeIJS134UvuUnuD9bQFhkBPstTBulpZV1QQivYaQ5k5bYVE2Q
7n/XscrWzbRkK/qEwmztjVUH7dQAHSSKEjYcbH1fREx5nR5oNEelrUH2RrGM98Y7ln+Ch4oCl8yCS6gjI6hfaDxwqo2oImmGE+Qi06SljoWGBCr5EIAhvlNuWe
2rWD5uEe4ivaL6KoAR9sADqhGBoaYs/wIVUcv5DHtSjU+yZIz/sVspJehvmb/979eDsct2IddCHLrjUet3DiKz4imIyh+cv9VTEI6MWbk5WIiKW2fFzZ6Oei/ddzmTq
VoNdn+d18tWwIf7hQIDAQAB';
=> ALTER AUTHENTICATION v_oauth_jwt SET jwt_issuer = 'token_issuer';
=> ALTER AUTHENTICATION v_oauth_jwt SET jwt_user_mapping = 'oauth_user';
=> ALTER AUTHENTICATION v_oauth_jwt SET jwt_accepted_audience_list = 'vertica,local';
=> ALTER AUTHENTICATION v_oauth_jwt SET jwt_accepted_scope_list = 'email,profile,user';
```

For example, to reject all plaintext client connections, specify the reject authentication method and the HOST NO TLS access method as follows:

```
=> CREATE AUTHENTICATION RejectNoSSL METHOD 'reject' HOST NO TLS '0.0.0.0/0';  --IPv4
=> CREATE AUTHENTICATION RejectNoSSL METHOD 'reject' HOST NO TLS '::/0';       --IPv6
```

See also

- Deleting Authentication Records
- Enabling and disabling authentication methods
- Granting and revoking authentication methods
- Modifying authentication records

Modifying authentication records

To modify existing authentication records, you must first be connected to your database. The following examples show how to make changes to your authentication records. For more information see ALTER AUTHENTICATION .

Rename an authentication method

Rename the v_kerberos authentication method to K5 , and enable it. All users who have been associated with the v_kerberos authentication method are now associated with the K5 method granted instead.

```
=> ALTER AUTHENTICATION v_kerberos RENAME TO K5 ENABLE;
```

Specify a priority for an authentication method

Specify a priority of 10 for K5 authentication:

```
=> ALTER AUTHENTICATION K5 PRIORITY 10;
```

For more information see Authentication record priority .

Change a parameter

Set the system_users parameter for ident1 authentication to root :

```
=> CREATE AUTHENTICATION ident1 METHOD 'ident' LOCAL;
=> ALTER AUTHENTICATION ident1 SET system_users='root';
```

Change the IP address and specify the parameters for an LDAP authentication method named Ldap1 .

In this example, you specify the bind parameters for the LDAP server. Vertica connects to the LDAP server, which authenticates the Vertica client. If the authentication succeeds, Vertica authenticates any users who have been granted the Ldap1 authentication method on the designated LDAP server:

```
=> CREATE AUTHENTICATION Ldap1 METHOD 'ldap' HOST '172.16.65.196';
=> ALTER AUTHENTICATION Ldap1 SET host='ldap://172.16.65.177',
  binddn_prefix='cn=', binddn_suffix=',dc=qa_domain,dc=com';
```

Change the IP address, and specify the parameters for an LDAP authentication method named Ldap1 . Assume that Vertica does not have enough information to create the distinguished name (DN) for a user attempting to authenticate. Therefore, in this case, you must specify to use LDAP search and bind:

```
=> CREATE AUTHENTICATION LDAP1 METHOD 'ldap' HOST '172.16.65.196';
=> ALTER AUTHENTICATION Ldap1 SET host='ldap://172.16.65.177',
basedn='dc=qa_domain,dc=com',binddn='cn=Manager,dc=qa_domain,
dc=com',search_attribute='cn',bind_password='secret';
```

Change the associated method

Change the localpwd authentication from trust to hash:

```
=> CREATE AUTHENTICATION localpwd METHOD 'trust' LOCAL;
=> ALTER AUTHENTICATION localpwd METHOD 'hash';
```

ALTER AUTHENTICATION validates the parameters you enter. If there are errors, it disables the authentication method that you are trying to modify.

Using the administration tools

The advantages of using the Administration Tools are:

- You do not have to connect to the database
- The editor verifies that records are correctly formed
- The editor maintains records so they are available to you to edit later

> **Note**
> You must restart the database to implement your changes.

For information about using the Administration Tools to create and edit authentication records, see Creating authentication records .

Deleting authentication records

To delete client authentication record, use DROP AUTHENTICATION . To use this approach, you have to be connected to your database.

To delete an authentication record for md5_auth use the following command:

```
=> DROP AUTHENTICATION md5_auth;
```

To delete an authentication record for a method that has been granted to a user, use the CASCADE keyword:

```
=> CREATE AUTHENTICATION localpwd METHOD 'password' LOCAL;
=> GRANT AUTHENTICATION localpwd TO jsmith;
=> DROP AUTHENTICATION localpwd CASCADE;
```

See also

- Creating authentication records
- Granting and revoking authentication methods

Authentication record priority

Each authentication record has a priority . If a user is granted more than one authentication record, Vertica attempts to authenticate the user with the authentication record with the highest priority and rejects the user if authentication fails.

There are two ways to authenticate with a record other than that with the highest priority:

- Fallthrough authentication : If authentication fails, Vertica attempts to authenticate the client with the record with the next highest priority.
- Authentication filtering : Clients can send the credentials required for a particular authentication method to authenticate with a record that uses that method.

Determining authentication priority

The following factors contribute to an authentication record's priority, as reflected in the CLIENT_AUTH system table:

```
=> SELECT auth_name, auth_method, auth_priority, method_priority, address_priority FROM client_auth;
  auth_name   | auth_method | auth_priority | method_priority | address_priority
--------------+-------------+---------------+-----------------+------------------
 ldap_auth    | LDAP        |             5 |               5 |               96
 hash_auth    | HASH        |             5 |               2 |              126
 tls_auth     | TLS         |             0 |               5 |               96
 oauth_auth   | OAUTH       |             0 |               5 |               96
 gss_auth     | GSS         |             0 |               5 |               96
 trust_auth   | TRUST       |             0 |               0 |               96
 reject_auth  | REJECT      |             0 |              10 |               96
(7 rows)
```

Note
Greater values indicate higher priorities. For example:

- A priority of 10 is higher than a priority of 5.
- A priority 0 is the lowest possible value.

Priorities are divided into tiers and listed in order of importance; in the event of a tie at one priority tier, Vertica checks the next priority tier. For example, if a user had both ldap and hash authentication records with an auth_priority of 5, Vertica would attempt to use the ldap authentication record because it has a greater method_priority value:

1. auth_priority : The priority explicitly set with ALTER AUTHENTICATION (default: 0).
2. method_priority : The priority specific to the authentication method. These priorities are as follows:
   - trust : 0
   - hash : 2
   - ldap : 5
   - tls : 5
   - oauth : 5
   - gss : 5
   - reject : 10
3. address_priority : The priority for IP address specified in HOST [TLS | NO TLS] ' *host-ip-address* ' . This priority is determined by the size of the netmask of the address; fewer zeros indicate greater specificity, and therefore higher priority. LOCAL has the lowest priority: 0.

Setting authentication priority

To set authentication priority:

```
=> ALTER AUTHENTICATION authentication_name PRIORITY value;
```

See also

- CLIENT_AUTH
- CLIENT_AUTH_PARAMS
- Client authentication

Viewing information about client authentication records

For information about client authentication records that you have configured for your database, query the following system tables in the V_CATALOG schema:

- CLIENT_AUTH
- CLIENT_AUTH_PARAMS
- PASSWORD_AUDITOR
- USER_CLIENT_AUTH

To determine the details behind the client authentication used for a particular user session, query the following tables in the V_MONITOR schema:

- SESSIONS
- USER_SESSIONS

Enabling and disabling authentication methods

When you create an authentication method, Vertica stores it in the catalog and enables it automatically. To enable or disable an authentication method, use the ALTER AUTHENTICATION statement. To use this approach, you must be connected to your database.

If an authentication method has not been enabled, Vertica cannot use it to authenticate users and clients trying to connect to the database.

To enable an authentication method:

```
ALTER AUTHENTICATION v_kerberos ENABLE;
```

To disable this authentication method:

```
ALTER AUTHENTICATION v_kerberos DISABLE;
```

See also

- Creating authentication records
- Deleting Authentication Records
- Granting and revoking authentication methods
- Modifying authentication records

Granting and revoking authentication methods

Before Vertica can validate a user or client through an authentication method, you must first associate that authentication method with the user or role that requires it, with GRANT (authentication) . When that user or role no longer needs to connect to Vertica using that method, you can disassociate that authentication from that user with REVOKE AUTHENTICATION.

Grant authentication methods

You can grant an authentication method to a specific user or role. You can also specify the default authentication method by granting an authentication method to PUBLIC , as in the following examples.

- Associate v_ldap authentication with user jsmith :

```
=> GRANT AUTHENTICATION v_ldap TO jsmith;
```

- Associate v_gss authentication to the role DBprogrammer :

```
=> CREATE ROLE DBprogrammer;
=> GRANT AUTHENTICATION v_gss TO DBprogrammer;
```

- Associate client authentication method v_localpwd with role PUBLIC , which is assigned by default to all users:

```
=> GRANT AUTHENTICATION v_localpwd TO PUBLIC;
```

Revoke authentication methods

If you no longer want to authenticate a user or client with a given authentication method, use the REVOKE (authentication) statement as in the following examples.

- Revoke v_ldap authentication from user jsmith :

```
=> REVOKE AUTHENTICATION v_ldap FROM jsmith;
```

- Revoke v_gss authentication from the role DBprogrammer :

```
=> REVOKE AUTHENTICATION v_gss FROM DBprogrammer;
```

- Revoke localpwd as the default client authentication method:

```
=> REVOKE AUTHENTICATION localpwd FROM PUBLIC;
```

Hiding database usernames

If you want to keep certain database usernames a secret from connecting clients and your authentication records do not use Fallthrough authentication , then these two user groups must share the same single authentication method (not necessarily the same authentication record):

- Users whose usernames must be kept secret.
- Users with the PUBLIC role.

If your authentication records use fallthrough, then ensure that the first authentication method that prompts for a password in the authentication chain is the same for both the secret users and the PUBLIC role. The following methods prompt for a password:

- hash
- ldap

A simple way to satisfy this condition is by duplicating the fallthrough chain for both groups with the same methods. For example, a valid authentication chain would be tls > ldap for both the secret users and the PUBLIC role.

Another valid configuration would be tls > ldap > hash for secret users, and ldap for the PUBLIC role.

Hash authentication

The hash authentication method authenticates users with passwords.

In general, you should prefer other authentication methods over hash .

In this section

- Password hashing algorithm
- Configuring hash authentication
- Passwords

Password hashing algorithm

> **Note**
> Vertica strongly recommends that you use SHA-512 for hash authentication.

Vertica does not store user passwords for the hash authentication method. Rather, Vertica stores a hash of the password. The hashing algorithm is determined by two parameters:

- A system-level configuration parameter, SecurityAlgorithm :
  ```
  => ALTER DATABASE DEFAULT SET PARAMETER SecurityAlgorithm = 'hashing_algorithm';
  ```
- A user-level parameter, SECURITY_ALGORITHM :
  ```
  => ALTER USER username SECURITY_ALGORITHM 'hashing_algorithm' IDENTIFIED BY 'new_password';
  ```

The system-level parameter, SecurityAlgorithm , can have the following values:

- SHA512 (default)
- MD5

The user-level parameter, SECURITY_ALGORITHM , can have the following values. Values other than NONE will take priority over the system-level parameter:

- NONE (default, uses algorithm specified by the system-level parameter SecurityAlgorithm)
- SHA512
- MD5

> **Note**
> If user's password is hashed with MD5, you cannot change their username with ALTER USER .

A user's EFFECTIVE_SECURITY_ALGORITHM is determined by a combination of the system-level and user-level parameters. If the user-level parameter is set to NONE , the effective security algorithm will be that of the system-level parameter. You can override the system-level parameter for a particular user by setting the user-level parameter to a non- NONE value.

You can view these parameters and their effects on each user by querying the system table PASSWORD_AUDITOR .

The following table shows the various combinations of the system-level and user-level parameters and the effective security algorithm for each.

FIPS mode forces the effective security algorithm to be SHA-512.

Parameter value		**Effective Security Algorithm**	
System level: SecurityAlgorithm	**User-level: SECURITY_ALGORITHM**	**Algorithm Used**	**Algorithm Used (FIPS mode)**
MD5	NONE	MD5	SHA-512
SHA512	NONE	SHA-512	SHA-512
MD5	MD5	MD5	SHA-512
SHA512	MD5	MD5	SHA-512
MD5	SHA512	SHA-512	SHA-512

SHA512	SHA512	SHA-512	SHA-512

Configuring hash authentication

The hash authentication method allows users to authenticate with a password.

Vertica stores hashes (SHA-512 by default) of passwords and not the passwords themselves. For details, see Password hashing algorithm .

1. Create an authentication record with the hash method. Authentication records are automatically enabled after creation. For example, to create the authentication record v_hash for users that log in from the IP address 192.0.2.0/24:
```
=> CREATE AUTHENTICATION v_hash METHOD 'hash' HOST '192.0.2.0/24';
```
2. Associate the v_hash authentication method with the desired users or roles, using a GRANT statement:
```
=> GRANT AUTHENTICATION v_hash to user1, user2, ...;
```

Passwords

Assign a password to a user to allow that user to connect to the database using password authentication. When the user supplies the correct password a connection to the database occurs.

Vertica hashes passwords according to each user's EFFECTIVE_SECURITY_ALGORITHM . However, the transmission of the hashed password from the client to Vertica is in plaintext. Thus, it is possible for a "man-in-the-middle" attack to intercept the plaintext password from the client.

Configuring Hash authentication ensures secure login using passwords.

About password creation and modification

You must be a superuser to create passwords for user accounts using the CREATE USER statement. A superuser can set any user account's password.

- To add a password, use the ALTER USER statement.
- To change a password, use ALTER USER or the vsql meta-command \password .

Users can also change their own passwords.

To make password authentication more effective, Vertica recommends that you enforce password policies that control how often users are forced to change passwords and the required content of a password. You set these policies using Profiles .

Default password authentication

When you have not specified any authentication methods, Vertica defaults to using password authentication for user accounts that have passwords.

If you create authentication methods, even for remote hosts, password authentication is disabled. In such cases, you must explicitly enable password authentication. The following commands create the local_pwd authentication method and make it the default for all users. When you create an authentication method, Vertica enables it automatically:

```
=> CREATE AUTHENTICATION local_pwd METHOD hash' LOCAL;
=> GRANT AUTHENTICATION local_pwd To Public;
```

In this section

- Profiles
- Password guidelines
- Password expiration
- Account locking

Profiles

You can set password policies for users by assigning them profiles. You can create multiple profiles to manage the password policies for several categories of users. For example, you could create one profile for interactive users that requires frequent password changes and another profile for user accounts that never requires password changes.

Defining profiles

You create profiles with CREATE PROFILE and alter existing profiles with ALTER PROFILE . Both statements let you set one or more profile parameters which can control, among other things, the minimum lifetime of a password, password complexity, and password-reset rules.

Each profile can specify one or more of the following policies.

- How often users must change their passwords
- How long a password must be set before it can be reset
- How many times users must change their passwords before they can reuse an old password
- How many times a user can fail to log in before the account is locked
- The required length and content of the password:
  - Maximum and minimum number of characters
  - Minimum number of capital letters, lowercase letters, digits, and symbols required in a password
  - How different a new password must be from the old password

Assigning profiles

After you define a profile, you can assign it to new and existing users with CREATE USER and ALTER USER , respectively.

Changes to profile policies for password content—for example, PASSWORD_MAX_LENGTH and PASSWORD_MIN_SYMBOLS —affect users only when they change their passwords. Vertica does not test existing passwords to verify that they comply with new password requirements. To enforce immediate compliance with new profile requirements, use ALTER USER...PASSWORD EXPIRE to immediately expire the current user's password. The next time the user logs in, Vertica prompts them to supply a new password, which must comply with the current policy.

Default profile

Each database contains a DEFAULT profile. Vertica assigns the default profile to users who are not explicitly assigned a profile. The default profile also sets parameters of non-default profiles in two cases:

- Profile parameters that are not explicitly set by CREATE PROFILE
- Parameters that ALTER PROFILE sets to DEFAULT

All parameters in the default profile are initially set to unlimited . You can use ALTER PROFILE to change these settings. For example, the following statement modifies the default profile parameter PASSWORD_MIN_SYMBOLS . The change requires passwords to contain at least one symbol, such as $, #, @. This change affects all profiles where PASSWORD_MIN_SYMBOLS is set to default :

```
ALTER PROFILE DEFAULT LIMIT PASSWORD_MIN_SYMBOLS 1;
```

Profile settings and client authentication

The following profile settings affect client authentication methods , such as LDAP or GSS:

- FAILED_LOGIN_ATTEMPTS
- PASSWORD_LOCK_TIME

All other profile settings are used only by Vertica to manage its passwords.

See also

- PROFILES
- Creating a database name and password

Password guidelines

For passwords to be effective, they must be hard to guess. You need to protect passwords from:

- Dictionary-style, brute-force attacks
- Users who have knowledge of the password holder (family names, birth dates , etc.)

Use Profiles to enforce good password practices (password length and required content). Make sure database users know the password guidelines, and encourage them not to use personal information in their passwords.

For guidelines on creating strong passwords go to Microsoft Tips for Creating a Strong Password .

See also

- Creating a database name and password

Password expiration

The following PROFILE parameters control the conditions for password expiration, new passwords, and minimum lifetime:

- PASSWORD_LIFE_TIME - The number of days a password remains valid
- PASSWORD_MIN_LIFE_TIME - The number of days a password must be set before it can be changed
- PASSWORD_GRACE_TIME - The number of days a password can be used after it expires

- PASSWORD_REUSE_MAX - The number of times you must change your password before you can reuse an earlier password
- PASSWORD_REUSE_TIME - The number of days that must pass after a password is set before you can reuse it
- PASSWORD_MIN_CHAR_CHANGE - Minimum number of characters that must be different from the previous password

For more details on these and other parameters, see CREATE PROFILE and ALTER PROFILE .

Important
Password expiration has no effect on current sessions.

Password expiration and grace period behavior

The profile parameter PASSWORD_LIFE_TIME controls the life time of a password in days. By default, the DEFAULT profile sets PASSWORD_LIFE_TIME to UNLIMITED , which disables password expiration. You can change this for the DEFAULT and custom profiles with ALTER PROFILE.

Normally, when a password expires, Vertica forces users to change their passwords the next time they log in. However, you can set a PASSWORD_GRACE_TIME to allow users to log in after their password expires. If a user logs in during their grace period, Vertica warns the user that their password has expired. Once this grace period ends, Vertica will issue the standard prompt to change the user's password.

Expire a password

You can expire a user's password immediately using the ALTER USER statement's PASSWORD EXPIRE parameter. By expiring a password, you can:

- Force users to comply with a change to password policy.
- Set a new password when a user forgets the old password.

Account locking

In a profile, you can set a password policy for how many consecutive failed login attempts a user account is allowed before locking. This locking mechanism helps prevent dictionary-style brute-force attempts to guess users' passwords.

Set account locking

Set this value using the FAILED_LOGIN_ATTEMPTS parameter using the CREATE PROFILE or ALTER PROFILE statement.

Vertica locks any user account that has more consecutive failed login attempts than the value to which you set FAILED_LOGIN_ATTEMPTS . The user cannot log in to a locked account, even by supplying the correct password.

Unlock a locked account

You can unlock accounts in one of two ways, depending on your privileges.

- **Manually** : If you are a superuser , you can manually unlock the account using the ALTER USER command.

  > **Note**
  > A superuser account cannot be locked, because it is the only user that can unlock accounts. For this reason, choose a very secure password for a superuser account. See Password guidelines for suggestions.

- **Password Lock Time Setting** : PASSWORD_LOCK_TIME specifies the number of days (units configurable with PasswordLockTimeUnit) an account is locked after a specified number of failed login attempts (configurable with FAILED_LOGIN_ATTEMPTS). Vertica automatically unlocks the account after the specified number of days has passed.

  If you set this parameter to UNLIMITED , the user's account is never automatically unlocked and a superuser must manually unlock it.

Ident authentication

The Ident protocol, defined in RFC 1413 , authenticates a database user with a system user name.To see if that system user can log in without specifying a password, you configure Vertica client authentication to query an Ident server. With this feature, the DBADMIN user can run automated scripts to execute tasks on the Vertica server.

> **Caution**
> Ident responses can be easily spoofed by untrusted servers. Use Ident authentication only on local connections, where the Ident server is installed on the same computer as the Vertica database server.

Following the instructions in these topics to install, set up, and configure Ident authentication for your database:

- Installing and setting up an ident server
- Configuring ident authentication for database users

Examples

The following examples show several ways to configure Ident authentication.

Allow system_user1 to connect to the database as Vertica vuser1 :

```
=> CREATE AUTHENTICATION v_ident METHOD 'ident' LOCAL;
=> ALTER AUTHENTICATION v_ident SET system_users='system_user1';
=> GRANT AUTHENTICATION v_ident to vuser1;
=> ALTER AUTHENTICATION v_ident ENABLE;
```

Allow system_user1 , system_user2 , and system_user3 to connect to the database as vuser1 . Use colons (:) to separate the user names:

```
=> CREATE AUTHENTICATION v_ident METHOD 'ident' LOCAL;
=> ALTER AUTHENTICATION v_ident SET system_users='system_user1:system_user2:system_user3';
=> GRANT AUTHENTICATION v_ident TO vuser1;
=> ALTER AUTHENTICATION v_ident ENABLE;
```

Associate the authentication with Public using a GRANT AUTHENTICATION statement. The users, system_user1 , system_user2 , and system_user3 can now connect to the database as any database user:

```
=> CREATE AUTHENTICATION v_ident METHOD 'ident' LOCAL;
=> ALTER AUTHENTICATION v_ident SET system_users='system_user1:system_user2:system_user3';
=> GRANT AUTHENTICATION v_ident to Public;
=> ALTER AUTHENTICATION v_ident ENABLE;
```

Set the system_users parameter to * to allow any system user to connect to the database as vuser1 :

```
=> CREATE AUTHENTICATION v_ident METHOD 'ident' LOCAL;
=> ALTER AUTHENTICATION v_ident SET system_users='*';
=> GRANT AUTHENTICATION v_ident TO vuser1;
=> ALTER AUTHENTICATION v_ident ENABLE;
```

Using a GRANT statement, associate the v_ident authentication with Public to allow system_user1 to log into the database as any database user:

```
=> CREATE AUTHENTICATION v_ident METHOD 'ident' LOCAL;
=> ALTER AUTHENTICATION v_ident SET system_users='system_user1';
=> GRANT AUTHENTICATION v_ident to Public;
=> ALTER AUTHENTICATION v_ident ENABLE;
```

In this section

- Installing and setting up an ident server
- Configuring ident authentication for database users

Installing and setting up an ident server

To use Ident authentication, you must install one or more packages, depending on your operating system, and enable the Ident server on your Vertica server. oidentd is an Ident daemon that is compatible with Vertica and compliant with RFC 1413 .

> **Note**
> You can find the source code and installation instructions for oidentd at the oidentd website .

To install and configure Ident authentication for use with your Vertica database, follow the appropriate steps for your operating system.

Red hat 7.x/CentOS 7.x

Install an Ident server on Red Hat 7.x or CentOS 7.x by installing the authd and xinetd packages:

```
$ yum install authd
$ yum install xinetd
```

Ubuntu/debian

Install oidentd on Ubuntu or Debian by running this command:

```
$ sudo apt-get install oidentd
```

SUSE Linux enterprise server

Install the pidentd and xinetd RPMs from the following locations:

- https://www.suse.com/LinuxPackages/packageRouter.jsp?product=server&version=11&service_pack=&architecture=i386&package_name=pidentd
- https://www.suse.com/LinuxPackages/packageRouter.jsp?product=server&version=11&service_pack=&architecture=i386&package_name=xinetd

Post-installation steps for ubuntu/debian

After you install oidentd on your Ubuntu/Debian system, continue with the following steps:

1. Verify that the Ident server accepts IPv6 connections to prevent authentication failure. To do so, you must enable this capability. In the script /etc/init.d/oidentd , change the line from:
   ```
   exec="/usr/sbin/oidentd"
   ```
   to
   ```
   exec="/usr/sbin/oidentd -a ::"
   ```
   Then, at the Linux prompt, start oidentd with -a :: .
2. Restart the server with the following command:
   ```
   $ /etc/init.d/oidentd restart
   ```

Post-installation steps for red hat 7.x/CentOS 7.x and SUSE Linux enterprise server

After you install the required packages on your Red Hat 7.x/CentOS 7.x or SUSE Linux Enterprise Server system, continue with the following steps:

1. Enable the auth service by setting disable = no in the configuration file /etc/xinetd.d/auth . If this file does not exist, create it. The following is a sample configuration file:
   ```
   service auth
   {
         disable = no
         socket_type = stream
         wait = no
         user = ident
         cps = 4096 10
         instances = UNLIMITED
         server = /usr/sbin/in.authd
         server_args = -t60 --xerror --os
   }
   ```
2. Restart the xinetd service with the following command:
   ```
   $ service xinetd restart
   ```

Configuring ident authentication for database users

To configure Ident authentication, take the following steps:

1. Create an authentication method that uses Ident.
   The Ident server must be installed on the same computer as your database, so specify the keyword LOCAL. Vertica requires that the Ident server and database always be on the same computer as the database.
   ```
   => CREATE AUTHENTICATION v_ident METHOD 'ident' LOCAL;
   ```
2. Set the Ident authentication parameters, specifying the system users who should be allowed to connect to your database.
   ```
   => ALTER AUTHENTICATION v_ident SET system_users='user1:user2:user3';
   ```
3. Associate the authentication method with the Vertica user. Use a GRANT statement that allows the system user user1 to log in using Ident authentication:
   ```
   => GRANT AUTHENTICATION v_ident TO user1;
   ```

Kerberos authentication

Kerberos authentication uses the following components to perform user authentication.

Client package

The Kerberos 5 client package communicates with the KDC server. This package is not included as part of the Vertica Analytics Platform installation. Kerberos software is built into Microsoft Windows. If you are using another operating system, you must obtain and install the client package.

If you do not already have the Kerberos 5 client package on your system, download it from the MIT Kerberos Distribution page . Install the package on each Vertica server and client used in Kerberos authentication, except the KDC itself.

Refer to the Kerberos documentation for installation instructions.

Service principals

A service principal consists of a host name, a service name, and a realm to which a set of credentials gets assigned (service/hostname@REALM). These credentials connect to the service, which is a host that you connect to over your network and authenticate using the KDC.

See Specify KDC information and configure realms to create the realm name. The host name must match the value supplied by the operating system. Typically this is the fully qualified host name. If the host name part of your principal does not match the value supplied by the operating system, Kerberos authentication fails.

Some systems use a hosts file (/etc/hosts or /etc/hostnames) to define host names. A hosts file can define more than one name for a host. The operating system supplies the first entry, so use that in your principal. For example, if your hosts file contains:

```
192.168.1.101 v_vmart_node0001.example.com v_vmart_node0001
```

then use v_vmart_node0001.example.com as the hostname value.

> **Note**
> Depending on your configuration it may be safer to use the fully qualified domain name rather than the hostname.

Configure the following as Kerberos principals:

- Each client (users or applications that connects to Vertica)
- The Vertica server

See the following topics for more information:

- Configure Vertica for Kerberos authentication
- Configure Clients for Kerberos Authentication

Keytab files

Principals are stored in encrypted keytab files. The keytab file contains the credentials for the Vertica principal. The keytab allows the Vertica server to authenticate itself to the KDC. You need the keytab so that Vertica Analytic Database does not have to prompt for a password.

Create one service principal for each node in your cluster. You can then either create individual keytab files (one for each node containing only that node's principal) or create one keytab file containing all the principals.

- **Create one keytab file with all principals** to simplify setup: all nodes have the same file, making initial setup easier. If you add nodes later you either update (and redistribute) the global keytab file or make separate keytabs for the new nodes. If a principal is compromised it is compromised on all nodes where it is present in a keytab file.
- **Create separate keytab files on each node** to simplify maintenance. Initial setup is more involved as you must create a different file on each node, but no principals are shared across nodes. If you add nodes later you create keytabs on the new nodes. Each node's ke ytab contains only one principal, the one to use for that node.

Ticket-granting ticket

The Ticket-Granting Ticket (TGT) retrieves service tickets that authenticates users to servers in the domain. Future login requests use the cached HTTP Service Ticket for authentication, unless it has expired as set in the ticket_lifetime parameter in krb5.conf.

Multi-realm support

> **Note**
> When assigning multiple realms to an authentication record, keep in mind that Vertica cannot distinguish between users from one realm and

users from the Vertica realm. This allows the same user to log in to Vertica from multiple realms at the same time.

Vertica provides multi-realm support for Kerberos authentication using the SET param=value parameter in ALTER AUTHENTICATION with REALM as the parameter:

```
=> ALTER AUTHENTICATION krb_auth_users set REALM='USERS.COM';
=> ALTER AUTHENTICATION krb_auth_realmad set REALM='REALM_AD.COM';
```

This allows you to assign a different realm so that users from another realm can authenticate to Vertica.

Mutli-realm support applies to GSS authentication types only. You can have one realm per authentication method. If you have multiple authentication methods, each can have its own realm:

```
=> SELECT * FROM client_auth;
auth_oid | auth_name | is_auth_enabled | auth_host_type | auth_host_address | auth_method | auth_parameters     | auth_priority
---------+-----------+-----------------+----------------+-------------------+-------------+-----------------+----------------
45035996 | krb001    |  True           |  HOST          |  0.0.0.0/0        |  GSS        | realm=USERS.COM     |  0
45035997 | user_auth |  True           |  LOCAL         |                   |  TRUST      |                     |  1000
45035737 | krb002    |  True           |  HOST          |  0.0.0.0/0        |  GSS        | realm=REALM_AD.COM  |  1
```

In this section

- Configure Vertica for Kerberos authentication
- Configure clients for Kerberos authentication
- Troubleshooting Kerberos authentication

Configure Vertica for Kerberos authentication

Kerberos provides a strong cryptographic authentication against the devices which lets the client & servers to communicate in a more secured manner. It addresses network security problems.

Your system must have one or more Kerberos Key Distribution Centers (KDC) installed and configured. The KDCs must be accessible from every node in your Vertica Analytic Database cluster.

The KDC must support Kerberos 5 using GSS-API. For details, see the MIT Kerberos Distribution Page.

In this section

In this section

- Create the Vertica principals and keytabs on Linux KDC
- Specify KDC information and configure realms
- Inform Vertica about the Kerberos principal
- Configure the authentication method for all clients
- Creating the principals and keytab on active directory
- Get the Kerberos ticket and authenticate Vertica

Create the Vertica principals and keytabs on Linux KDC

Vertica uses service principals for system-level operations. These principals identify the Vertica service and are used as follows:

- Kerberized Vertica clients request access to this service when they authenticate to the database.
- System processes like the Tuple Mover use this identity when they authenticate to external services such as Hadoop.

Create principals and keys as follows:

1. Start the Kerberos 5 database administration utility (kadmin or kadmin.local) to create Vertica principals on a Linux KDC.
   - Use kadmin if you are accessing the KDC on a remote server. If you have access to the Kerberos administrator password, you can use kadmin on any machine where the Kerberos 5 client package is installed. When you start kadmin , the utility prompts you for the Kerberos administrator's password. You might need root privileges on the client to run kadmin .
   - Use kadmin.local if:
     - The KDC is on the machine that you are logging in to.
     - You have root privileges on that server.

   kadmin.local does not require the administrators login credentials.

   For more information about the kadmin and kadmin.local commands, see the kadmin documentation .

2. Create one service principal for Vertica on each node. The host name must match the value supplied by the operating system. The following example creates the service principal vertica for the node named v_vmart_node0001.example.com :

```
$ sudo /usr/kerberos/sbin/kadmin.local
kadmin.local add_principal vertica/v_vmart_node0001.example.com
```

   Repeat the ktadd command once per principal. You can create separate keytabs for each principal user or add them all to a single keytab file (such as krb5.keytab). If you are using a single file, see the documentation for the -glob option in the MIT Kerberos documentation .
   You must have a user principal for each Vertica Analytic Database user that uses Kerberos Authentication. For example:

```
$ sudo /usr/kerberos/sbin/kadmin.local
kadmin.local add_principal [options] VerticaUser1
```

3. Copy each keytab file to the /etc folder on the corresponding cluster node. Use the same path and file name on all nodes.
4. On each node, make the keytab file readable by the file owner who is running the database process (typically, the Linux dbadmin user). For example, you can change ownership of the files to dbadmin as follows:

```
$ sudo chown dbadmin *.keytab
```

   Important
   In a production environment, you must control who can access the keytab file to prevent unauthorized users from delegating your server. For more information about delegation (also known as impersonation), see Technet.Microsoft.com .

   After you create a keytab file, you can use the klist command to view keys stored in the file:

```
$ sudo /usr/kerberos/bin/klist -ke -t
Keytab name: FILE:/etc/krb5.keytab
KVNO    Timestamp        Principal
---- ------------------- ------------------------------------------------------------------------
4     08/15/2017 7:35:41 vertica/v_vmart_node0001.example.com@EXAMPLE.COM (aes256-cts-hmac-sha1-96)
4     08/15/2017 7:35:41 vertica/v_vmart_node0001.example.com@EXAMPLE.COM (aes128-cts-hmac-sha1-96)
```

5. On Vertica run the following to ensure the Kerberos parameters are set correctly:

```
=> select parameter_name, current_value from configuration_parameters where parameter_name like 'Ker%';
parameter_name         |                  current_value
-----------------------+--------------------------------------------------------------------------
KerberosHostname       | v_vmart_node0001.example.com
KerberosKeytabFile     | /etc/krb5.keytab
KerberosRealm          | EXAMPLE.COM
KerberosTicketDuration | 0
KerberosServiceName    | vertica
(5 rows)
```

6. Ensure that all clients use the gss authentication method.
   From Vertica:

```
=> CREATE USER bob;
CREATE USER

=> CREATE AUTHENTICATION v_kerberos method 'gss' host '0.0.0.0/0';
CREATE AUTHENTICATION

=> ALTER AUTHENTICATION v_kerberos enable;
ALTER AUTHENTICATION

=> GRANT AUTHENTICATION v_kerberos to bob;
GRANT AUTHENTICATION
```

   From the operating system command line:

```
$ kinit bob

$ vsql -U bob -k vertica -K v_vmart_node0001.example.com -h v_vmart_node0001 -c "select client_authentication_name,
authentication_method from sessions;"
 client_authentication_name | authentication_method--
----------------------------+-----------------------
 v_kerberos                 |    GSS-Kerberos

(1 row)
```

7. On Vertica, run KERBEROS_CONFIG_CHECK to verify the Kerberos configuration. KERBEROS_CONFIG_CHECK verifies the following:
   - The existence of the kinit and kb5.conf files.
   - Whether the keytab file exists and is set
   - The Kerberos configuration parameters set in the database:
     - KerberosServiceName
     - KerberosHostname
     - KerberosRealm
     - Vertica Principal
   - That Kerberos can read the Vertica keys
   - That Kerberos can get the tickets for the Vertica principal
   - That Vertica can initialize the keys with kinit

Specify KDC information and configure realms

Each client and Vertica Analytic Database server in the Kerberos realm must have a valid, identically configured Kerberos configuration (krb5.conf) file. Without this file, the client does not know how to reach the KDC.

If you use Microsoft Active Directory, you do not need to perform this step. Refer to the Kerberos documentation for your platform for more information about the Kerberos configuration file on Active Directory.

At a minimum, you must configure the following sections in the krb5.conf file.

- [libdefaults]—Settings used by the Kerberos 5 library
- [realms]—Realm-specific contact information and settings
- [domain_realm]—Maps server hostnames to Kerberos realms

See the Kerberos documentation for information about other sections in this configuration file.

You must update the /etc/krb5.conf file to reflect your site's Kerberos configuration. The simplest way to enforce consistency among all clients and servers in the Kerberos realm is to copy the /etc/krb5.conf file from the KDC. Then, place this file in the /etc directory on each Vertica cluster node.

Inform Vertica about the Kerberos principal

Follow these steps to inform Vertica about the principal name and keytab location.

For information about the parameters that you are setting in this procedure, see Kerberos parameters .

1. Log in to the database as an administrator (typically dbadmin).
2. Set the KerberosKeyTabFile configuration parameter to point to the location of the keytab file:
   ```
   => ALTER DATABASE DEFAULT SET PARAMETER KerberosKeytabFile = '/etc/krb5.keytab';
   ```
   The keytab file must be in the same location (/etc/krb5.keytab in this example) on all nodes.
3. Set the service name for the Vertica principal; for example, vertica :
   ```
   => ALTER DATABASE DEFAULT SET PARAMETER KerberosServiceName = 'vertica';
   ```
4. Provide the realm portion of the principal, for example, EXAMPLE.COM :
   ```
   => ALTER DATABASE DEFAULT SET PARAMETER KerberosRealm = 'EXAMPLE.COM'
   ```

Configure the authentication method for all clients

To make sure that all clients use the gss authentication method, run the following statements:

```
=> CREATE AUTHENTICATION <method_name> METHOD 'gss' HOST '0.0.0.0/0';
=> GRANT AUTHENTICATION <method_name> TO Public;
```

For more information, see Implementing Client Authentication .

Creating the principals and keytab on active directory

Active Directory stores information about members of the Windows domain, including users and hosts.

Vertica uses the Kerberos protocol to access this information in order to authenticate Windows users to the Vertica database. The Kerberos protocol uses principals to identify users and keytab files to store their cryptographic information. You need to install the keytab files into Vertica to enable the Vertica database to cryptographically authenticate windows users.

This procedure describes:

- Creating a Vertica service principal.
- Exporting the keytab files for these principals
- Installing the keytab files in the Vertica database. This allows Vertica to authenticate Windows users and grant them access to the Vertica database.

1. Create a Windows account (principal) for the Vertica service and one Vertica host for each node/host in the cluster. This procedure creates Windows accounts for host verticanode01 and service vertica running on this node.
   When you create these accounts, select the following:
   - User cannot change password
   - Password never expires

   > **Note**
   > You can deselect **Password never expires** . However, if you change these user passwords, you must recreate the keytab files and reinstall them into Vertica. This includes repeating the entire procedure.

2. If you are using external tables on HDFS that are secured by Kerberos authentication, you *must* enable Delegation. To do so, access the Active Directory Users and Computers dialog, right-click the Windows account (principal) for the Vertica service, and select Delegation. Trust this user for delegation to any service.
3. Run the following command to create the keytab for the host verticanode01.dc.com node/host:

```
$ ktpass -out ./host.verticanode01.dc.com.keytab -princ host/verticanode01.dc.com@DC.COM -mapuser verticanode01
 -mapop set -pass secret  -ptype KRB5_NT_SRV_HST
```

4. Run the following command to create the keytab for the vertica service :

```
$ ktpass -out ./vertica.verticanode01dc.com.keytab -princ vertica/verticanode01.dc.com@DC.COM -mapuser vertica
 -mapop set -pass secret  -ptype KRB5_NT_PRINCIPAL
```

   For more information about keytab files, see Technet.Microsoft.com.
5. Run the following commands to verify that the service principal name is mapped correctly. You must run these commands for each node in your cluster:

```
$ setspn -L vertica
   Registered ServicePrincipalNamefor CN=vertica,CN=Users,DC=dc,DC=com
     vertica/verticanode01.dc.com

$ setspn -L verticanode01
   Registered ServicePrincipalNamefor CN=verticanode01,CN=Users,DC=dc,DC=com
     host/verticanode01.dc.com
```

6. Copy the keytabs you created above, vertica.verticanode01.dc.com.keytab and host.verticanode01.dc.com.keytab , to the Linux host verticanode01.dc.com .
7. Combine the keytab files into a single keytab:

```
[release@vertica krbTest]$ /usr/kerberos/sbin/ktutil
ktutil:  rkt host.verticanode01.dc.com.keytab
ktutil:  rkt vertica.verticanode01.dc.com.keytab
ktutil:  list
slot KVNO Principal
---- ---- ---------------------------------------------------------------------
   1    3  host/verticanode01.dc.com@DC.COM
   2   16  vertica/verticanode01.dc.com@DC.COM
ktutil:  wkt verticanode01.dc.com.keytab
ktutil:  exit
```

This creates a single keytab file that contains the server principal for authentication.

8. Copy the new keytab file to the catalog directory. For example:

```
$ cp verticanode01.dc.com.keytab /home/dbadmin/VMart/v_vmart_nodennnn_catalog
```

9. Test the keytab file's ability to retrieve a ticket to ensure it works from the Vertica node:

```
$ kinit vertica/verticanode01.dc.com -k -t verticanode01.dc.com.keytab
$ klist

Ticket cache: KFILE:/tmp/krb_ccache_1003
Default principal: vertica/verticanode01.dc.com@DC.COM

Valid starting Expires Service principal
04/08/2017 13:35:25 04/08/2017 23:35:25 krbtgt/DC.COM@DC.COM
        renew until 04/15/2017 14:35:25
```

When the ticket expires or not automatically retrieved you need to manually run the kinit command. See Get the Kerberos ticket and authenticate Vertica .

10. Set the right permissions and ownership on the keytab files:

```
$ chmod 600 verticanode01.dc.com.keytab
$ chown dbadmin:verticadba verticanode01.dc.com.keytab
```

11. Set the following Kerberos parameters using ALTER DATABASE to inform Vertica about the Kerberos principal:

```
KerberosKeytabFile=<CATALOGDIR>/verticanode01.dc.com.keytab
KerberosRealm=DC.COM
KerberosServiceName=vertica
KerberosTicketDuration = 0
KerberosHostname=verticanode01.dc.com
```

12. Restart the Vertica server.
13. Test your Kerberos setup as follows to ensure that all clients use the gss authentication method.

From Vertica:

```
=> CREATE USER windowsuser1;
CREATE USER

=> CREATE AUTHENTICATION v_kerberos method 'gss' host '0.0.0.0/0';
CREATE AUTHENTICATION

=> ALTER AUTHENTICATION v_kerberos enable;
ALTER AUTHENTICATION

=> GRANT AUTHENTICATION v_kerberos to windowsuser1;
GRANT AUTHENTICATION
```

From the operating system command line:

```
$ kinit windowsuser1

$ vsql -U windowsuser1 -k vertica -K verticanode01.dc.com -h verticanode01.dc.com -c "select client_authentication_name,
authentication_method from sessions;"
 client_authentication_name | authentication_method--
----------------------------+-----------------------
 v_kerberos                 |    GSS-Kerberos

(1 row)
```

14. Run KERBEROS_CONFIG_CHECK to verify the Kerberos configuration. KERBEROS_CONFIG_CHECK verifies the following:
    - The existence of the kinit and kb5.conf files.
    - Whether the keytab file exists and is set
    - The Kerberos configuration parameters set in the database:
        - KerberosServiceName
        - KerberosHostname
        - KerberosRealm
        - Vertica Principal
    - That Kerberos can read the Vertica keys
    - That Kerberos can get the tickets for the Vertica principal
    - That Vertica can initialize the keys with kinit

Get the Kerberos ticket and authenticate Vertica

If your organization uses Kerberos as part of the login process, Kerberos tickets are automatically retrieved upon login. Otherwise, you need to run kinit to retrieve the Kerberos ticket.

The following example shows how to retrieve the ticket and authenticate Vertica Analytic Database with the KDC using the kinit command. EXAMPLE.COM is the realm name. You must use the realm name with your username to retrieve a Kerberos ticket. See Specify KDC information and configure realms .

```
$ kinit
Password for principal_user@EXAMPLE.COM: kpasswd
```

You are prompted for the password of the principal user name created when you created the principals and keytabs (see Create the Vertica principals and keytabs on Linux KDC).

The Kerberos ticket gets cached for a pre-determined length of time. See Ticket Management in the Kerberos documentation for more information on setting expiration parameters.

Upon expiration, you need to run the kinit command again to retrieve another Kerberos ticket.

Configure clients for Kerberos authentication

Each supported platform has a different security framework. Thus, the steps required to configure and authenticate against Kerberos differ among clients.

On the server side, you construct the Vertica Kerberos service name principal using this format:

```
Kerberos_Service_Name/Kerberos_Host_Name@Kerberos_Realm
```

For each client, the GSS libraries require the following format for the Vertica service principal:

```
Kerberos_Service_Name@Kerberos_Host_Name
```

You can omit the realm portion of the principal because GSS libraries use the realm name of the configured default (Kerberos_Realm) realm.

For information about client connection strings, see the following topics:

- ODBC DSN connection properties
- JDBC connection properties
- ADO.NET connection properties
- (vsql) Command-line options

Note
A few scenarios exist in which the Vertica server principal name might not match the host name in the connection string. See Troubleshooting Kerberos Authentication for more information.

In this section

- Configure ODBC and vsql clients on non-windows platforms
- Configure ODBC and vsql Clients on Windows and ADO.NET
- Configure JDBC Clients on all Platforms

In this section

- Configure ODBC and vsql clients on non-windows platforms
- Configure ADO.NET, ODBC, and vsql clients on Windows
- Configure JDBC clients on all platforms

Configure ODBC and vsql clients on non-windows platforms

To configure an ODBC or vsql client on Linux or MAC OSX, you must first install the Kerberos 5 client package. See Kerberos authentication .

After you install the Kerberos 5 client package, you must provide clients with a valid Kerberos configuration file (krb5.conf). To communicate with the KDC, each client participating in Kerberos authentication must have a valid, identically configured krb5.conf file. The default location for the Kerberos configuration file is /etc/krb5.conf.

Tip
To enforce consistency among clients, Vertica Analytic Database, and the KDC, copy the /etc/krb5.conf file from the KDC to the client's/etc directory.

The Kerberos configuration (krb5.conf) file contains Kerberos-specific information, including:

- How to reach the KDC
- Default realm name
- Domain
- Path to log files
- DNS lookup
- Encryption types to use
- Ticket lifetime

The default location for the Kerberos configuration file is /etc/krb5.conf .

When configured properly, the client can authenticate with Kerberos and retrieve a ticket through the kinit utility (see Acquire an ODBC Authentication Request and Connection below). Likewise, the server can then use ktutil to store its credentials in a keytab file

Authenticating ODBC and vsql clients requests and connections on non-windows platforms

ODBC and vsql use the client's ticket established by kinit to perform Kerberos authentication. These clients rely on the security library's default mechanisms to find the ticket file and the and Kerberos configuration file.

To authenticate against Kerberos, call the kinit utility to obtain a ticket from the Kerberos KDC server. The following two examples show how to send the ticket request using ODBC and vsql clients.

Acquire an ODBC authentication request and connection

1. On an ODBC client, acquire a ticket for the kuser user by calling the kinit utility.
```
$ kinit kuser@EXAMPLE.COM
Password for kuser@EXAMPLE.COM:
```
2. Connect to Vertica, and provide the principals in the connection string:
```
char outStr[100];
SQLLEN len;
SQLDriverConnect(handle, NULL, "Database=VMart;User=kuser;
Server=myserver.example.com;Port=5433;KerberosHostname=vcluster.example.com",
SQL_NTS, outStr, &len);
```

Acquire a vsql authentication request connection

If the vsql client is on the same machine you are connecting to, vsql connects through a UNIX domain socket. This connection bypasses Kerberos authentication. When you authenticate with Kerberos, especially if the client authentication method is configured as 'local', you must include the -h hostname option. See Command Line Options .

1. On the vsql client, call the kinit utility:

```
$ kinit kuser@EXAMPLE.COM
Password for kuser@EXAMPLE.COM:
```

2. Connect to Vertica, and provide the host and user principals in the connection string:

```
$ ./vsql -K vcluster.example.com -h myserver.example.com -U kuser
Welcome to vsql, the Vertica Analytic Database
interactive terminal.

Type:  \h or \? for help with vsql commands
\g or terminate with semicolon to execute query
\q to quit
```

In the future, when you log in to vsql as kuser , vsql uses your cached ticket without prompting you for a password.

Verify the authentication method

You can verify the authentication method by querying the SESSIONS system table:

```
=> SELECT authentication_method FROM sessions;
 authentication_method
-----------------------
GSS-Kerberos
(1 row)
```

See also

- ODBC DSN connection properties
- (vsql) Command-line options

Configure ADO.NET, ODBC, and vsql clients on Windows

The Vertica client drivers support the Windows SSPI library for Kerberos authentication. Windows Kerberos configuration is stored in the registry.

You can choose between two different setup scenarios for Kerberos authentication on ODBC and vsql clients on Windows and ADO.NET:

- Windows KDC on Active Directory with Windows Built-in Kerberos Client and
- Linux KDC with Windows Built-in Kerberos Client and

Windows KDC on active directory with windows built-in Kerberos client and Vertica

Kerberos authentication on Windows is commonly used with Active Directory, Microsoft's enterprise directory service/Kerberos implementation.Typically your organization's network or IT administrator performs the setup.

Windows clients have Kerberos authentication built into the authentication process. You do not need any additional software.

Your login credentials authenticate you to the Kerberos server (KDC) when you:

- Log in to Windows from a client machine
- Use a Windows instance that has been configured to use Kerberos through Active Directory

To use Kerberos authentication on Windows clients, log in as REALM\user.

Important

When you use the ADO.NET driver to connect to Vertica, you can optionally specify IntegratedSecurity=true in the connection string. This informs the driver to authenticate the calling user against the user's Windows credentials. As a result, you do not need to include a user name or password in the connection string. Any user=< *username* > entry to the connection string is ignored.

Linux KDC with windows built-in Kerberos client and Vertica

A simple, but less common scenario is to configure Windows to authenticate against a non-Windows KDC. In this implementation, you use the ksetup utility to point the Windows operating system native Kerberos capabilities at a non-Active Directory KDC. By logging in to Windows, you obtain a ticket-granting ticket, similar to the Active Directory implementation. However, in this case, Windows is internally communicating with a Linux KDC. See the

Microsoft Windows Server Ksetup page for more information.

When a database/windows user logs into their Windows machine (or after performing a kinit on Windows) the Kerberos ticket MUST have ok_as_delegate and forwardable flag set to be able to access webhdfs based external tables as follows:

```
$ CMD \> klist
#2>     Client: release @ VERT.LOCAL
Server: vertica/example.com @ VERT.LOCAL
KerbTicket Encryption Type: RSADSI RC4-HMAC(NT)
Ticket Flags 0x40a50000 forwardable renewable pre_authent ok_as_delegate name_canonicalize
Start Time: 9/27/2017 13:24:43 (local)
End Time:   9/27/2017 20:34:45 (local)
Renew Time: 10/3/2017 15:04:45 (local)
Session Key Type: RSADSI RC4-HMAC(NT)
Cache Flags: 0
Kdc Called: ADKDC01
```

> **Note**
> The Ticket Flags setting above must contain ok_as_delegate and forwardable entries. For information on these parameters see Kerberos documentation .

Configure windows clients for Kerberos authentication

Depending on which implementation you want to configure, refer to one of the following pages on the Microsoft Server website:

- To set up Windows clients with Active Directory, refer to Step-by-Step Guide to Kerberos 5 (krb5 1.0) Interoperability .
- To set up Windows clients with the ksetup utility, refer to the Ksetup page.

Authenticate and connect clients

The KDC can authenticate both an ADO.NET and a vsql client.

> **Note**
> Use the fully-qualified domain name as the server in your connection string; for example, use host.example.com instead of just host . That way, if the server moves location, you do not have to change your connection string.

Verify an ADO.NET authentication request and connection

This example shows how to use the IntegratedSecurity=true , setting to specify that the ADO.NET driver authenticate the calling user's Windows credentials:

```
VerticaConnection conn = new
VerticaConnection("Database=VMart;Server=host.example.com;
Port=5433;IntegratedSecurity=true;
KerberosServiceName=vertica;KerberosHostname=vcluster.example.com");
conn.open();
```

Verify a vsql authentication request and connection

1. Log in to your Windows client, for example, as EXAMPLE\kuser .
2. Run the vsql client and supply the connection string to Vertica:

```
C:\Users\kuser\Desktop>vsql.exe -h host.example.com -K vcluster -U kuser

Welcome to vsql, the Vertica Analytic Database interactive terminal.
Type:  \h or \? for help with vsql commands
\g or terminate with semicolon to execute query
\q to quit
```

See also

- Kerberos authentication
- Command-line options
- ADO.NET connection properties

Configure JDBC clients on all platforms

Kerberos authentication on JDBC clients uses Java Authentication and Authorization Service (JAAS) to acquire the initial Kerberos credentials. JAAS is an API framework that hides platform-specific authentication details and provides a consistent interface for other applications.

You specify the client login process through the JAAS Login Configuration File. This file contains options that specify the authentication method and other settings to use for Kerberos. A class called the LoginModule defines valid options in the configuration file.

The JDBC client principal is crafted as jdbc-username@server-from-connection-string .

Implement the LoginModule

Vertica recommends that you use the JAAS public class com.sun.security.auth.module.Krb5LoginModul provided in the Java Runtime Environment (JRE).

The Krb5LoginModule authenticates users using Kerberos protocols and is implemented differently on non-Windows and Windows platforms:

- **On non-Windows platforms:** The Krb5LoginModule defers to a native Kerberos client implementation. Thus, you can use the same /etc/krb5.conf setup as you use to configure ODBC and vsql clients on Linux and MAC OSX platforms.
- **On Windows platforms:** The Krb5LoginModule uses a custom Kerberos client implementation bundled with the Java Runtime Environment (JRE). Windows settings are stored in a %WINDIR%\krb5.ini file, which has similar syntax and conventions to the non-Windows krb5.conf file. You can copy a krb5.conf from a non-Windows client to %WINDIR%\krb5.ini .

You can find documentation for the LoginModules in the com.sun.security.auth package, and on the Krb5LoginModule web page.

Create the JAAS login configuration

The JAASConfigName connection property identifies a specific configuration within a JAAS configuration that contains the Krb5LoginModule and its settings. The JAASConfigName setting lets multiple JDBC applications with different Kerberos settings coexist on a single host. The default configuration name is verticajdbc .

Important
Carefully construct the JAAS login configuration file. If syntax is incorrect, authentication fails.

You can configure JAAS-related settings in the java.security master security properties file. This file resides in the lib/security directory of the JRE. For more information, see Appendix A in the Java ™ Authentication and Authorization Service (JAAS) Reference Guide.

Create a JDBC login context

The following example shows how to create a login context for Kerberos authentication on a JDBC client. The client uses the default JAASConfigName of verticajdbc and specifies that:

- The ticket-granting ticket will be obtained from the ticket cache
- The user will not be prompted for a password if credentials cannot be obtained from the cache, keytab file, or through a shared state.

```
verticajdbc {
  com.sun.security.auth.module.Krb5LoginModule
  required
  useTicketCache=true
  doNotPrompt=true;
};
```

JDBC authentication request and connection

You can configure the Krb5LoginModule to use a cached ticket or keytab. The driver can also acquire a ticket or keytab automatically if the calling user provides a password.

In the preceding example, the login process uses a cached ticket and does not prompt for a password because both useTicketCache and doNotPrompt are set to true . If doNotPrompt=false and you provide a user name and password during the login process, the driver provides that information to the LoginModule. The driver then calls the kinit utility on your behalf.

1. On a JDBC client, call the kinit utility to acquire a ticket:
   ```
   $ kinit kuser@EXAMPLE.COM
   ```
   If you prefer to use a password instead of calling the kinit utility, see the next section.
2. Connect to Vertica:

```
Properties props = new Properties();
props.setProperty("user", "kuser");
props.setProperty("KerberosServiceName", "vertica");
props.setProperty("KerberosHostName", "vcluster.example.com");
props.setProperty("JAASConfigName", "verticajdbc");
Connection conn = DriverManager.getConnection
"jdbc:vertica://myserver.example.com:5433/VMart", props);
```

Have the driver acquire a ticket

Sometimes, you may want to bypass calling the kinit utility yourself but still use encrypted, mutual authentication. In such cases, you can optionally pass the driver a clear text password to acquire the ticket from the KDC. The password is encrypted when sent across the network. For example, useTicketCache and doNotPrompt are both false in the following example. Thus, the calling user's credentials are not obtained through the ticket cache or keytab.

```
$ verticajdbc  {
  com.sun.security.auth.module.Krb5LoginModule
  required
  useTicketCache=false
  doNotPrompt=false;
};
```

The preceding example demonstrates the flexibility of JAAS. The driver no longer looks for a cached ticket, and you do not have to call kinit . Instead, the driver takes the password and user name and calls kinit on your behalf.

See also

- Kerberos Client/Server Requirements
- JDBC connection properties
- Java ™ Authentication and Authorization Service (JAAS) Reference Guide (external website)

Troubleshooting Kerberos authentication

These tips can help you avoid issues related to Kerberos authentication with Vertica and to troubleshoot any problems that occur.

JDBC client authentication fails

If Kerberos authentication fails on a JDBC client, check the JAAS login configuration file for syntax issues. If syntax is incorrect, authentication fails.

Working domain name service (DNS) not configured

Verify that the DNS entries and the system host file (/etc/hosts or /etc/hostnames) on the network are all properly configured for your environment. If you are using a fully qualified domain name, ensure that is properly configured as well. Refer to the Kerberos documentation for your platform for details.

System clocks out of sync

System clocks in your network must remain in sync for Kerberos authentication to work properly. If you access data in HDFS, then Vertica nodes must also be in sync with Hadoop.

All systems except red hat 7/CentOS 7

To keep system clocks in sync:

1. Install NTP on the Kerberos server (KDC).
2. Install NTP on each server in your network.
3. Synchronize system clocks on all machines that participate in the Kerberos realm within a few minutes of the KDC and each other

Clock skew can be a problem on Linux virtual machines that need to sync with the Windows Time Service. Use the following stepsto keep time in sync:

1. Using any text editor, open /etc/ntp.conf .
2. Under the Undisciplined Local Clock section, add the IP address for the Vertica server. Then, remove existing server entries.
3. Log in to the server as root, and set up a cron job to sync time with the added IP address every half hour, or as often as needed. For example:
   ```
   # 0 */2 * * * /etc/init.d/ntpd restart
   ```
4. Alternatively, run the following command to force clock sync immediately:
   ```
   $ sudo /etc/init.d/ntpd restart
   ```

For more information, see Enabling network time protocol (NTP) and the Network Time Protocol website .

Red hat 7/CentOS 7 systems

In Red Hat 7/CentOS 7, ntpd is deprecated in favor of chrony . To keep system clocks in your network in sync for Kerberos authentication to work properly, do the following:

1. Install chrony on the Kerberos server (KDC).
2. Install chrony on each server in your network.
3. Synchronize system clocks on all machines that participate in the Kerberos realm within a few minutes of the KDC and each other.

Clock skew on Linux virtual machines

Clock skew can be problematic on Linux virtual machines that need to sync with the Windows Time Service. Try the following to keep time in sync:

1. Using any text editor, open /etc/chrony.conf .
2. Under the Undisciplined Local Clock section, add the IP address for the Vertica server. Then, remove existing server entries.
3. Log in to the server as root, and set up a cron job to sync time with the added IP address every half hour, or as often as needed. For example:
```
# 0 */2 * * * systemctl start chronyd
```
4. Alternatively, run the following command to force clock sync immediately:
```
$ sudo systemctl start chronyd
```

For more information, see the Red Hat chrony guide .

Kerberos ticket is valid, but Hadoop access fails

Vertica uses Kerberos tickets to obtain Hadoop tokens. It then uses the Hadoop tokens to access the Hadoop data. Hadoop tokens expire after a period of time, so Vertica periodically refreshes them. However, if your Hadoop cluster is set to expire tokens frequently, it is possible that tokens might not be refreshed in time. If the token expires, you cannot access data.

Setting the HadoopFSTokenRefreshFrequency configuration parameter allows you to specify how often Vertica should refresh the token. Specify this value, in seconds, to be smaller than the expiration period set for Hadoop. For example:

```
=> ALTER DATABASE exampledb SET HadoopFSTokenRefreshFrequency = '86400';
```

For another cause of Hadoop access failure, see System Clocks Out of Sync .

Encryption algorithm choices

Kerberos is based on symmetric encryption. Be sure that all Kerberos parties used in the Kerberos realm agree on the encryption algorithm to use. If they do not agree, authentication fails. You can review the exceptions in the vertica.log .

On a Windows client, be sure the encryption types match the types set on Active Directory. See Configure Vertica for Kerberos authentication .

Be aware that Kerberos is used only for securing the login process. After the login process completes, by default, information travels between client and server without encryption. If you want to encrypt traffic, use SSL. For details, see Implementing SSL .

Kerberos passwords not recognized

If you change your Kerberos password, you must re-create all of your keytab files.

Using the ODBC data source configuration utility

On Windows vsql clients, you may choose to use the ODBC Data Source Configuration utility and supply a client Data Source. If so, be sure you enter a Kerberos host name in the Client Settings tab to avoid client connection failures with the Vertica Analytic Database server.

Authentication failure in backup, restore, or admin tools

This problem can arise in configurations where each Vertica node uses its own Kerberos principal. (This configuration is recommended.) When using vbr or admintools you might see an error such as the following:

```
$ vsql: GSSAPI continuation error: Miscellaenous failure
GSSAPI continuation error: Server not found in Kerberos database
```

Backup/restore and the admin tools use the value of KerberosHostname, if it is set, in the Kerberos principal used to authenticate. The same value is used on all nodes. If you have defined one Kerberos principal per node, as recommended, this value does not match. To correct this, unset the KerberosHostname parameter:

```
=> ALTER DATABASE DEFAULT CLEAR KerberosHostname;
```

Server's principal name does not match host name

This problem can arise in configurations where a single Kerberos principal is used for all nodes. Vertica recommends against using a single Kerberos principal for all nodes. Instead, use one principal per node and do not set the KerberosHostname parameter.

In some cases during client connection, the Vertica server's principal name might not match the host name in the connection string. (See also Using the ODBC Data Source Configuration Utility in this topic.)

On Windows vsql clients, you may choose to use the ODBC Data Source Configuration utility and supply a client Data Source. If so, be sure you enter a Kerberos host name in the Client Settings tab to avoid client connection failures with the Vertica server.

On ODBC, JDBC, and ADO.NET clients, set the host name portion of the server's principal using the KerberosHostName connection string.

> **Tip**
> On vsql clients, you set the host name portion of the server's principal name using the -K KRB HOST command-line option. The default value is specified by the -h switch, which is the host name of the machine on which the Vertica server is running. -K is equivalent to the drivers' KerberosHostName connection string value.
>
> For details, see Command Line Options .

Principal/host mismatch issues and resolutions

The following issues can occur if the principal and host are mismatched.

The KerberosHostName configuration parameter has been overridden

For example, consider the following connection string:

```
jdbc:vertica://v_vmart_node0001.example.com/vmart?user=kuser
```

Because the this connection string includes no explicit KerberosHostName parameter, the driver defaults to the host in the URL (v_vmart_node0001.example.com). If you overwrite the server-side KerberosHostName parameter as " abc ", the client generates an incorrect principal.

To resolve this issue, explicitly set the client's KerberosHostName to the connection string, as in this example:

```
jdbc:vertica://v_vmart_node0001.example.com/vmart?user=kuser&kerberoshostname=abc
```

Connection load balancing is enabled... but the node against which the client authenticates might not be the node in the connection string.

In this situation, consider changing all nodes to use the same KerberosHostName setting. When you use the default to the host that was originally specified in the connection string, load balancing cannot interfere with Kerberos authentication.

A DNS name does not match the Kerberos host name

For example, imagine a cluster of six servers, where you want hr-servers and finance-servers to connect to different nodes on the Vertica cluster. Kerberos authentication, however, occurs on a single (the same) KDC. In the following example, the Kerberos service host name of the servers is server.example.com .

Suppose you have the following list of example servers:

```
server1.example.com 192.16.10.11
server2.example.com 192.16.10.12
server3.example.com 192.16.10.13
server4.example.com 192.16.10.14
server5.example.com 192.16.10.15
server6.example.com 192.16.10.16
```

Now, assume you have the following DNS entries:

```
finance-servers.example.com 192.168.10.11, 192.168.10.12, 192.168.10.13
hr-servers.example.com 192.168.10.14, 192.168.10.15, 192.168.10.16
```

When you connect to finance-servers.example.com , specify:

- Kerberos -h host name option as server.example.com
- -K host option for hr-servers.example.com

For example:

```
$ vsql -h finance-servers.example.com -K server.example.com
```

No DNS is set up on the client machine... so you must connect by IP only

To resolve this issue, specify:

- Kerberos -h host name option for the IP address
- -K host option for server.example.com

For example:

```
$ vsql -h 192.168.1.12 -K server.example.com
```

There is a load balancer involved (Virtual IP)... but there is no DNS name for the VIP

Specify:

- Kerberos -h host name option for the Virtual IP address
- -K host option for server.example.com

For example:

```
$ vsql -h <virtual IP> -K server.example.com
```

You connect to Vertica using an IP address... but there is no host name to construct the Kerberos principal name.

Provide the instance or host name for the Vertica as described in Inform Vertica about the Kerberos principal

The server-side KerberosHostName configuration parameter is set to a name other than the Vertica node's host name... but the client cannot determine the host name based on the host name in the connection string alone.

Reset KerberosHostName to match the name of the Vertica node's host name. For more information, see the following topics:

- ODBC DSN Parameters
- JDBC Connection Properties
- ADO.NET Connection Properties

LDAP authentication

Lightweight Directory Access Protocol (LDAP) is an authentication method that works like password authentication. The main difference is that the LDAP method authenticates clients trying to access your Vertica database against an LDAP or Active Directory server. Use LDAP authentication when your database needs to authenticate a user with an LDAP or Active Directory server.

In this section

- LDAP prerequisites and definitions
- LDAP authentication parameters
- TLS for LDAP authentication
- Authentication fallthrough for LDAP
- LDAP bind methods

LDAP prerequisites and definitions

Prerequisites

Before you configure LDAP authentication for your Vertica database you must have:

- IP address and host name for the LDAP server. Vertica supports IPv4 and IPv6 addresses.
- Your organization's Active Directory information.
- A service account for search and bind.
- Administrative access to your Vertica database.
- open-ldap-tools package installed on at least one node. This package includes ldapsearch .

Definitions

The following definitions are important to remember for LDAP authentication:

Parameter name	Description

Host	IP address or host name of the LDAP server. Vertica supports IPv4 and IPv6 addresses. For more information, see IPv4 and IPv6 for Client Authentication .
Common name (CN)	Depending on your LDAP environment, this value can be either the username or the first and last name of the user.
Domain component (DC)	Comma-separated list that contains your organization's domain component broken up into separate values, for example: dc=vertica, dc=com
Distinguished name (DN)	*domain* .com. A DN consists of two DC components, as in "DC=example, DC= com".
Organizational unit (OU)	Unit in the organization with which the user is associated, for example, Vertica Users.
sAMAccountName	An Active Directory user account field. This value is usually the attribute to be searched when you use bind and search against the Microsoft Active Directory server.
UID	A commonly used LDAP account attribute used to store a username.
Bind	LDAP authentication method that allows basic binding using the DN.
Search and bind	LDAP authentication method that must log in to the LDAP server to search on the specified attribute.
Service account	An LDAP user account that can be used to log in to the LDAP server during bind and search. This account's password is usually shared.
Anonymous binding	Allows a client to connect and search the directory (search and bind) without needing to log in.
ldapsearch	A command-line utility to search the LDAP directory. It returns information that you use to configure LDAP search and bind.
basedn	Distinguished name where the directory search should begin.
binddn	Domain name to find in the directory search.
search_attribute	Text to search for to locate the user record. The default is UID.

LDAP authentication parameters

There are several parameters that you need to configure for LDAP authentication.

General LDAP parameters

Use the following parameters to configure for either LDAP bind or LDAP bind and search:

Parameter name	**Description**
host	LDAP server URL in the following format: *schema* ://host: *optional_port* Where *schema* is one of the following: • ldap : The connection between Vertica and the LDAP server uses plaintext if TLSMODE of LDAPAuth is DISABLE . Set TLSMODE to ENABLE or higher for StartTLS (LDAP over TLS). • ldaps : If the TLSMODE of LDAPAuth is ENABLE or higher, the connection between Vertica and the LDAP server uses LDAPS.

ldap_continue	When set to yes, this parameter allows a connection retry when a user not found error occurs during the previous connection attempt. For any other failure error, the system automatically retries the connection.
starttls	Whether to request the connection between Vertica and the LDAP server during user authentication to be upgraded to TLS. You must configure the LDAPAuth TLS Configuration before using this parameter. starttls can be set to one of the following: • soft : If the server does not support TLS, use a plaintext connection. This value is equivalent to the -Z option in ldapsearch . If you use soft , Vertica ignores the certificate verification policies of the TLSMODE in the LDAPAuth TLS configuration . • hard : If the LDAP server does not support TLS, reject the connection. This value is equivalent to the -ZZ in ldapsearch . Using ldaps is equivalent to starttls='hard' . However, if you use them together in the same connection string, authentication fails and the following error appears: FATAL 2248: Authentication failed for username "<user_name>" If starttls is not set, whether TLS is requested and required depends on the value of the TLSMODE of the LDAPAuth TLS Configuration .

LDAP bind parameters

The following parameters create a bind name string, which specifies and uniquely identifies a user to the LDAP server. For details, see Workflow for configuring LDAP bind .

To create a bind name string, you must set one (and only one) of the following:

- Both binddn_prefix and binddn_suffix (must be set together)
- domain_prefix
- email_suffix

For example, if you set binddn_prefix and binddn_suffix , you cannot also set email_suffix . Conversely, if you set email_suffix , you cannot set binddn_prefix and binddn_suffix .

If you do not set a bind parameter, Vertica performs bind and search operations instead of a bind operation.

The following examples use the authentication record v_ldap :

```
=> CREATE AUTHENTICATION v_ldap METHOD 'ldap' HOST '10.0.0.0/23';
```

Parameter name	Description

binddn_prefix	First half of the bind string. If you set this parameter, you must also set binddn_suffix .
	
	For example, to construct the bind name **cn=** exampleusername,cn=Users,dc=ExampleDomain,dc=com :
	
	`=> ALTER AUTHENTICATION v_ldap SET`
	`binddn_prefix='cn=', binddn_suffix=',cn=Users,dc=ExampleDomain,dc=com';`

binddn_suffix	Second half of bind string.
	
	If you set this parameter, you must also set binddn_prefix .
	
	For example, to construct the bind name cn=exampleusername, **ou=ExampleUsers,dc=example,dc=com** :
	
	`=> ALTER AUTHENTICATION v_ldap SET`
	`binddn_prefix='cn=', binddn_suffix=',ou=OrgUsers,dc=example,dc=com';`

domain_prefix	The domain that contains the user. For example, to construct the bind name **Example** \exampleusername : => ALTER AUTHENTICATION v_ldap SET domain_prefix='Example';
email_suffix	The email domain. For example, to construct the bind name exampleusername@ **example.com** => ALTER AUTHENTICATION v_ldap SET email_suffix='example.com';

LDAP search and bind parameters

Use the following parameters when authenticating with LDAP search and bind. For more information see Workflow for configuring LDAP search and bind .

Parameter name	Description
basedn	Base DN for search.
binddn	Bind DN. Domain name to find in the directory search.
bind_password	Bind password. Required if you specify a binddn.
search_attribute	Optional attribute to search for on the LDAP server.

The following example shows how to set these three attributes. In this example, it sets

- binddn to cn=Manager,dc=example,dc=com
- bind_password to secret
- search_attribute to cn

```
=> ALTER AUTHENTICATION auth_method_name SET host='ldap://example13',
basedn='dc=example,dc=com',binddn='cn=Manager,dc=example,dc=com',
bind_password='secret',search_attribute='cn';
```

The binddn and bind_password parameters are optional. If you omit them, Vertica performs an anonymous search.

TLS for LDAP authentication

Vertica establishes a connection to an LDAP server in two contexts, and each context has a corresponding TLS Configuration that controls if each connection should use TLS:

1. **LDAPLink** : using the LDAPLink service or its dry run functions to synchronize users and groups between Vertica and the LDAP server.
2. **LDAPAuth** : when a user with an ldap authentication method attempts to log into Vertica, Vertica attempts to bind the user to a matching user in the LDAP server. If the bind succeeds, Vertica allows the user to log in.

Query TLS_CONFIGURATIONS to view existing TLS Configurations:

```
=> SELECT * FROM tls_configurations WHERE name IN ('LDAPLink', 'LDAPAuth');
   name   |  owner  | certificate | ca_certificate | cipher_suites |  mode
----------+---------+-------------+----------------+---------------+----------
 LDAPLink | dbadmin | client_cert | ldap_ca        |               | VERIFY_CA
 LDAPAuth | dbadmin | client_cert | ldap_ca        |               | DISABLE
(2 rows)
```

This page covers the LDAPAuth context. For details on the LDAPLink context, see TLS for LDAP link .

Keep in mind that configuring TLS for LDAP authentication does not encrypt the connection between Vertica and the client with TLS. To configure client-server TLS, see Configuring client-server TLS .

Configuring LDAP authentication

After a client successfully establishes a connection with Vertica, they must authenticate as a user before they can interact with the database. If the user has the ldap authentication method, Vertica connects to the LDAP server to authenticate the user. To configure TLS for this context, use the following procedure.

Setting the LDAPAuth TLS configuration

The LDAPAuth TLS Configuration takes a client certificate and CA certificate created or imported with CREATE CERTIFICATE . Vertica presents the client certificate to the LDAP server for verification by its CA. Vertica uses the CA certificate to verify the LDAP server's certificate.

For details on key and certificate generation, see Generating TLS certificates and keys .

1. If you want Vertica to verify the LDAP server's certificate before establishing the connection, generate or import a CA certificate and add it to the LDAPAuth TLS CONFIGURATION.
   For example, to import the existing CA certificate LDAP_CA.crt :

```
=> \set ldap_ca '\''`cat ldap_ca.crt`'\''
=> CREATE CA CERTIFICATE ldap_ca AS :ldap_ca;
CREATE CERTIFICATE
```

   Then, to add the ldap_ca CA certificate to LDAPAuth:

```
ALTER TLS CONFIGURATION LDAPAuth ADD CA CERTIFICATES ldap_ca;
```

2. If your LDAP server verifies client certificates, you must generate or import a client certificate and its key and add it to the LDAPAuth TLS Configuration. Vertica presents this certificate to the LDAP server for verification by its CA.
   For example, to import the existing certificate client.crt (signed by the imported CA) and key client.key :

```
=> \set client_key '\''`cat client.key`'\''
=> CREATE KEY client_key TYPE 'RSA' AS :client_key;
CREATE KEY

=> \set client_cert '\''`cat client.crt`'\''
=> CREATE CERTIFICATE client_cert AS :client_cert SIGNED BY ldap_ca KEY client_key;
CREATE CERTIFICATE
```

   Then, to add client_cert to LDAPAuth:

```
=> ALTER TLS CONFIGURATION LDAPAuth CERTIFICATE client_cert;
```

3. Enable TLS or LDAPS (the exact protocol used depends on the value of host in the AUTHENTICATION object) by setting the TLSMODE to one of the following. TRY_VERIFY or higher requires a CA certificate:
   - **ENABLE** : Enables TLS. Vertica does not check the LDAP server's certificate.
   - **TRY_VERIFY** : Establishes a TLS connection if one of the following is true:
     - The LDAP server presents a valid certificate.
     - The LDAP server doesn't present a certificate.

     If the LDAP server presents an invalid certificate, a plaintext connection is used.
   - **VERIFY_CA** : Connection succeeds if Vertica verifies that the LDAP server's certificate is from a trusted CA. Using this TLSMODE forces all connections without a certificate to use plaintext.
   - **VERIFY_FULL** : Connection succeeds if Vertica verifies that the LDAP server's certificate is from a trusted CA and the cn (Common Name) or subjectAltName attribute matches the hostname or IP address of the LDAP server.
     The cn is used for the username, so subjectAltName must match the hostname or IP address of the LDAP server.

   > **Note**
   > The value of TLSMODE only applies to authentication records where the starttls LDAP authentication parameter is set to hard or not set at all. If starttls is set to soft , Vertica establishes a TLS connection without verifying the LDAP server's certificate and falls back to a plaintext connection if the LDAP server does not support TLS. For details, see the next section.

   For example:

```
=> ALTER TLS CONFIGURATION LDAPAuth TLSMODE 'verify_ca';
ALTER TLS CONFIGURATION
```

4. Verify that the LDAPAuthConfigParameter parameter is using the TLS Configuration:

```
=> SHOW CURRENT LDAPAuthTLSConfig;
 level |       name        | setting
---------+-------------------+----------
 DEFAULT | LDAPAuthTLSConfig | LDAPAuth
(1 row)
```

Creating an LDAP authentication record

After a client successfully establishes a connection with Vertica, they must authenticate as a user before they can interact with the database. If the user has the ldap authentication method, Vertica connects to the LDAP server and attempts a bind to authenticate the user.

To view existing authentication records, query CLIENT_AUTH .

For details on the parameters referenced in this procedure, see LDAP authentication parameters .

1. CREATE an authentication record with an LDAP method.
   Syntax for creating an LDAP authentication record:
   ```
   => CREATE AUTHENTICATION auth_record_name method 'ldap' HOST 'user_connection_source';
   ```
   For example, to create an LDAP authentication record that applies to users that connect from any host:
   ```
   => CREATE AUTHENTICATION ldap_auth METHOD 'ldap' HOST '0.0.0.0/0';
   ```
2. ALTER the authentication record to to set the host and port (optional) of the LDAP server and the domain name (*basedn*) and bind distinguished name (binddn).
   - To use a plaintext connection between Vertica and the LDAP server (disable TLS):
     - Begin the host URL with ldap:// .
     - Set the TLSMODE of LDAPAuth to DISABLE and verify that starttls is not set.
   - To use StartTLS and reject plaintext connections:
     - Begin the host URL with ldap:// .
     - Set the TLSMODE of LDAPAuth to ENABLE or higher. Vertica only verifies the LDAP server's certificate if TLSMODE is set to TRY_VERIFY or higher.
     - Verify that starttls is set to hard or not set.
   - To use StartTLS, but still accept a plaintext connection if the LDAP server cannot be upgrade the connection to TLS:
     - Begin the host URL with ldap:// .
     - Set starttls to soft and the TLSMODE of LDAPAuth to ENABLE or higher. Vertica does not verify the server's certificate before establishing the connection and ignores the certificate verification policy of the LDAPAuth TLSMODE.
   - To use LDAPS:
     - Begin the host URL with ldaps://
     - TLSMODE of LDAPAuth to ENABLE or higher.

   This example authentication record searches for users in the active directory orgunit.example.com on an LDAP server with an IP address of 192.0.2.0 on port 5389 and requires a TLS connection to the LDAP server:
   ```
   => ALTER AUTHENTICATION ldap_auth SET
      host='ldap://192.0.2.0:5389',
      basedn='ou=orgunit,dc=example,dc=com',
      binddn_prefix='cn=',
      binddn_suffix=',ou=orgunit,dc=example,dc=com',
      starttls='hard';
   ```
   The binddn_prefix and binddn_suffix combine to create the full DN. That is, for some Vertica user asmith, ' cn= **asmith** ,ou=orgunit,dc=example,dc=com ' is the full DN when Vertica attempts the bind.
   To modify the ldap_auth authentication record to request StartTLS, but still accept plaintext connections, set the starttls parameter to soft :
   ```
   => ALTER AUTHENTICATION ldap_auth SET starttls='soft';
   ```
3. Enable the authentication record:
   ```
   => ALTER AUTHENTICATION ldap_auth ENABLE;
   ```
4. GRANT the authentication record to a user or role.
   For example:
   ```
   => GRANT AUTHENTICATION ldap_auth TO asmith;
   ```
   In this case, when the user asmith attempts to log in, Vertica constructs the distinguished name 'cn=asmith,ou=orgunit,dc=example,dc=com' from the search base specified in the ldap_auth, connects to the LDAP server, and attempts to bind it to the Vertica user. If the bind succeeds, Vertica allows asmith to log in.

Authentication fallthrough for LDAP

To use multiple search attributes for a single LDAP server or to configure multiple LDAP servers, create a separate authentication record for each search attribute or server and enable authentication fallthrough on each ldap record except the last (in order of priority).

Examples

The following example creates two authentication records, vldap1 and vldap2 . Together, they specify that the LDAP server should first search the entire directory (basedn=dc=example,dc=com) for a DN with an OU attribute Sales . If the first search returns no results or otherwise fails, the LDAP server should then search for a DN with the OU attribute Marketing :

```
=> CREATE AUTHENTICATION vldap1 method 'ldap' HOST '10.0.0.0/8' FALLTHROUGH;
=> ALTER AUTHENTICATION vldap1 PRIORITY 1;
=> ALTER AUTHENTICATION vldap1
    SET host='ldap://ldap.example.com/search',
    basedn='dc=example,dc=com',
    search_attribute='Sales';
=> GRANT AUTHENTICATION vldap1 to public;
```

```
=> CREATE AUTHENTICATION vldap2 method 'ldap' HOST '10.0.0.0/8';
=> ALTER AUTHENTICATION vldap2 PRIORITY 0;
=> ALTER AUTHENTICATION vldap2 SET
    host='ldap://ldap.example.com/search',
    basedn='dc=example,dc=com',
    search_attribute='Marketing';
=> GRANT AUTHENTICATION vldap2 to public;
```

LDAP bind methods

There are two LDAP methods that you use to authenticate your Vertica database against an LDAP server.

- Bind—Use LDAP bind when Vertica connects to the LDAP server and binds using the CN and password. (These values are the username and password of the user logging into the database). Use the bind method when your LDAP account's CN field matches that of the username defined in your database. For more information see Workflow for configuring LDAP bind .
- Search and Bind —Use LDAP search and bind when your LDAP account's CN field is a user's full name or does not match the username defined in your database. For search and bind, the username is usually in another field such as UID or sAMAccountName in a standard Active Directory environment. Search and bind requires your organization's Active Directory information. This information allows Vertica to log into the LDAP server and search for the specified field. For more information see Workflow for configuring LDAP search and bind .
  If you are using search and bind, having a service account simplifies your server side configuration. In addition, you do not need to store your Active Directory password.

LDAP anonymous binding

Anonymous binding is an LDAP server function. Anonymous binding allows a client to connect and search the directory (bind and search) without logging in because binddn and bindpasswd are not needed.

You also do not need to log in when you configure LDAP authentication using Management Console.

In this section

- Workflow for configuring LDAP bind
- Workflow for configuring LDAP search and bind

Workflow for configuring LDAP bind

To configure your Vertica database to authenticate clients using LDAP bind, follow these steps:

1. Obtain a service account. For information see the LDAP product documentation .You cannot use the service account in the connection parameters for LDAP bind.
2. Compare the user's LDAP account name to their Vertica username. For example, if John Smith's Active Directory (AD) sAMAccountName = jsmith, his Vertica username must also be jsmith.
   However, the LDAP account does not have to match the database user name, as shown in the following example:

```
=> CREATE USER r1 IDENTIFIED BY 'password';
=> CREATE AUTHENTICATION ldap1  METHOD 'ldap' HOST '172.16.65.177';
=> ALTER AUTHENTICATION ldap1 SET HOST=
   'ldap://172.16.65.10',basedn='dc=dc,dc=com',binddn_suffix=',ou=unit2,dc=dc,dc=com',binddn_prefix='cn=use';
=> GRANT AUTHENTICATION ldap1 TO r1;
\! ${TARGET}/bin/vsql -p $PGPORT -U r1 -w $LDAP_USER_PASSWD -h ${HOSTNAME} -c
   "select user_name, client_authentication_name from sessions;"
 user_name | client_authentication_name
-----------+----------------------------
 r1        | ldap
(1 row)
```

3. Run ldapsearch from a Vertica node against your LDAP or AD server. Verify the connection to the server and identify the values of relevant fields. Running ldapsearch helps you build the client authentication string needed to configure LDAP authentication.
   In the following example, ldapsearch returns the CN, DN, and sAMAccountName fields (if they exist) for any user whose CN contains the username jsmith. This search succeeds only for LDAP servers that allow anonymous binding:

```
$ ldapsearch -x -h 10.10.10.10 -b "ou=Vertica Users,dc=CompanyCorp,dc=com"
'(cn=jsmith*)' cn dn uid sAMAccountName
```

   ldapsearch returns the following results. The relevant information for LDAP bind is in **bold** :

```
# extended LDIF
#
# LDAPv3
# base <ou=Vertica Users,dc=CompanyCorp,dc=com> with scope subtree
# filter: (cn=jsmith*)
# requesting: cn dn uid sAMAccountName
#
# jsmith, Users, CompanyCorp.com
dn:cn=jsmith,ou=Vertica Users,dc=CompanyCorp,dc=com
cn: jsmith
uid: jsmith
# search result
search: 2
result: 0 Success
# numResponses: 2
# numEntries: 1
```

4. Create a new authentication record based on the information from ldapsearch . In the ldapsearch entry, the CN is username jsmith, so you do not need to set it. Vertica automatically sets the CN to the username of the user who is trying to connect. Vertica uses that CN to bind against the LDAP server.

```
=> CREATE AUTHENTICATION v_ldap_bind METHOD 'ldap' HOST '0.0.0.0/0';
=> GRANT AUTHENTICATION v_ldap_bind TO public;
=> ALTER AUTHENTICATION v_ldap_bind SET
host='ldap://10.10.10.10/',
basedn='DC=CompanyCorp,DC=com',
binddn_prefix='cn=',
binddn_suffix=',OU=Vertica Users,DC=CompanyCorp,DC=com';
```

For more information see LDAP Bind Parameters .

Workflow for configuring LDAP search and bind

To configure your Vertica database to authenticate clients using LDAP search and bind, follow these steps:

1. Obtain a service account. For information see the LDAP product documentation .
2. From a Vertica node, run ldapsearch against your LDAP or AD server. Verify the connection to the server, and identify the values of relevant fields. Running ldapsearch helps you build the client authentication string needed to configure LDAP authentication.
   In the following example, ldapsearch returns the CN, DN, and sAMAccountName fields (if they exist) for any user whose CN contains the username, John. This search succeeds only for LDAP servers that allow anonymous binding:

```
$ ldapsearch -x -h 10.10.10.10 -b 'OU=Vertica Users,DC=CompanyCorp,DC=com' -s sub -D
'CompanyCorp\jsmith' -W '(cn=John*)' cn dn uid sAMAccountName
```

3. Review the results that ldapsearch returns.The relevant information for search and bind is in bold:

```
# extended LDIF
#
# LDAPv3
# base <OU=Vertica Users,DC=CompanyCorp,DC=com> with scope subtree
# filter: (cn=John*)
# requesting: cn dn sAMAccountName
#
# John Smith, Vertica Users, CompanyCorp.com
dn: CN=jsmith,OU=Vertica Users,DC=CompanyCorp,DC=com
cn: Jsmith
sAMAccountName: jsmith
# search result
search: 2
result: 0 Success
# numResponses: 2
# numEntries: 1
```

4. Create the client authentication record. The cn attribute contains the username you want—jsmith. Set your search attribute to the CN field so that the search finds the appropriate account.

```
=> CREATE AUTHENTICATION v_ldap_bind_search METHOD 'ldap' HOST '10.10.10.10';
=> GRANT AUTHENTICATION v_ldap_bind_search TO public;
=> ALTER AUTHENTICATION v_ldap_bind_search SET
host='ldap://10.10.10.10',
basedn='OU=Vertica,DC=CompanyCorp,DC=com',
binddn='CN=jsmith,OU=Vertica Users,DC=CompanyCorp,DC=com',
bind_password='password',
search_attribute='CN';
```

For more information see LDAP Bind and Search Parameters

OAuth 2.0 authentication

Rather than with a username and password, users can authenticate to Vertica by first verifying their identity with an identity provider, receiving an OAuth token, and then passing the token to Vertica.

OAuth in Vertica is tested with Keycloak and Okta, but other providers should work if they support the RFC 7662 Token Introspection standard.

In this section

- Configuring OAuth authentication
- Just-in-time user provisioning
- OAuth authentication parameters

Configuring OAuth authentication

For a list of ODBC OAuth connection properties, see ODBC DSN connection properties .

The following procedure performs the following actions:

1. Configures an identity provider for OAuth integration with Vertica (either Okta or Keycloak).
2. Creates an OAuth authentication record.
3. Retrieves an access token with a POST request.
4. Uses a sample application to authenticate to Vertica, passing the access token as an argument and, optionally, parameters for token refresh.

Configure the identity provider

Vertica officially tests and supports OAuth integration with Keycloak and Okta. Other identity providers should also work as long as they implement the RFC 7662 Token Introspection standard. The following example configurations are provided for reference:

- Configure Keycloak
- Configure Okta

Create an authentication record

In Vertica, create an authentication record for OAuth. This uses the client ID, client secret, and either the discovery (Keycloak) or introspect (Okta) endpoint used by your identity provider.

The following authentication record v_oauth authenticates users from any IP address by contacting the identity provider to validate the OAuth token (rather than a username and password) and uses the following parameters:

- validate_type : The method used to validate the OAuth token. This should be set to IDP (default) to validate the OAuth token by contacting the identity provider.
- client_id : The client in the identity provider.
- client_secret : The client secret generated by the identity provider.
- discovery_url : Also known as the OpenID Provider Configuration Document , Vertica uses this endpoint to retrieve information about the identity provider's configuration and other endpoints (Keycloak only).
- introspect_url : Used by Vertica to introspect (validate) access tokens. You must specify the introspect_url if you do not specify the discovery_url and are not using JSON Web Token validation.

If discovery_url and introspect_url are both set, discovery_url takes precedence. The following example sets both for demonstration purposes; in general, you should prefer to set the discovery_url :

```
=> CREATE AUTHENTICATION v_oauth METHOD 'oauth' HOST '0.0.0.0/0';
=> ALTER AUTHENTICATION v_oauth SET validate_type = 'IDP';
=> ALTER AUTHENTICATION v_oauth SET client_id = 'vertica';
=> ALTER AUTHENTICATION v_oauth SET client_secret = 'client_secret';
=> ALTER AUTHENTICATION v_oauth SET discovery_url = 'https://203.0.113.1:8443/realms/myrealm/.well-known/openid-configuration';
=> ALTER AUTHENTICATION v_oauth SET introspect_url = 'https://203.0.113.1:8443/realms/myrealm/protocol/openid-connect/token/introspect';
```

Alternatively, if your identity provider supports the OpenID Connect protocol and your client is public, Vertica can use JWT validation, where Vertica validates OAuth tokens by verifying that it was signed by the identity provider's private key.

While Vertica does not contact the identity provider for JWT validation, the client does for token refresh if either the discovery or token endpoints are set in oauthjsonconfig.

JWT validation requires the following parameters:

- validate_type : The validation method, IDP by default. Setting this to JWT enables JWT validation.
- jwt_rsa_public_key : In PEM format, the public key used to sign the client's OAuth token. Vertica uses this to validate the OAuth token. If your identity provider does not natively provide PEM-formatted public keys, you must convert them to PEM format. For example, keys retrieved from an Okta endpoint are in JWK format and must be converted.
- jwt_issuer : The issuer of the OAuth token. This value is set by the identify provider.
- jwt_user_mapping : The name of the Vertica user.

You can also specify the following parameters to define a whitelist based on fields of the OAuth token:

- jwt_accepted_audience_list : Optional, a comma-delimited list of values to accept from the client JWT's aud field. If set, tokens must include in aud one of the accepted audiences to authenticate.
- jwt_accepted_scope_list : Optional, a comma-delimited list of values to accept from the client JWT's scope field. If set, tokens must include in scope at least one of the accepted scopes to authenticate.

The following authentication record v_oauth_jwt authenticates users from any IP address by verifying that the client's OAuth token was signed by the identity provider's private key. It also requires the user to provide the proper values in the token's aud and scope fields:

```
=> CREATE AUTHENTICATION v_oauth_jwt METHOD 'oauth' HOST '0.0.0.0/0';
=> ALTER AUTHENTICATION v_oauth_jwt SET validate_type = 'JWT';
=> ALTER AUTHENTICATION v_oauth_jwt SET jwt_rsa_public_key =
'MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAkjXjd6F8PyKkQtY5q2cJ9jNT9y+PeIJS134UvuUnuD9bQFhkBPstTBulpZV1QQivYaQ5k5bYVE2Q
7n/XscrWzbRkK/qEwmztjVUH7dQAHSSKEjYcbH1fREx5nR5oNEelrUH2RrGM98Y7ln+Ch4oCl8yCS6gjI6hfaDxwqo2oImmGE+Qi06SIjoWGBCr5EIAhvlNuWe
2rWD5uEe4ivaL6KoAR9sADqhGBoaYs/wIVUcv5DHtSjU+yZIz/sVspJehvmb/979eDsct2IddCHLrjUet3DiKz4imIyh+cv9VTEI6MWbk5WIiKW2fFzZ6Oei/ddzmTq
VoNdn+d18tWwlf7hQIDAQAB';
=> ALTER AUTHENTICATION v_oauth_jwt SET jwt_issuer = 'token_issuer';
=> ALTER AUTHENTICATION v_oauth_jwt SET jwt_user_mapping = 'oauth_user';
=> ALTER AUTHENTICATION v_oauth_jwt SET jwt_accepted_audience_list = 'vertica,local';
=> ALTER AUTHENTICATION v_oauth_jwt SET jwt_accepted_scope_list = 'email,profile,user';
```

To enable the authentication record:

```
=> ALTER AUTHENTICATION v_oauth ENABLE;
```

For a full list of OAuth authentication parameters, see OAuth authentication parameters .

Create a Vertica user

Vertica users map to the identity provider's users with the same username. You can either create the user manually or enable just-in-time (JIT) user provisioning in the authentication record to automatically create users with valid tokens.

To manually create the user:

1. To map to the user oauth_user in the identity provider, create a Vertica user with the same name. You do not need to specify a password because authentication is performed by the identity provider:
   ```
   => CREATE USER oauth_user;
   ```
2. Grant the OAuth authentication record to the user (or their role):
   ```
   => GRANT AUTHENTICATION v_oauth TO oauth_user;
   => GRANT ALL ON SCHEMA PUBLIC TO oauth_user;
   ```

To enable JIT user provisioning:

```
=> ALTER AUTHENTICATION v_oauth SET oauth2_jit_enabled = 'yes';
```

If the user already exists and JIT user provisioning is enabled and you use Keycloak as your identity provider, Vertica automatically assigns the roles associated with the user as specified by the identity provider if the roles also exist in Vertica. For details, see Just-in-time user provisioning .

Retrieve an access token

The simple way to get an OAuth access token and refresh token is to send a POST request to the token endpoint, providing the credentials of the user. You can then use the returned access token, refresh token, and scope with the oauthaccesstoken and oauthrefreshtoken connection properties for your client. For details, see JDBC connection properties and ODBC DSN connection properties .

For example, to get an access token for oauth_user from Keycloak:

```
$ curl --location --request POST 'http://203.0.113.1:8080/realms/master/protocol/openid-connect/token' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'username=oauth_user' \
--data-urlencode 'password=oauth_user_password' \
--data-urlencode 'client_id=vertica' \
--data-urlencode 'client_secret=client_secret' \
--data-urlencode 'grant_type=password'
```

Keycloak responds with a json string containing the access_token and refresh_token if you authenticated correctly.

```
{
  "access_token":"access_token",
  "expires_in":60,
  "refresh_expires_in":1800,
  "refresh_token":"refresh_token",
  "token_type":"Bearer",
  "not-before-policy":0,
  "session_state":"6745892a-aa74-452f-b6b9-c45637193859",
  "scope":"profile email"
}
```

Similarly, to retrieve an access token for oauth_user from Okta:

```
$ curl --insecure -d "client_id=0oa5cgdga1fb812rW697" -d "client_secret=aq22wRl3Z3mmtuoB13omRo6Cl03LtafeI4xYI77p" -d "username=oauth_user" -d "pa

{"token_type":"Bearer","expires_in":3600,"access_token":"access_token","id_token":"id_token"}
```

Run the sample applications

The OAuth sample applications, at a minimum, take an access token as an argument to authenticate to the database until the token expires. If you want the sample application to refresh the token after it expires, you must specify the following. The sample applications put these into a JSON string, OAuthJsonConfig or (ODBC) oauthjsonconfig (JDBC).

- Refresh token
- Client ID
- Client secret
- Token URL

ODBC

1. Follow the instructions in the README .
2. Run the sample application , passing the OAuth parameters as arguments:
   - To authenticate until the token expires:

     ```
     $ ./a.out --access-token OAuthAccessToken
     ```

   - To authenticate and silently refresh the access token when it expires:

     ```
     $ ./a.out --access-token OAuthAccessToken
       --refresh-token OAuthRefreshToken
       --client-id OAuthClientID
       --client-secret OAuthClientSecret
       --token-url OAuthTokenURL
     ```

For a list of all ODBC OAuth parameters, see ODBC DSN connection properties .

JDBC

1. Follow the instructions in the README .
2. Run the sample application , passing the OAuth parameters as arguments:
   - To authenticate until the token expires:

     ```
     $ mvn compile exec:java -Dexec.mainClass=OAuthSampleApp -Dexec.args="vertica_host database_name --access-token oauthaccesstoken"
     ```

   - To authenticate and silently refresh the access token when it expires:

     ```
     $ mvn compile exec:java -Dexec.mainClass=OAuthSampleApp -Dexec.args="vertica_host database_name --access-token oauthaccesstoken
       --refresh_token oauthrefreshtoken
       --client-id oauthclientid
       --client-secret oauthclientsecret
       --token-url oauthtokenurl"
     ```

For a list of all JDBC OAuth parameters, see JDBC connection properties .

Troubleshooting

To get debugging information for TLS, use the -Djavax.net.debug=ssl flag.

Custom CA certificates

A truststore is a container for trusted certificate authority (CA) certificates. These CA certificates are used to verify the identities of other systems when establishing a TLS connection. When your JDBC client connects to the identity provider through an HTTPS endpoint, the JDBC client verifies the identity provider's certificate by making sure that it was issued by a CA in the truststore.

If you configure your identity provider with TLS (that is, if you use HTTPS endpoints for your token or refresh URLs) and its certificate is not issued by a well-known CA, you must either specify a custom truststore or import the issuer's CA certificate into the system truststore with keytool .

To specify a custom truststore, set the JDBC connection properties oauthtruststorepath and oauthtruststorepassword :

```
connProps = new Properties(connProps);
connProps.setProperty("oauthtruststorepath", "/path/to/truststore/customoauth.truststore");
connProps.setProperty("oauthtruststorepassword", "password");
```

To add the certificate keycloak/cert.crt to the Java truststore:

```
$ keytool -trustcacerts -keystore /usr/lib/jvm/java-1.7.0-openjdk-1.7.0.261-2.6.22.2.el7_8.x86_64/jre/lib/security/cacerts -storepass changeit -importcert -alias keycloak -file /keycloak/cert.crt
```

In this section

- Configure Keycloak
- Configure Okta

Configure Keycloak

The following procedure configures a Keycloak 18.0.0 server on 203.0.113.1 for integration with Vertica. For details, see Configuring OAuth authentication .

The goals of this procedure are to configure Keycloak and obtain the following information:

- Client ID: The ID used to identify the Vertica database. This is configured by the user and set to vertica in the example procedure.
- Client secret: A Keycloak-generated string used to refresh the OAuth token when it expires.
- Discovery endpoint: The endpoint that serves information for all other endpoints as a JSON string. The endpoint for a Keycloak server on 203.0.113.1 is one of the following:
  - https://203.0.113.1:8443/realms/myrealm/.well-known/openid-configuration (if TLS is configured)
  - http://203.0.113.1:8443/realms/myrealm/.well-known/openid-configuration

Configure TLS (optional)

If you want to use TLS, you must obtain a certificate and key for Keycloak signed by a trusted CA. This example uses a self-signed CA for convenience. The following example creates a certificate and key in Vertica:

1. Generate the CA certificate:

```
=> CREATE KEY SSCA_key TYPE 'RSA' LENGTH 2048;
CREATE KEY

=> CREATE CA CERTIFICATE SSCA_cert
SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=Micro Focus/OU=Vertica/C N=Vertica Root CA'
VALID FOR 3650
EXTENSIONS 'nsComment' = 'Self-signed root CA cert'
KEY SSCA_key;
CREATE CERTIFICATE
```

2. Generate a server key and certificate, signed by your CA, setting the **subjectAltName** of the certificate to the DNS server and/or IP address of your Keycloak server:

```
=> CREATE KEY keycloak_key TYPE 'RSA' LENGTH 2048;
CREATE KEY

=> CREATE CERTIFICATE keycloak_cert
SUBJECT '/C=US/ST=Massachussets/L=Cambridge/O=Micro Focus/OU=Vertica/CN=Vertica Server'
SIGNED BY SSCA_cert
EXTENSIONS 'nsComment' = 'Keycloak CA', 'extendedKeyUsage' = 'serverAuth', 'subjectAltName' = 'DNS.1:dnsserver,IP:203.0.113.1'
KEY keycloak_key;
CREATE CERTIFICATE
```

3. Create the file *keycloak_directory* /conf/keyfile.pem with the content from the key column for the generated key:

```
=> SELECT key FROM cryptographic_keys WHERE name = 'keycloak_key';
```

4. Create the file *keycloak_directory* /conf/certfile.pem with the content from the certificate_text column for the generated certificate:

```
=> SELECT certificate_text FROM certificates WHERE name = 'keycloak_cert';
```

5. Append to your system's CA bundle the content from the certificate_text column for the generated CA certificate. The default CA bundle path and format varies between distributions; for details, see SystemCABundlePath :

```
=> SELECT certificate_text FROM certificates WHERE name = 'SSCA_cert';
```

6. Set the SystemCABundlePath configuration parameter:

```
=> ALTER DATABASE DEFAULT SET SystemCABundlePath = 'path/to/ca_bundle';
```

Start Keycloak

1. Enter the following commands for a minimal configuration to create the Keycloak admin and to start Keycloak in start-dev mode:

```
$ KEYCLOAK_ADMIN=kcadmin
$ export KEYCLOAK_ADMIN
$ KEYCLOAK_ADMIN_PASSWORD=password
$ export KEYCLOAK_ADMIN_PASSWORD
$ cd keycloak_directory/bin/
$ ./kc.sh start-dev --hostname 203.0.113.1 --https-certificate-file ../conf/certfile.pem --https-certificate-key-file=../conf/keyfile.pem
```

2. Open the Keycloak console with your browser (these examples use the default ports):
   - For HTTP: http://203.0.113.1:8080

    - For HTTPS: http://203.0.113.1:8443
3. Sign in as the admin.
4. (Optional) To make testing OAuth more convenient, go to **Realm Settings** > **Tokens** and increase **Access Token Lifespan** to a greater value (the default is 5 minutes).

Create the Vertica client

1. Go to **Clients** and select on **Create** . The **Add Client** page appears.
2. In **Client ID** , enter vertica .
3. Select **Save** . The client configuration page appears.
4. On the **Settings** tab, use the **Access Type** dropdown to select **confidential** .
5. On the **Credentials** tab, copy the **Secret** . This is the client secret used to refresh the token when it expires.

Create a Keycloak user

Keycloak users map to Vertica users with the same name. This example creates a the Keycloak user oauth_user .

1. On the **Users** tab, select **Add user** . The **Add user** page appears.
2. In **Username** , enter oauth_user .
3. On the **Credentials** tab, enter a password.

Configure Okta

The following procedure configures Okta for integration with Vertica and requires administrator privileges. For details, see Configuring OAuth authentication .

The goals of this procedure are to configure Okta and obtain the following information:

- Client ID: The ID used to identify the Vertica database. This is generated by Okta.
- Client secret: An Okta-generated string used to refresh the OAuth token when it expires.
- Token endpoint: Used by the client to retrieve the OAuth token.
- Introspection endpoint: Used by Vertica to validate the OAuth token.

These values are used to create the oauth authentication record in Vertica and act as instructions for Vertica to communicate with Okta when a user attempts to authenticate.

Create an OIDC application

1. From the Okta dashboard, go to **Applications** > **Applications** and select **Create App Integration** . The **Create a new app integration** dialog box appears.
2. For the **Sign-in method** , select **OIDC - OpenID Connect** . The **Application type** section appears.
3. For the **Application type** , select **Native Application** .
4. Select **Next** . The **New Native App Integration** window opens.
5. In the **App integration name** , enter a name for your application. This example uses **Demo_Vertica** .
6. For the **Grant Type** , select **Authorization Code** , **Refresh Token** , and **Resource Owner Password** .
7. For **Controlled access** , select the option applicable to your organization. This example uses **Allow everyone in your organization to access** .
8. Select **Save** to save your application.

Retrieve the client ID and client secret

1. From the Okta dashboard, go to **Applications** > **Applications** and select the name of your OIDC application.
2. In the **General** tab in the **Client Credentials** section, select **Edit** .
3. For **Client authentication** , select **Client secret** and select **Save** . This generates a new client secret.
4. Copy the client ID and client secret.

Set the authentication policy

1. From the Okta dashboard, go to **Applications** > **Applications** and select the name of your OIDC application.
2. In the **Sign On** tab in the **User authentication** section, select **Edit** .
3. Select **Password only** .
4. Select **Save** to save the new policy.

Retrieve Okta endpoints

1. From the Okta dashboard, go to **Security** > **API** .
2. In the **Authorization Servers** tab, select the name of your authorization server. By default, the name of this server is **default** .
3. Select the **Metadata URI** to get a list of all endpoints for your authorization server as a json string.
4. Copy the values for the **token_endpoint** and **introspection_endpoint** .

Test the configuration

1. Verify that you can retrieve an OAuth token from the token endpoint. If successful, Okta respond with an access_token and refresh_token :

```
$ curl --insecure -d "client_id=client_id" -d "client_secret=client_secret" -d "username=okta_username" -d "password=okta_password" -d "grant_type=pas
```

2. Verify that the tokens are valid with the introspection endpoint. If successful, Okta responds with a json string containing "active":true :
   To verify the access token:

```
$ curl --insecure -d "client_id=client_id" -d "client_secret=client_secret" -d "token=access_token" introspection_endpoint
```

   Similarly, to verify the refresh token:

```
$ curl --insecure -d "client_id=client_id" -d "client_secret=client_secret" -d "token=refresh_token" introspection_endpoint
```

   The access_token and refresh_token can then be used for the oauthaccesstoken and oauthrefreshtoken parameters. For details, see JDBC connection properties and ODBC DSN connection properties .

Just-in-time user provisioning

Just-in-time (JIT) user provisioning is the act of automatically configuring an authenticated user and their roles based on information provided by the identity provider.

When a client uses an OAuth authentication record that enables JIT user provisioning, Vertica automatically performs the following actions:

1. Creates the user if they do not already exist in the database. The length of the username in the identity provider cannot be greater than 128 characters.
2. (Keycloak only) Grants to the user and sets as default the roles associated with the user (as specified by the identity provider), provided the roles already exist in Vertica.
3. Grants to the user the authentication record used to authenticate them if neither their user nor role has a grant on that record.

For example, if a client presents an OAuth token to authenticate as user Alice with role director , and Alice does not exist in Vertica, Vertica automatically creates the user Alice , grants to her the authentication record, and grants to her the director role as a default role .

To view users created by JIT user provisioning and which authentication record they use, query the USERS system table:

```
=> SELECT user_name, managed_by_oauth2_auth_id FROM users;
 user_name |  managed_by_oauth2_auth_id
-----------+-----------------------------
 dbadmin   |
 Bob       | 45035996273853300
 Margie    |
 Alice     | 45035996273866484
(4 rows)
```

For details on using JIT user provisioning with OAuth authentication, see Configuring OAuth authentication .

Enabling just-in-time user provisioning

JIT user provisioning is enabled at the authentication record level by setting the oauth2_jit_enabled parameter :

```
=> CREATE AUTHENTICATION v_oauth METHOD 'oauth' HOST '0.0.0.0/0';
=> ALTER AUTHENTICATION v_oauth SET oauth2_jit_enabled = 'yes';

=> SELECT auth_name, is_oauth2_jit_enabled FROM client_auth WHERE auth_name='v_oauth';
  auth_name  |  is_oauth_2_jit_enabled
-------------+--------------------------
 v_oauth     | True
(1 row)
```

Automatic role assignment

The roles automatically assigned to JIT-provisioned active users are based on the information provided by the identity provider (either from the endpoints introspect_url or userinfo_url) and are identified by the client/application specified by the OAuth2JITClient configuration parameter (vertica by default):

```
=> ALTER DATABASE DEFAULT SET OAuth2JITClient = "vertica";
```

With OAuth2JITClient set to vertica , roles that both exist in Vertica and are listed in resource_access.vertica.roles are automatically granted to and set as default roles for JIT-provisioned users.

The identity provider shares user roles through the introspect_url or userinfo_url endpoints. Vertica first sends a request to the introspect_url . If no roles are found in the response, Vertica sends a request to userinfo_url .

For example, if a client attempts to authenticate as the user bob and no user with the same name exists in the database, Vertica sends the following request to the identity provider's introspect_url to retrieve the roles given to bob by the identity provider (this example is truncated):

```
{
   ...
   "resource_access": {
      "vertica": {
         "roles": [
            "customer-facing",
            "order-management",
            "idp-exclusive-role"
         ]
      },
      "account": {
         "roles": [
            "manage-account",
            "manage-account-links",
            "view-profile"
         ]
      }
   },
   "scope": "email profile roles",
   "sid": "dcdd14b1-fe47-491e-b62b-10d1e05c6ffe",
   "client_id": "vertica",
   "username": "bob",
   "active": true
}
```

Because the active field is true and bob has no corresponding user in the Vertica server, Vertica automatically creates the user bob and grants to him the roles that exist in Vertica and are listed in resource_access.vertica.roles : customer-facing and order-management . The role idp-exclusive-role does not exist in Vertica, so it is ignored.

Automatic user pruning

You can enable automatic user pruning to periodically drop users created by JIT user provisioning if they do not log in after a certain period of time. This cleanup service is managed by the following database-level configuration parameters :

- EnableOAuthJITCleanup: Whether to enable cleanup (disabled by default).

```
=> ALTER DATABASE DEFAULT SET EnableOAuthJITCleanup = 1; --enables the pruning service
=> ALTER DATABASE DEFAULT SET EnableOAuthJITCleanup = 0; --disables the pruning service
```

- OAuth2UserExpiredInterval: The number of days a user must be inactive before it is dropped (14 by default). This is calculated based on the current date and the LAST_LOGIN_TIME in the USERS system table.

> **Note**
> The LAST_LOGIN_TIME as recorded by the USERS system table is not persistent; if the database is restarted, the LAST_LOGIN_TIME for users created by just-in-time user provisioning is set to the database start time (this appears as an empty value in LAST_LOGIN_TIME).
>
> You can view the database start time by querying the DATABASES system table:
>
> ```
> => SELECT database_name, start_time FROM databases;
> database_name |          start_time
> ---------------+-------------------------------
> VMart         | 2023-02-06 14:26:50.630054-05
> (1 row)
> ```

```
=> ALTER DATABASE DEFAULT SET OAuth2UserExpiredInterval = 20;
```

- GlobalHeirUsername: The user to reassign objects to if the owner is a JIT-provisioned (or LDAP) user that got dropped by the pruning service. If set to <auto> , objects are reassigned to the dbadmin .

```
=> ALTER DATABASE DEFAULT SET GlobalHeirUsername = <auto>
```

The cleanup service runs daily and there can be a delay of up to 24 hours for dropping an expired user.

OAuth authentication parameters

Vertica OAuth authentication records use the following parameters to determine how to validate client OAuth tokens and how to contact the identity provider during the validation process. These parameters should be set with ALTER AUTHENTICATION .

Just-in-time provisioning parameters

The optional oauth2_jit_enabled parameter specifies whether to enable just-in-time user provisioning . If set to 'yes' , when the user authenticates, Vertica automatically performs the following actions:

1. Creates the user if they do not already exist in the database. The length of the username in the identity provider cannot be greater than 128 characters.
2. (Keycloak only) Grants to the user and sets as default the roles associated with the user (as specified by the identity provider), provided the roles already exist in Vertica.
3. Grants to the user the authentication record used to authenticate them if neither their user nor role has a grant on that record.

If set to 'no' (default), users must be manually created and granted an oauth authentication record to authenticate to Vertica with OAuth tokens.

Validation modes

OAuth authentication records have two modes for validating OAuth tokens, each specified with the authentication parameter validate_type .

The validate_type parameter takes one of the following values:

- IDP (default): Validate OAuth tokens by contacting the identity provider. This validation type requires the client to specify their client secret. This should be used with confidential clients (set for each client by the identity provider).
- JWT : Validate OAuth tokens by verifying that it was signed by the identity provider's private key. While Vertica does not contact the identity provider for JWT validation, the client does for token refresh if either the discovery or token endpoints are set in oauthjsonconfig. This should be used for public clients (set for each client by the identity provider). Additionally, clients can connect if they leave oauthjsonconfig or OAuthJsonConfig empty.

Each validation mode uses a different set of parameters, which are detailed in the tables below.

IDP validation parameters

Parameter name	Description	Required/Optional
client_id	The ID of the confidential client application registered in the identity provider. Vertica uses this ID to call the introspection API to retrieve user grants.	Required
client_secret	The secret of the confidential client application registered in the identity provider. This value is not shared with other clients.	Required
discovery_url (Keycloak only)	Also known as the OpenID Provider Configuration Document or the well-known configuration endpoint, this endpoint contains information about the configuration and endpoints of the identity provider. If you specify the discovery_url and not the introspect_url , Vertica automatically retrieves the introspect_url from the identity provider. If you specify both the discovery_url and introspect_url , the discovery_url takes precedence.	Required for IDP validation if introspect_url is not specified.
introspect_url	Used by Vertica to introspect (validate) access tokens. You must specify this parameter if you do not specify the discovery_url . For examples, see the Keycloak and Okta documentation.	Required if discovery_url is not specified.

JWT validation parameters

The following table lists the parameters used to configure OAuth authentication records that use the JWT validation mode:

Parameter name	Description	Required/Optional
jwt_rsa_public_key	In PEM format, the public key that corresponds to the private key used to sign the client's OAuth token. Vertica uses this to validate the OAuth token. If your identity provider does not natively provide PEM-formatted public keys, you must convert them to PEM format. For example, keys retrieved from an Okta endpoint are in JWK format and must be converted.	Required
jwt_issuer	The issuer of the OAuth token. This value is set by the identify provider.	Required
jwt_user_mapping	The name of the Vertica user.	Required
jwt_accepted_audience_list	A comma-delimited list of values to accept from the client OAuth token's aud field. If set, tokens must include in aud one of the accepted audiences to authenticate.	Optional
jwt_accepted_scope_list	A comma-delimited list of values to accept from the client OAuth token's scope field. If set, tokens must include in scope at least one of the accepted scopes to authenticate.	Optional

TLS authentication

The tls authentication method authenticates users that can establish a mutual mode client-server TLS connection using a certificate that specifies a valid database username in the Common Name (CN) field.

Before you create and use a tls authentication method, you must configure Vertica for client-server TLS in mutual mode (disabled by default).

In this section

- Client authentication with TLS

Client authentication with TLS

Database users or roles granted a tls authentication record can authenticate to Vertica with a TLS certificate.

Prerequisites

You must configure Vertica for mutual mode client-server TLS .

In mutual mode, the client and server must verify each other's identity before connecting. This mode allows Vertica to verify the identity of the client and allow them to authenticate the client through their certificate.

Configuring TLS authentication

The following sections generate a private key and certificate for the client. For simplicity, the example signs the client certificate with the following self-signed CA certificate (which has also, in the context of the example, signed the Vertica database's server certificate):

```
=> CREATE KEY SSCA_key TYPE 'RSA' LENGTH 2048;
=> CREATE CA CERTIFICATE ca_certificate
SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=OpenText/OU=Vertica/CN=Vertica Root CA'
VALID FOR 3650
EXTENSIONS 'nsComment' = 'Self-signed root CA cert'
KEY SSCA_key;
```

In a production environment, you should instead use a CA certificate from a trusted certificate authority.

Create client keys

The following steps generate a client key and certificate, and then make them available to the client:

1. Generate the client key:

   ```
   => CREATE KEY client_private_key TYPE 'RSA' LENGTH 2048;
   CREATE KEY
   ```

2. Generate the client certificate. Mutual TLS requires that the Common Name (CN) in the SUBJECT specifies a database username:

```
=> CREATE CERTIFICATE client_certificate
SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=OpenText/OU=Vertica/CN=dbadmin/emailAddress=example@example.com'
SIGNED BY ca_certificate
EXTENSIONS 'nsComment' = 'Vertica client cert', 'extendedKeyUsage' = 'clientAuth'
KEY client_private_key;
CREATE CERTIFICATE
```

3. On the client machine, export the client key and client certificate to the client filesystem. The following commands use the vsql client :

```
$ vsql -At -c "SELECT key FROM cryptographic_keys WHERE name = 'client_private_key';" -o client_private_key.key
$ vsql -At -c "SELECT certificate_text FROM certificates WHERE name = 'client_certificate';" -o client_cert.pem
```

   In the preceding command:
   - -A : enables unaligned output.
   - -t : prevents the command from outputting metadata, such as column names.
   - -c : instructs the shell to run one command and then exit.
   - -o : writes the query output to the specified filename.

   For details about all vsql command line options, see Command-line options
4. Copy or move the client key and certificate to a location that your client recognizes.
   The following commands move the client key and certificate to the hidden directory ~/.client-creds , and then grants the file owner read and write permissions with chmod :

```
$ mkdir ~/.client-creds
$ mv client_private_key.key ~/.client-creds/client_key.key
$ mv client_cert.pem ~/.client-creds/client_cert.pem
$ chmod 600 ~/.client-creds/client_key.key ~/.client-creds/client_cert.pem
```

Create an authentication record

Next, you must create an authentication record in the database. An authentication record defines a set of authentication and the access methods for the database. You grant this record to a user or role to control how they authenticate to the database:

1. Create the authentication record . The tls method requires that clients authenticate with a certificate whose Common Name (CN) specifies a database username:

```
=> CREATE AUTHENTICATION auth_record METHOD 'tls' HOST TLS '0.0.0.0/0';
CREATE AUTHENTICATION
```

2. Grant the authentication record to a user or to a role. The following example grants the authentication record to PUBLIC , the default role for all users:

```
=> GRANT AUTHENTICATION auth_record TO PUBLIC;
GRANT AUTHENTICATION
```

Reject plaintext connections

You can create an authentication record that rejects remote connections from a specified IP range.

For example, to reject all plaintext client connections, specify the reject authentication method and the HOST NO TLS access method as follows:

```
=> CREATE AUTHENTICATION RejectNoSSL METHOD 'reject' HOST NO TLS '0.0.0.0/0';  --IPv4
=> CREATE AUTHENTICATION RejectNoSSL METHOD 'reject' HOST NO TLS '::/0';       --IPv6
```

Internode TLS

Internode TLS secures communication between nodes within a cluster. It is important to secure communications between nodes if you do not trust the network between the nodes.

Before setting up internode TLS, check the current status of your configuration with SECURITY_CONFIG_CHECK .

```
=> SELECT SECURITY_CONFIG_CHECK('NETWORK');
```

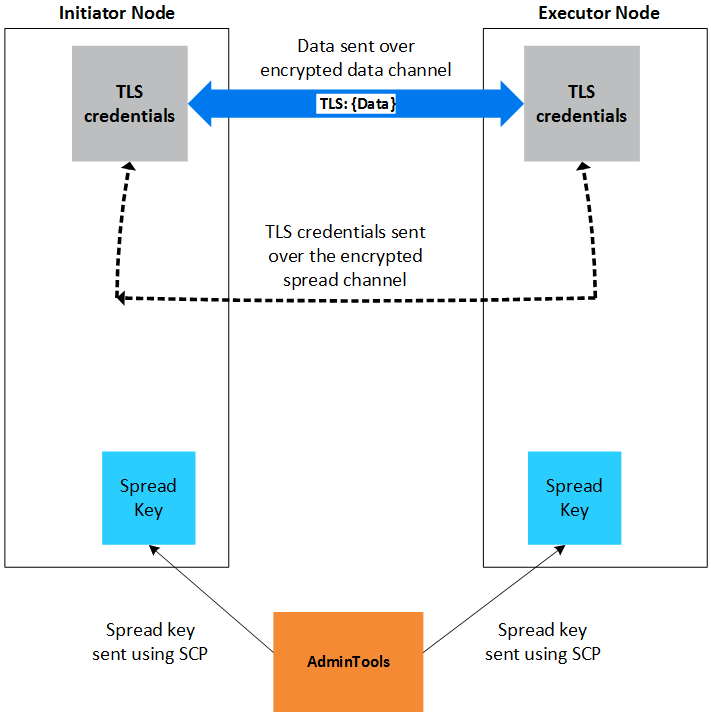
Communication between the server nodes uses two channels: the control channel and data channel. To enable internode encryption, set the EncryptSpreadComm parameter (disabled by default) to encrypt Spread communication on the control channel and configure the data_channel TLS Configuration to encrypt the data channel:

1. Encrypt Spread communication on the control channel with EncryptSpreadComm . See Control channel spread TLS for details.
2. Encrypt the data channel with the data_channel TLS Configuration. See Data channel TLS for details.

If you enable internode encryption, some of your queries might run slower than expected. Performance depends on the data sent and network quality.

Admintools generates or retrieves the spread key to encrypt all traffic on the control channel and ships the spread key to all nodes. Vertica uses TLS to encrypt all traffic on the data channel. TLS credentials are shared between nodes over the encrypted control channel.

The following graphic illustrates the internode encryption process.



See also

- Control channel spread TLS
- Data channel TLS
- TLS overview

In this section

- Control channel spread TLS
- Data channel TLS

Control channel spread TLS

The control channel allows nodes to exchange plan information with one another, and to distribute calls among nodes. Enabling spread security secures this communication. See Internode TLS for more information.

Internode TLS uses the following channels. Both must be enabled, and in the following order, before setting other parameters:

1. Control Channel to exchange plan information and distribute calls. It is implemented using Spread . For more information, visit spread.org .
2. Data Channel to exchange table data. It is implemented using TCP.

Enabling EncryptSpreadComm

EncryptSpreadComm can be set with one of two values:

- *vertica* . Vertica generates the spread encryption key for the cluster when the database starts up.
- aws-kms| *<key_name>* . Vertica fetches the user-specified key from the AWS Key Management Service when the database starts up, rather than generating one itself.

In general, you should set the EncryptSpreadComm parameter to enable spread encryption before setting any other security parameters.

1. Set the EncryptSpreadComm parameter with ALTER DATABASE .
```
=> ALTER DATABASE DEFAULT SET PARAMETER EncryptSpreadComm = 'vertica';
```
2. Restart the database.
3. Verify your settings with SECURITY_CONFIG_CHECK .
```
=> SELECT SECURITY_CONFIG_CHECK('NETWORK');
```

Privileges

Superuser

Restrictions

After setting this parameter, you must restart your database.

Example

This enables the EncryptSpreadComm parameter and tells Vertica to generate a spread encryption key the next time the database starts up.

```
=> ALTER DATABASE DEFAULT SET PARAMETER EncryptSpreadComm = 'vertica';
```

For more information on this and other security parameters, see Security parameters .

See also

- Internode TLS
- Data channel TLS
- TLS overview

Data channel TLS

Nodes use the data channel to exchange table data during operations such as queries.

Internode communication uses the following channels. Their associated components and parameters must be enabled in the following order:

1. Control Channel to exchange plan information and distribute calls. It is implemented using Spread . For more information, visit spread.org .
2. Data Channel to exchange table data. It is implemented using TCP.

Configuring data channel TLS

This procedure configures TLS between Vertica nodes and uses the predefined TLS Configuration data_channel . To use a custom TLS Configuration, see TLS configurations .

1. Enable TLS on the control channel .
2. Generate or import a CA (Certificate Authority) certificate. For example, to create a self-signed CA certificate, generate a key and sign CA certificate with the key:
```
=> CREATE KEY SSCA_key TYPE 'RSA' LENGTH 2048;
```
```
=> CREATE CA CERTIFICATE SSCA_cert
SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=OpenText/OU=Vertica/CN=Vertica Root CA'
VALID FOR 3650
EXTENSIONS 'nsComment' = 'Self-signed root CA cert'
KEY SSCA_key;
```
3. Generate or import a private key. For example, to generate the private key:
```
=> CREATE KEY internode_key TYPE 'RSA' LENGTH 2048;
```
4. Generate or import a TLS certificate. The certificate must have a full chain that ends in a CA, and must be either a x509v1 certificate or use the extendedKeyUsage extensions serverAuth and clientAuth . For example, to generate internode_cert and sign it with SSCA_cert :
```
=> CREATE CERTIFICATE internode_cert
SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=Micro Focus/OU=Vertica/CN=data channel'
SIGNED BY SSCA_cert
EXTENSIONS 'nsComment' = 'Vertica internode cert', 'extendedKeyUsage' = 'serverAuth, clientAuth'
KEY internode_key;
```

5. Set the certificate, and optionally the TLSMODE for data_channel TLS configuration. If the TLSMODE is set to TRY_VERIFY or higher, the certificate's signing CA is added to the TLS Configuration's list of CA certificates :

```
=> ALTER TLS CONFIGURATION data_channel CERTIFICATE internode_cert TLSMODE 'TRY_VERIFY'
```

   If you do not specify a TLSMODE, and the TLSMODE was previously set to DISABLE (default), TRY_VERIFY , VERIFY_CA , or VERIFY_FULL (which behaves like VERIFY_CA), the TLSMODE automatically changes to VERIFY_CA :

```
=> ALTER TLS CONFIGURATION data_channel CERTIFICATE internode_cert;
```

   If the certificate is not signed by a known CA, the TLSMODE is set to DISABLE .
6. Verify that the InternodeTLSConfig parameter uses the TLS Configuration:

```
=> SHOW CURRENT InternodeTLSConfig;
 level |       name        |   setting
---------+-------------------+-------------
 DEFAULT | InternodeTLSConfig | data_channel
(1 row)
```

7. Verify that data channel encryption is enabled with SECURITY_CONFIG_CHECK('NETWORK') :

```
=> SELECT SECURITY_CONFIG_CHECK('NETWORK');
  SECURITY_CONFIG_CHECK
---------------------------
 Spread security details:
* EncryptSpreadComm = [vertica]
Spread encryption is enabled
It is now safe to set/change other security knobs

Data Channel security details:
 TLS Configuration 'data_channel' TLSMODE is VERIFY_CA
TLS on the data channel is enabled
```

Privileges

Superuser

Restrictions

- In general, you should set EncryptSpreadComm before configuring data channel TLS.
- Changes to the InternodeTLSConfig parameter and its underlying TLS Configuration take effect immediately and interrupt all ongoing queries in order to update node connections.

See also

- Internode TLS
- Control channel spread TLS
- TLS overview

TLS protocol

TLS (Transport Layer Security) is a cryptographic protocol used to secure communications between servers, their nodes, and clients.

When enabled, a Vertica database and the clients that connect to it use TLS 1.2.

Although TLS, SSL, and TLS/SSL are often used interchangeably, the Vertica documentation always uses TLS to reference the protocol. Some Vertica parameters and components use SSL, and in these cases the documentation uses SSL to reference them, but these too can be categorized under the TLS umbrella.

Enabling TLS is a multi-step process. First, check the status of your security configuration with SECURITY_CONFIG_CHECK . Then, you can configure TLS authentication records to reject non-TLS connections.

In this section

- TLS overview

TLS overview

To secure communications and verify data integrity, you can configure Vertica and database clients to use TLS. The TLS protocol uses a key and certificate exchange system along with a trusted third party called a Certificate Authority (CA). Both the owner of a certificate and the other party that relies on the certificate must trust the CA to confirm the certificate holder's identity.

Vertica also supports the following authentication methods using the Transport Layer Security (TLS) v1.2 protocol. Both methods encrypt and verify the integrity of the data in transit:

- **Server Mode** - In server mode, the client must confirm the server's identity before connecting. The client verifies that the server's certificate and public key are valid and were issued by a certificate authority (CA) listed in the client's list of trusted CAs. This helps prevent man-in-the-middle attacks.
- **Mutual Mode** - In mutual mode, the client and server must verify each other's identity before connecting.

In addition to the requirements detailed in this section, you must create TLS authentication records to reject non-TLS client connections.

TLS handshake process

The following is a high-level/simplified overview of one possible "handshake" process for the client to verify the identity of the server in **Server Mode** . Additional actions taken in **Mutual Mode** for the server to identify the client are marked as such.

Public and Private Key Pairs - Key pairs are generated by clients and servers. The owner of a public key must be verified by a certificate authority. The key pairs are used to encrypt messages. For example, suppose Alice wants to send confidential data to Bob. Because she wants only Bob to read it, she encrypts the data with Bob's public key. Even if someone else gains access to the encrypted data, it remains protected. Because only Bob has access to his corresponding private key, he is the only person who can decrypt Alice's encrypted data back into its original form.

Certificates - Certificates contain a public key and identify the owner of the key. They are issued by the certificate authority (CA).

Certificate Authority (CA) - A certificate authority is a trusted party that verifies the identity of public key owners.

Client and Server Random - Client Random and Server Random are random strings that used to created a shared secret which encrypts communication if the handshake succeeds.

1. Before connecting, the server and client generate their own public and private key pairs. The CA then distributes identifying certificates to the server and client for their respective public keys.
2. The client sends its Client Random to the server and requests the server's certificate.
3. The server sends its certificate and its Server Random, encrypted with its private key, to the client. In **Mutual Mode** , the server also requests the client's certificate.
4. In **Mutual Mode** , the client sends its certificate.
5. The client uses the certificate to verify that the server owns its public key, then decrypts the Server Random with the server's public key to verify that the server owns its private key.
6. In **Mutual Mode** , the server uses the certificate to verify that the client owns its public key.
7. The server and client use the Client and Server Randoms to generate a new secret, called a session key, which encrypts future communication.

In this section

- TLS configurations
- Generating TLS certificates and keys
- Configuring client-server TLS
- Managing CA bundles
- Generating certificates and keys for MC
- Importing a new certificate to MC
- Replacing the agent certificate
- Importing and exporting data with TLS

TLS configurations

A TLS Configuration is a database object that encapsulates all settings and certificates needed to configure TLS. After setting up a TLS Configuration, you can use it by setting it as the value for one or more of the following database parameters, each of which controls TLS for a certain type of connection between the Vertica database and a client or server:

- ServerTLSConfig
- LDAPLinkTLSConfig
- LDAPAuthTLSConfig
- InternodeTLSConfig

These parameters are set to predefined TLS Configurations by default so if you just want to configure TLS, you should use ALTER TLS CONFIGURATION to modify a predefined TLS Configuration . Otherwise, you can use CREATE TLS CONFIGURATION to create a custom TLS Configuration .

Reusing an existing TLS configurations

To reuse an existing TLS Configuration , use ALTER TLS CONFIGURATION .

The following table lists each TLS connection type parameter with its associated connection type and predefined TLS Configuration:

Note
For OAuth, Vertica is the client and the identity provider is the server. TLS for this connection type is not controlled by a TLS Configuration. For details, see Configure Keycloak .

Connection Type	Parameter	Default TLS Configuration	Example
Client-server where Vertica is the server	ServerTLSConfig	server	Configuring client-server TLS
Connections for the LDAP Link service	LDAPLinkTLSConfig	LDAPLink	TLS for LDAP link
Connections between Vertica and an LDAP server to authenticate users	LDAPAuthTLSConfig	LDAPAuth	TLS for LDAP authentication
Connections between Vertica nodes	InternodeTLSConfig	data_channel	Internode TLS

Creating custom TLS configurations

You can create TLS Configurations with CREATE TLS CONFIGURATION .

The following example creates a TLS Configuration and enables it for client-server TLS by setting it in ServerTLSConfig:

1. Create the keys and certificates:

```
-- create CA certificate
=> CREATE KEY k_ca TYPE 'RSA' LENGTH 4096;
=> CREATE CA CERTIFICATE ca
  SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=Micro Focus/OU=Vertica/CN=Vertica Root CA'
  VALID FOR 3650
  EXTENSIONS 'nsComment' = 'Vertica generated root CA cert'
  KEY k_ca;

-- create server certificate
=> CREATE KEY k_server TYPE 'RSA' LENGTH 2048;
=> CREATE CERTIFICATE server
   SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=Micro Focus/OU=Vertica/CN=Vertica Cluster/emailAddress=example@example.com'
   SIGNED BY ca
   KEY k_server;
```

2. Create the TLS Configuration with the server's certificate:

```
=> CREATE TLS CONFIGURATION new_tls_config CERTIFICATE server TLSMODE 'ENABLE';
```

3. Set the ServerTLSConfig parameter to use the new TLS Configuration for client-server TLS:

```
=> ALTER DATABASE DEFAULT SET ServerTLSConfig = 'new_tls_config';
```

Generating TLS certificates and keys

This page includes examples and sample procedures for generating certificates and keys with CREATE KEY and CREATE CERTIFICATE . To view your keys and certificates, query the CRYPTOGRAPHIC_KEYS and CERTIFICATES system tables.

For more detailed information on creating signed certificates, OpenSSL recommends the OpenSSL Cookbook .

For more information on x509 extensions, see the OpenSSL documentation .

Importing keys and certificates

Keys

You only need to import private keys if you intend to use its associated certificate to sign something, like a message in client-server TLS, or another certificate. That is, you only only need to import keys if its associated certificate is one of the following:

- Client/server certificate

- CA certificate used to sign other certificates while in Vertica

If you only need your CA certificate to validate other certificates, you do not need to import its private key.

To import a private key:

```
=> CREATE KEY imported_key TYPE 'RSA' AS '-----BEGIN PRIVATE KEY-----...-----END PRIVATE KEY-----';
```

Certificates

To import a CA certificate that only validates other certificates (no private key):

```
=> CREATE CA CERTIFICATE imported_validating_ca AS '-----BEGIN CERTIFICATE-----...-----END CERTIFICATE-----';
```

To import a CA that can both validate and sign other certificates (private key required):

```
=> CREATE CA CERTIFICATE imported_signing_ca AS '-----BEGIN CERTIFICATE-----...-----END CERTIFICATE-----'
KEY ca_key;
```

To import a certificate for server mode TLS:

```
=> CREATE CERTIFICATE server_mode_cert AS '-----BEGIN CERTIFICATE-----...-----END CERTIFICATE-----' KEY imported_key;
```

To import a certificate for mutual mode TLS or client authentication, you must specify its CA:

```
=> CREATE CERTIFICATE imported_cert AS '-----BEGIN CERTIFICATE-----...-----END CERTIFICATE-----'
SIGNED BY imported_ca KEY imported_key;
```

Generating private keys and certificates

Keys

To generate an 2048-bit RSA private key:

```
=> CREATE KEY new_key TYPE 'RSA' LENGTH 2048;
```

Self-signed CA certificates

Important

A self-signed CA certificate is convenient for development purposes, but you should always use a proper certificate authority in a production environment.

A CA is a trusted entity that signs and validates other certificates with its own certificate. The following example generates a self-signed root CA certificate:

1. Generate or import a private key. The following command generates a new private key:

   ```
   => CREATE KEY ca_private_key TYPE 'RSA' LENGTH 4096;
   CREATE KEY
   ```

2. Generate the certificate with the following format. Sign the certificate the with the private key that you generated or imported in the previous step:

   ```
   => CREATE CA CERTIFICATE ca_certificate
   SUBJECT '/C=country_code/ST=state_or_province/L=locality/O=organization/OU=org_unit/CN=Vertica Root CA'
   VALID FOR days_valid
   EXTENSIONS 'authorityKeyIdentifier' = 'keyid:always,issuer', 'nsComment' = 'Vertica generated root CA cert'
   KEY ca_private_key;
   ```

   Note

   The CA certificate SUBJECT must be different from the SUBJECT of any certificate that it signs.

   For example:

```
=> CREATE CA CERTIFICATE SSCA_cert
SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=OpenText/OU=Vertica/CN=Vertica Root CA'
VALID FOR 3650
EXTENSIONS 'nsComment' = 'Self-signed root CA cert'
KEY SSCA_key;
```

Intermediate CA certificates

In addition to server certificates, CAs can also sign the certificates of other CAs. This process produces an intermediate CA and a chain of trust between the top-level CA and the intermediate CA. These intermediate CAs can then sign other certificates.

> **Note**
> Intermediate CA certificates generated with CREATE CERTIFICATE cannot sign other CA certificates.

1. Generate or import the CA that signs the intermediate CA. The example that follows generates and uses a self-signed root CA:

   ```
   => CREATE KEY SSCA_key TYPE 'RSA' LENGTH 2048;
   ```

   ```
   => CREATE CA CERTIFICATE SSCA_cert
   SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=OpenText/OU=Vertica/CN=Vertica Root CA'
   VALID FOR 3650
   EXTENSIONS 'nsComment' = 'Self-signed root CA cert'
   KEY SSCA_key;
   ```

2. Generate or import a private key:

   ```
   => CREATE KEY intermediate_key TYPE 'RSA' LENGTH 2048;
   ```

3. Generate the intermediate CA certificate, specifying its private key and signing CA using the following format:

   ```
   => CREATE CERTIFICATE intermediate_certificate_name
   SUBJECT '/C=country_code/ST=state_or_province/L=locality/O=organization/OU=org_unit/CN=Vertica intermediate CA'
   SIGNED BY ca_name
   KEY intermediate_key;
   ```

   For example:

   ```
   => CREATE CA CERTIFICATE intermediate_CA
   SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=OpenText/OU=Vertica/CN=Vertica Intermediate CA'
   SIGNED BY SSCA_cert
   KEY intermediate_key;
   ```

Client/server certificates

CREATE CERTIFICATE generates x509v3 certificates, which allow you to specify extensions to restrict how the certificate can be used. The value for the extendedKeyUsage extension will differ based on your use case:

- Server certificate:

  ```
  'extendedKeyUsage' = 'serverAuth'
  ```

- Client certificate:

  ```
  'extendedKeyUsage' = 'clientAuth'
  ```

- Server certificate for internode encryption :

  ```
  'extendedKeyUsage' = 'serverAuth, clientAuth'
  ```

Because these certificates are used for client/server TLS, you must import or generate their private keys.

The following example certificates are all signed by this self-signed CA certificate:

```
=> CREATE KEY SSCA_key TYPE 'RSA' LENGTH 2048;
```

```
=> CREATE CA CERTIFICATE SSCA_cert
SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=OpenText/OU=Vertica/CN=Vertica Root CA'
VALID FOR 3650
EXTENSIONS 'nsComment' = 'Self-signed root CA cert'
KEY SSCA_key;
```

To generate a server certificate:

```
=> CREATE KEY server_key TYPE 'RSA' LENGTH 2048;
```

```
=> CREATE CERTIFICATE server_cert
SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=OpenText/OU=Vertica/CN=Vertica server/emailAddress=example@example.com'
SIGNED BY SSCA_cert
EXTENSIONS 'nsComment' = 'Vertica server cert', 'extendedKeyUsage' = 'serverAuth'
KEY server_key;
```

To generate a client certificate:

```
=> CREATE KEY client_key TYPE 'RSA' LENGTH 2048;
```

```
=> CREATE CERTIFICATE client_cert
SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=OpenText/OU=Vertica/CN=Vertica client/emailAddress=clientexample@example.com'
SIGNED BY SSCA_cert
EXTENSIONS 'nsComment' = 'Vertica client cert', 'extendedKeyUsage' = 'clientAuth'
KEY client_key;
```

To generate an internode TLS certificate:

```
=> CREATE KEY internode_key TYPE 'RSA' LENGTH 2048;
```

```
=> CREATE CERTIFICATE internode_cert
SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=Micro Focus/OU=Vertica/CN=data channel'
SIGNED BY SSCA_cert
EXTENSIONS 'nsComment' = 'Vertica internode cert', 'extendedKeyUsage' = 'serverAuth, clientAuth'
KEY internode_key;
```

Configuring client-server TLS

Vertica offers two connection modes for client-server TLS:

- In **Server Mode** , the client must verify the host's certificate. Hosts must have a server private key and certificate.
- In **Mutual Mode** , the client and host must each verify the other's certificate. Hosts must have a server private key, server certificate, and CA certificate(s).

Client-server TLS secures the connection step between Vertica and clients, not the following authentication step to authenticate these clients as users in the database. To configure authentication for TLS connections or to reject plaintext connections, see TLS authentication .

Setting certificates with TLS configuration

This procedure creates keys and certificates for client-server TLS and sets them in the predefined TLS Configuration server , which is the default TLS configuration for ServerTLSConfig. To create a custom TLS configuration, see TLS configurations .

1. Generate or import the following according to your use case:
   - **Server Mode** : server certificate private key, server certificate
   - **Mutual Mode** : server certificate private key, server certificate, CA certificate(s)
2. Run the following commands according to your desired configuration. New connections will use TLS.
   - To use **Server Mode** , set the server certificate for the server's TLS Configuration:
     ```
     => ALTER TLS CONFIGURATION server CERTIFICATE server_cert;
     ```
   - To use **Mutual Mode** , set a server and CA certificate. This CA certificate is used to verify client certificates:
     ```
     => ALTER TLS CONFIGURATION server CERTIFICATE server_cert ADD CA CERTIFICATES ca_cert;
     ```

To use multiple CA certificates, separate them with commas:

```
=> ALTER TLS CONFIGURATION server CERTIFICATE server_cert
   ADD CA CERTIFICATES intermediate_ca_cert, ca_cert;
```

3. Enable TLS (disabled by default). Choose one of the following TLSMODEs, listed in ascending security.
   - DISABLE : Disables TLS. All other options for this parameter enable TLS.
   - ENABLE : Enables TLS. Vertica does not verify client certificates.
   - TRY_VERIFY : Establishes a TLS connection if one of the following is true:
     - The client presents a valid certificate.
     - The client doesn't present a certificate

     If the client presents an invalid certificate, the connection is rejected.
   - VERIFY_CA : Connection succeeds if Vertica verifies that the client certificate is from a trusted CA. If the client does not present a client certificate, the connection is rejected.

   TLS Configurations also support the TLSMODE VERIFY_FULL , but this TLSMODE is unsupported for client-server TLS (the connection type handled by ServerTLSConfig) and behaves like VERIFY_CA .

   For **Server Mode** , choose ENABLE :

   ```
   => ALTER TLS CONFIGURATION server TLSMODE 'ENABLE';
   ```

   For **Mutual Mode** , choose TRY_VERIFY or higher:

   ```
   => ALTER TLS CONFIGURATION server TLSMODE 'VERIFY_CA';
   ```

4. Verify that the ServerTLSConfig parameter is set to the server TLS Configuration:

   ```
   => SHOW CURRENT ServerTLSConfig;
    level |      name       | setting
   ---------+------------------+---------
    DEFAULT | ServerTLSConfig  | server
   (1 row)
   ```

   If not, set the ServerTLSConfig parameter:

   ```
   => ALTER DATABASE DEFAULT SET ServerTLSConfig = 'server';
   ```

See also

- Internode TLS
- SECURITY_CONFIG_CHECK

Managing CA bundles

Certificate authority (CA) bundles allow you to group CA certificates together and use them to validate connections to your database.

You can view existing CA bundles by querying the CA_BUNDLES system table.

Creating a CA bundle

To create a CA bundle, use CREATE CA BUNDLE and specify one or more CA certificates. If you don't specify a CA certificate, the CA bundle will be empty.

This example creates a CA bundle called ca_bundle that contains CA certificates root_ca and root_ca2:

```
=> CREATE CA BUNDLE ca_bundle CERTIFICATES root_ca, root_ca2;
CREATE CA BUNDLE

=> SELECT * FROM ca_bundles WHERE name='ca_bundle';
       oid       |   name    |       owner       |                 certificates
-----------------+-----------+-------------------+--------------------------------------
 45035996274026954 | ca_bundle | 45035996273704962 | [45035996274026764, 45035996274026766]
(1 row)
```

Modifying existing CA bundles

CA_BUNDLES only stores OIDs. Since operations on CA bundles require certificate and owner names, you can use the following query to map bundles to certificate and owner names:

```
=> SELECT user_name AS owner_name,
      owner     AS owner_oid,
      b.name    AS bundle_name,
      c.name    AS cert_name
FROM   (SELECT name,
               STRING_TO_ARRAY(certificates) :: array[INT] AS certs
        FROM   ca_bundles) b
      LEFT JOIN certificates c
             ON CONTAINS(b.certs, c.oid)
      LEFT JOIN users
             ON user_id = owner
ORDER  BY 1;

 owner_name |     owner_oid     | bundle_name  | cert_name
------------+-------------------+--------------+-----------
 dbadmin    | 45035996273704962 | ca_bundle    | root_ca
 dbadmin    | 45035996273704962 | ca_bundle    | ca_cert
(2 rows)
```

Adding and removing CA certificates

If you have ownership of a CA bundle, you can add and remove certificates with ALTER CA BUNDLE .

This example modifies ca_bundle by adding ca_cert and removing root_ca2:

```
=> ALTER CA BUNDLE ca_bundle ADD CERTIFICATES ca_cert;
ALTER CA BUNDLE

=> SELECT * FROM ca_bundles WHERE name='ca_bundle';
       oid        |   name    |       owner       |                        certificates
------------------+-----------+-------------------+-----------------------------------------------------------
 45035996274027356 | ca_bundle | 45035996273704962 | [45035996274027342, 45035996274027348, 45035996274027396]
(1 row)

=> ALTER CA BUNDLE ca_bundle REMOVE CERTIFICATES root_ca2;
ALTER CA BUNDLE

=> SELECT * FROM CA_BUNDLES;
       oid        |   name    |       owner       |              certificates
------------------+-----------+-------------------+---------------------------------------
 45035996274027356 | ca_bundle | 45035996273704962 | [45035996274027342, 45035996274027396]
(1 row)
```

Managing CA bundle ownership

Superusers and CA bundle owners can see whether a bundle exists by querying the CA_BUNDLES system table, but only owners of a given bundle can see the certificates inside.

In the following example, the dbadmin user owns ca_bundle. After giving ownership of the bundle to 'Alice', the dbadmin can no longer see the certificates inside the bundle:

```
=> => SELECT * FROM ca_bundles WHERE name='ca_bundle';
       oid       |   name    |       owner       |                certificates
-----------------+-----------+-------------------+--------------------------------------
 45035996274027356 | ca_bundle | 45035996273704962 | [45035996274027342, 45035996274027396]
(1 row)

=> ALTER CA BUNDLE ca_bundle OWNER TO Alice;
ALTER CA BUNDLE

=> SELECT * FROM ca_bundles WHERE name='ca_bundle';
       oid       |   name    |       owner       | certificates
-----------------+-----------+-------------------+--------------
 45035996274027356 | ca_bundle | 45035996274027586 | []
(1 row)
```

Dropping CA bundles

You must have ownership of a CA bundle to drop it:

```
=> DROP CA BUNDLE ca_bundle;
DROP CA BUNDLE
```

Generating certificates and keys for MC

A *certificate signing request (CSR)* is a block of encrypted text generated on the server on which the certificate is used. You send the CSR to a certificate authority (CA) to apply for a digital identity certificate. The CA uses the CSR to create your SSL certificate from information in your certificate; for example, organization name, common (domain) name, city, and country.

Management Console (MC) uses a combination of OAuth (Open Authorization), Secure Socket Layer (SSL), and locally-encrypted passwords to secure HTTPS requests between a user's browser and MC, and between MC and the agents . Authentication occurs through MC and between agents within the cluster. Agents also authenticate and authorize jobs.

The MC configuration process sets up SSL automatically, but you must have the openssl package installed on your Linux environment first.

When you connect to MC through a client browser, Vertica assigns each HTTPS request a self-signed certificate, which includes a timestamp. To increase security and protect against password replay attacks, the timestamp is valid for several seconds only, after which it expires.

To avoid being blocked out of MC, synchronize time on the hosts in your Vertica cluster, and on the MC host if it resides on a dedicated server. To recover from loss or lack of synchronization, resync system time and the Network Time Protocol.

Create a certificate and submit it for signing

For production, you must use certificates signed by a certificate authority. You can create and submit a certificate and when the certificate returns from the CA, import the certificate into MC .

Use the openssl command to generate a new CSR, entering the passphrase "password" when prompted:

```
$ sudo openssl req -new -key /opt/vconsole/config/keystore.key -out server.csr
Enter pass phrase for /opt/vconsole/config/keystore.key:
```

When you press **Enter** , you are prompted to enter information to be incorporated into your certificate request. Some fields contain a default value, which you should change for security reasons. Other fields you can leave blank, such as password and optional company name. To leave the field blank, type '.' .

Important

The keystore.key value for the -key option creates private key for the keystore. If you generate a new key and import it using the Management Console interface, the MC process does restart properly. You must restore the original keystore.jks file and restart Management Console .

This information is contained in the CSR and shows both the default and replacement values:

```
Country Name (2 letter code) [GB]:USState or Province Name (full name) [Berkshire]:Massachusetts
Locality Name (eg, city) [Newbury]: Cambridge
Organization Name (eg, company) [My Company Ltd]:Vertica
Organizational Unit Name (eg, section) []:Information Management
Common Name (eg, your name or your server's hostname) []:console.vertica.com
Email Address []:mcadmin@vertica.com
```

The **Common Name** field is the fully qualified domain name of your server. Your entry must exactly match what you type in your web browser, or you receive a name mismatch error.

Self-sign a certificate for testing

To test your new SSL implementation, you can self-sign a CSR using either a temporary certificate or your own internal CA, if one is available.

> **Note**
> A self-signed certificate generates a browser-based error notifying you that the signing certificate authority is unknown and not trusted. For testing purposes, accept the risks and continue.

The following command generates a temporary certificate, which expires after 365 days:

```
$ sudo openssl x509 -req -days 365 -in server.csr -signkey /opt/vconsole/config/keystore.key -out server.crt
Enter passphrase for /opt/vconsole/config/keystore.key:
Enter same passphrase again:
```

The previous example prompts you for a passphrase. This is required for Apache to start. To implement a passphrase you must put the SSLPassPhraseDialog directive in the appropriate Apache configuration file. For more information see your Apache documentation.

This example shows the command's output to the terminal window:

```
Signature oksubject=/C=US/ST=Massachusetts/L=Cambridge/O=Vertica/OU=IT/
CN=console.vertica.com/emailAddress=mcadmin@vertica.com
Getting Private key
```

You can now import the self-signed key , server.crt , into Management Console.

See also

- Configuring TLS for JDBC clients
- Configuring TLS for ODBC Clients
- Key and Certificate Management Tool

Importing a new certificate to MC

Use this procedure to import a new certificate into Management Console.

> **Note**
> To generate a new certificate for Management Console, you must use the keystore.key file, which is located in /opt/vconsole/config on the server on which you installed MC. Any other generated key/certificate pair causes MC to restart incorrectly. You will then have to restore the original keystore.jks file and restart Management Console . See Generating Certifications and Keys for Management Console .

1. Connect to Management Console, and log in as an administrator.
2. On the Home page, click MC **Settings** .
3. In the button panel on the left, click **SSL certificates** .
4. To the right of "Upload a new SSL certificate," click **Browse** to import the new key.
5. Click **Apply** .
6. Restart Management Console .

Replacing the agent certificate

The Agent uses a preinstalled Certificate Authority (CA) certificate. You can replace it copying the your preferred certificate and its private key to the host.

To view your current agent certificate:

```
$ openssl s_client -prexit -connect database_IP:database_port
```

Generating a certificate

If you don't already have one, you can generate a self-signed certificate. For more information, see Generating TLS certificates and keys

1. Generate the private key and certificate.

```
$ openssl req -new -newkey rsa:4096 -x509 -sha256 -days 365 -nodes -out agent.cert -keyout agent.key

You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MA
Locality Name (eg, city) []:Cambridge
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Company
Organizational Unit Name (eg, section) []:IT
Common Name (e.g. server FQDN or YOUR name) []:*.mycompany.com
Email Address []:myaddress@mycompany.com
```

2. Make a copy of the certificate in PEM format.

```
$ openssl x509 -in agent.cert -out agent.pem -outform PEM
```

3. Review the certificate.

```
$ openssl x509 -in agent.pem -text
```

Replacing the agent certificate on a host

The following procedure replaces the Agent's current private key and certificate on a single host. To replace this certificate and key across an entire cluster, repeat this procedure for all the hosts.

1. Stop the Agent service on the host.

```
$ /etc/init.d/vertica_agent stop
```

2. Backup and rename the existing agent certificate and key.

```
$ cd /opt/vertica/config/share
$ mv agent.cert agent.cert.bck
$ mv agent.key agent.key.bck
$ mv agent.p em agent.pem.bck
```

3. Transfer the new certificate and key to the host's /opt/vertica/config/share directory.

```
$ scp agent.* root@123.12.12.123:/opt/vertica/config/share
```

4. Change the owner of the certificate and key to uidbadmin and the group to verticadba .

```
$ chown installed_Vertica_user:installed_Vertica_group agent.*
```

5. Make the certificate and key files read-only.

```
$ chmod -R 400 agent.*
```

6. Start the Agent service.

```
$ /etc/init.d/vertica_agent start
starting agent
Opening PID file "/opt/vertica/log/agent.pid".
Overwriting /opt/vertica/log/agent_uidbadmin.log
Overwriting /opt/vertica/log/agent_uidbadmin.err
start OK for user: uidbadmin
```

7. Verify that you can view information about your database with your API key.

```
$ curl -X GET https://10.20.80.145:5444/databases -H "VerticaApiKey:wCgXny3Wm+8OhEvGkAclv7v9+VIlxgXblpr4rf" -k
```

8. Verify that the Agent is using the new certificate.

```
$ openssl s_client -prexit -connect 10.20.80.145:5444
```

Importing and exporting data with TLS

Vertica uses TLS to secure connections and communications between clients and servers. When you import or export data between Vertica clusters, one of the clusters functions as a client, which means you can use TLS to protect that connection, too.

The ImportExportTLSMode parameter controls the strictness of TLS when importing or exporting data.

By default, ImportExportTLSMode is set to PREFER. With this setting, Vertica attempts to use TLS and falls back to plaintext; you can change this to always require encryption and, further, to validate the certificate on each connection. For more information about TLS during import and export operations, see Configuring connection security between clusters .

LDAP link service

LDAP Link enables synchronization between the LDAP and Vertica servers. This eliminates the need for you to manage two sets of users and groups or roles, one on the LDAP server and another on the Vertica server. With LDAP synchronization, the Vertica server becomes a replication database for the LDAP server.

> **Note**
> Users created by the LDAP Link service are compatible with native Vertica roles and do not require LDAP roles for functions like column access policies .

Automatic synchronization

With LDAP Link the Vertica server closely integrates with an existing directory service such as MS Active Directory or OpenLDAP. The Vertica server automatically synchronizes:

- LDAP users to Vertica users
- LDAP groups to Vertica roles

You manage all user and group properties in the LDAP server. If you are the Vertica database administrator, you need only to set up permissions for Vertica Analytic Database access on the users and groups.

Configure LDAP Link with LDAP Link connection parameters that reside in the catalog. See General and Connection Parameters for more information.

Configure LDAP link with dry runs

The LDAP Link dry run meta-functions allow you to configure the service in discrete stages before making any changes to your database. These stages are:

1. LDAP Link Bind : Establishing a connection between the LDAP server and the Vertica database
2. LDAP Link Search : Searching the LDAP server for users and groups
3. LDAP Link Sync : Mapping LDAP users and groups to their equivalents in Vertica

Query the system table LDAP_LINK_DRYRUN_EVENTS to view the results of each dry run.

For more information on dry runs and configuring LDAP Link, see Configuring LDAP link with dry runs .

Enable LDAP link

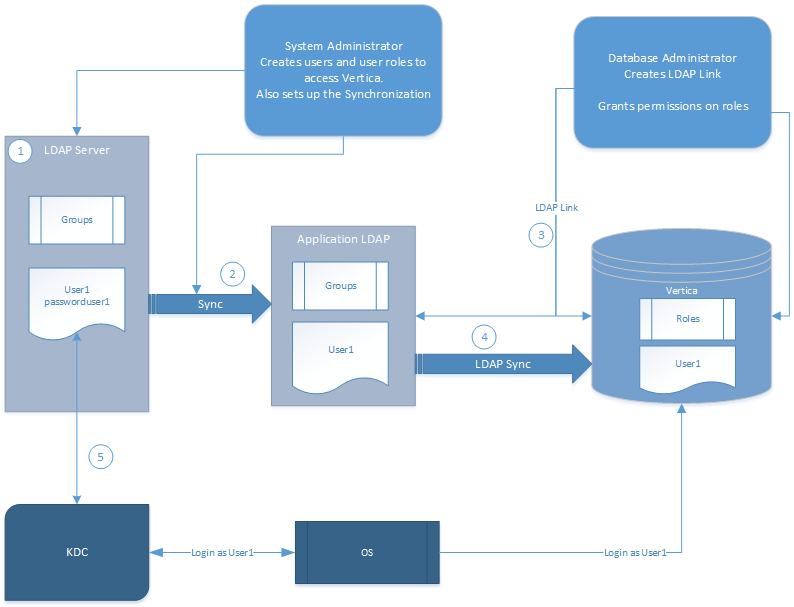
Enable LDAP Link as shown:

```
=> ALTER DATABASE dbname SET PARAMETER LDAPLinkURL='ldap://example.dc.com',
   LDAPLinkSearchBase='dc=DC,dc=com', LDAPLinkBindDN='CN=jsmith,OU=QA,DC=dc,DC=com,
   LDAPLinkBindPswd='password',LDAPLinkFilterUser='(objectClass=inetOrgPerson)', LDAPLinkFilterGroup='(objectClass=group)', LDAPLinkOn=1;
=> SELECT ldap_link_sync_start();
```

See LDAP link parameters .

LDAP link workflow

After you enable LDAP Link, synchronization occurs according to this workflow:



1. The System Administrator creates users and user groups on the LDAP server.
2. The System Administrator sets up LDAP Link service parameters as required and enables the service.
3. Using the LDAP Link service, Vertica Analytic Database replicates the users and user groups from the Application LDAP to the Vertica database, creating Vertica users and roles.
4. The LDAP server uses Kerberos (KDC) to authenticate the user logging in to Vertica.
   - The LDAP user can log in to Vertica if assigned the appropriate authentication type.
   - After logging in, you can grant users privileges using GRANT statements or as part of a Group.

> **Note**
> After synchronization, the Vertica user does not have an associated authentication method. To allow the user to log in, you must assign an authentication method to the user. See Configuring client authentication .

In this section

- Configuring LDAP link with dry runs
- Using LDAP link
- LDAP link parameters
- TLS for LDAP link
- Troubleshooting LDAP link issues

Configuring LDAP link with dry runs

Vertica supports several meta-functions that let you tweak LDAP Link settings before syncing with Vertica. Each meta-function takes LDAP Link parameters as arguments and tests a separate part of LDAP Link:

- LDAP_LINK_DRYRUN_CONNECT connects to the LDAP server.
- LDAP_LINK_DRYRUN_SEARCH searches for LDAP users and groups.
- LDAP_LINK_DRYRUN_SYNC maps and synchronizes LDAP users and groups to their equivalents in Vertica, creating and orphaning them accordingly.

These meta-functions should be used and tested in succession, and their arguments are cumulative. That is, the parameters you use to configure LDAP_LINK_DRYRUN_CONNECT are used for LDAP_LINK_DRYRUN_SEARCH, and the arguments for those functions are used for LDAP_LINK_DRYRUN_SYNC.

The dryrun and LDAP_LINK_SYNC_START functions must be run from the clerk node. To determine the clerk node, query NODE_RESOURCES :

```
=> SELECT node_name, dbclerk FROM node_resources WHERE dbclerk='t';
    node_name     | dbclerk
------------------+---------
v_vmart_node0001 | t
(1 row)
```

Be sure to query the LDAP_LINK_DRYRUN_EVENTS system table to verify the results of each dry run before moving to the next meta-function.

Configuring TLS for dry runs

Like the standard LDAP Link functions, LDAP Link dry-run functions pull from the 'LDAPLink' TLS Configuration for managing TLS connections. Query the TLS_CONFIGURATIONS system table to view existing TLS Configurations.

```
=> SELECT * FROM tls_configurations WHERE name='LDAPLink';
   name   |  owner  | certificate | ca_certificate | cipher_suites |  mode
----------+---------+-------------+----------------+---------------+---------
 LDAPLink | dbadmin | client_cert | ldap_ca        |               | DISABLE
(1 row)
```

For instructions on configuring TLS for LDAP Link and its dry run functions, see TLS for LDAP link .

Configuring LDAP link bind

Before configuring LDAP users and importing them to Vertica, you must first connect or "bind," with the LDAP server. Connections are managed with several parameters. For more information on each parameter, related functions, options, and default values, see LDAP link parameters .

LDAP_LINK_DRYRUN_CONNECT requires a Distinguished Name (DN), a password to authenticate with the LDAP server, and the URL to the LDAP server.

To encrypt the connection, configure the LDAPLink TLS Configuration .

By providing an empty string for the LDAPLinkBindPswd argument, you can also perform an anonymous bind if your LDAP server allows unauthenticated binds.

```
=> SELECT LDAP_LINK_DRYRUN_CONNECT('LDAPLinkURL','LDAPLinkBindDN','LDAPLinkBindPswd');
```

Dry run bind example

This tests the connection to an LDAP server at ldap://example.dc.com with the DN CN=amir,OU=QA,DC=dc,DC=com .

```
=> SELECT LDAP_LINK_DRYRUN_CONNECT('ldap://example.dc.com','CN=amir,OU=QA,DC=dc,DC=com','password');

               ldap_link_dryrun_connect
-------------------------------------------------------------------------------
Dry Run Connect Completed. Query v_monitor.ldap_link_dryrun_events for results.
```

To check the results of the bind, query the system table LDAP_LINK_DRYRUN_EVENTS.

```
=> SELECT event_timestamp, event_type, entry_name, role_name, link_scope, search_base from LDAP_LINK_DRYRUN_EVENTS;
       event_timestamp       |       event_type       |       entry_name       | link_scope | search_base
-----------------------------+------------------------+------------------------+------------+-------------
2019-12-09 15:41:43.589398-05 | BIND_STARTED           | ---------- | ---------- | ----------
2019-12-09 15:41:43.590504-05 | BIND_FINISHED          | ---------- | ---------- | ----------
```

Configuring LDAP link search

After a successful connection between Vertica and the LDAP server, you should configure and test your user and group search space for correctness and efficiency.

To search for users and groups on the LDAP server to import to your database, pass both the connection and search parameters to the LDAP_LINK_DRYRUN_SEARCH meta-function. The LDAP server responds with a list of users and groups that would be imported into Vertica with the given parameters.

By providing an empty string for the LDAPLinkBindPswd argument, you can also perform an anonymous search if your LDAP server's Access Control List (ACL) is configured to allow unauthenticated searches. The settings for allowing anonymous binds are different from the ACL settings for allowing anonymous searches.

```
=> SELECT LDAP_LINK_DRYRUN_SEARCH('LDAPLinkURL','LDAPLinkBindDN','LDAPLinkBindPswd','LDAPLinkSearchBase',
'LDAPLinkScope','LDAPLinkFilterUser','LDAPLinkFilterGroup','LDAPLinkUserName','LDAPLinkGroupName',
'LDAPLinkGroupMembers',[LDAPLinkSearchTimeout],['LDAPLinkJoinAttr']);
```

Dry run search example

This searches for users and groups in the LDAP server. In this case, the LDAPLinkSearchBase parameter specifies the dc.com domain and a sub scope, which replicates the entire subtree under the DN.

To further filter results, the function checks for users and groups with the person and group objectClass attributes. It then searches the group attribute cn , identifying members of that group with the member attribute, and then identifying those individual users with the attribute uid .

```
=> SELECT LDAP_LINK_DRYRUN_SEARCH('ldap://example.dc.com','CN=amir,OU=QA,DC=dc,DC=com','$vertica$','dc=DC,dc=com','sub',
'(objectClass=person)','(objectClass=group)','uid','cn','member',10,'dn');

            ldap_link_dryrun_search
-------------------------------------------------------------------------------
Dry Run Search Completed. Query v_monitor.ldap_link_dryrun_events for results.
```

To check the results of the search, query the system table LDAP_LINK_DRYRUN_EVENTS.

```
=> SELECT event_timestamp, event_type, entry_name, ldapurihash, link_scope, search_base from LDAP_LINK_DRYRUN_EVENTS;
       event_timestamp          |    event_type    |       entry_name       | ldapurihash | link_scope | search_base
--------------------------------+------------------+------------------------+-------------+------------+-------------
2020-01-03 21:03:26.411753+05:30 | BIND_STARTED     | ---------------------- |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:26.422188+05:30 | BIND_FINISHED    | ---------------------- |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:26.422223+05:30 | SYNC_STARTED     | ---------------------- |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:26.422229+05:30 | SEARCH_STARTED   | **********             |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:32.043107+05:30 | LDAP_GROUP_FOUND | Account Operators      |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:32.04312+05:30  | LDAP_GROUP_FOUND | Administrators         |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:32.043182+05:30 | LDAP_USER_FOUND  | user1                  |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:32.043186+05:30 | LDAP_USER_FOUND  | user2                  |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:32.04319+05:30  | SEARCH_FINISHED  | **********             |           0 | sub        | dc=DC,dc=com
```

Configuring LDAP link sync

After configuring the search space, you'll have a list of users and groups. LDAP sync maps LDAP users and groups to their equivalents in Vertica. The LDAPLinkUserName maps to the Vertica usernames and the LDAPLinkGroupName maps to Vertica roles.

```
=> SELECT LDAP_LINK_DRYRUN_SYNC('LDAPLinkURL','LDAPLinkBindDN','LDAPLinkBindPswd','LDAPLinkSearchBase',
'LDAPLinkScope','LDAPLinkFilterUser','LDAPLinkFilterGroup','LDAPLinkUserName','LDAPLinkGroupName',
'LDAPLinkGroupMembers',[LDAPLinkSearchTimeout],['LDAPLinkJoinAttr']);
```

Dry run sync example

To perform a dry run to map the users and groups returned from LDAP_LINK_DRYRUN_SEARCH, pass the same parameters as arguments to LDAP_LINK_DRYRUN_SYNC.

```
=> SELECT LDAP_LINK_DRYRUN_SYNC('ldap://example.dc.com','CN=amir,OU=QA,DC=dc,DC=com','$vertica$','dc=DC,dc=com','sub',
'(objectClass=person)','(objectClass=group)','uid','cn','member',10,'dn');

                    LDAP_LINK_DRYRUN_SYNC
-----------------------------------------------------------------------------------------
Dry Run Connect and Sync Completed. Query v_monitor.ldap_link_dryrun_events for results.
```

To check the results of the sync, query the system table LDAP_LINK_DRYRUN_EVENTS.

```
=> SELECT event_timestamp, event_type, entry_name, ldapurihash, link_scope, search_base from LDAP_LINK_DRYRUN_EVENTS;
       event_timestamp          |     event_type     |       entry_name       | ldapurihash | link_scope | search_base
--------------------------------+--------------------+------------------------+-------------+------------+-------------
2020-01-03 21:08:30.883783+05:30 | BIND_STARTED       | ---------------------- |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:30.890574+05:30 | BIND_FINISHED      | ---------------------- |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:30.890602+05:30 | SYNC_STARTED       | ---------------------- |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:30.890605+05:30 | SEARCH_STARTED     | **********             |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939369+05:30 | LDAP_GROUP_FOUND   | Account Operators      |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939395+05:30 | LDAP_GROUP_FOUND   | Administrators         |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939461+05:30 | LDAP_USER_FOUND    | user1                  |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939463+05:30 | LDAP_USER_FOUND    | user2                  |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939468+05:30 | SEARCH_FINISHED    | **********             |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939718+05:30 | PROCESSING_STARTED | **********             |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939887+05:30 | USER_CREATED       | user1                  |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939895+05:30 | USER_CREATED       | user2                  |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939949+05:30 | ROLE_CREATED       | Account Operators      |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939959+05:30 | ROLE_CREATED       | Administrators         |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.940603+05:30 | PROCESSING_FINISHED | **********            |           0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.940613+05:30 | SYNC_FINISHED      | ---------------------- |           0 | sub        | dc=DC,dc=com
```

Using LDAP link

When you use LDAP Link, the following are directly affected and help you manage and monitor the LDAP Link - Vertica Analytic Database synchronization:

- User and Group management
- LDAP Link User Flag
- Blocked Commands
- Client Authentication types

To cancel an in-progress synchronization, use LDAP_LINK_SYNC_CANCEL .

User and group management

Users and groups created on the LDAP server have a specific relationship with those users and roles replicated to the Vertica server:

- The user-group relationship on the LDAP server is maintained when those users and groups (roles) are synchronized with Vertica Analytic Database.
- If a user or group name exists on the Vertica database and a user or group with the same names is synchronized from the LDAP Server using LDAP Link, the users or groups become conflicted. Vertica cannot support multiple users with the same name. To resolve this, see User Conflicts .
- If the LDAP server contains a circular relationship, Vertica accepts and creates roles for the first non-circular part of the relationship returned by the LDAP server and ignores the rest.

  For example, suppose the LDAP server contains groups A and B , where A contains B , and B contains A , creating a circular relationship.

  If the LDAP server first returns that A contains B , Vertica creates roles A and B , and grants role A to role B . Vertica then ignores the fact that group B also contains A .

LDAP Link uses the entries in the dn: section of the LDAP configuration file as the unique user identifier when synchronizing a user to the Vertica Analytic Database:

```
dn: cn=user1,ou=dev,dc=example,dc=com
cn: user1
ou: dev
id: user1
```

The uid parameter in the LDAP configuration file indicates the LDAP user name.

```
uid: user1
```

Upon synchronization, the dn: entry gets mapped to the uid: to identify the Vertica Analytic Database user.

If you change a setting in the dn: and do not change the uid: , LDAP Link interprets the user as a new user when re-synchronizing with the Vertica Analytic Database. In this case, the existing Vertica Analytic Database user with that uid: gets deleted from Vertica and a new Vertica Analytic Database user is created.

If you change the uid: and not the dn: on LDAP, the uid on the Vertica Analytic Database gets updated to the new uid. Since you did not change the dn: LDAP Link does not interpret the user as a new user.

LDAP link user flag

As a dbadmin user, you can access the vs_users table to monitor user behavior on the Vertica Analytic Database. The users table contains an ldap_dn field that identifies whether or not the Vertica Analytic Database user is also an LDAP Link user. This example shows the ldap_dn field set to dn indicating the Vertica Analytic Database user is also an LDAP Link user:

```
=> SELECT * FROM vs_users;
-[ RECORD 1 ]---------+----------------------------------------------------
user_id               | 45035996273704962
user_name             | dbadmin
is_super_user         | t
profile_name          | default
is_locked             | f
lock_time             |
resource_pool         | general
memory_cap_kb         | unlimited
temp_space_cap_kb     | unlimited
run_time_cap          | unlimited
max_connections       | unlimited
connection_limit_mode | database
idle_session_timeout  | unlimited
all_roles             | dbduser*, dbadmin*, pseudosuperuser*
default_roles         | dbduser*, dbadmin*, pseudosuperuser*
search_path           |
ldap_dn               | dn
ldap_uri_hash         | 0
is_orphaned_from_ldap | f
```

Blocked commands

Be aware that the following SQL statements are blocked for Vertica users with ldapdn set to dn in the vs_users table:

- DROP USER and DROP ROLE
- ALTER ROLE RENAME
- ALTER USER name IDENTIFIED BY 'password' [REPLACE 'old_password']
- ALTER USER name PASSWORD EXPIRE
- ALTER USER name PROFILE
- ALTER USER name SECURITY_ALGORITHM...
- ALTER USER name DEFAULT ROLE role-name
- GRANT (Role)

Client authentication types

LDAP user and groups cannot log in to Vertica if client authentication is not assigned to the user or group. You can use the following valid authentication types for LDAP users and groups:

- GSS
- Ident
- LDAP
- Reject
- Trust

LDAP link parameters

Use LDAP Link parameters to determine:

- LDAP Link operations, such as enabling or disabling LDAP Link and how often to perform replication
- Authentication parameters, including SSL authentication parameters
- Users and groups that inherit unowned objects
- How to resolve conflicts

To configure TLS for LDAP Link, see TLS for LDAP link .

Set LDAP link parameters

This example shows how you can set:

- LDAPLinkURL , the URL of the LDAP server.
- LDAPLinkSearchBase , the base DN from which to start replication.

You also see how to set the LDAP Link Bind authentication parameters (LDAPLinkBindDN and LDAPLinkBindPswd) and enables LDAP Link (LDAPLinkOn).

```
=> ALTER DATABASE myDB1 SET PARAMETER LDAPLinkURL='ldap://10.60.55.128',
LDAPLinkSearchBase='dc=corp,dc=com',LDAPLinkBindDN='dc=corp,dc=com',LDAPLinkBindPswd='password';

=> ALTER DATABASE myDB1 SET PARAMETER LDAPLinkOn = '1';
```

General and connection parameters

Parameter	Description
LDAPLinkOn	Enables or disables LDAP Link. **Valid Values:** **0** —LDAP Link disabled **1** —LDAP Link enabled **Default:** 0
LDAPLinkURL	The LDAP server URL. To use a plaintext connection between Vertica and the LDAP server, begin the LDAPLinkURL with ldap:// and set the TLSMODE of LDAPLink to DISABLE . To use StartTLS, begin the LDAPLinkURL with ldap:// and set the TLSMODE of LDAPLink to ENABLE or higher. To use LDAPS, begin the LDAPLinkURL with ldaps:// and set the TLSMODE of LDAPLink to ENABLE or higher. **Example:** => SET PARAMETER LDAPLinkURL='ldap://example.dc.com';
LDAPLinkInterval	The time interval, in seconds, by which the LDAP Server and Vertica server synchronize. **Default:** 86400 (one day).
LDAPLinkFirstInterval	The first interval, in seconds, for LDAP/Vertica synchronization after the clerk node joins the cluster. **Default:** 120
LDAPLinkRetryInterval	The time, in seconds, the system waits to retry a failed synchronization. **Default:** 10
LDAPLinkRetryNumber	The number of retry attempts if synchronization failed. **Default:** 10.
LDAPLinkSearchBase	The base dn from where to start replication. **Example:** => SET PARAMETER LDAPLinkSearchBase='ou=vertica,dc=mycompany,dc=com'; Vertica recommends using a separate OU for database users.
LDAPLinkSearchTimeout	The timeout length, in seconds, for the LDAP search operation during an LDAP Link Service run. **Default:** 10

LDAPLinkScope	Indicates what dn level to replicate. **Valid Values:** • **sub** —Replicate entire subtree under baseDN • **one** —Replicate to one level under baseDN • **base** —Replicate only the baseDN level If you decrease the scope (for example, sub to one), some users may not be recognized during the next synchronization. **Default:** sub
LDAPLinkFilterUser	Determines how to filter users to be replicated. **Default:** "(objectClass=inetOrgPerson)"
LDAPLinkFilterGroup	Determines how to filter groups to be replicated. **Default:** "(objectClass=groupofnames)"
LDAPLinkGroupName	[Optional] The LDAP field to use when creating a role name in Vertica. **Default:** cn
LDAPLinkGroupMembers	The LDAP group that identifies the members of an LDAP group. This attribute returns a Fully Qualified Domain Name (FQDN). **Default:** member
LDAPLinkUserName	The LDAP field to use when creating a user name in Vertica. **Default:** uid
LDAPLinkJoinAttr	Specifies the attribute on which you want to join to assign users to their roles. **Default:** dn **Example:** POSIX groups associate users and groups with the uid attribute instead of dn . => SET PARAMETER LDAPLinkJoinAttr='uid';
LDAPLinkAddRolesAsDefault	Specifies whether the users synchronized through LDAP Link should have their groups set as default roles. If LDAPLinkAddRolesAsDefault is disabled (default), then the users are granted their groups as non- default roles, which must be manually enabled with SET ROLE . **Default:** 0 (disabled) **Example:** To enable: => ALTER DATABASE DEFAULT SET LDAPLinkAddRolesAsDefault = 1; To disable: => ALTER DATABASE DEFAULT SET LDAPLinkAddRolesAsDefault = 0;

Authentication parameters

Parameter	Description
LDAPLinkBindDN	The LDAP Bind DN used for authentication. **Example:** => SET PARAMETER LDAPLinkBindDN='CN=amir,OU=QA,DC=dc,DC=com';
LDAPLinkBindPswd	The valid password for the LDAP Bind DN to access the server. Only accessible by the dbadmin user. **Example:** => SET PARAMETER LDAPLinkBindPswd='password';

Miscellaneous parameters

Parameter	Description
LDAPLinkConflictPolicy	Determines how to resolve a user conflict. **Valid Values:** IGNORE—Ignores the incoming LDAP user and maintains the existing Vertica user. MERGE—Converts the existing user to an LDAP user. **Default:** MERGE
LDAPLinkStopIfZeroUsers	Enables or disables the shutdown of LDAPLink synchronization if no users are found in LDAP. **Valid values:** 0 - Disables the LDAPLink synchronization shutdown if no users are found. This may lead to inadvertent dropping of Vertica users. 1 - Enables the LDAPLink synchronization shutdown if no users are found. This prevents inadvertent dropping of Vertica users.
LDAPLinkDryRun	[Optional] Tests the connection to the LDAP server and logs the response without doing a synchronization. Also tests if parameters are correctly set. Note that this parameter is not the preferred dry run method. Instead, the LDAP_Link_Dryrun family of meta-functions provides more granular control over configurations and is the preferred way to perform LDAP Link dry runs . **Valid Values:** 0 - Disables LDAPLinkDryRun 1 - Enables LDAPLinkDryRun **Default:** 0
LDAPLinkConfigFile	[Optional] If this parameter is set with the path to a .LDIF file, the LDAP Link service will use the file as the source tree instead of connecting to the LDAP server.

See Configuration parameter management for information on setting LDAP Link parameters.

> **Note**
> When you change any Connection or Authentication parameter, LDAP Link reconnects and re-initializes the synchronization.

TLS for LDAP link

Vertica establishes a connection to an LDAP server in two contexts, and each context has a corresponding TLS Configuration that controls if each connection should use TLS:

1. **LDAPLink** : using the LDAPLink service or its dry run functions to synchronize users and groups between Vertica and the LDAP server.
2. **LDAPAuth** : when a user with an ldap authentication method attempts to log into Vertica, Vertica attempts to bind the user to a matching user in the LDAP server. If the bind succeeds, Vertica allows the user to log in.

Query TLS_CONFIGURATIONS to view existing TLS Configurations:

```
=> SELECT * FROM tls_configurations WHERE name IN ('LDAPLink', 'LDAPAuth');
   name   |  owner  | certificate | ca_certificate | cipher_suites |  mode
----------+---------+-------------+----------------+---------------+----------
 LDAPLink | dbadmin | client_cert | ldap_ca        |               | VERIFY_CA
 LDAPAuth | dbadmin | client_cert | ldap_ca        |               | DISABLE
(2 rows)
```

This page covers the LDAP Link service context. For details on the LDAP authentication context, see TLS for LDAP authentication .

Configuring LDAP link TLS

Vertica uses the LDAP Link service to retrieve users and groups from the LDAP server and to create corresponding users and roles in the database. To configure TLS for LDAP Link and its dry run functions , use the following procedure.

This procedure uses the predefined TLS Configuration LDAPLink . To create a custom TLS Configuration, see TLS configurations .

For details on key and certificate generation, see Generating TLS certificates and keys .

1. If you want Vertica to verify the LDAP server's certificate before establishing the connection, generate or import a CA certificate and add it to the LDAPLink TLS Configuration.
   For example, to import the existing CA certificate LDAP_CA.crt :

   ```
   => \set ldap_ca '\''`cat ldap_ca.crt`'\''
   => CREATE CA CERTIFICATE ldap_ca AS :ldap_ca;
   CREATE CERTIFICATE
   ```

   Then, to add the ldap_ca CA certificate to LDAPLink:

   ```
   ALTER TLS CONFIGURATION LDAPLink ADD CA CERTIFICATES ldap_ca;
   ```

2. If your LDAP server verifies client certificates, you must generate or import a client certificate and its key and add it to the LDAPLink TLS Configuration. Vertica presents this certificate to the LDAP server for verification by its CA.
   For example, to import the existing certificate client.crt (signed by the imported CA) and key client.key :

   ```
   => \set client_key '\''`cat client.key`'\''
   => CREATE KEY client_key TYPE 'RSA' AS :client_key;
   CREATE KEY

   => \set client_cert '\''`cat client.crt`'\''
   => CREATE CERTIFICATE client_cert AS :client_cert SIGNED BY ldap_ca KEY client_key;
   CREATE CERTIFICATE
   ```

   Then, to add client_cert to LDAPLink:

   ```
   => ALTER TLS CONFIGURATION LDAPLink CERTIFICATE client_cert;
   ```

3. Enable TLS or LDAPS (the exact protocol used depends on the value of host in the AUTHENTICATION object) by setting the TLSMODE to one of the following. TRY_VERIFY or higher requires a CA certificate:
   - **ENABLE** : Enables TLS. Vertica does not check the LDAP server's certificate.
   - **TRY_VERIFY** : Establishes a TLS connection if one of the following is true:
     - The LDAP server presents a valid certificate.
     - The LDAP server doesn't present a certificate.

     If the LDAP server presents an invalid certificate, a plaintext connection is used.
   - **VERIFY_CA** : Connection succeeds if Vertica verifies that the LDAP server's certificate is from a trusted CA. Using this TLSMODE forces all connections without a certificate to use plaintext.
   - **VERIFY_FULL** : Connection succeeds if Vertica verifies that the LDAP server's certificate is from a trusted CA and the cn (Common Name) or subjectAltName attribute matches the hostname or IP address of the LDAP server.

The cn is used for the username, so subjectAltName must match the hostname or IP address of the LDAP server.

For example:

```
=> ALTER TLS CONFIGURATION LDAPLink TLSMODE 'verify_ca';
ALTER TLS CONFIGURATION
```

4. Verify that the LDAPLinkTLSConfig parameter is using the TLS Configuration:

```
=> SHOW CURRENT LDAPLinkTLSConfig;
 level |       name        | setting
---------+-------------------+----------
 DEFAULT | LDAPLinkTLSConfig | LDAPLink
(1 row)
```

5. Set the LDAP Link Parameters according to your use case.

Troubleshooting LDAP link issues

Various issues can arise with LDAP Link service, including:

- Disconnected (Orphaned) users and roles
- Lost objects
- User conflicts

Disconnected (orphaned) users and roles

Vertica users and roles synchronized through LDAP Link can become disconnected, or orphaned, if an issue arises with the LDAP Link service. For example, users and roles become orphaned when you change the connection to the LDAP server as the following scenario describes:

1. Create an LDAP connection as follows:

```
=> ALTER DATABASE MyDB1 SET PARAMETER LDAPLinkURL='ldap://ebuser',
LDAPLinkSearchBase='dc=example,dc=com', LDAPLinkBindDN='mega',
LDAPLinkBindPswd='$megapassword$';
=> ALTER DATABASE MyDB1 SET PARAMETER LDAPLinkOn = '1';
```

2. Run an LDAP Link session to synchronize LDAP and Vertica users.
3. Change one or more connection parameters from Step 1. You can change the connection only if you change one of the LDAPLinkURL or LDAPLinkSearchBase parameters. Users will not be orphaned if the new and old LDAPLinkURL and LDAPLinkSearchBase contain the same set of users.
4. Run another LDAP Link session. The system attempts to re-synchronize LDAP and Vertica users. Because the connection has changed, the existing Vertica users cannot be synchronized with the LDAP users from the new connection. These Vertica users become orphaned.

As the dbadmin, you can identify orphaned users by checking the is_orphaned_from_ldap column in the USERS system table:

```
=> SELECT is_orphaned_from_ldap FROM users;
```

A field value of t indicates that the user is an orphaned user. Orphaned Vertica users cannot connect to the LDAP server and cannot login to Vertica using LDAP authentication (however, other authentication methods assigned to the user still work). In this case, you can delete the orphaned Vertica user and run the LDAP Link service to resynchronize users.

Re-parented objects

When you delete users or groups from the LDAP server, the LDAP Link service removes the same users and roles from Vertica, but does not delete objects owned by the deleted users and roles. To give these unowned objects a new owner, use the GlobalHeirUsername parameter, which specifies a user as the new parent for all objects originally owned by deleted users.

For example, to give ownership of unowned objects to user1, creating the user if it does not already exist:

```
=> ALTER DATABASE example_db SET PARAMETER GlobalHeirUsername=user1;
```

By default, this parameter is set to <auto> which re-parents objects to the dbadmin user.

If GlobalHeirUsername is empty, objects are not re-parented to another user.

For details, see Security Parameters .

User conflicts

Vertica users and roles synchronized using LDAP Link can become conflicted. Such conflicts can occur, for example, when you create a new user or group on the LDAP server and another user or role with the same name exists on the Vertica.

As the dbadmin, use one of the following parameters to resolve user conflicts:

- LDAPLinkConflictPolicy
- LDAPLinkStopIfZeroUsers

LDAPLinkConflictPolicy

LDAPLinkConflictPolicy controls how Vertica behaves when it encounters a conflict. Changes to this parameter take effect during the next synchronization.

- LDAPLinkConflictPolicy=IGNORE ignores the incoming LDAP users and retains the existing Vertica user
- LDAPLinkConflictPolicy=MERGE (default) merges the incoming LDAP user with the Vertica user and converts the database user to an LDAP user, retaining the database user's objects.

For example, to set the parameter:

```
=> ALTER DATABASE example_db SET PARAMETER LDAPLinkConflictPolcy='MERGE';
```

LDAPLinkStopIfZeroUsers

LDAPLinkStopIfZeroUsers controls how Vertica behaves when the LDAP server has zero users during synchronization.

- LDAPLinkStopIfZeroUsers=0 does not stop the synchronization if no users are found in the LDAP server and all Vertica users with are dropped.
- LDAPLinkStopIfZeroUsers=1 stops the synchronization if no users are found in the LDAP server and returns an error. No Vertica users are dropped.

LDAP_LINK_DRYRUN and LDAP_LINK_SYNC_START do not populate tables

The dryrun and LDAP_LINK_SYNC_START functions must be run from the clerk node. To determine the clerk node, query NODE_RESOURCES :

```
=> SELECT node_name, dbclerk FROM node_resources WHERE dbclerk='t';
   node_name      | dbclerk
------------------+---------
 v_vmart_node0001 | t
(1 row)
```

Monitoring LDAP link

Use the ldap_link_events table to monitor LDAP Link synchronization:

```
=> SELECT transaction_id, event_type, entry_name, entry_oid FROM ldap_link_events;
   transaction_id |     event_type      | entry_name | entry_oid
------------------+---------------------+------------+-----------
45035996273705317 | SYNC_STARTED        |            |         0
45066962732553589 | SYNC_FINISHED       |            |         0
45066988112255317 | PROCESSING_STARTED  |            |         0
23411234566789765 | USER_CREATED        | tuser      | 234548899
(4 rows)
```

Connector framework service

The Connector Framework Service (CFS) allows secure indexing of documents from IDOL to the Vertica Analytic Database. Access control lists determine which users have permissions to access documents. Documents transferred from IDOL are stored in a flex table (Flex tables).

Important

Vertica 9.1.1 does not support the IdolLib function library. If you have the IdolLib function library installed and are upgrading to Vertica 9.1.1, you see an error and you cannot access the IdolLib function library.

To determine if the IdolLib library is installed, run the following script:

```
$ /opt/vertica/packages/idol/ddl/isinstalled.sql
```

To uninstall the IdolLib library, run the following script:

```
$ /opt/vertica/packages/idol/ddl/uninstall.sql
```

CFS components

Use the following CFS components to implement the service on the Vertica:

- IDOL document metadata
- CFS Configuration file

For detailed information, see Implementing CFS .

IDOL document metadata

Vertica Analytic Database stores IDOL document metadata in a flex table. Set the name of the flex table with the TableName parameter in the CFS configuration file (see Implementing CFS). The metadata includes the following:

- AUTONOMYMETADATA (Mandatory): An alphanumeric designation for the ACL designated for the document.
- DREFIELD: Assigns permission levels to users and groups for accessing IDOL documents.
- DRETITLE: The document title.

CFS configuration file

You must index IDOL metadata in Vertica Analytic Database to be available for queries. See Implementing CFS .

In this section
- Implementing CFS

Implementing CFS

After Vertica ingests documents from IDOL into flex tables, you can implement CFS to secure those documents. Implementing the security requires that the Vertica database administrator modify the CFS configuration file.

Modify the CFS configuration file

The database administrator must modify the following in the CFS configuration file to have CFS automatically index the metadata:

1. In the [Indexing] section, set the IndexerSections parameter to vertica:
```
[Indexing]
IndexerSections=vertica
IndexBathSize=1
IndexTimeInterval=30
```
2. Create a new section with the same name you entered in the IndexerSections parameter and enter the following parameters and keywords:
```
[vertica]
IndexerType=Library
ConnectionString=Driver=Vertica;Server=123.456.478.900;Databaswe=myDb;UID=dbadmin;PWD=password
TableName=myFlexTable
LibraryDirectory= ./shared_library_indexers
LibraryName=VerticaIndexer
```

The VerticaIndexer (LibraryName above) is part of CFS. To use this tool, you must install and configure the Vertica ODBC drivers on the same machine as CFS. CFS sends JSON-formatted data to the Flex table using ODBC.

For more information, see Installing the ODBC client driver .

Query the IDOL data

To query the IDOL data in a flex table, run a simple SELECT query. In this example, idol_table is the name of the flex table:

```
=> SELECT * FROM idol_table;
```

Federal information processing standard

When running on a FIPS-compliant operating system that Vertica supports , Vertica uses a certified OpenSSL FIPS 140-2 cryptographic module. This meets the security standards set by the National Institute of Standards and Technology (NIST) for Federal Agencies in the United States or other countries.

The standard specifies the security requirements that a cryptographic module needs in a system protecting sensitive information. For details on the standard see the Computer Security Resource Center .

> **Note**
> Vertica itself is not FIPS compliant but it is compatible with running on a FIPS-enabled system using FIPS resources.

For a list of FIPS prerequisites, see FIPS 140-2 supported platforms .

In this section

- OpenSSL behavior
- FIPS-Enabled databases: limitations
- Implementing FIPS 140-2
- FIPS 140-2 compliance statement

OpenSSL behavior

Dynamic OpenSSL linking is a requirement for a FIPS implementation on the client and server. The Vertica server uses the OpenSSL that resides on the host system (as indicated in FIPS 140-2 supported platforms). OpenSSL dynamically links with LDAP and Kerberos.

For more information see Locate OpenSSL Libraries .

Libraries on CentOS systems

On a FIPS-compliant CentOS system, Vertica runs only with the OpenSSL libraries listed in FIPS 140-2 supported platforms . Other versions of these libraries do not run on a FIPS system. This incompatibility occurs because the FIPS security policy checksums the library to which an application is linked and verifies that the library the application executes with the same checksum.

Library versioning on Non-FIPS systems

Be aware that on some non-FIPS systems, versioning anomalies can occur when you install a new version of OpenSSL. Sometimes, the default OpenSSL build procedure produces libraries with versions named 1.0.0. For Vertica to recognize that a library has a higher version number, you must provide the library name with a higher version number. For example, when installing OpenSSL version 1.0.1t, name the libraries libcrypto.so.1.0.1t or libssl.1.0.1t (symbolic links with these names are sufficient).

FIPS-Enabled databases: limitations

FIPS-enabled databases have the following limitations:

- You cannot create a FIPS-enabled database on a non-FIPS machine.
- You cannot create a non-FIPS database on a FIPS-enabled machine.
- The Management Console and its daemon, Agent , are not available on FIPS-enabled databases.
- Copying data generated with the MD5 hashing algorithm from a non-FIPS machine to a FIPS-enabled machine results in data corruption.
- Due to limitations in the FIPS cryptographic module, Vertica does not recommend enabling internode encryption in FIPS environments. If you use FIPS and internode encryption, you may experience occasional query failure due to socket closure in workloads that send a high volume of data across the network.

Implementing FIPS 140-2

Implementing FIPS 140-2 on your Vertica Analytic Database requires configuration on the server and client. While Vertica server uses FIPS-approved algorithms, Vertica clients may be running on non-FIPS-approved systems. Therefore, you must implement FIPS 140-2 compliance from end to end.

For more information on implementing FIPS, see:

- FIPS compliance for the Vertica server
- Implement FIPS on the client

In this section

- FIPS compliance for the Vertica server
- Implement FIPS on the client

FIPS compliance for the Vertica server

To make Vertica FIPS-compliant, you must:

- Set the RequireFIPS parameter to 1.
- Hash your passwords with SHA-512. See Hash authentication for details.
- Generate a signed TLS certificate to establish a secure connection to the client.

RequireFIPS parameter

Vertica sets the RequireFIPS configuration parameter on the server on startup to reflect the state of FIPS on the system: 1 if FIPS is enabled and 0 if FIPS is disabled.

The value of RequireFIPS matches the value of crypto.fips_enabled file.

Vertica sets the RequireFIPS parameter based on the contents of crypto.fips_enabled :

- If the file /proc/sys/crypto/fips_enabled exists and contains a 1 (FIPS-enabled), Vertica sets RequireFIPS to 1.
- If the file /proc/sys/crypto/fips_enabled does not exist, or exists and contains a 0 (non-FIPS), Vertica automatically sets RequireFIPS to 0.
- If the FIPS state of a node, as determined from the existence of /proc/sys/crypto/fips_enabled , differs from the state received from the cluster initiator, the node fails. This behavior prevents the creation of clusters of mixed FIPS and non-FIPS systems.

Important
If you attempt to restore a FIPS-enabled node to a non-FIPS cluster, the restore will fail.

Secure client-server connection

It's important to secure client-server connections with TLS. For instructions on setting up client-server TLS, see Configuring client-server TLS .

FIPS-Compliant AWS endpoints

To configure AWS to use a FIPS-compliant S3 Endpoint , set the following S3 parameters :

```
AWSEndpoint = s3-fips.dualstack.us-east-1.amazonaws.com
S3EnableVirtualAddressing = 1
```

Implement FIPS on the client

Vertica provides a FIPS-compliant client driver, which you can install on a FIPS-enabled system. The 64-bit client includes vsql and ODBC drivers.

For information about installing the FIPS client, and installation, refer to the following

- Installing the FIPS client driver for ODBC and vsql
- Installing the FIPS client driver for JDBC

FIPS 140-2 compliance statement

Contents

1. Summary

2. Overview

a. About Vertica

b. About FIPS 140-2

3. Vertica and FIPS 140-2

1. summary

Vertica complies with Federal Information Processing Standard 140-2 (FIPS 140-2), which defines the technical requirements to be used by Federal Agencies when these organizations specify cryptographic-based security systems for protection of sensitive or valuable data. The compliance of Vertica with FIPS 140-2 is ensured by: 1) Integrating validated and NIST-certified third party cryptographic module(s), and using the module(s) as the only provider(s) of cryptographic services; 2) Using FIPS-approved cryptographic functions; 3) Using FIPS-approved and NIST-validated technologies applicable for Vertica design, implementation and operation.

2. overview

a. About Vertica

- Vertica is a high performance relational database management system used for advanced analytics applications. Its performance and scale is achieved through a columnar storage and execution architecture that offers a massively parallel processing solution. Aggressive encoding and compression allows Vertica analytics to perform by reducing CPU, memory and disk I/O Processing times.
- For more details about Vertica and its usage, see Architecture .

b. About FIPS 140-2

FIPS (Federal Information Processing Standard) 140-2, *Security requirements for cryptographic modules* , is the Federal standard for proper cryptography for computer systems purchased by the government.

The Federal Information Processing Standards Publication (FIPS) 140-2, “Security Requirements for Cryptographic Modules,” was issued by the National Institute of Standards and Technology (NIST) in May, 2001.

The benefits of using FIPS 140-2 validated crypto module is that the crypto algorithms are deemed appropriate and that they perform the encrypt/decrypt/hash functions correctly. The standard specifies the security requirements for cryptographic modules utilized within a security system that protects sensitive or valuable data. The requirements can be found in the following documents:

- SECURITY REQUIREMENTS FOR CRYPTOGRAPHIC MODULES
- Annex A: Approved Security Functions for FIPS PUB 140-2, Security Requirements for Cryptographic Modules

3. Vertica and FIPS 140-2

FIPS 140-2 validated third party module

Vertica conforms with FIPS 140-2 Level 1 compliance by dynamically linking to the FIPS 140-2 approved OpenSSL cryptographic module provided by the Operating System, which in our initial release is Red Hat Enterprise Linux 6.6 OpenSSL Module.

Vertica can be configured to operate in FIPS-compliant mode ensuring its functions and procedures like SSL/TLS connections, which require cryptography (secure hash, encryption, digital signatures, etc.) makes use of the crypto services provided by RedHat Enterprise Linux 6.6 OpenSSL Module v3.0 which is validated for FIPS 140-2. If you are not running on a FIPS-compliant operating system that Vertica supports , you will not be able to run Vertica on FIPS mode. The assurance that Vertica is using the right FIPS 140-2 encryption modules is managed at the operating system level by RedHat’s implementation.

Vertica checks the OS level flag setting /proc/sys/crypto/fips_enabled to kick off Vertica’s FIPS mode installation. Further details about how to install and configure Vertica and its components to conform to FIPS 140-2 standard appear in the installation and security guides:

- Install Vertica with the installation script
- Federal information processing standard

Modes of Operation

Vertica Server operates in one of two modes determined by the OS configuration.

- FIPS-compliant mode – supports FIPS 140-2 compliant cryptographic functions. In this mode, all cryptographic functions, default algorithms and key lengths are bound to those allowed by FIPS 140-2.
- Standard mode – non-FIPS 140-2 compliant mode which utilizes all existing Vertica cryptography functions.

TLS/SSL3.x

All the Vertica client/server communications can be secured with FIPS-compliant Transport Layer Security TLS1.2/SSL3.1 or higher. It is relying on FIPS 140-2 approved hash algorithms and ciphers.

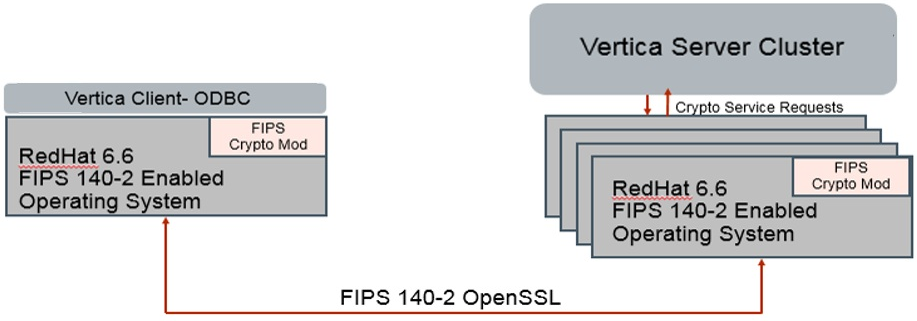
- TLS handshake, key negotiation and authentication provides data integrity and uses secure hash and FIPS 140-2 approved cryptography and digital signature.
- TLS encryption of data in transit provides confidentiality and making use of FIPS 140-2 approved cryptography.

Secure Hash

Per FIPS 140-2 standards, Vertica, in the FIPS 140-2 compliant mode, can be configured to use only the SHA-512 algorithm.

FIPS 140-2 Architecture

Vertica is a relational database system that is comprised of a client component and a server component. On the Client Side, we offer a suite of drivers for host clients to access the Vertica Server Side component. Both client and server Vertica components conform to FIPS 140-2 Level 1 compliance by dynamically linking to the FIPS 140-2 approved OpenSSL cryptographic module provided by RedHat Enterprise Linux 6.6 OpenSSL Module.



Supported Platforms

See FIPS 140-2 supported platforms for information about FIPS-compliant operating systems and client drivers that Vertica supports.

Design Assurance

Vertica uses the security provider Red Hat Enterprise Linux 6.6 OpenSSL Module v3.0 . This is the only supported security provider for FIPS 140-2.

Once you have configured Vertica to be compliant with FIPS 140-2, you cannot revert back to the standard configuration unless you disable FIPS 140-2 at the operating system level. Please reference the following documentation section for considerations:

- FIPS compliance for the Vertica server
- Implement FIPS on the client

Database auditing

Database auditing often involves observing a database to be aware of the actions the database users are taking. Auditing can help with security, for example, to ensure that a user does not change information to which they should not have access. Audit categories make it easier to track changes within the database. You can see system tables that will bring together logged queries, tables, and changes to configuration parameters.

For example, the authentication audit category tracks queries, system tables, and configuration parameters related to security and authentication, such as:

- DROP AUTHENTICATION statement
- GRANT/REVOKE authentication statements
- LDAP Link related configuration parameters

You can also use the authentication audit category for weekly security reports to better understand attempts to gain access to data or to view unauthorized changes.

There are three system tables that can be used to track changes for queries, parameters, and tables as follows:

- LOG_PARAMS
- LOG_QUERIES
- LOG_TABLES

There is also a system table for tracking changes to privileges for users:

- AUDIT_MANAGING_USERS_PRIVILEGES

System table restriction and access

Two functions let you restrict and open access to system tables for a given session:

- RESTRICT_SYSTEM_TABLES_ACCESS restricts access to non-superuser-only tables that are not accessible during lockdown.
- RELEASE_SYSTEM_TABLES_ACCESS allows access to non-superuser-only tables that are not accessible during lockdown.

Connecting to Vertica

This book explains several methods of connecting to Vertica, including:

- Directly connecting to Vertica using the vsql client application.
- Installing and configuring the Vertica client libraries to allow client applications to access Vertica.
- Developing your own client applications using the Vertica client libraries.

In this section

- Using vsql
- Client libraries
- Management API
- Connect with an SSH tunnel
- SQLTools Vertica driver

Using vsql

vsql is a character-based, interactive, front-end utility that lets you type SQL statements and see the results. It also provides a number of meta-

commands and various shell-like features that facilitate writing scripts and automating a variety of tasks.

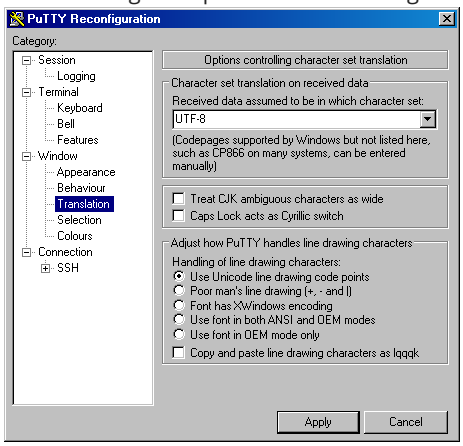
If you are using the vsql client installed on the server, then you can connect from the:

- Administration Tools
- Linux command line

You can also install the vsql client for other supported platforms.

General notes

- SQL statements can be spread over several lines for clarity.
- vsql can handle input and output in UTF-8 encoding. The terminal emulator running vsql must be set up to display the UTF-8 characters correctly. The following example shows the settings in PuTTy:



  See also Best Practices for Working with Locales .
- Cancel SQL statements by typing Ctrl+C.
- Traverse command history by typing Ctrl+R.
- When you disconnect a user session, any transactions in progress are automatically rolled back.
- To view wide result sets, use the Linux less utility to truncate long lines.
  1. Before connecting to the database, specify that you want to use less for query output:
     ```
     $ export PAGER=less
     ```
  2. Connect to the database.
  3. Query a wide table:
     ```
     => select * from wide_table;
     ```
  4. At the less prompt, type:
     ```
     -S
     ```

  If a shell running vsql fails (crashes or freezes), the vsql processes continue to run even if you stop the database. In that case, log in as root on the machine on which the shell was running and manually terminate the vsql process. For example: $ ps -ef | grep vertica ... fred 2401 1 0 06:02 pts/1 00:00:00 /opt/vertica/bin/vsql -p 5433 -h test01_site01 quick_start_single ... $ kill -9 2401

Enabling autocommit

By default, you must COMMIT to save changes made in a transaction. To enable automatic commits, see SET SESSION AUTOCOMMIT .

In this section

- Installing the vsql client
- vsql usage on Windows
- Connecting from the administration tools
- Connecting from the command line
- Meta-commands
- Variables
- Prompting
- Command line editing
- vsql environment variables
- Locales
- Entering data with vsql
- Files
- Exporting data using vsql
- Copying data using vsql

- Output formatting examples

Installing the vsql client

This page covers a non-FIPS installation. To install on a FIPS-compliant system, see Installing the FIPS client driver for ODBC and vsql .

Linux and macOS

> **Note**
> For Linux: The vsql client is automatically installed as part of the Vertica server .rpm .

To install vsql manually on another system:

1. Download vsql.
2. Extract or install vsql:
   - If you downloaded the .tar , create the /opt/vertica/ directory if it does not already exist, copy the .tar to it, navigate to it, and extract the .tar :

     ```
     $ mkdir -p /opt/vertica/
     $ cp driver_name.tar.gz /opt/vertica/
     $ tar vzxf driver_name.tar.gz
     ```

   - If you downloaded the .rpm , install it with:

     ```
     $ rpm -Uvh driver_name.rpm
     ```

3. Optionally add the vsql directory to your PATH. For example:

   ```
   $ export PATH=$PATH:\opt\vertica\bin
   ```

4. Make the vsql client executable. For example, to allow all users to run vsql:

   ```
   $ chmod ugo+x /path/to/vsql
   ```

5. Set your shell locale to a locale supported by vsql (which ones?). For example, in your .profile , add:

   ```
   export LANG=end_US.UTF-8
   ```

Windows

To install the vsql client:

1. Download the Windows client driver installer. For details on the drivers included in this installer, see Windows client driver installer .
2. Run the installer and follow the prompts to install the drivers.
3. Reboot your system.

After installing the driver, you can optionally add the vsql directory to your PATH. For example, to append the vsql directory to your PATH with Windows PowerShell for the current session:

```
PS C:\> $Env:PATH += ";C:\Program Files\Vertica Systems\VSQL64\"
```

You can verify that the vsql directory is in your PATH by running vsql -? :

```
PS C:\> vsql -?
This is vsql, the Vertica Analytic Database interactive terminal.

Usage:
  vsql [OPTIONS]... [DBNAME [USERNAME]]
```

For usage details, see vsql usage on Windows

vsql usage on Windows

Font

The default raster font does not work well with the ANSI code page. Set the console font to "Lucida Console."

Console encoding

vsql is built as a "console application." The Windows console windows use a different encoding than the rest of the system, so take care when you use 8-bit characters within vsql. If vsql detects a problematic console code page, it warns you at startup.

To change the console code page, set the code page by entering cmd.exe /c chcp 1252 .

> **Note**
> 1252 is a code page that is appropriate for European languages. Replace it with your preferred locale code page.

Running under cygwin

Verify that your cygwin.bat file does not include the "tty" flag. If the "tty" flag is included in your cywgin.bat file, then banners and prompts are not displayed in vsql.

To verify, enter:

```
set CYGWIN=binmode tty ntsec
```

To remove the "tty" flag, enter:

```
set CYGWIN=binmode ntsec
```

Additionally, when running under Cygwin, vsql uses Cygwin shell conventions as opposed to Windows console conventions.

Tab completion

Tab completion is a function of the shell, not vsql. Because of this, tab completion does not work the same way in Windows vsql as it does on Linux versions of vsql.

On Windows, instead of using tab-completion, press F7 to pop-up a history window of commands. You can also press F8 after typing a few letters of a command to cycle through commands in the history buffer which begin with the same letters.

Connecting from the administration tools

You can use the Administration tools to connect to a database using vsql on any node in the cluster.

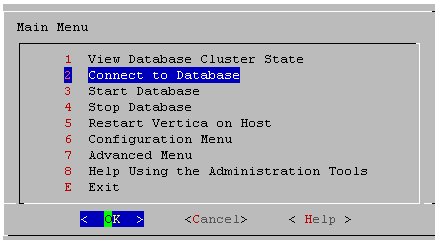
1. Log in as the database administrator user; for example, dbadmin.

   > **Note**
   > Vertica does not allow users with root privileges to connect to a database for security reasons.

2. Run the Administration Tools.

   ```
   $ /opt/vertica/bin/admintools
   ```

3. On the Main Menu, select **Connect to Database** .

4. If prompted, enter the database password:

   ```
   Password:
   ```

   When you create a new user with the CREATE USER command, you can configure the password or leave it empty. You cannot bypass the password if the user was created with a password configured. You can change a user's password using the ALTER USER command.

5. The Administration Tools connect to the database and transfer control to vsql .

   ```
   Welcome to vsql, the Vertica Analytic Database interactive terminal.
   Type:  \h or \? for help with vsql commands
        \g or terminate with semicolon to execute query
        \q to quit

   =>
   ```

> **Note**
> See Meta-commands for the various commands you can run while connected to the database through the Administration Tools.

Connecting from the command line

You can connect to a database using vsql from the command line on multiple client platforms.

If the connection cannot be made for any reason—for example, you have insufficient privileges, or the server is not running on the targeted host—vsql returns an error and terminates.

Syntax

```
/opt/vertica/bin/vsql [-h host] [ -p port ] [ option...] [ dbname [ username ] ]
```

Parameters

host

Optional if you connect to a local server. You can provide an IPv4 or IPv6 IP address or a host name.

For Vertica servers that have both IPv4 and IPv6 addressed and you have provided a host name instead of an IP address, you can prefer to use an IPv4 address with the -4 option and to use the IPv6 adddress with the -6 option if the DNS is configured to provide both IPv4 and IPv6 addresses. If you are using IPv6 and provide an IP address, you must append the address with an % *interface name* .

port

The database server port.

Default: 5433

option

One or more vsql command-line options .

If the database is password protected, you must specify the -w or --password command line option.

dbname

The name of the target database. If unspecified, vsql automatically connects to the database on the specified *host* and *port* .

username

A database username, by default your system username.

Exit codes

vsql returns 0 to the shell when it terminates normally. Otherwise, it returns one of the following:

- 1: A fatal error occurred—for example, out of memory or file not found.
- 2: The connection to the server went bad and the session was not interactive
- 3: An error occurred in a script and the variable ON_ERROR_STOP was set.
- Unrecognized words in the command line might be interpreted as database or user names.

Examples

The following example shows how to capture error messages by redirecting vsql output to the output file retail_queries.out :

```
$ vsql --echo-all < retail_queries.sql > retail_queries.out 2>&1
```

In this section

- Command-line options
- Connecting from a non-cluster host

Command-line options

This section contains the command-line options for vsql.

General options

--command command

-c command

Runs one command and exits. This command is useful in shell scripts.

Variables set with -v are not processed when referenced in a -c command. To use variables, create a .sql file that references the variable and pass it to vsql with the -f option.

--dbname *dbname*
-d *dbname*
Specifies the name of the database to which you want to connect. Using this command is equivalent to specifying **dbname** as the first non-option argument on the command line.

--file *filename*
-f *filename*
Uses the *filename* as the source of commands instead of reading commands interactively. After the file is processed, vsql terminates.

--help
Displays help about vsql command line arguments and exits.

--timing
-i
Enables the \timing meta-command.

--list
-l
Returns all available databases, then exits. Other non-connection options are ignored. This command is similar to the internal command \list .

--set *assignment*
--variable *assignment*
-v *assignment*
Performs a variable assignment, like the vsql command \set .

--version -V
Prints the vsql version and exits.

--no-vsqlr
-X
Disables all command line editing and history functionality.

Connection options

-4
When resolving hostnames in dual stack environments, prefer IPv4 addresses.

-6
When resolving hostnames in dual stack environments, prefer IPv6 addresses.

-B *server:port* [,...]
Sets connection backup server/port. Use comma-separated multiple hosts (default: not set). If using an IPv6 address, enclose the address in brackets ([,]) and place the port outside of the brackets. For example \B [2620:0:a13:8a4:9d9f:e0e3:1181:7f51]:5433

--enable-connection
-load-balance -C
Enables connection load balancing (default: not enabled).

> **Note**
> You can only use load balancing with one address family in dual stack environments. For example, if you've configured load balancing for IPv6 addresses, then when an IPv4 client connects and requests load balancing the server does not allow it.

--host *hostname*
-h *hostname*
Specifies the host name of the machine on which the server is running.

-k *krb-service*
Provides the service name portion of the Kerberos principal (default: vertica). Using -k is equivalent to using the drivers' KerberosServiceName connection string.

-K *krb-host*
Provides the instance or host name portion of the Kerberos principal. -K is equivalent to the drivers' KerberosHostName connection string.

-g *client-label*

--label *client-label*

Sets the client label for the connection.

--sslmode

-m

Specifies the policy for making SSL connections to the server. Options are require, prefer, allow, and disable. You can also set the VSQL_SSLMODE variable to achieve the same effect. If the variable is set, the command-line option overrides it.

--port *port*

-p *port*

Specifies the TCP port or the local socket file extension on which the server is listening for connections. Defaults to port 5433.

`--username` *`username`*

`-U` *`username`*

Connects to the database as the user *username* instead of the default.

-w *password*

Specifies the password for a database user.

> **Note**
> Using this command-line option displays the database password in plain text. Use it with care, particularly if you are connecting as the database administrator, to avoid exposing sensitive information.

--password

-W

Forces vsql to prompt for a password before connecting to a database.The password is not displayed on the screen. This option remains set for the entire session, even if you change the database connection with the meta-command \connect .

Output formatting

--no-align

-A

Switches to unaligned output mode. (The default output mode is aligned.)

-b

Beep on command completion.

--field-separator *separator*

-F *separator*

Specifies the field separator for unaligned output (default: "|") (-P fieldsep=). (See -A --no-align .) Using this command is equivalent to \pset fieldsep or \f .

--html

-H

Turns on HTML tabular output. Using this command is equivalent to using the \pset format html or the \H command.

--pset *assignment*

-P *assignment*

Lets you specify printing options in the style of \pset on the command line. You must separate the name and value with an equals (=) sign instead of a space. Thus, to set the output format to LaTeX, you could write -P format=latex .

-Q

Turns on trailing record separator. Use \pset trailingrecordsep to toggle the trailing record separator on or off.

--record-separator *separator*

-R *separator*

Uses *separator* as the record separator. Using this command is equivalent to using the \pset recordsep command.

--tuples-only

-t

Disables printing of column names, result row count footers, and so on. This is equivalent to the vsql meta-command \t .

--table-attr *options*

-T *options*

Allows you to specify options to be placed within the HTML table tag. See \pset for details.

--expanded

-x

Enables extended table formatting mode. This is equivalent to the vsql meta-command \x .

Input and output options

--echo-all

-a

Prints all input lines to standard output as they are read. This approach is more useful for script processing than interactive mode. It is the same as setting the variable ECHO to all .

--echo-queries

-e

Copies all SQL commands sent to the server to standard output. Using this command is equivalent to setting the variable ECHO to queries.

-E

Displays queries generated by internal commands.

-n

Disables command line editing.

--output *filename*

-o *filename*

Writes all query output to *filename* . Using this command is equivalent to using the vsql meta-command \o .

--quiet

-q

Specifies that vsql do its work quietly (without informational output, such as welcome messages). This command is useful with the -c option. Within vsql you can also set the QUIET variable to achieve the same effect.

--single-step

-s

Runs in single-step mode for debugging scripts. Forces vsql to prompt before each statement is sent to the database and allows you to cancel execution.

--single-line -S

Runs in single-line mode where a newline terminates a SQL command, as if you are using a semicolon.

> **Note**
> This mode is provided only by customer request. Vertica recommends that you not use single-line mode in cases where you mix SQL and meta-commands on a line. In single-line mode, the order of execution might be unclear to the inexperienced user.

In this section

- -A --no-align
- -a --echo-all
- -c --command
- -d --dbname
- -E
- -e --echo-queries
- -F --field-separator
- -f --file
- ? --help
- -H --html
- -h --host
- -i -- timing
- -g --label
- -l --list
- -m --sslmode
- -n
- -o --output
- -P --pset
- -p --port
- -q --quiet

- -R --record-separator
- -S --single-line
- -s --single-step
- -T --table-attr
- -t --tuples-only
- -V --version
- -v --variable --set
- -X --no-vsqlrc
- -x --expanded

-A --no-align

-A or --no-align switches to unaligned output mode. The default output mode is aligned.

-a --echo-all

-a or --echo-all prints all input lines to standard output as they are read. This is more useful for script processing than interactive mode. It is equivalent to setting the variable ECHO to all .

-c --command

-c *command* or --command *command* runs one command and exits. This is useful in shell scripts.

Use either:

- A command string that can be completely parsed by the server that does not contain features specific to vsql
- A single meta-command

You cannot mix SQL and vsql meta-commands. You can, however, pipe the string into vsql as shown:

```
echo "\\timing\\\\select * from t" | ../Linux64/bin/vsql
        Timing is on.
        i | c | v
        ---+---+---
        (0 rows)
```

Note
If you use double quotes (") with echo , you must double the backslashes ().

-d --dbname

-d *db-name* or --dbname *db-name* specifies the name of the database to connect to. This is equivalent to specifying *db-name* as the first non-option argument on the command line.

-E

-E displays queries generated by internal commands.

-e --echo-queries

-e --echo-queries copies all SQL commands sent to the server to standard output as well. This is equivalent to setting the variable ECHO to queries .

-F --field-separator

-F *separator* or --field-separator *separator* specifies the field separator for unaligned output (default: "|") (-P fieldsep=). (See -A --no-align .) This is equivalent to \pset fieldsep or \f .

To set the field separator value to a control character, use your shell's control character escape notation. In Bash, you specify a control character in an argument using a dollar sign ($) followed by a string contained in single quotes. This string can contain C-string escapes (such as \t for tab), or a backslash () followed by an octal value for the character you want to use.

The following example demonstrates setting the separator character to tab (\t), vertical tab (\v) and the octal value of vertical tab (\013).

```
$ vsql -At -c "SELECT * FROM testtable;"
A|1|2|3
B|4|5|6

$ vsql -F $'\t' -At -c "SELECT * FROM testtable;"
A	1	2	3
B	4	5	6

$ vsql -F $'\v' -At -c "SELECT * FROM testtable;"
A
1
2
3
B
4
5
6
$ vsql -F $'\013' -At -c "SELECT * FROM testtable;"
A
1
2
3
B
4
5
6
```

-f --file

-f *filename* or --file *filename* uses *filename* as the source of commands instead of reading commands interactively. After the file is processed, vsql terminates.

If *filename* is a hyphen (-), standard input is read.

Using this option is different from writing vsql < *filename* . Using -f enables some additional features such as error messages with line numbers. Conversely, the variant using the shell's input redirection should always yield exactly the same output that you would have gotten had you entered everything manually.

? --help

-? --help displays help about vsql command line arguments and exits.

-H --html

-H --html turns on HTML tabular output. This is equivalent to \pset format html or the \H command.

-h --host

-h *hostname* or --host *hostname* specifies the host name of the machine on which the server is running. Use this flag to connect to Vertica remotely.

The following requirements and restrictions apply:

- If you use client authentication with a Kerberos connection method of either gss or krb5 , you must specify -h *hostname* .
- Use the -h option if you want to connect to Vertica from a local connection, but want to use the an authentication record with the access method HOST (rather than LOCAL).

-i -- timing

Enables the \timing meta-command. You can only use this command with the -c --command and -f --file commands:

```
$VSQL -h host1 -U user1 -d VMart -p 15 -w ****** -i -f transactions.sql
```

You can only use -i with the -c (command) and -f (filename) commands. For more information see Command-line options .

From the command line enter the -i option before running a session to turn timing on. For example:

```
$VSQL -h host1 -U user1 -d VMart -p 15 -w ****** -i -f transactions.sql
```

```
$VSQL-h host1 -U user1 -d VMart -p 15 -w ****** -i -c "SELECT user_name,
ssl_state, authentication_method, client_authentication_name, client_type FROM sessions
WHERE session_id=(SELECT session_id FROM current_session);"
```

-g --label

Assigns a client label to the connection at the start of the session. Client connections and their labels appear in the SESSIONS and some Data collector tables like DC_REQUESTS_ISSUED. Client labels set with this option appear in DC_SESSION_STARTS.

If used, this option takes precedence over the VSQL_CLIENT_LABEL environment variable.

To set client labels for ongoing sessions, use SET_CLIENT_LABEL .

-l --list

-l or --list returns all available databases, then exits. Other non-connection options are ignored. This command is similar to the internal command \list .

-m --sslmode

-m or --sslmode specifies the policy for making SSL connections to the server. Options are verify_full , verify_ca require , prefer , allow , and disable . You can also set the VSQL_SSLMODE variable to achieve the same effect. If the variable is set, the command-line option overrides it.

For information on these modes see Configuring TLS for ODBC Clients .

-n

-n disables command line editing.

-o --output

-o *filename* or --output *filename* writes all query output into file *filename* . This is equivalent to the vsql meta-command \o .

-P --pset

-P *assignment* or --pset *assignment* lets you specify printing options in the style of \pset on the command line. Note that you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write -P format=latex .

-p --port

-p *port* or --port *port* specifies the TCP port or the local socket file extension on which the server is listening for connections. Defaults to port 5433.

-q --quiet

-q or --quiet specifies that vsql do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this appears. This is useful with the -c option. Within vsql you can also set the QUIET variable to achieve the same effect.

-R --record-separator

-R *separator* or --record-separator *separator* specifies *separator* as the record separator. This is equivalent to the \pset recordsep command.

-S --single-line

-S --single-line runs in single-line mode where a newline terminates a SQL command, like the semicolon does.

> **Note**
> This mode is provided for those who insist on it, but you are not necessarily encouraged to use it, particularly if you mix SQL and meta-commands on a line. The order of execution might not always be clear to the inexperienced user.

-s --single-step

-s --single-step runs in single-step mode for debugging scripts. Forces vsql to prompt before each statement is sent to the database and allows you to cancel execution.

-T --table-attr

-T *table-options* or --table-attr *table-options* lets you specify options to be placed within the HTML table tag. See \pset for details.

-t --tuples-only

-t or --tuples-only disables printing of column names, result row count footers, and so on. This is equivalent to the vsql meta-command \t .

-V --version

-V or --version prints the vsql version and exits.

-v --variable --set

-v *assignment* , --variable *assignment* , and --set *assignment* perform a variable assignment, like the vsql meta-command \set .

> **Note**
> You must separate name and value, if any, by an equals sign (=) on the command line.

To unset a variable, omit the equal sign. To set a variable without a value, use the equals sign but omit the value. Make these assignments at a very early stage of start-up, so that variables reserved for internal purposes can get overwritten later.

-X --no-vsqlrc

-X --no-vsqlrc prevents the start-up file from being read: the system-wide vsqlrc file or the user's ~/.vsqlrc file.

-x --expanded

-x or --expanded enables extended table formatting mode. This is equivalent to the vsql meta-command \x .

Connecting from a non-cluster host

You can use the Vertica vsql executable image on a non-cluster Linux host to connect to a Vertica database.

- On Red Hat, CentOS, and SUSE systems, you can install the client driver RPM, which includes the vsql executable. See Installing the vsql client for details.
- If the non-cluster host is running the same version of Linux as the cluster, copy the image file to the remote system. For example:
  ```
  $ scp host01:/opt/vertica/bin/vsql .$ ./vsql
  ```
- If the non-cluster host is running a different distribution or version of Linux than your cluster hosts, you must install the Vertica server RPM in order to get vsql:
  1. Download the appropriate RPM package by browsing to Vertica website. On the **Support** tab, select **Customer Downloads** .
  2. If the system you used to download the RPM is not the non-cluster host, transfer the file to the non-cluster host.
  3. Log into the non-cluster host as root and install the RPM package using the command:
     ```
     # rpm -Uvh filename
     ```
     Where *filename* is the package you downloaded. Note that you do not have to run the install_vertica script on the non-cluster host to use vsql.

Notes

- Use the same Command-line options that you would on a cluster host.
- You cannot run vsql on a Cygwin bash shell (Windows). Use ssh to connect to a cluster host, then run vsql.

Meta-commands

Anything you enter in vsql that begins with an unquoted backslash is a vsql meta-command that is processed by vsql itself. These commands help make vsql more useful for administration or scripting. Meta-commands are more commonly called slash or backslash commands.

The format of a vsql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of whitespace characters.

To include whitespace into an argument you can quote it with a single quote. To include a single quote into such an argument, precede it by a backslash. Anything contained in single quotes is furthermore subject to C-like substitutions for \n (new line), \t (tab), \ *digits* , \0 *digits* , and \0x *digits* (the character with the given decimal, octal, or hexadecimal code).

If an unquoted argument begins with a colon (:), it is taken as a vsql variable and the value of the variable is used as the argument instead.

Arguments that are enclosed in backquotes (```) are taken as a command line that is passed to the shell. The output of the command (with any trailing newline removed) is taken as the argument value. The above escape sequences also apply in backquotes.

Some commands take a SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL: Unquoted letters are forced to lowercase, while double quotes (") protect letters from case conversion and allow incorporation of whitespace into the identifier. Within double quotes, paired double quotes reduce to a single double quote in the resulting name. For example, FOO"BAR"BAZ is interpreted as fooBARbaz , and "A weird"" name" becomes A weird" name .

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence \\ (two backslashes) marks the end of arguments and continues parsing SQL commands, if any. That way SQL and vsql commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

In this section

- Meta-commands quick reference
- \connect
- \d meta-commands
- \edit
- \i
- \locale
- \pset
- \set
- \timing

Meta-commands quick reference

Syntax	**Summary**	**Command-line Options**
\! [*cmd*]	Executes a command in a Linux shell (passing arguments as entered) or starts an interactive shell.	
\?	Displays help information about all meta-commands, the same as \h .	?
\a	Toggles output format alignment. For details, see \pset format aligned .	-A -no-align
\b	Toggles beep on command completion.	
\c[onnect] [*db* [*user-name*]]	Establishes a connection to database *db* , under the specified user *user-name* . For details, see \connect .	
\C [' *title-str* ']	Sets a title *title-str* that precedes query result output. For details, see \pset title *title-str* .	
\cd [*dir*]	Changes the current working directory to *dir* , changes to your home directory if you omit specifying a directory.	
\d commands	See \d meta-commands	
\e[dit] [*file*]	Edits the query buffer (or specified file) with an external editor. For details, see \edit .	

\echo [*str*]	Writes *str* to standard output. **Tip** Use \qecho to redirect query output to the query output stream, as set by \o .	
\f [*str*]	Sets the field separator for unaligned query output. The default is the vertical bar (|).	-F --field-separator
\g [*file-name* | *shell-command*]	Sends the query in the input buffer (see \p) to the server. You can send query results to *file-name* , or pipe results to *shell-comand* ; otherwise, \g sends query results to standard output.	
\H	Renders output in HTML markup as a table. For details, see \pset format aligned .	-H --html
\h[elp]	Displays help information about the meta-commands, the same as \? .	--help
\i *file*	Reads and executes input from filename.	-f --file
\l \list	Lists available databases and owners.	-l --list
\locale [*locale*]	Displays the current locale setting or sets a new locale for the session. For details, see \locale .	
\o [*file-name* | *shell-command*]	Controls where vsql directs query output. You can send query results to *file-name* , or pipe results to *shell-comand* ; otherwise, \o sends query results to standard output.	-o --output
\p	Prints the current query buffer to standard output.	
\password [*user-name*]	Starts the password change process. Superusers can specify a user name to change that user's password; otherwise, users can only change their own passwords.	
\pset *output-option*	Sets options that control how Vertica formats query result output. For details, see \pset .	-P --pset
\q	Quits the vsql program	
\qecho [*str*]	Writes *str* to the query output stream, as specified by by \o .	
\r	Clears (resets) the query buffer	
\s [*file*]	Valid only if vsql is configured to use the GNU Readline library, prints or saves the command line history to *file* , or to standard output if no file name is supplied.	
\set [*var* [*value*]...]	Sets internal variable *var* to *value* . If you specify multiple values, var is set to their concantenated values. If no values are specified, *var* is set to no value.	--set -v --variable
\t	Toggles between tuples only and full display. For details, see \pset format tuples_only .	-t --tuples-only
\T *html-attribute* [...]	Specifies attributes to be placed inside the HTML table tag—for example, cellpadding or bgcolor , the same as \pset tableattr *html-attribute* [...] . For sample usage, see Output formatting examples .	-T --table-attr

\timing	If set to on, returns how long (in milliseconds) each SQL statement runs. For details, see \timing .	-i -- timing
\unset *var*	Deletes internal variable *var* that was set by the meta-command \set .	
\w *file-name*	Outputs the current query buffer to file *file-name* .	
\x	Toggles between regular and expanded format. For details, see \pset format expanded .	-x --expanded

\z	Returns a summary of privileges on all objects in system table V_CATALOG.GRANTS : grantee, grantor, privileges, schema, and object name (equivalent to \dp).
	
	\z supports the same options as \dp for filtering output by schema and object name patterns, . For example:
	
	`> \z *.*myseq*`
	`  Access privileges for database "dbadmin"`
	`Grantee | Grantor | Privileges | Schema | Name`
	`---------+---------+------------+--------+--------`
	`dbadmin | dbadmin | SELECT* | public | mySeq`
	`dbadmin | dbadmin | SELECT* | public | mySeq2`
	`(2 rows)`	

\connect

Establishes a connection to database *db* , under the specified user *user-name* . The previous connection is closed. If you omit specifying a database name, Vertica connects to the current database. If you omit specifying a user name argument, Vertica assumes the current user.

Syntax

```
\c[onnect] [db [user-name]]
```

Error handling

Errors that prevent execution include specifying an unknown user and denial of access to the specified database. Vertica handles errors differently, depending on whether this command is executed interactively in vsql, or in a script:

- VSQL handling: The current connection is maintained.
- Script: Processing immediately stops with an error. This is prevents scripts from acting on the wrong database.

\d meta-commands

Vertica supports a number of \d commands, which return information on different categories of database objects. For a full list, see \d Reference below.

Syntax

Unless otherwise noted, \d commands generally conform to the following syntax:

```
\dCommand [ [schema.]pattern ]
```

Arguments

You can supply most \d commands with a string pattern argument, which filters the results that the command returns. The pattern can optionally be qualified by a schema name.

schema

Valid for most \d commands, restricts output to only database objects in *schema* . For example, the following \dp command obtains privileges information for all V_MONITOR tables that contain the string resource :

```
=> \dp V_MONITOR.*resource*
                Access privileges for database "dbadmin"
 Grantee | Grantor | Privileges |  Schema   |            Name
---------+---------+------------+-----------+----------------------------
 public  | dbadmin | SELECT     | v_monitor | resource_rejections
 public  | dbadmin | SELECT     | v_monitor | disk_resource_rejections
 public  | dbadmin | SELECT     | v_monitor | resource_usage
 public  | dbadmin | SELECT     | v_monitor | resource_acquisitions
 public  | dbadmin | SELECT     | v_monitor | resource_rejection_details
 public  | dbadmin | SELECT     | v_monitor | resource_pool_move
 public  | dbadmin | SELECT     | v_monitor | host_resources
 public  | dbadmin | SELECT     | v_monitor | node_resources
 public  | dbadmin | SELECT     | v_monitor | resource_queues
 public  | dbadmin | SELECT     | v_monitor | resource_pool_status
(10 rows)
```

pattern

Returns only the database objects that match the specified string. Pattern strings can include the following wildcards:

- * (asterisk): zero or more characters.
- ? (question mark): any single character.

For example, the following \dt command returns tables that start with the string store :

```
=> \dt store*
                  List of tables
 Schema |       Name        | Kind  |  Owner  | Comment
--------+-------------------+-------+---------+---------
 public | store_orders      | table | dbadmin |
 public | store_orders_2018 | table | dbadmin |
 public | store_overseas    | table | dbadmin |
 store  | store_dimension   | table | dbadmin |
 store  | store_orders_fact | table | dbadmin |
 store  | store_sales_fact  | table | dbadmin |
(6 rows)
```

\d reference

\d

Unqualified by a pattern argument, returns all tables with their schema names, owners, and comments. If qualified by a pattern argument, \d returns all matching tables and all columns in each table, with details about each column, such as data type, size, and default value.

\df

Returns all function names, the function return data type, and the function argument data type. Also returns the procedure names and arguments for all procedures that are available to the user.

\dj

Returns all projections showing the schema, projection name, owner, and node. The returned rows include superprojections, live aggregate projections, Top-K projections, and projections with expressions.

\dn

Returns the schema names and schema owner.

\dp

Returns a summary of privileges on all objects in system table V_CATALOG.GRANTS : grantee, grantor, privileges, schema, and object name (equivalent to \z).

\dS

Unqualified by a pattern argument, returns all V_CATALOG and V_MONITOR system tables. To obtain system tables for just one schema, qualify the command with the schema name, as follows:

```
\dS { V_CATALOG | V_MONITOR }.*
```

\ds

Returns sequences and their parameters.

\dT

Returns all data types that Vertica supports.

> **Note**
> \dT returns no results if qualified with a pattern argument.

\dt

Unqualified by a pattern argument, returns the same information as an unqualified \d command. If qualified by a pattern argument, \dt returns matching tables with the same level of detail as an unqualified \dt command.

\dtv

Returns tables and views.

\du

Returns database users and whether they are superusers.

\dv

Unqualified by a pattern argument, returns all views with their schema names, owners, and comments. If qualified by a pattern argument, \dv returns all matching views and the columns in each view, with each column's data type and size.

\edit

Edits the query buffer (or specified file) with an external editor. When the editor exits, its content is copied back to the query buffer. If no argument is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

The new query buffer is then re-parsed according to the normal rules of vsql, where the whole buffer up to the first semicolon is treated as a single line. (Thus you cannot make scripts this way. Use \i for that.) If there is no semicolon, vsql waits for one to be entered (it does not execute the query buffer).

> **Tip**
> vsql searches the environment variables VSQL_EDITOR, EDITOR, and VISUAL (in that order) for an editor to use. If all of them are unset, vi is used on Linux systems, notepad.exe on Windows systems.

Syntax

```
\e[dit] [ file ]
```

\i

Reads and executes input from the specified file.

> **Note**
> To see the lines on the screen as they are read, set the variable ECHO to all.

Syntax

```
\i filename
```

Examples

The Vertica vsql client on Linux supports backquote (backtick) expansion. For example:

1. Set an environment variable to a path that contains scripts you want to run:
   ```
   $ export MYSCRIPTS=/home/dbadmin/testscripts
   ```
2. Issue the vsql command.
   ```
   $ vsql
   ```
3. Use backquote expansion to include the path for running an existing script—for example, sample.sql .
   ```
   => \i `echo $MYSCRIPTS/sample.sql`
   ```

\locale

Displays or sets the locale setting for the current session.

> **Note**
> This command does not alter the default locale for all database sessions. To change the default for all sessions, set configuration parameter DefaultSessionLocale .

Syntax

```
\locale [locale-identifier]
```

Arguments

locale-identifier

Specifies the ICU locale identifier to use, by default set to:

```
en_US@collation=binary
```

If set to an empty string, Vertica sets locale to en_US_POSIX .

If you omit this argument, \locale returns the current locale setting.

For details on identifier options, see About locale . For a complete list of locale identifiers, see the ICU Project .

Examples

View the current locale setting:

```
=> \locale
en_US@collation=binary
```

Change the default locale for this session:

```
=> \locale en_GBINFO:
INFO 2567:  Canonical locale: 'en_GBINFO:'
Standard collation: 'LEN'
English (GBINFO:)
```

Notes

The server locale settings impact only the collation behavior for server-side query processing. The client application is responsible for ensuring that the correct locale is set in order to display the characters correctly. Below are the best practices recommended by Vertica to ensure predictable results:

- The locale setting in the terminal emulator for vsql (POSIX) should be set to be equivalent to session locale setting on server side (ICU) so data is collated correctly on the server and displayed correctly on the client.
- The vsql locale should be set using the POSIX LANG environment variable in terminal emulator. Refer to the documentation of your terminal emulator for how to set locale.
- Server session locale should be set using the set as described in Specify the default locale for the database .
- All input data for vsql should be in UTF-8 and all output data is encoded in UTF-8.
- Non UTF-8 encodings and associated locale values are not supported.

\pset

Sets options that control how Vertica formats query result output.

Syntax

```
\pset output-option
```

Output options

> **Note**
> Unless otherwise specified, output options are valid for all formats.

format *format-option*

Sets output format, where *format-option* is one of the following:

- u[naligned] writes all column data of each row on a single line, where each field is separated only by the current separator character. Use this output for use as input to other programs—for example, comma-delimited fields for CSV input.
- a[ligned] (default): Renders column-aligned output.
- h[tml] : Renders output in HTML markup as a table.
- l[atex] : Renders output in LaTex markup.

border *int*

Valid only if output format is set to html , specifies the table border, where *int* specifies the border type.

expanded

Toggles between regular and expanded format. When expanded format is enabled, all output has two columns with the column name on the left and the data on the right. This mode is especially useful for wide tables.

fieldsep ' *arg* '

Valid only if output format is set to unaligned , specifies the field separator, by default | (vertical bar).

For example, to specify tab as the field separator:

```
\pset fieldsep '\t'
```

footer

Toggles display of the default footer:

(*int* rows)

null ' *string* '

Specifies to represent column null values as *string* . By default, Vertica renders null values as an empty field, which might be mistaken as an empty string.

For example:

```
\pset null '(null)'
```

pager [*always*]

Toggles use of a pager for query and vsql help output. If the environment variable PAGER is set, the output is piped to the specified program. Otherwise a platform-dependent default (such as more) is used.

When the pager is off, the pager is not used. When the pager is on, the pager is used only when appropriate; that is, the output is to a terminal and does not fit on the screen. (vsql does not do a perfect job of estimating when to use the pager.)
If qualified with the argument always, the pager is always used.

recordsep ' *char* '

Valid only if output format is set to unaligned , specifies the character used to delimit table records (tuples), by default a newline character.

tableattr *html-attribute* [...]

Specifies attributes to be placed inside the HTML table tag—for example, cellpadding or bgcolor .

title [' *title-str* ']

Sets a title that precedes query result output, to *title-str* . HTML output renders this as follows:

```
<caption>title-str</caption>
```

To remove the title, reissue the command omit the *title-str* argument.

trailingrecordsep

Toggles on or off the trailing record separator to use in unaligned output mode.

t[uples_only]

Toggles between tuples only and full display. Full display might show extra information such as column headers, titles, and various footers. In tuples only mode, only actual table data is shown.

Shortcuts

The following \pset commands have short-cuts:

\pset expanded

```
\x
```

\pset fieldsep ' *arg* '

\f

\pset format aligned

\a

\pset format html

\H

\pset tableattr *html-attribute* [...]

\T *html-attribute*[...]

\pset title *title-str*

\C ['*title-str*']

\pset tuples_only

\t

Examples

See Output formatting examples .

\set

S ets an internal variable to one or more values. If multiple values are specified, they are concantenated. An unqualified \set command lists all internal variables.

To unset a variable, use vsql meta-command \unset .

Syntax

\set [*var* [*value*]...]

Arguments

var

The name of an internal variable to set. Valid variable names are case sensitive and can contain characters, digits, and underscores. vsql treats several variables as special, which are described in Variables .

value

A value to set in variable *var* . If no value is specified, the variable is set to no value.

If set to an empty string, the variable is set to no value. If you omit this argument, \set returns all internal variables.

If no arguments are supplied, \set returns all internal variables. For example:

```
=> \set
VERSION = 'vsql'
AUTOCOMMIT = 'off'
VERBOSITY = 'default'
PROMPT1 = '%/%R%# '
PROMPT2 = '%/%R%# '
PROMPT3 = '>> '
ROWS_AT_A_TIME = '1000'
DBNAME = 'dbadmin'
USER = 'dbadmin'
PORT = '5433'
LOCALE = 'en_US@collation=binary'
HISTSIZE = '500'
```

\timing

If set to on, returns how long (in milliseconds) each SQL statement runs. Results include:

- Length of time required to fetch the first block of rows
- Total time until the last block is formatted.

Unqualified, \timing toggles timing on and off. You can explicitly turn timing on and off by qualifying the command with options ON and OFF , respectively.

> **Note**
> You can also enable \timing from the command line using the vsql -i command.

Syntax

```
\timing [ON | OFF]
```

Examples

The following unqualified \timing commands toggle timing on and off:

```
=> \timing
Timing is on
=> \timing
Timing is off
```

The following example shows a SQL command with timing on:

```
=> \timing
Timing is on.
=> SELECT user_name, ssl_state, authentication_method, client_authentication_name,
    client_type FROM sessions WHERE session_id=(SELECT session_id FROM current_session);
 user_name | ssl_state | authentication_method | client_authentication_name | client_type
-----------+-----------+-----------------------+----------------------------+-------------
 dbadmin   | None      | ImpTrust              | default: Implicit Trust    | vsql
(1 row)

Time: First fetch (1 row): 73.684 ms. All rows formatted: 73.770 ms
```

Variables

vsql provides variable substitution features similar to common Linux command shells. Variables are name/value pairs, where the value can be a string of any length. To set variables, use the vsql meta-command \set . For example, the following statement sets the variable fact to the value dim :

```
=> \set fact dim
```

If you call \set on a variable and supply no value, the variable is set to an empty string.

> **Note**
> The arguments of \set are subject to the same substitution rules as with other commands. For example, \set dim :fact is a valid way to copy a variable.

Getting variables

To retrieve the content of a given variable, precede the name with a colon and use it as the argument of any slash command. For example:

```
=> \echo :fact
dim
```

An unqualified \set command returns all current variables and their values:

```
dbadmin=> \set
VERSION = 'vsql'
AUTOCOMMIT = 'off'
VERBOSITY = 'default'
PROMPT1 = '%/%R%# '
PROMPT2 = '%/%R%# '
PROMPT3 = '>> '
ROWS_AT_A_TIME = '1000'
DBNAME = 'dbadmin'
USER = 'dbadmin'
PORT = '5433'
LOCALE = 'en_US@collation=binary'
HISTSIZE = '500'
```

Deleting variables

To unset (or delete) a variable, use the vsql meta-command \unset .

Variable naming conventions

vsql internal variable names can contain letters, numbers, and underscores in any order and any number. Some variables are treated specially by vsql. They indicate certain option settings that can be changed at run time by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid using such variable names for your own purposes.

SQL interpolation

You can substitute ("interpolate") vsql variables into regular SQL statements. You do so by prepending the variable name with a colon (:). For example, the following statements query the table my_table :

```
=> \set fact 'my_table'
=> SELECT * FROM :fact;
```

The value of the variable is copied literally, so it can even contain unbalanced quotes or backslash commands. Make sure that it makes sense where you put it. Variable interpolation is not performed into quoted SQL entities. One exception applies: the contents of backquoted strings (````) are passed to a system shell, and replaced with the shell's output. See Using Backquotes to Read System Variables below.

Using backquotes to read system variables

In vsql, the contents of backquotes are passed to the system shell to be interpreted (the same behavior as many UNIX shells). This is particularly useful in setting internal vsql variables, since you may want to access UNIX system variables (such as HOME or TMPDIR) rather than hard-code values.

For example, to set an internal variable to the full path for a file in your UNIX user directory, you can use backquotes to get the content of the system HOME variable, which is the full path to your user directory:

```
=> \set inputfile `echo $HOME`/myinput.txt=> \echo :inputfile
/home/dbadmin/myinput.txt
```

The contents of the backquotes are replaced with the results of running the contents in a system shell interpreter. In this case, the echo $HOME command returns the contents of the HOME system variable.

In this section

- DBNAME
- ECHO
- ECHO_HIDDEN
- ENCODING
- HISTCONTROL
- HISTSIZE
- HOST
- IGNOREEOF
- ON_ERROR_STOP
- PORT
- PROMPT1 PROMPT2 PROMPT3
- QUIET
- ROWS_AT_A_TIME
- SINGLELINE

- SINGLESTEP
- USER
- VERBOSITY
- VSQL_HOME
- VSQL_SSLMODE

DBNAME

The name of the database to which you are currently connected. DBNAME is set every time you connect to a database (including program startup), but it can be unset.

ECHO

If set to all , all lines entered from the keyboard or from a script are written to the standard output before they are parsed or run.

To select this behavior on program start-up, use the switch -a . If set to queries , vsql merely prints all queries as they are sent to the server. The switch for this is -e .

ECHO_HIDDEN

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the Vertica internals and provide similar functionality in your own programs. (To select this behavior on program start-up, use the switch -E .)

If you set the variable to the value noexec , the queries are just shown but are not actually sent to the server and run.

ENCODING

The current client character set encoding.

HISTCONTROL

If this variable is set to ignorespace , lines that begin with a space are not entered into the history list. If set to a value of ignoredups , lines matching the previous history line are not entered. A value of ignoreboth combines the two options. If unset, or if set to any other value than those previously mentioned, all lines read in interactive mode are saved on the history list.

Source: Bash.

HISTSIZE

Specifies how much storage space is allocated to store the history of SQL statements issued in the current vsql session. vsql uses this setting, by default 500, to calculate the size of the history buffer:

```
HISTSIZE * 50 (bytes)
```

where 50 bytes approximates the average length of a SQL statement. The actual length of SQL statements in the current session determines how many statements vsql stores.

HISTSIZE has no effect on the history that is stored in .vsql_history .

Source: Bash

HOST

The database server host you are currently connected to. This is set every time you connect to a database (including program startup), but can be unset.

IGNOREEOF

If unset, sending an EOF character (usually Control+D) to an interactive session of vsql terminates the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

Source: Bash.

ON_ERROR_STOP

By default, if a script command results in an error, for example, because of a malformed command or invalid data format, processing continues. If you set ON_ERROR_STOP to ON in a script and an error occurs during processing, the script terminates immediately.

For example:

```
=> \set ON_ERROR_STOP ON
```

> **Note**
> If you invoke the script on Linux with vsql -f , vsql returns with error code 3 to indicate that an error occurred in the script.

PORT

The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be unset.

PROMPT1 PROMPT2 PROMPT3

These specify what the prompts vsql issues look like. See Prompting for details.

QUIET

This variable is equivalent to the command line option -q . It is probably not too useful in interactive mode.

ROWS_AT_A_TIME

ROWS_AT_A_TIME is set by default to 1000, and retrieves results as blocks of rows of that size. The column formatting for the first block is used for all blocks, so in later blocks some entries could overflow.

When formatting results, Vertica buffers ROWS_AT_A_TIME rows in memory to calculate the maximum column widths. It is possible that rows after this initial fetch are not properly aligned if any of the field values are longer than those see in the first ROWS_AT_A_TIME rows. ROWS_AT_A_TIME can be unset with vsql meta-command \unset to guarantee perfect alignment. However, this requires re-buffering the entire result set in memory and might cause vsql to fail if the result set is too big.

SINGLELINE

This variable is equivalent to the command line option -S .

SINGLESTEP

This variable is equivalent to the command line option -s .

USER

The database user you are currently connected as. This is set every time you connect to a database (including program startup), but can be unset.

VERBOSITY

This variable can be set to the values default , verbose , or terse to control the verbosity of error reports.

VSQL_HOME

By default, the vsql program reads configuration files from the user's home directory. In cases where this is not desirable, the configuration file location can be overridden by setting the VSQL_HOME environment variable in a way that does not require modifying a shared resource.

In the following example, vsql reads configuration information out of /tmp/jsmith rather than out of ~.

```
# Make an alternate configuration file in /tmp/jsmith
mkdir -p /tmp/jsmith
echo "\\echo Using VSQLRC in tmp/jsmith" > /tmp/jsmith/.vsqlrc
# Note that nothing is echoed when invoked normally
vsql
# Note that the .vsqlrc is read and the following is
# displayed before the vsql prompt
#
# Using VSQLRC in tmp/jsmith
VSQL_HOME=/tmp/jsmith vsql
```

VSQL_SSLMODE

VSQL_SSLMODE specifies how (or whether) clients (like admintools) use SSL when connecting to servers. The default value is prefer , meaning to use SSL if the server offers it. Legal values are require , prefer , allow , and disable . This variable is equivalent to the command-line -m option (or --sslmode).

Prompting

The prompts vsql issues can be customized to your preference. The three variables PROMPT1 , PROMPT2 , and PROMPT3 contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when vsql requests a new command. Prompt 2 is issued when more input is expected during command input because the command was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you run a SQL COPY command and you are expected to type in the row values on the terminal.

The value of the selected prompt variable is printed literally, except where a percent sign (%) is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

%M

The full host name (with domain name) of the database server, or [local] if the connection is over a socket, or [local:/dir/name], if the socket is not at the compiled in default location.

%m

The host name of the database server, truncated at the first dot, or [local].

%>

The port number at which the database server is listening.

%n

The database session user name.

%/

The name of the current database.

%~

Like %/, but the output is ~ (tilde) if the database is your default database.

%#

If the session user is a database superuser, then a #, otherwise a >. (The expansion of this value might change during a database session as the result of the command SET SESSION AUTHORIZATION.)

%R

In prompt 1 normally =, but ^ if in single-line mode, and ! if the session is disconnected from the database (which can happen if \connect fails). In prompt 2 the sequence is replaced by -, *, a single quote, a double quote, or a dollar sign, depending on whether vsql expects more input because the command wasn't terminated yet, because you are inside a /* ... */ comment, or because you are inside a quoted or dollar-escaped string. In prompt 3 the sequence doesn't produce anything.

%x

Transaction status: an empty string when not in a transaction block, or * when in a transaction block, or ! when in a failed transaction block, or ? when the transaction state is indeterminate (for example, because there is no connection).

%digits

The character with the indicated numeric code is substituted. If digits starts with 0x the rest of the characters are interpreted as hexadecimal; otherwise if the first digit is 0 the digits are interpreted as octal; otherwise the digits are read as a decimal number.

%:name:

The value of the vsql variable name. See the section Variables for details.

%`command`

The output of command, similar to ordinary "back- tick" substitution.

%[... %]

Prompts may contain terminal control characters which, for example, change the color, background, or style of the prompt text, or change the title of the terminal window. In order for the line editing features of Readline to work properly, these non-printing control characters must be designated as invisible by surrounding them with %[and %]. Multiple pairs of these may occur within the prompt. The following example results in a boldfaced (1;) yellow-on-black (33;40) prompt on VT100-compatible, color-capable terminals:

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%#%[%033[0m%] '
```

To insert a percent sign into your prompt, write %%. The default prompts are '%/%R%# ' for prompts 1 and 2, and '>> ' for prompt 3.

Note: See the specification for terminal control sequences (applicable to gnome-terminal and xterm).

Command line editing

vsql supports the tecla library for line editing and retrieval. You can define a tecla configuration with the following files:

- ~/.vsqlrc (user)
- /opt/vertica/config/vsqlrc (global)

For details, see the tecla documentation .

Command history is automatically saved in ~/.vsql_history when vsql exits and is reloaded when vsql starts.

Disabling tab completion

To disable tab completion, add the following to .vsqlrc :

```
\bind ^I
```

Key bindings

Key bindings are read from a global configuration at /opt/vertica/config/vsqlrc , if present. To override key bindings, add definitions to ~/.vsqlrc .

Key bindings must be prefixed with a backslash (\ ). For example, the following definition binds the "backward-word" action to Ctrl+B:

```
\bind ^B backward-word
```

The following key bindings are specific to vsql :

- Insert switches between insert mode (the default) and overwrite mode.
- Delete deletes the character to the right of the cursor.
- Home moves the cursor to the front of the line.
- End moves the cursor to the end of the line.
- ^R Performs a history backwards search.

Implementation differences

The vsql implementation of the tecla library deviates from the tecla documentation in the following ways:

- Unlike the standard tecla library, which saves all executed lines in the command history, vsql only saves unique non-empty lines.
- vsql standardizes the name and location of the history file (~/.vsql_history).
- vsql does not support 8-bit meta characters. This can affect international character sets, meta keys, and locales. You can verify that a meta character sends an escape by setting the EightBitInput X resource to False . You can do this in the following ways:
  - Add the following to ~/.Xdefaults :
    ```
    XTerm*EightBitInput: False
    ```
  - Start an xterm session with the -xrm '*EightBitInput: False' .

vsql environment variables

Set one or more of the following environment variables to be used by the defined properties automatically, each time you start vsql:

PAGER

If the query results do not fit on the screen, they are piped through this command. Typical values are more or less . The default is platform-dependent. Use the \pset command to enable/disable the pager.

VSQL_CLIENT_LABEL

The label to identify the vsql client in various system tables like SESSIONS . This is an alternative to setting the client label with the --label option or SET_CLIENT_LABEL , but if either of these is used, they take precedence over VSQL_CLIENT_LABEL .

VSQL_DATABASE

The database to which you are connecting. For example, VMart .

TMPDIR

Directory for storing temporary files. The default is platform-dependent. On Unix-like systems the default is /tmp .

VSQL_EDITOR

EDITOR

VISUAL

Editor used by the \e command. The variables are examined in the order listed; the first that is set is used.

VSQL_HOME

By default, the vsql program reads configuration files from the user's home directory. In cases where this is not desirable, the configuration file location can be overridden by setting the VSQL_HOME environment variable in a way that does not require modifying a shared resource.

VSQL_HOST

Host name or IP address of the Vertica node.

VSQL_PASSWORD

The database password. Using this environment variable increases site security by precluding the need to enter the database password on the command line.

VSQL_PORT

Port to use for the connection.

VSQL_SSLMODE

Specifies whether and how clients such as admintools use SSL when connecting to servers.

VSQL_USER

User name to use for the connection.

Locales

The default terminal emulator under Linux is gnome-terminal, although xterm can also be used.

Vertica recommends that you use gnome-terminal with vsql in UTF-8 mode, which is its default.

To change settings on Linux

1. From the tabs at the top of the vsql screen, select Terminal.
2. Click **Set Character Encoding** .
3. Select **Unicode (UTF-8)** .

> **Note**
> This works well for standard keyboards. xterm has a similar UTF-8 option.

To change settings on Windows using PuTTy

1. Right click the vsql screen title bar and select **Change Settings** .
2. Click **Window** and click **Translation** .
3. Select **UTF-8** in the drop-down menu on the right.

Notes

- vsql has no way of knowing how you have set your terminal emulator options.
- The tecla library is prepared to do POSIX-type translations from a local encoding to UTF-8 on interactive input, using the POSIX LANG, etc., environment variables. This could be useful to international users who have a non-UTF-8 keyboard. See the tecla documentation for details. Vertica recommends the following (or whatever other .UTF-8 locale setting you find appropriate):

```
export LANG=en_US.UTF-8
```

- The vsql \locale command invokes and tracks the server SET LOCALE TO command, described. vsql itself currently does nothing with this locale setting, but rather treats its input (from files or from tecla), all its output, and all its interactions with the server as UTF-8. vsql ignores the POSIX locale variables, except for any "automatic" uses in printf , and so on.

Entering data with vsql

You often need to insert literal data when using vsql. For example:

- Adding a row of data to a table using an INSERT statement.
- Adding multiple rows of data through a COPY FROM STDIN statement.

The following table lists the data types that Vertica supports, and the format you use to enter that data in queries when using vsql.

Data Type	**Inserting to vsql using**	**Example Use in INSERT INTO *table* ...**	**For More Information See...**
Binary types, such as BINARY and VARBINARY	Helper functions such as HEX_TO_BINARY, octal strings, specified data format in COPY statements, casting string values to binary.	VALUES(HEX_TO_BINARY('0x3D'), '\\141\\337\\');	• Binary data types (BINARY and VARBINARY) • Binary (native) data
BOOLEAN	Literal values TRUE and FALSE or strings such as 'y' , 't' , 'true' , or 'false' .	VALUES(TRUE, 'f');	Boolean data type
Character data types such as CHAR or LONG VARCHAR	Strings enclosed in single quotes.	VALUES('my string');	Character data types (CHAR and VARCHAR)
Date and time data types, such as TIMESTAMPTZ	Formatted text string	VALUES('16:43:00', '2016-09-15 04:55:00 PDT');	• Date/time data types • Date/time expressions
Numeric Data Types	Literal numeric values, including scientific notation, hexadecimal, and BINARY scaling.	VALUES(3.1415, 42, 6.0221409e23);	Numeric data types
UUID	Formatted text string	VALUES('12345678-1234-1234-1234-123456789012');	UUID data type

Files

Before starting up, vsql attempts to read and execute commands from the system-wide vsqlrc file and the user's ~/.vsqlrc file. The command-line history is stored in the file ~/.vsql_history.

> **Tip**
> If you want to save your old history file, open another terminal window and save a copy to a different file name.

Exporting data using vsql

You can use vsql for simple data-export tasks by changing its output format options so the output is suitable for importing into other systems (tab-delimited or comma-separated files, for example). These options can be set either from within an interactive vsql session, or through command-line arguments to the vsql command (making the export process suitable for automation through scripting). After you have set vsql's options so it outputs the data in a format your target system can read, you run a query and capture the result in a text file.

The following table lists the meta-commands and command-line options that are useful for changing the format of vsql's output.

Description	Meta-command	Command-line Option
Disable padding used to align output.	\a	-A or --no-align
Show only tuples, disabling column headings and row counts.	\t	-t or --tuples-only
Set the field separator character.	\pset fieldsep	-F or --field-separator
Send output to a file.	\o	-o or --output
Specify a SQL statement to execute.	N/A	-c or --command

The following example demonstrates disabling padding and column headers in the output, and setting a field separator to dump a table to a tab-separated text file within an interactive session.

```
=> SELECT * FROM my_table;
 a |   b   | c
---+-------+---
 a | one   | 1
 b | two   | 2
 c | three | 3
 d | four  | 4
 e | five  | 5
(5 rows)
=> \a
Output format is unaligned.
=> \t
Showing only tuples.
=> \pset fieldsep '\t'
Field separator is "    ".
=> \o dumpfile.txt
=> select * from my_table;
=> \o
=> \! cat dumpfile.txt
a       one     1
b       two     2
c       three   3
d       four    4
e       five    5
```

Note

You could encounter issues with empty strings being converted to NULLs or the reverse using this technique. You can prevent any confusion by explicitly setting null values to output a unique string such as NULLNULLNULL (for example, \pset null 'NULLNULLNULL'). Then, on the import end, convert the unique string back to a null value. For example, if you are copying the file back into a Vertica database, you would give the argument NULL 'NULLNULLNULL' to the COPY statement.

When logged into one of the database nodes, you can create the same output file directly from the command line by passing the right parameters to vsql:

```
$ vsql -U username -F $'\t' -At -o dumpfile.txt -c "SELECT * FROM my_table;"
Password:
$ cat dumpfile.txt
a       one     1
b       two     2
c       three   3
d       four    4
e       five    5
```

If you want to convert null values to a unique string as mentioned earlier, you can add the argument -P null='NULLNULLNULL' (or whatever unique string you choose).

By adding the -w vsql command-line option to the example command line, you could use the command within a batch script to automate the data export. However, the script would contain the database password as plain text. If you take this approach, you should prevent unauthorized access to the batch script, and also have the script use a database user account that has limited access.

To set the field separator value to a control character, use your shell's control character escape notation. In Bash, you specify a control character in an argument using a dollar sign ($) followed by a string contained in single quotes. This string can contain C-string escapes (such as \t for tab), or a backslash () followed by an octal value for the character you want to use.

The following example demonstrates setting the separator character to tab (\t), vertical tab (\v) and the octal value of vertical tab (\013).

```
$ vsql -At -c "SELECT * FROM testtable;"
A|1|2|3
B|4|5|6

$ vsql -F $'\t' -At -c "SELECT * FROM testtable;"
A	1	2	3
B	4	5	6

$ vsql -F $'\v' -At -c "SELECT * FROM testtable;"
A
1
2
3
B
4
5
6
$ vsql -F $'\013' -At -c "SELECT * FROM testtable;"
A
1
2
3
B
4
5
6
```

Copying data using vsql

You can use vsql to copy data between two Vertica databases. This technique is similar to the technique explained in Exporting data using vsql , except instead of having vsql save data to a file for export, you pipe one vsql's output to the input of another vsql command that runs a COPY statement from STDIN. This technique can also work for other databases or applications that accept data from an input stream.

> **Note**
> The following technique only works for individual tables. To copy an entire database to another cluster, see Copying the database to another cluster .

The easiest way to copy using vsql is to log in to a node of the target database, then issue a vsql command that connects to the source Vertica database to dump the data you want. For example, the following command copies the store.store_sales_fact table from the vmart database on node testdb01 to the vmart database on the node you are logged into:

```
vsql -U username -w passwd -h testdb01 -d vmart -At -c "SELECT * from store.store_sales_fact"  \
| vsql -U username -w passwd -d vmart -c "COPY store.store_sales_fact FROM STDIN DELIMITER '|';"
```

> **Note**
> The above example copies the data only, not the table design. The target table for the data copy must already exist in the target database. You can export the design of the table using EXPORT_OBJECTS or EXPORT_CATALOG .

If you are using the Bash shell, you can escape special delimiter characters. For example, DELIMITER E'\t' specifies tab. Shells other than Bash may have other string-literal syntax.

Monitoring progress (optional)

You may want some way of monitoring progress when copying large amounts of data between Vertica databases. One way of monitoring the progress of the copy operation is to use a utility such as Pipe Viewer that pipes its input directly to its output while displaying the amount and speed of data it passes along. Pipe Viewer can even display a progress bar if you give it the total number of bytes or lines you expect to be processed. You can get the number of lines to be processed by running a separate vsql command that executes a SELECT COUNT query.

Note
Pipe Viewer isn't a standard Linux command, so you will need to download and install it yourself. See the Pipe Viewer page for download packages and instructions. Vertica does not support Pipe Viewer. Install and use it at your own risk.

The following command demonstrates how you can use Pipe Viewer to monitor the progress of the copy shown in the prior example. The command is complicated by the need to get the number of rows that will be copied, which is done using a separate vsql command within a Bash backquote string, which executes the string's contents and inserts the output of the command into the command line. This vsql command just counts the number of rows in the store.store_sales_fact table.

```
vsql -U username -w passwd -h testdb01 -d vmart -At -c "SELECT * from store.store_sales_fact"  \
| pv -lpetr -s `vsql -U username -w passwd -h testdb01 -d vmart -At -c "SELECT COUNT (*) FROM store.store_sales_fact;"` \
| vsql -U username -w passwd -d vmart -c "COPY store.store_sales_fact FROM STDIN DELIMITER '|';"
```

While running, the above command displays a progress bar that looks like this:

```
0:00:39 [12.6M/s] [==============================>                    ] 50% ETA 00:00:40
```

Output formatting examples

By default, Vertica formats query output as follows:

```
=> SELECT DISTINCT category_description FROM product_dimension ORDER BY category_description;
     category_description
-----------------------------------
 Food
 Medical
 Misc
 Non-food
(4 rows)
```

You can control the format of query output in various ways with the \pset command—for example, change the border:

```
=> \pset border 2
Border style is 2.
=> SELECT DISTINCT category_description FROM product_dimension ORDER BY category_description;
+----------------------------------+
|       category_description       |
+----------------------------------+
| Food                             |
| Medical                          |
| Misc                             |
| Non-food                         |
+----------------------------------+
(4 rows)
=> \pset border 0
Border style is 0.
=> SELECT DISTINCT category_description FROM product_dimension ORDER BY category_description;
     category_description
--------------------------------
Food
Medical
Misc
Non-food
(4 rows)
```

The following sequence of pset commands change query output in several ways:

- Set border style to 1.
- Remove column alignment.
- Change the field separator to a comma.
- Remove column headings

```
=> \pset border 1
Border style is 1.
=> \pset format unaligned
Output format is unaligned.
=> \pset fieldsep ','
Field separator is ",".
=> \pset tuples_only
Showing only tuples.
=> SELECT product_key, product_description, category_description FROM product_dimension LIMIT 10;
1,Brand #2 bagels,Food
1,Brand #1 butter,Food
2,Brand #6 chicken noodle soup,Food
3,Brand #11 vanilla ice cream,Food
4,Brand #14 chocolate chip cookies,Food
4,Brand #12 rash ointment,Medical
6,Brand #18 bananas,Food
7,Brand #25 basketball,Misc
8,Brand #27 french bread,Food
9,Brand #32 clams,Food
```

The following example uses meta-commands to toggle output format—in this case, \a (alignment), \t (tuples only), and -x (extended display):

```
=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is off.
=> SELECT product_key, product_description, category_description FROM product_dimension LIMIT 10;
 product_key |       product_description        |       category_description
-------------+----------------------------------+----------------------------------
           1 | Brand #2 bagels                  | Food
           1 | Brand #1 butter                  | Food
           2 | Brand #6 chicken noodle soup     | Food
           3 | Brand #11 vanilla ice cream      | Food
           4 | Brand #14 chocolate chip cookies | Food
           4 | Brand #12 rash ointment          | Medical
           6 | Brand #18 bananas                | Food
           7 | Brand #25 basketball             | Misc
           8 | Brand #27 french bread           | Food
           9 | Brand #32 clams                  | Food
(10 rows)
```

The following example sets output format to HTML, so Vertica renders query results in HTML markup as a table:

```
=> \pset format html
Output format is html.
=> \pset tableattr 'border="2" cellpadding="3"'
Table attribute is "border="2" cellpadding="3"".
=> SELECT product_key, product_description, category_description FROM product_dimension LIMIT 2;
<table border="1" border="2" cellpadding="3">
 <tr>
  <th align="center">product_key</th>
  <th align="center">product_description</th>
  <th align="center">category_description</th>
 </tr>
 <tr valign="top">
  <td align="right">1</td>
  <td align="left">Brand #2 bagels</td>
  <td align="left">Food                          </td>
 </tr>
 <tr valign="top">
  <td align="right">1</td>
  <td align="left">Brand #1 butter</td>
  <td align="left">Food                          </td>
 </tr>
</table>
<p>(2 rows)<br />
</p>
```

Client libraries

The Vertica client driver libraries provide interfaces for connecting your client applications (or third-party applications such as Cognos and MicroStrategy) to your Vertica database. The drivers simplify exchanging data for loading, report generation, and other common database tasks.

There are three separate client drivers:

- Open Database Connectivity (ODBC)—the most commonly-used interface for third-party applications and clients written in C, Python, PHP, Perl, and most other languages.
- Java Database Connectivity (JDBC)—used by clients written in the Java programming language.
- ActiveX Data Objects for .NET (ADO.NET)—used by clients developed using Microsoft's .NET Framework and written in C#, Visual Basic .NET, and other .NET languages.

Client driver standards

The Vertica client drivers are compatible with the following driver standards:

- The ODBC driver complies with version 3.5.1 of the ODBC standard.
- Vertica's JDBC driver is a type 4 driver that complies with the JDBC 3.0 standard. It is compiled using JDK version 1.5, and is compatible with client applications compiled using JDK versions 1.5 and 1.6.
- ADO.NET drivers conform to .NET framework 3.0 specifications.

The drivers do not support some of the optional features in the standards. See ODBC feature support and JDBC feature support and Using ADO.NET for details.

In this section
- Client driver and server version compatibility
- Client drivers
- Accessing Vertica
- Managing query execution between the client and Vertica

Client driver and server version compatibility

Backward compatibility between Vertica server and client drivers works in both directions; Vertica server is compatible with all previous versions of client drivers, and all new client drivers are compatible with most versions of Vertica server. This compatibility lets you upgrade your Vertica server without having to immediately upgrade your client software, and use new client software with older versions of Vertica. Occasionally, however, individual features of a new server version might be unavailable through older drivers.

Client	Compatible Server Versions
ODBC	7.1.x and above
JDBC	7.2.x and above
ADO.NET	7.2.x and above
FIPS-enabled ODBC	FIPS-enabled 8.0.x and above (FIPS cannot be enabled in Vertica 9.3.x and 10.0.x.).
FIPS-enabled JDBC	FIPS-enabled 8.1.x and above (FIPS cannot be enabled in Vertica 9.3.x and 10.0.x.)

Client drivers

You must install the Vertica client drivers to access Vertica from your client application. The drivers create and maintain connections to the database and provide APIs that your applications use to access your data. The client drivers support connections using JDBC, ODBC, and ADO.NET .

Client driver standards

The client drivers support the following standards:

- ODBC drivers conform to ODBC 3.5.1 specifications.
- JDBC drivers conform to JDK 5 specifications.
- ADO.NET drivers conform to .NET framework 3.0 specifications.

In this section
- Installing and configuring client drivers
- Upgrading the client drivers
- Setting a client connection label
- Using legacy drivers

Installing and configuring client drivers

You can access your Vertica database with various programming languages and tools by installing the appropriate client driver. The following table lists the required client drivers for each access method:

Client Driver	Language/Tool
JDBC	Java

ODBC	• Python (pyodbc) • PHP • Perl
vertica-python	Python (native client)
ADO.NET	C#
vertica-nodejs	JavaScript
vertica-sql-go	Go

In this section

- Windows client driver installer
- FIPS client drivers
- JDBC client driver
- ODBC client driver
- Python client drivers
- Node.js client driver
- Go client driver
- OLE DB client driver
- ADO.NET client driver

Windows client driver installer

All available client drivers for Windows are included in the Vertica Client Drivers and Tools installer . This installs the following components on systems that meet the prerequisites . The individual components may require additional configuration before use, so navigate to their pages linked below for more information:

- ODBC
- JDBC
- OLE DB
- vsql
- ADO.NET
- The Microsoft connectivity pack for windows
- The visual studio plug-in for windows

In this section

- System prerequisites
- The visual studio plug-in for windows
- Uninstalling, modifying, or repairing the client drivers and tools

System prerequisites

The Vertica Client Drivers and Tools for Windows has basic system prerequisite requirements. The pack also requires that specific Microsoft components be installed for full integration.

For a list of all prerequisites, see Client drivers support in the Supported Platforms document.

Fully update your system

Before you install the Vertica driver package, verify that your system is fully up to date with all Windows updates and patches. See the documentation for your version of Windows for instructions on how to run Windows update. The Vertica client libraries and vsql executable install updated Windows libraries that depend on Windows service packs. Be sure to resolve any issues that block the installation of Windows updates.

If your system is not fully up-to-date, you may receive error messages about missing libraries such as api-ms-win-crt-runtime-l1-1-0.dll when starting vsql.

In this section

- .NET framework
- Microsoft visual studio
- Microsoft SQL server

.NET framework

The Vertica Client Drivers and Tools for Windows requires and prompts you to install the Microsoft .NET Framework 4.6 if it is not installed.

To manually install the Microsoft .NET Framework 4.6, see the Microsoft documentation .

Microsoft visual studio

The Vertica Client Drivers and Tools for Windows installer provides a Visual Studio plug-in which allows you to use Vertica as a Visual Studio Data Source for Visual Studio 2012, 2013, or 2015. The connection properties for the plug-in are the same as ADO.NET connection properties .

Important

Important: You must have Visual Studio and the matching SDK installed to use the Visual Studio plug-in.

After installing the plug-in, you can use it to access your Vertica database from within Visual Studio. If you do not have the SDK installed, download the SDK specific to your version of Visual Studio.

> **Note**
> For Visual Studio 2015, you do not need to download the SDK separately as it is included as an installation option with the Visual Studio installation. For more information, refer to the Microsoft documentation .

- Microsoft Visual Studio 2012 SDK
- Microsoft Visual Studio 2013 SDK

If the Microsoft Visual Studio SDK is missing when you begin the installation, a dialog box will open to install it. You can choose to ignore this dialog box.

Configuring SSDT-BI integration

The Vertica Client Drivers and Tools for Windows installer provides SSDT-BI (Visual Studio 2012, 2013, or 2015) integration. To use SSDT-BI, follow this process:

1. Install the SSDT-BI development tool add-on for Visual Studio.
2. Verify that SQL Server is installed on the same or a different machine.
3. Verify that the SQL Server Shared Features for SSDT-BI have been activated.

You can then develop packages using SSDT-BI, creating your projects using SQL Server's SSIS, SSAS, SSRS features. To use these features, you must connect to Vertica through the Vertica ADO.NET driver (for SSIS and SSRS) or the OLE DB driver (for SSAS).

For more information, see Microsoft components .

Microsoft SQL server

Use SQL Server 2012, 2014 or 2016. The Vertica Client Drivers and Tools for Windows installer enables support for the following:

- **SQL Server 2012, 2014, and 2016:**
  - SQL Server Integration Services (SSIS)
  - SQL Server Reporting Services (SSRS)
  - SQL Server Analysis Services (SSAS)
- **SQL Server using 2012, 2013, and 2015** —SQL Server Data Tool - Business Intelligence (SSDT-BI)

> **Note**
> For SQL Server 2012, you can use either SQL Server 2012 or SQL Server 2012 SP1.

To use the enhanced Vertica .NET support, you must first install SQL Server. Then, you can install the Client Drivers and Tools for Windows. The following components must be installed on the SQL server:

For...	Install...
SSAS	The Analysis Services Instance Feature.

SSRS	The Reporting Services Instance Feature.
SSIS (Data Type Mappings)	The SQL Server Integration Services Shared Feature.
SSDT-BI (Visual Studio 2012, 2013, or 2015)	SQL Server Data Tool - Business Intelligence Shared Feature only *after* installing Microsoft Visual Studio 2012, 2013, or 2015.

The visual studio plug-in for windows

The Visual Studio plug-in for Windows is installed as part of the Client Drivers and Tools for Windows.

For information on how the Visual Studio plug-in integrates with Microsoft components previously installed on your system, see Microsoft components .

In this section

- Visual studio limitations

Visual studio limitations

Visual Studio 2012 May Require Update 3

You may need to install update 3 to Visual Studio 2012 if:

- You launch Server Explorer to view and work with your Vertica server, but the Vertica data source is not visible.
- You create a SSAS cube, connect to Vertica, and find either an empty list of tables or tables not functioning correctly.

This issue does not occur for other versions of Visual Studio supported by Vertica.

Results Viewer Limited to 655 Columns

The Visual Studio results viewer cannot execute a query that includes more than 655 columns. If a table includes more than 655 columns, select specific columns (up to 655 total) rather than selecting all columns.

Manually Refresh Settings for Visual Studio

If, after installing the Visual Studio plug-in, you do not see Vertica listed as a data provider, manually refresh.

To do so, run devenv.exe/setup , which you can find in the Visual Studio installation folder.

SQL Pane Issues

- **ALTER TABLE** or **CREATE TABLE**
  You use Visual Studio 2012, 2013, or 2015 and issue the ALTER TABLE or CREATE TABLE statement in the SQL pane. However, a message displays telling you that the statement is not supported. To resolve the error, click **Continue** , and the query executes.
- **Queries with Semicolons**
  You use Visual Studio 2012, 2013, or 2015 and execute a SQL query in the SQL pane. If you include a semicolon (;) with your query, the query executes, but the result returned cannot be edited. To avoid this issue, enter the same query in the SQL pane without the semicolon.
- **Quoting Boolean Values**
  You use Visual Studio 2012, 2013, or 2015 to connect to the Vertica database and execute a SQL query in the SQL pane. When attempting to insert a value into a Boolean column without putting quotes around the value, subsequent execution of the SQL statement returns an error. To work around this issue, include quotes.

Uninstalling Client Drivers and Tools for Windows **Error**

There is a scenario where an uninstall of the Client Drivers and Tools for Windows package fails with a message that the .NET framework is required. What follows is the scenario that causes this issue.

1. You Install the Client Drivers and Tools for Windows.
2. You then install Visual Studio 2012, which includes installation of the .NET framework 4.0 or 4.5.
3. You uninstall the .NET framework using the Windows Control Panel.
4. You then attempt to uninstall the Client Drivers and Tools for Windows. The uninstall fails with the message that .NET framework is required.

Perform the following to correct this issue:

1. Reinstall the .NET framework 4.0 or 4.5 manually, using the Windows Control Panel.

2. Uninstall the Client Drivers and Tools for Windows.

Uninstalling, modifying, or repairing the client drivers and tools

To uninstall, modify, or repair the client drivers and tools, run the Client Drivers and Tools for Windows installer.

The installer provides three options:

Action	Description
Modify	Remove installed client drivers and tools or install missing client drivers and tools.
Repair	Reinstall already-installed client drivers and tools.
Uninstall	Uninstall all of the client drivers and tools.

Silently uninstall the client drivers and tools

1. As a Windows Administrator, open a command-line session, and change directory to the folder that contains the installer.
2. Run the command:

```
VerticaSetup.exe -q -uninstall
```

The client drivers and tools are silently uninstalled.

FIPS client drivers

Vertica offers a FIPS-compliant version of the ODBC and JDBC client drivers.

In this section

- Installing the FIPS client driver for JDBC
- Installing the FIPS client driver for ODBC and vsql

Installing the FIPS client driver for JDBC

Vertica offers a JDBC client driver that is compliant with the Federal Information Processing Standard (FIPS). Use this JDBC client driver to access systems that are FIPS-compatible. For more information on FIPS, see Federal information processing standard .

Implementing FIPS on a JDBC client requires a third-party JRE extension called BouncyCastle , a collection of APIs used for cryptography. Use BouncyCastle APIs with JDK 1.7 and 1.8, and a supported FIPS-compliant operating system .

Important

When using the JDBC FIPS-compliant client, expect a slight delay for the client to establish a secure connection with the database. If necessary, increase your system's entropy to ensure a fast and secure connection.

The following procedure adds the FIPS BouncyCastle .jar as a JVM JSSE provider:

1. Download the BouncyCastle FIPS .jar file bc-fips-1.0.0.jar .
2. Add bc-fips-1.0.0.jar as a JRE library extension:

```
path/to/jre/lib/ext/bc-fips-1.0.0.jar
```

3. Add BouncyCastle as an SSL security provider in <path to jre>/lib/security/java.security :

```
security.provider.1=org.bouncycastle.jcajce.provider.BouncyCastle FipsProvider
security.provider.2=com.sun.net.ssl.internal.ssl.Provider BCFIPS
security.provider.3=sun.security.provider.Sun
```

4. Use the following JVM java -D system property command arguments to set the KeyStore and TrustStore files to BCFIPS:

```
export JAVA_OPTS="$JAVA_OPTS -Djavax.net.ssl.keyStoreProvider=BCFIPS
export JAVA_OPTS="$JAVA_OPTS -Djavax.net.ssl.trustStoreProvider=BCFIPS
```

5. Set the default type for the KeyStore implementation to BCFKS in *path/to/jre* /lib/security/java.security:

```
keystore type=BCFKS
ssl.keystore.type=BCFKS
```

> **Note**
> If you are using FIPS with BouncyCastle, you must create all client keys and certificates with the BCFKS store type, including the Vertica-to-Kafka keys and certificates.

6. Create the BCFKS-type keystore and truststore:

```
cd path/to/jre
-storetype BCFKS
-providername BCFIPS
-providerclass org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider
-provider org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider
-providerpath bc-fips-1.0.0.jar
-alias CARoot
-import -file path/to/server.crt.der
```

7. When prompted, enter the keystore password. The following message is displayed to confirm that a certificate was added to the keystore:

```
"Certificate was added to the keystore"
```

8. Run the Java program with SSL DB:
   1. Copy the vertica.kafka.keystore.bcfks keyStore from *path/to/jre* /lib/ext/ to the Java program folder.
   2. Convert the Vertica server certificate to a form that Java understands:

```
$ path/to/java/bin/keytool -keystore verticastore -keypasswd -storepass password
                    -importkeystore -noprompt -alias verticasql -import -file server.crt.der
```

   3. Install JDBC .
9. Test the implementation:

```
$ java -Djavax.net.debug=ssl -Djavax.net.ssl.keyStore='vertica.kafka.keystore.bcfks'
-Djavax.net.ssl.keyStorePassword='password'
-Djavax.net.ssl.trustStore='path/to/verticastore'
-Djavax.net.ssl.trustStorePassword='password'
-cp .:vertica-jdbc-12.0.0-0.jar FIPSTest
```

Installing the FIPS client driver for ODBC and vsql

Vertica offers a FIPS client for FIPS-compatible systems. A FIPS-compatible system is FIPS-enabled and includes the OpenSSL libraries.

The FIPS client supports ODBC and vsql and is offered in 64-bit only.

Prerequisites

Verify that your host system is running a FIPS-compliant operating system that Vertica supports .

The FIPS client installer checks your host system for the value of the sysctl parameter, crypto.fips_enabled. You must set this parameter to 1 (enabled). If your host is not enabled, the client does not install.

Installing the FIPS client

To install the FIPS client driver package:

1. Download the FIPS client package from the Vertica driver downloads page .
2. Log in to the client system as root.
3. Install the RPM package that you downloaded:

```
# rpm -Uvh package_name.rpm
```

For ODBC, after you have installed the client package, create a DSN and set some additional configuration parameters. For more information, see:

- Creating an ODBC DSN for Linux
- ODBC driver settings

You can optionally add the vsql client to your PATH environment variable so that you do not need to enter its full path to run it. To do so, add the following to the .profile file in your home directory or the global /etc/profile file:

```
export PATH=$PATH:/opt/vertica/bin
```

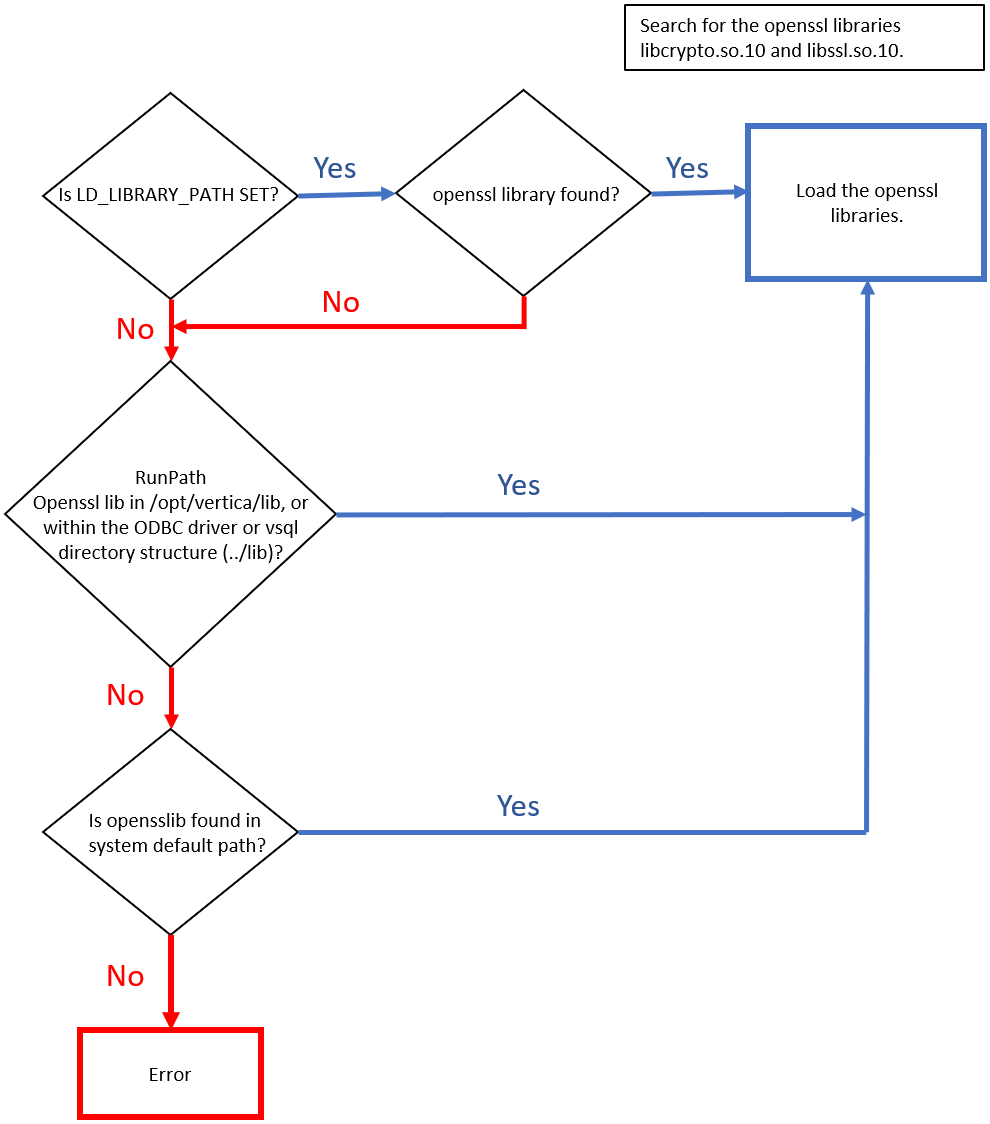
How the client searches for OpenSSL libraries

When you launch the client application to connect to the server, the client searches for and loads the OpenSSL libraries libcrypto.so.10 and libssl.so.10 for supported OpenSSL versions :

- The client first checks to see if LD_LIBRARY_PATH is set.
- If the LD_LIBARY_PATH location does not include the libraries, it checks RunPath , either /opt/vertica/lib or within the ODBC or vsql directory structure (../lib).

Important
The LD_LIBRARY_PATH , if set, directs the search path for the OpenSSL libraries. The client loads the libraries from any set or preset LD_LIBRARY_PATH location.

The following figure depicts the search process for the OpenSSL libraries:



JDBC client driver

The Vertica JDBC client driver conforms to JDK 5 specifications and provides an interface for communicating with the Vertica database with Java. For details on this and other APIs, see API Reference .

To install the JDBC client driver, see Installing the JDBC client driver .

In this section

- Installing the JDBC client driver
- Modifying the Java CLASSPATH

Installing the JDBC client driver

The JDBC client driver conforms to JDK 5 specifications. Download the JDBC client driver according to your environment and requirements. If you need a FIPS-compliant driver, see Installing the FIPS client driver for JDBC .

Installing Vertica from the RPM automatically installs the JDBC client driver. To use the JDBC client driver, you just need to add the Vertica JDBC .jar to your CLASSPATH .

To manually install the JDBC client driver:

1. Download the version of the JDBC client driver from the Client Drivers downloads page compatible with your version of Vertica.
2. Copy the .jar file to a directory in your Java CLASSPATH on every client system with which you want to access Vertica. You can either:
   - Copy the .jar file to its own directory (such as /opt/vertica/java/lib) and then add that directory to your CLASSPATH (recommended). See Modifying the Java CLASSPATH for details.
   - Copy the .jar file to directory that is already in your CLASSPATH (for example, a directory where you have placed other .jar files on which your application depends).
   - Copy the .jar file to the system-wide Java Extensions directory. The exact location differs between operating systems. Some examples include:
     - Windows: C:\Program Files\Java\jrex.x.x_x\lib\ext\
     - Mac OS: /Library/Java/Extensions or /Users/username/Library/Java/Extensions
3. Create a connection to test your configuration.

Modifying the Java CLASSPATH

The CLASSPATH environment variable contains a list of directories where the Java runtime looks for library class files. For your Java client code to access Vertica, you must add to the CLASSPATH the directory containing the Vertica JDBC .jar .

Using symbolic links for the CLASSPATH

You can optionally add to the CLASSPATH a symbolic link vertica-jdbc-x.x.x.jar (where x.x.x is a version number) that points to the JDBC library .jar file, rather than the .jar file itself.

Using the symbolic link ensures that any updates to the JDBC library .jar file (which will use a different filename) will not invalidate your CLASSPATH setting, since the symbolic link's filename will remain the same. You just need to update the symbolic link to point at the new .jar file.

Linux and OS X

The following examples use a POSIX-compliant shell.

To set the CLASSPATH for the current session:

```
$ export CLASSPATH=$CLASSPATH:/opt/vertica/java/lib/vertica-jdbc-x.x.x.jar
```

To set the CLASSPATH for every session, add the following to your start-up file (such as ~/.profile or /etc/profile :

```
$ export CLASSPATH=$CLASSPATH:/opt/vertica/java/lib/vertica-jdbc-x.x.x.jar
```

Windows

Provide the class paths to the .jar , .zip , or .class files.

```
C:> SET CLASSPATH=classpath1;classpath2...
```

For example:

```
C:> SET CLASSPATH=C:\java\MyClasses\vertica-jdbc-x.x.x.jar
```

As with the Linux/UNIX settings, this setting only lasts for the current session. To set the CLASSPATH permanently, set an environment variable:

1. On the Windows Control Panel, click **System** .
2. Click **Advanced** or **Advanced Systems Settings** .
3. Click **Environment Variables** .
4. Under User variables, click **New** .
5. In the Variable name box, type CLASSPATH .
6. In the Variable value box, type the path to the Vertica JDBC .jar file on your system (for example, C:\Program Files (x86)\Vertica\JDBC\vertica-jdbc-x.x.x.jar)

Specifying the library directory in the Java command

Another, OS-agnostic way to tell the Java runtime where to find the Vertica JDBC driver is to explicitly add the directory containing the .jar file to the Java command line using either the -cp or -classpath argument. For example, you can start your client application with:

```
java -classpath /opt/vertica/java/lib/vertica-jdbc-x.x.x.jar myapplication.class
```

Your Java IDE may also let you add directories to your CLASSPATH, or let you import the Vertica JDBC driver into your project. See your IDE documentation for details.

ODBC client driver

The Vertica ODBC client driver provides an interface for creating client applications with several languages:

- C/C++
- Python
- Perl
- PHP

To install ODBC, see Installing the ODBC client driver .

In this section

- Installing the ODBC client driver
- Upgrading and downgrading ODBC
- Uninstalling ODBC
- Creating an ODBC data source name (DSN)
- ODBC driver settings
- Configuring ODBC logs

Installing the ODBC client driver

To install ODBC, follow the instructions according to your platform. For a list of supported platforms, see Client drivers support .

This page covers a non-FIPS installation. To install ODBC on a FIPS-compliant system, see Installing the FIPS client driver for ODBC and vsql .

Installing on Linux

Installing Vertica from the RPM automatically installs the ODBC client driver, so you do not need to install them again on the machine running Vertica. To use the ODBC client driver in this case, create a DSN .

To install the ODBC client driver manually on other machines:

1. Log in to the client system as root.
2. Verify that your system has a supported ODBC driver manager.
3. Download the ODBC client driver for Linux in the format appropriate for your distribution.
4. Install or extract the driver:
   - If you downloaded the .rpm , install the driver:

     > **Note**
     > If the client driver is already installed on your system (either from a manual installation or from automatic installation from the Vertica RPM) and you attempt to reinstall them manually, you will receive error messages. To bypass these errors and overwrite the existing driver installations, use the --force flag.

     ```
     $ rpm -Uvh driver_name.rpm
     ```

   - If you downloaded the .tar , create the /opt/vertica/ directory if it does not already exist, copy the .tar to it, navigate to it, and extract the .tar :

     ```
     $ mkdir -p /opt/vertica/
     $ cp driver_name.tar.gz /opt/vertica/
     $ tar vzxf driver_name.tar.gz
     ```

     This creates two directories:
     - /opt/vertica/include : Contains the header file.
     - /opt/vertica/lib64/ (64-bit) or /opt/vertica/lib/ (32-bit): Contains library files.
5. Set the following ODBC driver settings in vertica.ini . For details on each, see ODBC driver settings :
   - ErrorMessagesPath : Required, the path of the directory containing the ODBC driver's error message files.

- ODBCInstLib : The path to the ODBC installer library. This is only required if the driver manager's installation library is not in the environment variables LD_LIBRARY_PATH or LIB_PATH .
- DriverManagerEncoding : The UTF encoding standard used by the driver manager. This is only required if your driver manager does not use UTF-8.

The following is an example configuration in vertica.ini :

- Use encoding for the 64-bit UNIXODBC driver manager.
- Use the error messages defined in the standard Vertica 64-bit ODBC driver installation directory.
- Log all warnings and more severe messages to log files in /tmp/

```
[Driver]
DriverManagerEncoding=UTF-16
ODBCInstLib=/usr/lib64/libodbcinst.so
ErrorMessagesPath=/opt/vertica
LogLevel=4
LogPath=/tmp
```

6. Create a DSN .

Installing on macOS

> **Note**
> You can only have one installation per version of the ODBC driver on a macOS system. This is because each installation is identified by a package ID and version number, and package ID does not change between versions of the driver.

To install the ODBC client driver on macOS:

1. Verify that your system has a compatible driver manager. The driver is designed to be used with the standard iODBC Driver Manager that ships with macOS. You can also use unixODBC .
2. Download the ODBC client driver.
3. If you installed a previous version of the ODBC driver, your system might already have a registered driver named "Vertica". You must remove or rename this older version of the driver before installing a new version from the .pkg installer. Renaming the older version allows you to retain the old version after you install the new one.
4. Run the installer.
5. Create a DSN .

Installing silently

1. Log into the client macOS in one of two ways:
   - As an administrator account if you are installing the driver for system-wide use.
   - As the user who needs to use the Vertica ODBC driver.
2. Open a terminal.
3. Install the .pkg file containing the ODBC driver using the command:

```
sudo installer -pkg path/to/client/driver/vertica-odbc-xx.x.x-x.pkg -target /
```

Installing on Windows

To install the ODBC client driver on Windows:

1. Download the client driver installer for Windows.
2. Run the installer.
3. Create a DSN .

Installing silently

1. Open a terminal as an Administrator.
2. Run the following command to silently install the drivers to C:\Program Files\Vertica Systems :

```
VerticaSetup.exe -q -install InstallFolder="C:\Program Files\Vertica Systems"
```

Upgrading and downgrading ODBC

Linux

To upgrade ODBC:

1. Uninstall the current version of the driver.
2. Install the new version of the driver.

macOS

To upgrade or downgrade ODBC:

- **Upgrade** : Newly installed versions of the Vertica ODBC driver for macOS automatically upgrade the relevant driver system settings. Any DSNs associated with a previous version of the driver are not affected, except that they begin using the newer version of the driver.
- **Downgrade** : Run the uninstall script to remove the current version of the Vertica ODBC driver for macOS. Complete this step before installing an older driver version.

Windows

1. Download the Windows client driver installer.
2. Run the installer and follow the prompts to upgrade the driver. The installer upgrades existing drivers in place.
3. Reboot your system.

Uninstalling ODBC

Linux

If you installed ODBC with the .rpm :

```
$ rpm -e package_name
```

If you installed ODBC with the .tar , delete the directory manually.

macOS

Uninstalling the macOS ODBC Client-Driver does not remove any existing DSNs associated with the driver.

To uninstall:

1. Open a terminal window.
2. Run the command:
   ```
   sudo /Library/Vertica/ODBC/bin/Uninstall
   ```

Windows

1. Open the **Add or Remove Programs** menu.
2. EIther uninstall the **Vertica Client Installer** to remove all client drivers from the system or, to only uninstall ODBC, uninstall the following applications:
   - Vertica ODBC Driver (32 Bit)
   - Vertica ODBC Driver (64 Bit)

Creating an ODBC data source name (DSN)

A Data Source Name (DSN) is the logical name that is used by Open Database Connectivity (ODBC) to refer to the driver and other information that is required to access data from a data source. Whether you are developing your own ODBC client code or you are using a third-party tool that needs to access Vertica using ODBC, you need to configure and test a DSN. The method you use depends upon the client operating system you are using.

Refer to the following sections for information specific to your client operating system.

In this section

- Creating an ODBC DSN for Linux
- Creating an ODBC DSN for windows clients
- Creating an ODBC DSN for macOS clients
- ODBC DSN connection properties
- Setting DSN connection properties

Creating an ODBC DSN for Linux

You define DSN on Linux and other UNIX-like platforms in a text file. Your client's driver manager reads this file to determine how to connect to your Vertica database. The driver manager usually looks for the DSN definitions in two places:

- /etc/odbc.ini
- ~/.odbc.ini (a file named .odbc.ini in the user's home directory)

Users must be able to read the odbc.ini file in order to use it to connect to the database. If you use a global odbc.ini file, consider creating a UNIX group with read access to the file. Then, add the users who need to use the DSN to this group.

The structure of these files is the same—only their location differs. If both files are present, the ~/.odbc.ini file usually overrides the system-wide /etc/odbc.ini file.

> **Note**
> See your ODBC driver manager's documentation for details on where these files should be located and any other requirements.

odbc.ini file structure

The odbc.ini is a text file that contains two types of lines:

- Section definitions, which are text strings enclosed in square brackets.
- Parameter definitions, which contain a parameter name, an equals sign (=), and then the parameter's value.

> **Caution**
> The unixODBC driver manager supports parameter values of up to 1000 characters in odbc.ini . If your parameter value is greater than 1000 characters (for example, OAuthAccessToken), you must pass it through a connection string rather than specifying it in odbc.ini .

The first section of the file is always named [ODBC Data Sources], and contains a list of all the DSNs that the odbc.ini file defines. The parameters in this section are the names of the DSNs, which appear as section definitions later in the file. The value is a text description of the DSN and has no function. For example, an odbc.ini file that defines a single DSN named Vertica DSN could have this ODBC Data Sources section:

```
[ODBC Data Sources]
VerticaDSN = "vmartdb"
```

Appearing after the ODBC data sources section are sections that define each DSN. The name of a DSN section must match one of the names defined in the ODBC Data Sources section.

Configuring the odbc.ini file:

To create or edit the DSN definition file:

1. Using the text editor of your choice, open odbc.ini or ~/.odbc.ini .
2. Create an ODBC Data Sources section and define a parameter:
   - Whose name is the name of the DSN you want to create
   - Whose value is a description of the DSN

   For example, to create a DSN named VMart, you would enter:

   ```
   [ODBC Data Sources]
   VMart = "VMart database on Vertica"
   ```

3. Create a section whose name matches the DSN name you defined in step 2. In this section, you add parameters that define the DSN's settings. The most commonly-defined parameters are:
   - **Description** – Additional information about the data source.
   - **Driver** – The location and designation of the Vertica ODBC driver, or the name of a driver defined in the odbcinst.ini file (see below). For future compatibility, use the name of the symbolic link in the library directory, rather than the library file:
     - (/opt/vertica/lib , on 32-bit clients
     - /opt/vertica/lib64 , on 64-bit clients

     For example, the symbolic link for the 64-bit ODBC driver library is:

     ```
     /opt/vertica/lib64/libverticaodbc.so
     ```

     The symbolic link always points to the most up-to-date version of the Vertica client ODBC library. Use this link so that you do not need to update all of your DSNs when you update your client drivers.
   - **Database** – The name of the database running on the server. This example uses vmartdb for the vmartdb.
   - **ServerName** — The name of the server where Vertica is installed. Use localhost if Vertica is installed on the same machine.
     You can provide an IPv4 address, IPv6 address, or host name.
     In mixed IPv4/IPv6 networks, the DNS server configuration determines which IP version address is sent first. Use the PreferredAddressFamily option to force the connection to use either IPv4 or IPv6.
   - **UID** — Either the database superuser (same name as database administrator account) or a user that the superuser has created and granted privileges. This example uses the user name dbadmin.
   - **PWD** —The password for the specified user name. This example leaves the password field blank.
   - **Port** — The port number on which Vertica listens for ODBC connections. For example, 5433.

- **ConnSettings** — Can contain SQL commands separated by a semicolon. These commands can be run immediately after connecting to the server.
- **SSLKeyFile** — The file path and name of the client's private key. This file can reside anywhere on the system.
- **SSLCertFile** —The file path and name of the client's public certificate. This file can reside anywhere on the system.
- **Locale** — The default locale used for the session. By default, the locale for the database is: en_US@collation=binary (English as in the United States of America). Specify the locale as an ICU Locale. See the ICU User Guide (http://userguide.icu-project.org/locale) for a complete list of parameters that can be used to specify a locale.
- **PreferredAddressFamily** :
  The IP version to use if the client and server have both IPv4 and IPv6 addresses and you have provided a host name, one of the following:
  - ipv4 : Connect to the server using IPv4.
  - ipv6 : Connect to the server using IPv6.
  - none : Use the IP address provided by the DNS server.

For example:

```
[VMart]
Description = Vmart Database
Driver = /opt/vertica/lib64/libverticaodbc.so
Database = vmartdb
Servername = host01
UID = dbadmin
PWD =
Port = 5433
ConnSettings =
AutoCommit = 0
SSLKeyFile = /home/dbadmin/client.key
SSLCertFile = /home/dbadmin/client.crt
Locale = en_US@collation=binary
```

See ODBC DSN connection properties for a complete list of parameters including Vertica-specific ones.

Using an odbcinst.ini file

Instead of giving the path of the ODBC driver library in your DSN definitions, you can use the name of a driver defined in the odbcinst.ini file. This method is useful method if you have many DSNs and often need to update them to point to new driver libraries. It also allows you to set some additional ODBC parameters, such as the threading model.

Just as in the odbc.ini file, odbcinst.ini has sections. Each section defines an ODBC driver that can be referenced in the odbc.ini files.

In a section, you can define the following parameters:

- **Description** — Additional information about the data source.
- **Driver** — The location and designation of the Vertica ODBC driver, such as /opt/vertica/lib64/libverticaodbc.so

For example:

```
[Vertica]
Description = Vertica ODBC Driver
Driver = /opt/vertica/lib64/libverticaodbc.so
```

Then, in your odbc.ini file, use the name of the section you created in the odbcinst.ini file that describes the driver you want to use. For example:

```
[VMart]
Description = Vertica Vmart database
Driver = Vertica
```

If you are using the unixODBC driver manager, you should also add an ODBC section to override its standard threading settings. By default, unixODBC serializes all SQL calls through ODBC, which prevents multiple parallel loads. To change this default behavior, add the following to your odbcinst.ini file:

```
[ODBC]
Threading = 1
```

Configuring additional ODBC settings

On Linux and UNIX systems, you need to configure some additional driver settings before you can use your DSN. See ODBC driver settings for details.

In this section

- Testing an ODBC DSN using isql

Testing an ODBC DSN using isql

The unixODBC driver manager includes a utility named isql, which is a simple ODBC command-line client. It lets you to connect to a DSN to send commands and receive results, similarly to vsql.

To use isql to test a DSN connection:

1. Run the following command:
   ```
   $ isql –v DSNname
   ```
   Where *DSNname* is the name of the DSN you created.
   A connection message and a SQL prompt display. If they do not, you could have a configuration problem or you could be using the wrong user name or password.
2. Try a simple SQL statement. For example:
   ```
   SQL> SELECT table_name FROM tables;
   ```
   The isql tool returns the results of your SQL statement.

> **Note**
> If you have not set the ErrorMessagesPath in the additional driver configuration settings, any errors during testing will trigger a missing error message file ("The error message NoSQLGetPrivateProfileString could not be found in the en-US locale"). See ODBC driver settings for more information.

Creating an ODBC DSN for windows clients

To create a DSN for Microsoft Windows clients, you must perform the following tasks:

In this section

- Setting up an ODBC DSN
- Encrypting passwords on ODBC DSN
- Testing an ODBC DSN using Excel

Setting up an ODBC DSN

A *Data Source Name (DSN)* is the ODBC logical name for the drive and other information the database needs to access data. The name is used by Internet Information Services (IIS) for a connection to an ODBC data source.

This section describes how to use the Vertica ODBC Driver to set up an ODBC DSN. This topic assumes that the driver is already installed, as described in Installing Client Drivers on Windows .

To set up a DSN

1. Open the ODBC Administrator. For example, you could navigate to **Start > Control Panel > Administrative Tools > Data Sources (ODBC)** .

   > **Note**
   > The method you use to open the ODBC Administrator depends on your version of Windows. Differences between Windows versions and **Start Menu** customizations could require you to take a different action to open the ODBC Administrator.

2. Decide if you want all users on your client system to be able to access to the DSN for the Vertica database.
   - If you want all users to have access, then click the **System DSN** tab.
   - Otherwise, click the **User DSN** tab to create a DSN that is only usable by your Windows user account.
3. Click **Add** to create a new DSN to connect to the Vertica database.
4. Scroll through the list of drivers in the Create a New Data Source dialog box to locate the Vertica driver. Select the driver, and then click **Finish** .

   > **Note**
   > If you have installed more than one version of the Vertica client drivers on your Windows client system, you may see multiple versions of the driver in this list. Choose the version that you know is compatible with your client application and Vertica Analytic Database server. If you are unsure, use the latest version of the driver.

The Vertica ODBC DSN configuration dialog box appears.

5. Click the **More >>>** button to view a description of the field you are editing and the connection string defined by the DSN.
6. Enter the information for your DSN. The following fields are required:
   - **DSN Name** — The name for the DSN. Clients use this name to identify the DSN to which they want to connect. The DSN name must satisfy the following requirements:
     - Its maximum length is 32 characters.
     - It is composed of ASCII characters except for the following: ⌐ { } , ; ? * = ! @ \
     - It contains no spaces.
   - **Server** — The host name or IP address of the Vertica server to which you want to connect. Use localhost, if Vertica is installed on the same machine.
     You can provide an IPv4 address, IPv6 address, or host name.
     In mixed IPv4/IPv6 networks, the DNS server configuration determines which IP version address is sent first. Use the PreferredAddressFamily option to force the connection to use either IPv4 or IPv6.
     The PreferredAddressFamily option is available on the Client Settings tab.
   - **Backup Servers** — A comma-separated list of host names or IP addresses used to connect to if the server specified by the Server field is down. Optional.
   - **Database** —The name of the Vertica database.
   - **User Name** — The name of the user account to use when connecting to the database. If the application does not supply its own user name when connecting to the DSN, this account name is used to log into the database.

   The rest of the fields are optional. See DSN Parameters for detailed information about the DSN parameters you can define.
7. If you want to test your connection:
   1. Enter at least a valid **DSN name** , **Server name** , **Database** , and either **User name** or select **Windows authentication** .
   2. If you have not selected **Windows authentication** , you can enter a password in the **Password** box. Alternately, you can select **Password for missing password** to have the driver prompt you for a password when connecting.

      > **Caution**
      > Passwords entered into the **Password** box are saved, in plaintext, to the Windows registry.

   3. Click **Test Connection** .
8. When you have finished editing and testing the DSN, click **OK** . The Vertica ODBC DSN configuration window closes, and your new DSN is listed in the ODBC Data Source Administrator window.
9. Click **OK** to close the ODBC Data Source Administrator.

After creating the DSN, you can test it using Microsoft Excel 2007 .

Setting up a 32-Bit DSN on 64-Bit versions of Microsoft windows

On 64-bit versions of Windows, the default ODBC Data Source Administrator creates and edits DSNs that are associated with the 64-bit Vertica ODBC library.

Attempting to use these 64-bit DSNs with a 32-bit client application results in an architecture mismatch error. Instead, you must create a specific 32-bit DSN for 32-bit clients by running the 32-bit ODBC Administrator usually located at:

```
c:\Windows\SysWOW64\odbcad32.exe
```

This administrator window edits a set of DSNs that are associated with the 32-bit ODBC library. You can then use your 32-bit client applications with the DSNs you create with this version of the ODBC administrator.

Encrypting passwords on ODBC DSN

When you install an ODBC driver and create a Data Source Name (DSN) the DSN settings are stored in the registry, including the password. Encrypting passwords on ODBC DSN applies only to Windows systems.

Encrypting passwords on an ODBC data source name (DSN) provides security against unauthorized database access. The password is not encrypted by default and is stored in plain-text.

> **Note**
> ODBC DSN passwords that were created in Vertica ≤8.0.x are not encrypted when you upgrade to a higher version, regardless of encryption settings.

Enable password encryption

Use the EncryptPassword parameter to enable or disable password encryption for an ODBC DSN:

- EncryptPassword = true enables password encryption
- EncryptPassword = false (default) disables password encryption

Set EncryptPassword in the Windows registry - HKEY_LOCAL_MACHINE > Software > Vertica > ODBC > Driver EncryptPassword=<true/false> .

Note

For 32 bit driver running on 64 bit windows verify password encryption here:

HKEY_LOCAL_MACHINE > Software > Wow6432Node > Vertica > ODBC >

Driver > EncryptPassword=<true/false>

Encrypted passwords get updated in the following registry locations:

For a user DSN:

HKEY_CURRENT_USER-> Software -> ODBC -> ODBC.INI -> DSNNAME -> PWD

For a system DSN:

HKEY_LOCAL_MACHINE-> Software -> ODBC -> ODBC.INI -> DSNNAME -> PWD

Verify password encryption

Use Windows Registry editor to determine if password encryption is enabled based on the value of EncryptPassword. Depending on the type of DSN you installed, check the following:

For a user DSN: HKEY_CURRENT_USER > Software > ODBC > ODBC.INI > dsn name > isPasswordEncrypted=<1/0>

For a system DSN: HKEY_LOCAL_MACHINE > Software > ODBC > ODBC.INI > dsn name > isPasswordEncrypted=<1/0>

For each DSN, the value of the isPasswordEncrypted parameter indicates the status of the password encryption, where 1 indicates an encrypted password and 0 indicates an unencrypted password.

Testing an ODBC DSN using Excel

You can use Microsoft Excel to verify that an application can connect to an ODBC data source or other ODBC application.

1. Open Microsoft Excel, and select **Data > Get External Data > From Other Sources > From Microsoft Query** .
2. When the Choose Data Source dialog box opens:
   1. Select **New Data Source** , and click **OK** .
   2. Enter the name of the data source.
   3. Select the Vertica driver.
   4. Click **Connect** .
3. When the Vertica Connection Dialog box opens, enter the connection information for the DSN, and click **OK** .
4. Click **OK** on the Create New Data Source dialog box to return to the Choose Data Source dialog box.
5. Select VMart_Schema*, and verify that the Use the Query Wizard check box is deselected. Click **OK** .
6. When the Add Tables dialog box opens, click **Close** .
7. When the Microsoft Query window opens, click the **SQL** button.
8. In the SQL window, write any simple query to test your connection. For example:

```
SELECT DISTINCT calendar_year FROM date_dimension;
```

* If you see the caution, "SQL Query can't be represented graphically. Continue anyway?" click **OK**. * The data values 2003, 2004, 2005, 2006, 2007 indicate that you successfully connected to and ran a query through ODBC.

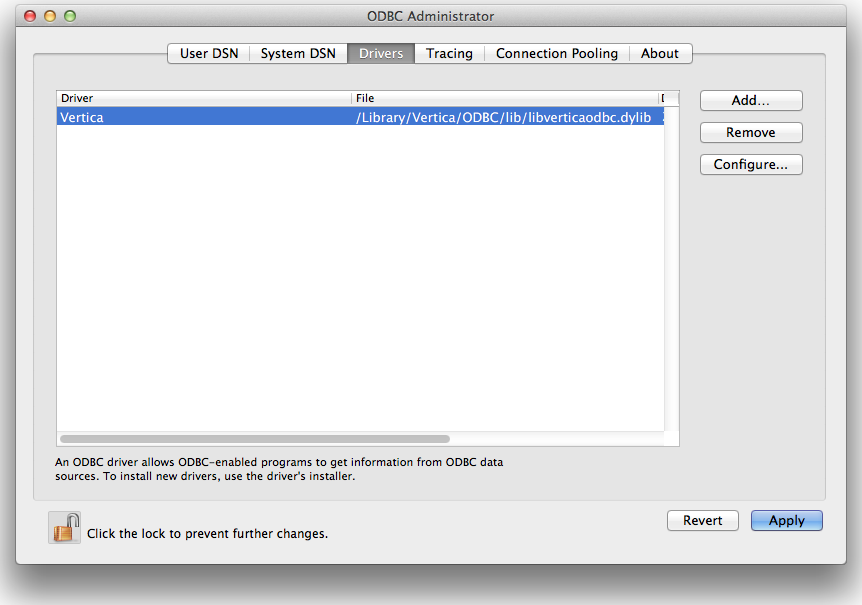
10. Select **File > Return Data to Microsoft Office Excel** .
11. In the Import Data dialog box, click **OK** .
    The data is now available for use in an Excel worksheet.

Creating an ODBC DSN for macOS clients

You can use the Vertica ODBC Driver to set up an ODBC DSN. This procedure assumes that the driver is already installed, as described in Installing the ODBC client driver .

Setting up a DSN

1. Using your web browser, download and install the Apple ODBC Administrator Tool .
2. Locate and open the ODBC Administrator Tool after installation:
    1. Navigate to **Finder** > **Applications** > **Utilities** .
    2. Open the ODBC Administrator Tool.
3. Click the **Drivers** tab, and verify that the Vertica driver is installed.



4. Specify if you want all users on your client system to be able to access the DSN for the Vertica database:
    - If you want all users to have access, then click the **System DSN** tab.
    - Otherwise, click the **User DSN** tab to create a DSN that is only usable by your Macintosh user account.
5. Click **Add...** to create a new DSN to connect to the Vertica database.
6. Scroll through the list of drivers in the Choose A Driver dialog box to locate the Vertica driver. Select the driver, and then click **OK** . A dialog box opens that requests DSN parameter information.
7. In the dialog box, enter the **Data Source Name (DSN)** and an optional **Description** . To do so, click **Add** to insert keywords (parameters) and values that define the settings needed to connect to your database, including database name, server host, database user name (such as dbadamin), database password, and port. Then, click **OK** .
8. In the ODBC Administrator dialog box, click **Apply** .
   See ODBC DSN connection properties for a complete list of parameters including those specific to Vertica.

After configuring the ODBC Administrator Tool, you may need to configure additional driver settings before you can use your DSN, depending on your environment. See Additional ODBC Driver Configuration Settings for details.

> **Note**
> To test your connection, use the iodbctest utility. For details, see Testing an ODBC DSN using iodbctest .

In this section

- Testing an ODBC DSN using iodbctest

Testing an ODBC DSN using iodbctest

The standard iODBC Driver Manager on OS X includes a utility named iodbctest that lets you test a DSN to verify that it is correctly configured. You pass this command a connection string in the same format that you would use to open an ODBC database connection. After configuring your DSN connection, you can run a query to verify that the connection works.

For example:

```
# iodbctest "DSN=VerticaDSN;UID=dbadmin;PWD=password"
iODBC Demonstration program
This program shows an interactive SQL processor
Driver Manager: 03.52.0607.1008
Driver: 07.01.0200 (verticaodbcw.so)
SQL> SELECT table_name FROM tables;
table_name
-----------------------------------------------------------------------------------------------------------------------
customer_dimension
product_dimension
promotion_dimension
date_dimension
vendor_dimension
employee_dimension
shipping_dimension
warehouse_dimension
inventory_fact
store_dimension
store_sales_fact
store_orders_fact
online_page_dimension
call_center_dimension
online_sales_fact
numbers
result set 1 returned 16 rows.
```

ODBC DSN connection properties

The following tables list the connection properties you can set in the DSNs for use with Vertica's ODBC driver. To set these parameters, see Setting DSN connection properties .

Required connection properties

These connection properties are the minimum required to create a functioning DSN.

> **Note**
> If you use a host name (Servername) whose DNS entry resolves to multiple IP addresses, the client attempts to connect to the first IP address returned by the DNS. If a connection cannot be made to the first address, the client attempts to connect to the second, then the third, continuing until it either connects successfully or runs out of addresses.

Property	Description
Driver	The file path and name of the driver used.
Database	The name of the database running on the server.
Servername	The host name or IP address of any active node in a Vertica cluster. You can provide an IPv4 address, IPv6 address, or host name. In mixed IPv4/IPv6 networks, the DNS server configuration determines which IP version address is sent first. Use the PreferredAddressFamily option to force the connection to use either IPv4 or IPv6. You can also use the aliases "server" and "host" for this property.
UID	The database username.

Optional properties

Property	Description
Port	The port number on which Vertica listens for ODBC connections. **Default:** 5433
PWD	The password for the specified user name. You may insert an empty string to leave this property blank. **Default:** None, login only succeeds if the user does not have a password set.
PreferredAddressFamily	The IP version to use if the client and server have both IPv4 and IPv6 addresses and you have provided a host name, one of the following: • ipv4 : Connect to the server using IPv4. • ipv6 : Connect to the server using IPv6. • none : Use the IP address provided by the DNS server. **Default:** none

Advanced settings

Property	Description
AutoCommit	A Boolean value that controls whether the driver automatically commits transactions after executing a DML statement. **Default:** true
BackupServerNode	A string containing the host name or IP address that client libraries can try to connect to if the host specified in ServerName is unreachable. Connection attempts continue until successful or until the list of server nodes is exhausted. **Valid values:** Comma-separated list of servers optionally followed by a colon and port number.
ConnectionLoadBalance	A Boolean value that indicates whether the connection can be redirected to a host in the database other than the ServerNode. This affects the connection only if the load balancing. is set to something other than "none". When the node differs from the node the client is connected to, the client disconnects and reconnects to the targeted node. See About Native Connection Load Balancing in the Administration Guide. **Default:** false
ConnSettings	A string containing SQL commands that the driver should execute immediately after connecting to the server. You can use this property to configure the connection, such as setting a schema search path. **Reserved symbol:** In the connection string semicolon (;) is a reserved symbol. To set multiple properties as part of ConnSettings properties, use %3B as the comma delimiter, and + (plus) for spaces.
ConnectionTimeout	The number of seconds to wait for a request to complete before returning to the client application. This is equivalent to the SQL_ATTR_CONNECTION_TIMEOUT parameter in the ODBC API. **Default** : 0 (no timeout)

ConvertSquareBracketIdentifiers	Controls whether square-bracket query identifiers are converted to a double quote identifier for compatibility when making queries to a Vertica database. **Default:** false
DirectBatchInsert	Deprecated, always set to true.
DriverStringConversions	Controls whether the ODBC driver performs type conversions on strings sent between the ODBC driver and the database. Possible values are: • NONE: No conversion in either direction. This results in the highest performance. • INPUT: Strings sent from the client to the server are converted, but strings sent from the server to the client are not. • OUTPUT: Strings sent by the server to the client are converted, but strings sent from the client to the server are not. • BOTH: Strings are converted in both directions. **Default:** OUTPUT
Locale	The locale used for the session. Specify the locale as an ICU Locale. **See **the ICU User Guide for a complete list of properties that can be used to specify a locale. **Default:** en_US@collation=binary
PromptOnNoPassword	[Windows only] Controls whether users are prompted to enter a password, if none is supplied by the connection string or DSN used to connect to Vertica. See Prompting windows users for passwords . **Default:** false
ReadOnly	A true or false value that controls whether the connection can read data only from Vertica. **Default:** false
ResultBufferSize	Size of memory buffer for the large result sets in streaming mode. A value of 0 means ResultBufferSize is turned off. **Default:** 131072 (128KB)
TransactionIsolation	Sets the transaction isolation for the connection, one of the following: • Read Committed • Serializable • Server Default See Changing Transaction Isolation Levels for an explanation of transaction isolation. **Default:** Server Default

Identification

Property	**Description**	**Standard/ Vertica**

Description	Description for the DSN entry. **Required?** No Insert an empty string to leave the description empty.	Standard
Label / SessionLabel	Sets a label for the connection on the server. This value appears in the client_label column of the V_MONITOR.SESSIONS system table. Label and SessionLabel are synonyms and can be used interchangeably.	Vertica

OAuth connection properties

The following connection properties pertain to OAuth in ODBC.

> **Caution**
> The unixODBC driver manager supports parameter values of up to 1000 characters in odbc.ini . If your parameter value is greater than 1000 characters (for example, OAuthAccessToken), you must pass it through a connection string rather than specifying it in odbc.ini .

Property	Description
OAuthAccessToken	Required if oauthrefreshtoken is unspecified, an OAuth token that authorizes a user to the database.
OAuthRefreshToken	Required if oauthaccesstoken is unspecified, allows a user to refresh and obtain a new oauthaccesstoken when their old one expires. If you set this parameter, you must also set the following refresh properties in oauthjsonconfig: • oauthrefreshtoken • oauthdiscoveryurl or oauthtokenurl • oauthclientid • oauthclientsecret In cases where introspection fails (e.g. when the access token expires), Vertica responds to the request with an error. If introspection fails and OAuthRefreshToken is specified, the driver attempts to refresh and silently retrieve a new access token. Otherwise, the driver passes error to the client application.

OAuthJsonConfig	A JSON string or file that lets you set the following:
	
	• oauthclientid : The client ID of the client application registered in the identity provider.
	• oauthclientsecret : The client secret of the client application registered in the identity provider.
	• oauthtokenurl : The endpoint to which token refresh requests are sent. The format for this depends on your provider. For examples, see the Keycloak and Okta documentation.
	If oauthdiscoveryurl is not set: this parameter is only required for IDP valdiation if oauthdiscoveryurl , but should also be set for JWT validation to support token refresh.
	• oauthauthurl : The authorization endpoint used for single sign-on. For examples, see the Keycloak and Okta documentation.
	• oauthdiscoveryurl : Also known as the OpenID Provider Configuration Document , this endpoint contains a list of all other endpoints supported by the IDP. If set, the other endpoints (such as oauthtokenurl and oauthauthurl) do not need to be specified. This parameter is only supported for Keycloak. For other identity providers like Okta, the endpoints must be set manually.
	If you set both oauthdiscoveryurl and another endpoint (like oauthtokenurl), oauthdiscoveryurl takes precedence.
	This parameter is only required for IDP valdiation , but should also be set for JWT validation to support token refresh.
	• oauthscope: The requested OAuth scopes, delimited with spaces. These scopes define the extent of access to the resource server (in this case, Vertica) granted to the client by the access token. For details, see the OAuth documentation .
	• oauthvalidatehostname: Boolean, whether to verify the subjectAltName of the identity provider host. If enabled, the IP address or hostname must be set as the subjectAltName in its certificate. Hostname verification is enabled by default.
	
	Unlike oauthaccesstoken or oauthrefreshtoken, which must be set programmatically by the client when they attempt to connect, the same oauthjsonconfig can be reused between connections to the database.
	
	For example, to set it as a JSON string in ODBC.ini as part of the DSN:
	
	`OAuthJsonConfig = {`
	`"oauthdiscoveryurl": "http://203.0.113.1:8080/realms/myrealm/.well-known/openid-configuration",`
	`"oauthtokenurl": "http://203.0.113.1:8080/auth/realms/myrealm/protocol/openid-connect/token",`
	`"oauthclientid": "vertica",`
	`"oauthclientsecret": "eba23135-834f-1341-aa34-bf9345713dfc",`
	`"oauthscope": "offline_access openid",`
	`"oauthvalidatehostname": "false"`
	`}`
	
	To set it inside separate a JSON configuration file:
	
	1. Create oauth_config.json :
	`{`
	`"oauthdiscoveryurl": "http://203.0.113.1:8080/realms/myrealm/.well-known/openid-configuration",`
	`"oauthtokenurl": "http://203.0.113.1:8080/auth/realms/myrealm/protocol/openid-connect/token",`
	`"oauthclientid": "vertica",`
	`"oauthclientsecret": "eba23135-834f-1341-aa34-bf9345713dfc",`
	`"oauthscope": "offline_access openid",`
	`"oauthvalidatehostname": "false"`
	`}`
	2. Specify oauth_config.json inside ODBC.ini :
	`OAuthJsonconfig = {"oauthjsonfile": "/path/to/oauth_config.json"}`

Encryption

Property	Description	Standard/ Vertica

SSLMode	Controls whether the connection to the database uses SSL encryption, one of the following. For information on using these parameters to configure TLS, see Configuring TLS for ODBC Clients : • require : Requires that the server use TLS. If the TLS connection attempt fails, the client rejects the connection. • prefer : Prefers that the server use TLS. The client first attempts to connect using TLS. If that attempt fails, the client attempts to connect again in plaintext. • allow : Makes a connection to the server whether the server uses TLS or not. The first connection attempt to the database is attempted over a clear channel. If that fails, a second connection is attempted over TLS. • verify_ca : The client verifies that the server's certificate was issued by a trusted certificate authority (CA). • verify_full : The client verifies that the following conditions are met: • The server's certificate was issued by a trusted CA. • One of the following: ◦ The server's hostname matches the common name specified in the server's certificate. ◦ The server's hostname or IP address appears in the Subject Alternative Name (SAN) field of the server's certificate. • disable : Never connect to the server using TLS. This setting is typically used for troubleshooting. **Default:** prefer	Vertica
SSLCertFile	The absolute path of the client's public certificate file. This file can reside anywhere on the system.	Vertica
SSLKeyFile	The absolute path to the client's private key file. This file can reside anywhere on the system.	Vertica

Third-party compatibility

Property	**Description**	**Default**	**Standard/ Vertica**
ColumnsAsChar	Specifies how character column types are reported when the driver is in Unicode mode. When set to false, the ODBC driver reports the data type of character columns as WCHAR. If you set ColumnsAsChar to true, the driver identifies character column as CHAR. You typically use this setting for compatibility with some third-party clients. **Default:** false	false	Vertica
ThreePartNaming	A Boolean value that controls how catalog names are interpreted by the driver. When this value is false, the driver reports that catalog names are not supported. When catalog names are not supported, they cannot be used as a filter in database metadata API calls. In this case, the driver returns NULL as the catalog name in all driver metadata results. When this value is true, catalog names can be used as a filter in database metadata API calls. In this case, the driver returns the database name as the catalog name in metadata results. Some third-party applications assume a certain catalog behavior and do not work properly with the default values. Enable this option if your client software expects to get the catalog name from the database metadata and use it as part of a three-part name reference. **Default:** false for UNIX, true for Windows	false (UNIX) true (Window)	Vertica

EnforceBatchInsertNullConstraints	Prevents NULL values from being loaded into columns with a NOT NULL constraint during batch inserts. When this value is set to true, batch inserts roll back when NULL values are inserted in to columns with NOT NULL constraints. When this value is set to false, batch insert behavior is unchanged. Vertica recommends only using this property with SAP Data Services as it could negatively impact database performance.	false	Vertica

Kerberos connection properties

Use the following properties for client authentication using Kerberos.

Property	Description	Standard/ Vertica
KerberosServiceName	Provides the service name portion of the Vertica Kerberos principal; for example: vertichost@EXAMPLE.COM **Default:** vertica	Vertica
KerberosHostname	Provides the instance or host name portion of the Vertica Kerberos principal; for example: verticaosEXAMPLE.COM **Default:** Value specified in the servername connection string property	Vertica

See also

ODBC driver settings

Setting DSN connection properties

The properties in the following tables are common for all user and system DSN entries. The examples provided are for Windows clients.

To edit DSN properties:

- On UNIX and Linux client platforms, you can edit the odbc.ini file. The location of this file is specific to the driver manager. See Creating an ODBC DSN for Linux .
- On Windows client platforms, you can edit some DSN properties using the Vertica ODBC client driver interface. See Creating an ODBC DSN for windows clients .
- You can also edit the DSN properties directly by opening the DSN entry in the Windows registry (for example, at HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\ *DSNname*). Directly editing the registry can be risky, so you should only use this method for properties that cannot be set through the ODBC driver's user interface, or via your client code.
- You can set properties in the connection string when opening a connection using the SQLDriverConnect() function:

```
sqlRet = SQLDriverConnect(sql_hDBC, 0, (SQLCHAR*)"DSN=DSNName;Locale=en_GB@collation=binary", SQL_NTS, szDNS, 1024,&nSize,
SQL_DRIVER_NOPROMPT);
```

> **Note**
> In the connection string ';' is a reserved symbol. If you need to set multiple properties as part of the ConnSettings property use '%3B' in place of ';'. Also use '+' instead of spaces.
>
> For example:
>
> ```
> sqlRet = SQLDriverConnect(sql_hDBC, 0, (SQLCHAR*)"DSN=Vertica
> SQL;ConnSettings=set+search_path+to+a,b,c%3Bset+locale=ch;SSLMode=prefer", SQL_NTS,
> szDNS, 1024,&nSize, SQL_DRIVER_NOPROMPT);
> ```

- Your client code can retrieve DSN property values after a connection has been made to Vertica using the SQLGetConnectAttr() and SQLGetStmtAttr() API calls. Some properties can be set and using SQLSetConnectAttr() and SQLSetStmtAttr() .
  For details of the list of properties specific to Vertica see ODBC Header Files specific to Vertica .

ODBC driver settings

> **Note**
> While required settings are required for all platforms, these settings automatically set by the Windows and macOS installers , so all directives to change these settings are for Linux users.

- DriverManagerEncoding : The UTF encoding standard used by the driver manager. This can be one of the following:
  - UTF-8
  - UTF-16
  - UTF-32

  The ODBC driver encoding must match that of your driver manager. The following table lists default encodings for various platforms that take effect if you do not set this parameter. If the defaults do not match the encoding used by your driver manager, you must set it manually. Consult your driver manager's documentation for details on its encoding.

  > **Note**
  > While both UTF-16 and UTF-8 are valid settings for the DataDirect driver manager, UTF-16 is recommended.

Client Platform	Default Encoding
Linux 32-bit	UTF-32
Linux 64-bit	UTF-32
Linux Itanium 64-bit	UTF-32
OS X	UTF-32
Windows 32-bit	UTF-16
Windows 64-bit	UTF-16

- ErrorMessagesPath : Required, the path of the directory containing the ODBC driver's error message files. These files (ODBCMessages.xml and VerticaMessages.xml) are stored in the same directory as the Vertica ODBC driver files (for example, opt/vertica/en-US in the downloaded .tar).
- ODBCInstLib : The path to the ODBC installer library. This setting is only required if the directory containing the library is not set in the LD_LIBRARY_PATH or LIB_PATH environment variables. The library files for the major driver managers are:
- UnixODBC: libodbcinst.so
- iODBC: libiodbcinst.so (libiodbcinst.2.dylib on macOS)
- DataDirect: libodbcinst.so

You can also control client-server message logging for both ODBC and ADO.NET. For details, see Configuring ODBC logs .

Linux and macOS

To set these parameters on Linux or macOS:

1. Create a file vertica.ini anywhere on the client system. Common locations are in /etc/ for a shared configuration, or the home directory for a per-user configuration.
2. Verify that users of the ODBC driver have read privileges on the file.
3. Set the VERTICAINI environment variable to the path of vertica.ini . For example:

```
$ export VERTICAINI=/etc/vertica.ini
```

1. Create a section called [Driver] in vertica.ini :

```
[Driver]
```

1. Under [Driver] , set parameters with the following format. Each parameter must have its own line:

```
[Driver]
DriverManagerEncoding=UTF-16
ODBCInstLib=/usr/lib64/libodbcinst.so
```

Windows

The Windows client driver installer automatically configures all necessary settings for the ODBC driver. Settings are stored in the registry in HKEY_LOCAL_MACHINE\SOFTWARE\Vertica\ODBC\Driver .

If you want to configure ODBC further, use the **ODBC Data Sources** program.

Configuring ODBC logs

The following parameters control whether and how the ODBC client driver logs messages between the client and server.

The way you set these parameters differs between operating systems:

- On Linux and macOS, edit vertica.ini you created during the installation . For example, to log all warnings and more severe messages to log files in /tmp/ :

  ```
  [Driver]
  LogLevel=4
  LogPath=/tmp
  ```

- On Windows, edit the keys in the Windows Registry under HKEY_LOCAL_MACHINE\SOFTWARE\Vertica\ODBC\Driver .

Parameters

- **LogLevel** : The severity of messages that are logged between the client and the server. The valid values are:
  - 0: No logging
  - 1: Fatal errors
  - 2: Errors
  - 3: Warnings
  - 4: Info
  - 5: Debug
  - 6: Trace (all messages)

  The value you specify for this setting sets the minimum severity for a message to be logged. For example, setting LogLevel to 3 means that the client driver logs all warnings, errors, and fatal errors.
- **LogPath** : The absolute path of a directory to store log files. For example: /var/log/verticaodbc

Diverting log entires to ETW (windows)

On Windows clients, ODBC log entries can be sent to Event Tracing for Windows (ETW) so they appear in the Windows Event Viewer:

- Register the driver as a Windows Event Log provider and enable the logs.
- Activate ETW by adding a string value LogType with data ETW to your Windows Registry.
- Understand how Vertica compresses log levels for the Windows Event Viewer.
- Know where to find the logs within Event Viewer.
- Understand the meaning of the Event IDs in your log entries.

Registering the ODBC driver as a windows event log provider

To use ETW logging, you must register the ODBC driver as a Windows Event Log provider. You can choose to register either the 32-bit or 64-bit driver. After you have registered the driver, you must enable the logs.

Important

If you do not both register the driver and enable the logs, output is directed to stdout.

1. Open a command prompt window as Administrator, or launch the command prompt with the Run as Administrator option.
   Important
   You must have administrator privileges to successfully complete the next step.

2. Run the command wevtutil im to register either the 32-bit or 64-bit version of the driver.
   - For the 64-bit ODBC driver, run:

     ```
     wevtutil im "c:\Program Files\Vertica Systems\ODBC64\lib\VerticaODBC64.man"
     /resourceFilePath:"c:\Program Files\Vertica Systems\ODBC64\lib\vertica_9.1_odbc_3.5.dll"
     /messageFilePath:"c:\Program Files\Vertica Systems\ODBC64\lib\vertica_9.1_odbc_3.5.dll"
     ```

- For the 32-bit ODBC driver, run:

```
wevtutil im "c:\Program Files (x86)\Vertica Systems\ODBC32\lib\VerticaODBC32.man"
/resourceFilePath:"c:\Program Files (x86)\Vertica Systems\ODBC32\lib\vertica_9.1_odbc_3.5.dll"
/messageFilePath:"c:\Program Files (x86)\Vertica Systems\ODBC32\lib\vertica_9.1_odbc_3.5.dll"
```

3. Run the command wevtutil sl to enable the logs.
   - For 64-bit ODBC driver logs, run:

```
wevtutil sl VerticaODBC64/e:true
```

   - For the 32-bit ODBC driver logs, run:

```
wevtutil sl VerticaODBC32/e:true
```

> **Note**
> Should you want to later disable the logs, you can use the same wevtutil sl command, substituting /e:false in place of /e:true when you issue the statement. Alternatively, you can enable or disable logs within the Windows Event Viewer itself.

Add the string value LogType

By default, Vertica does not send ODBC log entries to ETW. To activate ETW, add the string LogType to your Windows registry, and set its value to ETW .

1. Start the registry editor by typing regedit.exe in the Windows Run command box.
2. Navigate to HKEY_LOCAL_MACHINE\SOFTWARE\Vertica\ODBC\Driver in the registry.
3. Right-click in the right pane of the **Registry Editor** window.
4. Select **New** , then select **String Value** .
5. Change the name of the string value from New Value #1 to LogType .
6. Double-click the new LogType entry. When prompted for a new value, enter ETW .
7. Exit the registry editor.

ETW is disabled by default. When ETW is enabled, you can disable it by clearing the value ETW from the LogType string.

LogLevel in the windows event viewer

While LogLevel ranges from 0 through 6, this range is compressed for the Windows Event Viewer to a range of 0 through 3.

Vertica LogLevel Setting	Vertica LogLevel Description	Log level sent to the Windows Event Viewer	Log level displayed by the Windows Event Viewer
0	(No logging)	0	(No logging)
1	Fatal Errors	1	Critical
2	Errors	2	Error
3	Warnings	3	Warning
4	Info	4	Information
5	Debug	4	
6	Trace (all messages)	4	

The following examples show how LogLevel is converted when displayed in the Windows Event Viewer.

- A LogLevel of 5 sends fatal errors, errors, warnings, info and debug log level entries to Event Viewer as Level 4 (Information).
- A LogLevel of 6 sends fatal errors, errors, warnings, debug and trace log level entries to Event Viewer as Level 4.

Finding logs in the event viewer

1. Launch the **Windows Event Viewer** .
2. From **Event Viewer (Local)** , expand **Applications and Services Logs** .
3. Expand the folder that contains the log you want to review (for example, VerticaODBC64).
4. Select the Vertica ODBC log under the folder. Entries appear in the right pane.

5. Note the value in the **Event ID** field. Each Event Log entry includes one of four Event IDs:
   - 0: Informational (debug, info, and trace events)
   - 1: Error
   - 2: Fatal event
   - 3: Warning

Python client drivers

Vertica supports several Python drivers for creating client applications.

Prerequisites

To create Python client applications, you must install the required drivers.

In this section

- Installing Python client drivers

Installing Python client drivers

Vertica supports several Python client drivers.

Installing vertica-python

See the vertica-python repository for installation and usage instructions.

Installing pyodbc

The pyodbc module interacts with the Vertica ODBC client driver. To install it:

1. Install the ODBC client driver.
2. Install compatible versions of Python and pyodbc.

Node.js client driver

The open-source vertica-nodejs client driver lets you interact with your database with JavaScript. For details, see the vertica-nodejs package on npm.

Go client driver

The open-source vertica-sql-go driver lets you interact with your database with Go. For details, see vertica-sql-go.

OLE DB client driver

The OLE DB client driver is an interface for Microsoft Analysis Services (SSAS) and C# client applications to interact with your Vertica database.

In this section

- Installing the OLE DB client driver
- The Microsoft connectivity pack for windows

Installing the OLE DB client driver

To install the Vertica OLE DB client driver:

1. Download the Windows client driver installer. For details on the drivers included in this installer, see Windows client driver installer.
2. Run the installer and follow the prompts to install the drivers.
3. Reboot your system.

After installing the OLE DB client driver, you can configure ETW logging.

For details on how the OLE DB client driver integrates with other Microsoft components, see Microsoft component configuration.

For a list of connection properties, see OLE DB connection properties.

In this section

- OLE DB connection properties
- Configuring OLE DB logs

OLE DB connection properties

Use the Connection Manager to set the OLE DB connection string properties, which define your connection. You access the Connection Manager from within Visual Studio.

These connection parameters appear on the Connection page.

Parameters	Action
Provider	Select the native OLE DB provider for the connection.
OLE DB Provider	Indicates Vertica OLE DB Provider.
Server or file name	Enter the server or file name.
Location	Not supported.
Use Windows NT Integrated Security	Not supported.
Use a specific user name and password	Enter a user name and password. **Connect with No Password** : Select the **Blank password** check box. **Save and Encrypt Password:** Select **Allow saving password** .
Initial Catalog	The name of the database running on the server.

The **All** page from the **Connection Manager** dialog box includes all possible connection string properties for the provider.

The table that follows lists the connection parameters for the **All** page.

For OLE DB properties information specific to Microsoft, see the Microsoft documentation OLE DB Properties .

Parameters	Action
Extended Properties	Not supported.
Locale Identifier	Indicates the Locale ID. **Default:** 0
Mode	Specifies access permissions. **Default:** 0
Connect Timeout	Not supported. **Default:** 0
General Timeout	Not supported.
File Name	Not supported.
OLE DB Services	Specifies which OLE DB services to enable or disable.
Password	Specifies the password for the user ID. For no password, insert an empty string.

Persist Security Info	A security measure. When False, security sensitive-information, such as the password, is not returned as part of the connection if the connection is open or has ever been in an open state. **Default:** true
User ID	The database username.
Data Source	The host name or IP address of any active node in a Vertica cluster. You can provide an IPv4 address, IPv6 address, or host name. In mixed IPv4/IPv6 networks, the DNS server configuration determines which IP version address is sent first. Use the PreferredAddressFamily option to force the connection to use either IPv4 or IPv6.
Initial Catalog	The name of the database running on the server.
Provider	The name of the OLE DB Provider to use when connecting to the Data Source. **Default:** VerticaOLEDB.1
BackupServerNode	A designated host name or IP address to use if the ServerName host is unavailable. Enter as a string. Connection attempts continue until successful or until the list of server nodes is exhausted. **Valid values:** Comma-separated list of servers optionally followed by a colon and port number. For example: server1:5033,server2:5034
ConnectionLoadBalance	A Boolean value that determines whether the connection can be redirected to a host in the database other than the ServerNode. This parameter affects the connection only if load balancing is set to a value other than NONE. When the node differs from the node that the client is connected to, the client disconnects and reconnects to the targeted node. See About Native Connection Load Balancing in the Administration Guide. **Default:** false
ConnSettings	SQL commands that the driver should execute immediately after connecting to the server. Use to configure the connection, such as setting a schema search path. **Reserved symbol:** ';' To set multiple parameters in this field use '%3B' for ','. Spaces: Use '+'.
ConvertSquareBracketIdentifiers	Controls whether square-bracket query identifiers are converted to a double quote identifier for compatibility when making queries to a Vertica database. **Default:** false
DirectBatchInsert	Deprecated, always set to true.
KerberosHostName	Provides the instance or host name portion of the Vertica Kerberos principal; for example: verticaosEXAMPLE.COM
KerberosServiceName	Provides the service name portion of the Vertica Kerberos principal; for example: vertichost@EXAMPLE.COM

Label	Sets a label for the connection on the server. This value appears in the session_id column of system table SESSIONS .
LogLevel	Specifies the amount of information included in the log. Leave this field blank or set to 0 unless otherwise instructed by Vertica Customer Support.
LogPath	The path for the log file.
Port	The port number on which Vertica listens for OLE DB connections. **Default:** port 5433
PreferredAddressFamily	The IP version to use if the client and server have both IPv4 and IPv6 addresses and you have provided a host name, one of the following: • ipv4 : Connect to the server using IPv4. • ipv6 : Connect to the server using IPv6. • none : Use the IP address provided by the DNS server.
SSLCertFile	The absolute path of the client's public certificate file. This file can reside anywhere on the system.
SSLKeyFile	The absolute path to the client's private key file. This file can reside anywhere on the system.
SSLMode	Controls whether the connection to the database uses SSL encryption, one of the following: • require : Requires the server to use SSL. If the server cannot provide an encrypted channel, the connection fails. • prefer : Prefers that the server use SSL. If the server does not offer an encrypted channel, the client requests one. The first attempt is made with SSL. If that attempt fails, the second attempt is over a clear channel. • allow : Makes a connection to the server whether or not the server uses SSL. The first attempt is made over a clear channel. If that attempt fails, a second attempt is over SSL. • disable : Never connects to the server using SSL. Typically, you use this setting for troubleshooting. **Default:** prefer

Configuring OLE DB logs

The following parameters control how the OLE DB client driver logs messages between the client and server. To set them, edit the keys in the Windows Registry under HKEY_LOCAL_MACHINE\SOFTWARE\Vertica\OLEDB\Driver :

- **LogLevel** : The severity of messages that are logged between the client and the server. The valid values are:
  - 0: No logging
  - 1: Fatal errors
  - 2: Errors
  - 3: Warnings
  - 4: Info
  - 5: Debug
  - 6: Trace (all messages)

  The value you specify for this setting sets the minimum severity for a message to be logged. For example, setting LogLevel to 3 means that the client driver logs all warnings, errors, and fatal errors.
- **LogPath** : The absolute path of a directory to store log files. For example: /var/log/verticaoledb

Diverting OLE DB log entries to ETW

On Windows clients, you can direct Vertica to send OLE DB log entries to Event Tracing for Windows (ETW). Once set, OLE DB log entries appear in the Windows Event Viewer. To use ETW:

- Register the driver as a Windows Event Log provider, and enable the logs.
- Activate ETW by adding a string value to your Windows Registry.

- Understand how Vertica compresses log levels for the Windows Event Viewer.
- Know where to find the logs within Event Viewer.
- Understand the meaning of the Event IDs in your log entries.

Registering the OLE DB driver as a windows event log provider

To use ETW logging, you must register the OLE DB driver as a Windows Event Log provider. You can choose to register either the 32-bit or 64-bit driver. Once you have registered the driver, you must enable the logs.

Important
If you do not both register the driver and enable the logs, output is directed to stdout.

1. Open a command prompt window as Administrator, or launch the command prompt with the Run as Administrator option.
   Important
   You must have administrator privileges to successfully complete the next step.

2. Run the command wevtutil im to register either the 32-bit or 64-bit version of the driver.
   1. For the 64-bit OLE DB driver, run:
      ```
      wevtutil im "c:\Program Files\Vertica Systems\OLEDB64\lib\VerticaOLEDB64.man"
      /resourceFilePath:"c:\Program Files\Vertica Systems\OLEDB64\lib\vertica_8.1_oledb.dll"
      /messageFilePath:"c:\Program Files\Vertica Systems\OLEDB64\lib\vertica_8.1_oledb.dll"
      ```
   2. For the 32-bit OLE DB driver, run:
      ```
      wevtutil im "c:\Program Files (x86)\Vertica Systems\OLEDB32\lib\VerticaOLEDB32.man"
      /resourceFilePath:"c:\Program Files (x86)\Vertica Systems\OLEDB32\lib\vertica_8.1_oledb.dll"
      /messageFilePath:"c:\Program Files (x86)\Vertica Systems\OLEDB32\lib\vertica_8.1_oledb.dll"
      ```
3. Run the command wevtutil sl to enable the logs.
   1. For 64-bit OLE DB driver logs, run:
      ```
      wevtutil sl VerticaOLEDB64/e:true
      ```
   2. For the 32-bit ODBC driver logs, run:
      ```
      wevtutil sl VerticaOLEDB32/e:true
      ```

> **Note**
> Should you want to later disable the logs, you can use the same wevtutil sl command, substituting /e:false in place of /e:true when you issue the statement. Alternatively, you can enable or disable logs within the Windows Event Viewer itself.

Add the string value LogType

By default, Vertica does not send OLE DB log entries to ETW. To activate ETW, add the string LogType to your Windows registry, and set its value to ETW .

1. Start the registry editor by typing regedit.exe in the Windows Run command box.
2. Navigate, in the registry, to: HKEY_LOCAL_MACHINE\SOFTWARE\Vertica\OLEDB\Driver .
3. Right-click in the right pane of the **Registry Editor** window.
4. Select **New** , then select **String Value** .
5. Change the name of the string value from New Value #1 to LogType .
6. Double-click the new LogType entry. When prompted for a new value, enter ETW .
7. Exit the registry editor.

ETW is off by default. When ETW is activated, you can subsequently turn it off by clearing the value ETW from the LogType string.

LogLevel in the windows event viewer

While LogLevel ranges from 0 through 6, this range is compressed for the Windows Event Viewer to a range of 0 through 3.

Vertica LogLevel Setting	Vertica LogLevel Description	Log level sent to the Windows Event Viewer	Log level displayed by the Windows Event Viewer
0	(No logging)	0	(No logging)

1	Fatal Errors	1	Critical
2	Errors	2	Error
3	Warnings	3	Warning
4	Info	4	Information
5	Debug	4	
6	Trace (all messages)	4	

The following examples show how LogLevel is converted when displayed in the Windows Event Viewer.

- A LogLevel of 5 sends fatal errors, errors, warnings, info and debug log level entries to Event Viewer as Level 4 (Information).
- A LogLevel of 6 sends fatal errors, errors, warnings, debug and trace log level entries to Event Viewer as Level 4.

Finding logs in the event viewer

1. Launch the **Windows Event Viewer** .
2. From **Event Viewer (Local)** , expand **Applications and Services Logs** .
3. Expand the folder that contains the log you want to review (for example, VerticaOLEDB64).
4. Select the Vertica ODBC log under the folder. Entries appear in the right pane.
5. Note the value in the **Event ID** field. Each Event Log entry includes one of four Event IDs:
   - 0: Informational (debug, info, and trace events)
   - 1: Error
   - 2: Fatal event
   - 3: Warning

The Microsoft connectivity pack for windows

The Vertica Microsoft Connectivity Pack for Windows provides a configuration file for you to access Microsoft Business Intelligence tools. The Connectivity Pack is installed as part of the Client Drivers and Tools for Windows.

For details about the components included with the Microsoft Connectivity Pack, see Microsoft components .

In this section

- Microsoft components
- Compatibility issues and limitations

Microsoft components

This section describes the Microsoft Business Intelligence components you can use with Microsoft Visual Studio and Microsoft SQL Server. After configuration, you can use these Microsoft components to develop business solutions using your Vertica server.

Important

Client Drivers and Tools for Windows includes the Vertica Microsoft Connectivity Pack. To use the Connectivity Pack to access Microsoft Business Intelligence tools, reboot your system after installation to ensure integration.

In this section

- Microsoft component configuration
- SSDT-BI
- SQL server analysis services (SSAS) support
- SQL server integration services (SSIS) support
- SQL server reporting services (SSRS) support

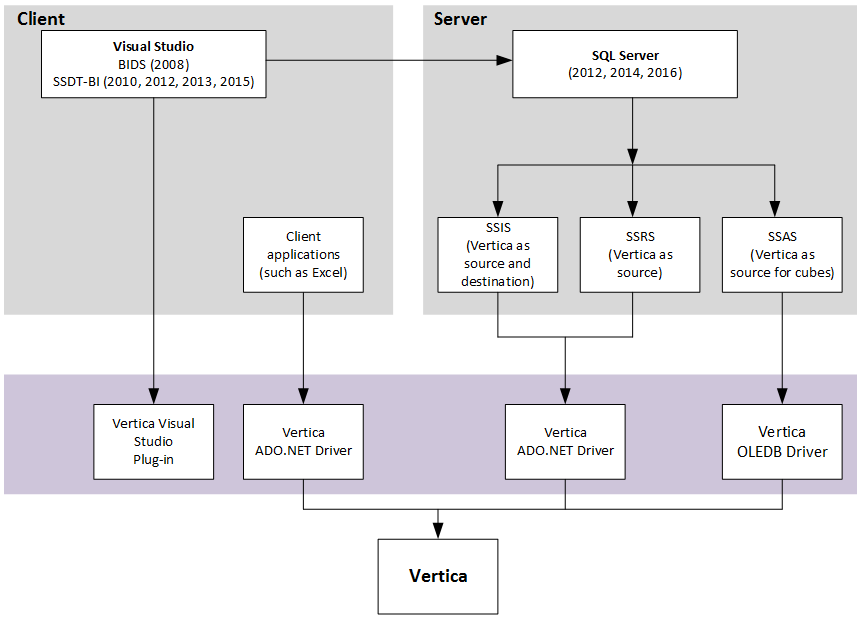
Microsoft component configuration

The Vertica ADO.NET driver, the Visual Studio plug-in, and the OLE DB driver allow you to integrate your Vertica server with an environment that includes Microsoft components previously installed on your system. Additional tools are also available for integration with Microsoft SQL Server.

The available drivers provide integration with the following Microsoft components:

- SQL Server Data Tool - Business Intelligence (SSDT-BI) for Visual Studio 2010/2012/2013/2015 for use with SQL Server 2012, 2014, and 2016. SSDT-BI replaces BIDS for 2010, 2012, 2013, and 2015. It serves the same purpose as BIDS, providing a development environment for developing business intelligence solutions.
- SQL Server Analysis Services (SSAS) for SQL Server 2012, 2014, and 2016. Use SSAS for OLAP and data mining, while using Vertica as the source for cube creation.
- SQL Server Integration Services (SSIS) for SQL Server 2012, 2014, and 2016. SSIS provides SQL Server Type Mappings to map data types between Vertica and SQL Server. Use SSIS for data migration, data integration and workflow, and ETL.

The following figure displays the relationship between Microsoft components and Vertica dependencies.



SSDT-BI

SQL Server Data Tool - Business Intelligence (SSDT-BI) is a client-based application that provides a development environment for creating business solutions in Visual Studio 2012, 2013, and 2015. SSDT-BI includes project types specific to SQL Server Business Intelligence.

You can use the Visual Studio Shell Integration plug-in to browse a database from within the Visual Studio Server Explorer, allowing you to work outside of SSDT-BI development to perform tasks, such as listing tables or inserting data. When you use Visual Studio in SSDT-BI mode, you can develop business solutions using the data in your Vertica database. For example, you can create cubes or open tables.

You cannot use Microsoft Visual Studio 2012/2013/2015 with SSDT-BI development to create a SQL Server 2008 Business Intelligence solution.

SQL server analysis services (SSAS) support

BIDS or SSDT-BI includes the Analysis Services project for developing online analytical processing (OLAP) for business intelligence applications. This project type includes templates for:

- Cubes
- Dimensions
- Data sources
- Data source views

It also provides the tools for working with these objects.

> **Note**
> **Note** : OpenText recommends that you use the Vertica OLE DB driver when connecting to the Vertica server from SSAS due to improved performance.

You can find the OLE DB connection properties in OLE DB connection properties .

SQL server integration services (SSIS) support

BIDS or SSDT-BI includes the Integration Services project for developing ETL solutions. This project type includes templates for:

- Packages
- Data sources
- Data source views

It also provides the tools for working with these objects.

You can find support for using Vertica as a data source and target from both SSIS and the import/export wizard. You must install mapping files specific to Vertica on the Integration Server and BIDS or SSDT-BI workstation to enable this capability. The Vertica Client Drivers and Tools for Windows installs these mapping files as the "SQL Server Type Mappings" component(s) in both 32-bit and 64-bit versions.

> **Note**
> **Note** : Always use the Vertica ADO.NET driver when connecting to the Vertica server from SSIS.

SQL server reporting services (SSRS) support

BIDS or SSDT-BI includes Report projects for developing reporting solutions.

You can use Vertica as a data source for Reporting Services. The installer implements various configuration file modifications to enable this capability on both the BIDS or SSDT-BI workstation and the Reporting Services server.

Compatibility issues and limitations

This section lists compatibility issues and limitations for integrating the Microsoft Connectivity Pack with Microsoft Visual Studio and Microsoft SQL Server.

In this section

- SSDT-BI limitations
- SSAS limitations
- SSIS data type limitations
- SSRS limitations

SSDT-BI limitations

SSDT-BI is a 32-bit development environments for Analysis Services, Integration Services, and Reporting Services projects. It is not designed to run on the Itanium 64-bit architecture and thus are not installed on Itanium servers.

SSAS limitations

- The SSAS Tabular Model is not supported.
- If, after installing the Vertica OLE DB driver, an SSAS cube build fails, restart the SSAS service.

SSIS data type limitations

The following sections cover data type limitations when using SQL Server Integration Services (SSIS).

Time data transfer

When transferring time data, SSIS uses the TimeSpan data type that supports precision greater than six digits. The Vertica ADO.NET driver translates TimeSpan as an Interval data type that supports up to six digits. The Interval type is not converted to the TimeSpan type during transfer. As a result, if the time value has a precision of more than six digits, the data is truncated, not rounded.

For information on ADO.NET data types, refer to ADO.NET data types .

DATE and DATETIME precision

To function without errors, DATE and DATETIME have a range from 0001-01-01 00:00:00.0000000 to 9999-12-31 23:59:59.999999.

In SSIS, the DATETIME type (DT_TIMESTAMP) supports only a scale up to three decimal places for seconds. Any decimal places after that are automatically discarded. You can perform derived column transformations only on DATETIME values between January 1, 1753 through December 31, 9999.

Numeric precision

The maximum and minimum decimal allowed is:

- Max: +79,228,162,514,264,337,593,543,950,335
- Min: -79,228,162,514,264,337,593,543,950,335

For example, if the scale is 16, the range of values is:

+/- 7,922,816,251,426.4337593543950335

The valid scale range is any number that is smaller than 29 and greater than 38. Using a scale between 29 and 38 does not generate an error.

See: http://msdn.microsoft.com/en-us/library/system.decimal.maxvalue.aspx

Unsupported floating point values

SQL Server does not support NaN, Infinity, or –Infinity values. These values are supported when you use SSIS to transfer between Vertica instances, but they are not supported with a SQL Server Destination.

String Conversion

The CHAR and VARCHAR data types used in SSIS are DT_WSTR, with a maximum length of 4000 characters.

In SSIS, Vertica strings are converted to Unicode strings in SSIS to handle multi-lingual data. You can convert these strings to ASCII using a Data Conversion Task.

Scale

Whenever you use a scale greater than 38, SSIS replaces it with a value of 4.

Interval conversion

SSIS does not support interval types. It converts them to TIME and strips off the day component. Any package that has interval types greater than a day returns incorrect results.

Data mapping issues with SQL server import and export wizard

When you create an Integrated Services package (SSIS) using the SQL Server Import and Export Wizard, certain data types do not automatically map correctly. A mapping issue can occur when you use the wizard with:

- SQL Server Native OLE DB Provider for SQL Server 2008 or 2012
- SQL Server Native Client 10.0/11.0 Provider for SQL Server 2010/2012

To avoid this issue, manually change the type mappings with BIDS or SSDT-BI.

Data transfer failures

When using an Integrated Services package (SSIS) with the SQL Server OLE DB Provider for SQL Server 2008 or 2012, certain data type transfers can fail when transferring from Vertica to SQL Server. To avoid this issue, use either BIDS or SSDT-BI when transferring data.

Batch insert of VARBINARY/LONG VARBINARY data types

Sometimes, one row of a batch insert of VARBINARY or LONG VARBINARY data types exceeds the data type limit:

- VARBINARY: 65 KB
- LONG VARBINARY: 32 MB

In such cases, all rows are rejected, rather than just the one row whose length exceeds the type limit. The batch insert fails with the message "row(s) are rejected".

To avoid this issue, use a predicate to filter out rows from the source that do not fit into the receiving database.

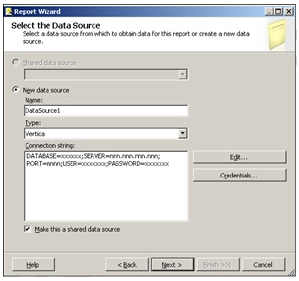
Boolean queries in SQL server query designer

When issuing a Boolean query in SQL Server Query Designer, you must enclose Boolean column values in quotes. Otherwise, you receive a SQL execution error (for example, `someboolean = 'true'`).

SSRS limitations

Data Connection Wizard Workaround

The SSRS Report Wizard provides a data connection wizard. After you select the wizard and enter all the connection information, the **OK** button is disabled. You cannot save your work and continue. The workaround is to not use the wizard and to use the following panel instead:



Report Wizard - Query Designer

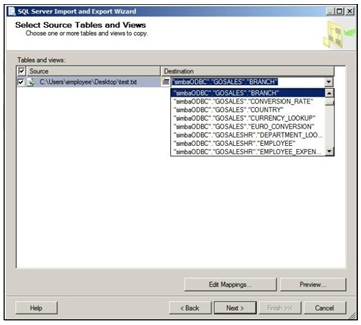
Vertica uses the Report Wizard's Generic Query Designer. Other data sources use a Graphical Query Designer that supports building queries visually. The Graphical Query Designer is a part of a package called Visual Data Tools (VDT). The Graphical Query Designer works only with Generic OLE DB providers and the built-in providers. You cannot use it with the Vertica Data Provider.

Report Builder

Report Builder is a web-based report design tool. It does not support creating reports using custom data extensions, so you cannot use it with Vertica. When you create a report using Report Builder, existing Vertica data sources appear in the list of available data sources. However, choosing a Vertica data source causes an error.

Schema Name Not Automatically Provided when Mapping Vertica **Destination**

Currently, when you map a Vertica destination, the schema name is not automatically provided. You must enter it manually or pick it from the drop-down menu as follows:



ADO.NET client driver

The Vertica ADO.NET driver lets you access Vertica with C# .

In this section

- Installing the ADO.NET client driver
- Configure ADO.NET logs

Installing the ADO.NET client driver

Prerequisites

The ADO.NET client driver requires the following:

- A supported operating system .
- At least 512MB of memory
- A supported version of the .NET Framework. If you do not meet this requirement, in most cases, the installer will offer to install this for you.

Installation

To install the ADO.NET client driver:

1. Download the Windows client driver installer. For details on the drivers included in this installer, see Windows client driver installer .
2. Run the installer and follow the prompts to install the drivers.
3. Reboot your system.

Configure ADO.NET logs

You can configure the ADO.NET driver's logging behavior at the registry level with HKEY_LOCAL_MACHINE\SOFTWARE\Vertica\ADO.NET\Driver and the class level with VerticaLogProperties .

Registry-level settings

The following parameters control how messages between the client and server are logged. If they are not set, then the client library does not log any messages.

To set these parameters on Windows, edit the keys in the Windows Registry under HKEY_LOCAL_MACHINE\SOFTWARE\Vertica\ADO.NET\Driver .

LogLevel

The minimum severity of a message for it to be logged:

- 0: No logging
- 1: Fatal errors
- 2: errors
- 3: Warnings
- 4: Info
- 5: Debug
- 6: Trace (all messages)

For example, a LogLevel of 3 means that the client driver logs messages with severities 0, 1, 2, and 3.

LogPath

The absolute path of a directory to store log files. For example: /var/log/verticaadonet .

LogNamespace

Limits logging to messages generated by certain objects in the client driver.

C#PreloadLogging

Normally, logging only starts after the driver has fully loaded. This option specifies whether the ADO.NET driver should start logging before the driver has fully loaded, set to one of the following:

- 0: Do not start logging until the driver has loaded.
- 1: Start logging as soon as possible.

Class-level settings

You can set the log properties of the ADO.NET driver with the VerticaLogProperties static class. This class includes the following methods:

- SetLogPath(String path, bool persist) : Must be set before the connection starts.
- SetLogNamespace(String lognamespace, bool persist)
- SetLogLevel(VerticaLogLevel loglevel, bool persist)

Logs are created when the first connection is opened, so you cannot change the log path after the connection starts. You can change the log level and log namespace at any time.

The persist parameter controls whether the setting is written to the client's Windows Registry, where it will be used for all subsequent connections. If set to false, then the setting only applies to the current session.

SetLogPath

The SetLogPath method takes as its arguments a string containing the path to the log file and the persist argument. If the path string contains only a directory path, then the log file is created with the name vdp-driver-MM-dd_HH.mm.ss.log (where MM-dd_HH.mm.ss is the date and time the log was created). If the path ends in a filename, such as log.txt or log.log, then the log is created with that filename.

If SetLogPath is called with an empty string for the path argument, then the client executable's current directory is used as the log path.

If SetLogPath is not called and no registry entry exists for the log path, and you have called any of the other VerticaLogProperties methods, then the client executable's current directory is used as the log path.

When the persist argument is set to true, the path specified is copied to the registry verbatim. If no filename was specified, then the filename is not saved to the registry.

> **Note**
> The path must exist on the client system prior to calling this method. The method does not create directories.

For example:

```
//set the log path
string path = "C:\\log";
VerticaLogProperties.SetLogPath(path, false);
```

SetLogNamespace

The SetLogNamespace method takes as its arguments a string containing the namespace to log and the persist argument. The namespace string to log can be one of the following:

- Vertica
- Vertica.Data.VerticaClient
- Vertica.Data.Internal.IO
- Vertica.Data.Internal.DataEngine
- Vertica.Data.Internal.Core

Namespaces can be truncated to include multiple child namespaces. For example, you can specify Vertica.Data.Internal to log for all of the Vertica.Data.Internal namespaces.

If a log namespace is not set, and no value is stored in the registry, then the "Vertica" namespace is used for logging.

For example:

```
//set namespace to log
string lognamespace = "Vertica.Data.VerticaClient";
VerticaLogProperties.SetLogNamespace(lognamespace, false);
```

SetLogLevel

The SetLogLevel method takes as its arguments a VerticaLogLevel type and the persist argument. The VerticaLogLevel argument can be one of:

- VerticaLogLevel.None
- VerticaLogLevel.Fatal
- VerticaLogLevel.Error
- VerticaLogLevel.Warning
- VerticaLogLevel.Info
- VerticaLogLevel.Debug
- VerticaLogLevel.Trace

If a log level is not set, and no value is stored in the registry, then VerticaLogLevel.None is used.

For example:

```
//set log level
VerticaLogLevel level = VerticaLogLevel.Debug;
VerticaLogProperties.SetLogLevel(level, false);
```

Getting Log Properties

You can get the log property values using the getters included in the VerticaLogProperties class. The properties are:

- LogPath
- LogNamespace
- LogLevel

For example:

```
//get current log settings
string logpath = VerticaLogProperties.LogPath;
VerticaLogLevel loglevel = VerticaLogProperties.LogLevel;
string logns = VerticaLogProperties.LogNamespace;
Console.WriteLine("Current Log Settings:");
Console.WriteLine("Log Path: " + logpath);
Console.WriteLine("Log Level: " + loglevel);
Console.WriteLine("Log Namespace: " + logns);
```

Examples

This complete example shows how to set and get log properties:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            //configure connection properties
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";

            //get current log settings
            string logpath = VerticaLogProperties.LogPath;
            VerticaLogLevel loglevel = VerticaLogProperties.LogLevel;
            string logns = VerticaLogProperties.LogNamespace;
            Console.WriteLine("\nOld Log Settings:");
            Console.WriteLine("Log Path: " + logpath);
            Console.WriteLine("Log Level: " + loglevel);
            Console.WriteLine("Log Namespace: " + logns);

            //set the log path
            string path = "C:\\log";
            VerticaLogProperties.SetLogPath(path, false);

            // set log level
            VerticaLogLevel level = VerticaLogLevel.Debug;
            VerticaLogProperties.SetLogLevel(level, false);

            //set namespace to log
            string lognamespace = "Vertica";
            VerticaLogProperties.SetLogNamespace(lognamespace, false);

            //open the connection
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();

            //get new log settings
            logpath = VerticaLogProperties.LogPath;
            loglevel = VerticaLogProperties.LogLevel;
            logns = VerticaLogProperties.LogNamespace;
            Console.WriteLine("\nNew Log Settings:");
            Console.WriteLine("Log Path: " + logpath);
            Console.WriteLine("Log Level: " + loglevel);
            Console.WriteLine("Log Namespace: " + logns);

            //close the connection
            _conn.Close();
        }
    }
```

The example produces the following output:

```
Old Log Settings:
Log Path:
Log Level: None
Log Namespace:
New Log Settings:
Log Path: C:\log
Log Level: Debug
Log Namespace: Vertica
```

Upgrading the client drivers

The Vertica client drivers are usually updated for each new release of the Vertica server. The client driver installation packages include the version number of the corresponding Vertica server release. Usually, the drivers are forward-compatible with the next release, so your client applications are still be able to connect using the older drivers after you upgrade to the next version of Vertica Analytics Platform server. See Client driver and server version compatibility for details on which client driver versions work with each version of Vertica server.

> **Note**
> Vertica ODBC, JDBC and ADO.NET client drivers are backwards compatible to all supported Vertica server versions.

You should upgrade your clients as soon as possible after upgrading your server to take advantage of new features and to maintain maximum compatibility with the server.

To upgrade your drivers, follow the same procedure you used to install them in the first place. The new installation will overwrite the old. See the specific instructions for installing the drivers on your client platform for any special instructions regarding upgrades.

> **Note**
> Installing new ODBC drivers does not alter existing DSN settings. You may need to change the driver settings in either the DSN or in the odbcinst.ini file, if your client system uses one. See Creating an ODBC Data Source Name for details.

Setting a client connection label

A client connection label identifies a connection to the database with a user-defined string. You can view the label for an existing session with GET_CLIENT_LABEL:

```
=> SELECT GET_CLIENT_LABEL();
     GET_CLIENT_LABEL
-----------------------------
 my_client_connection_label
(1 row)
```

New connections

In JDBC, ODBC, and ADO.NET, you use each client driver's "Label" connection property to set the client label before connecting to the database. Setting the label before you connect ensures that the connection is associated with the label in all system and Data collector tables. Examples of these tables include SESSIONS and DC_SESSION_STARTS.

- JDBC connection properties
- ODBC DSN connection properties
- ADO.NET connection properties

You can also preemptively set the client label with vsql by using the --label option. For details, see -g --label

Existing connections

You can set a client connection label after you connect to a Vertica database with SET_CLIENT_LABEL:

```
=> SELECT SET_CLIENT_LABEL('py_data_load_application');
            SET_CLIENT_LABEL
-------------------------------------------------
 client_label set to py_data_load_application
(1 row)

=> SELECT GET_CLIENT_LABEL();
    GET_CLIENT_LABEL
--------------------------
 py_data_load_application
(1 row)
```

Certain client drivers, like JDBC, have dedicated functions for setting the client connection label for existing connections. For details, see Setting and returning a client connection label .

Using legacy drivers

The Vertica server supports connections from previous versions of the client drivers. For detailed information the compatibility between versions of the Vertica server and Vertica client, see Client driver and server version compatibility .

Accessing Vertica

The following table shows which client drivers you have to set up to access Vertica with a supported programming language:

Client Driver	Language/Tool
JDBC	Java
ODBC	• C/C++ • Python (pyodbc) • PHP • Perl
vertica-python	Python (native client)
ADO.NET	C#
vertica-nodejs	JavaScript
vertica-sql-go	Go

In this section

- C/C++
- C#
- Go
- Java
- JavaScript
- Perl
- Python
- PHP

C/C++

> **Note**
> You must install the ODBC client driver before creating C/C++ client applications.

Vertica provides an Open Database Connectivity (ODBC) driver that allows applications to connect to the Vertica database. This driver can be used by custom-written client applications that use the ODBC API to interact with Vertica. ODBC is also used by many third-party applications to connect to Vertica, including business intelligence applications and extract, transform, and load (ETL) applications.

This section details the process for configuring the Vertica ODBC driver. It also explains how to use the ODBC API to connect to Vertica in your own client applications.

While client applications written in C, C++, Perl , PHP , etc. all use the ODBC client driver to connect to Vertica, this section only concerns C and C++ applications.

In this section

- ODBC architecture
- ODBC feature support
- Vertica and ODBC data type translation
- ODBC header file
- Connecting to the database
- Load balancing
- Configuring TLS for ODBC Clients
- Connection failover
- Prompting windows users for missing connection properties
- Prompting windows users for passwords
- Setting the locale and encoding for ODBC sessions
- AUTOCOMMIT and ODBC transactions
- Retrieving data
- Loading data

ODBC architecture

The ODBC architecture has four layers:

- **Client Application**
  Is an application that opens a data source through a Data Source Name (DSN). It then sends requests to the data source, and receives the results of those requests. Requests are made in the form of calls to ODBC functions.
- **Driver Manager**
  Is a library on the client system that acts as an intermediary between a client application and one or more drivers. The driver manager:
  - Resolves the DSN provided by the client application.
  - Loads the driver required to access the specific database defined within the DSN.
  - Processes ODBC function calls from the client or passing them to the driver.
  - Retrieves results from the driver.
  - Unloads drivers when they are no longer needed.

  On Windows and macOS client systems, the driver manager is provided by the operating system. On Linux systems, you usually need to install a driver manager. See Client drivers support for a list of driver managers that can be used with Vertica on your client platform.
- **Driver**
  A library on the client system that provides access to a specific database. It translates requests into the format expected by the database, and translates results back into the format required by the client application.
- **Database**
  The database processes requests initiated at the client application and returns results.

ODBC feature support

The ODBC driver for Vertica supports the most of the features defined in the Microsoft ODBC 3.5 specifications. The following features are *not* supported:

- Updatable result sets
- Backwards scrolling cursors
- Cursor attributes
- More than one open statement per connection. Simultaneously executing statements must each belong to a different connection. For example, you cannot execute a new statement while another statement has a result set open. To execute another statement with the same connection/session, wait for the current statement to finish executing and close its result set, then execute the new statement.
- Keysets
- Bookmarks

The Vertica ODBC driver accurately reports its capabilities. If you need to determine whether it complies with a specific feature, you should query the driver's capabilities directly using the SQLGetInfo() function.

Vertica and ODBC data type translation

Most data types are transparently converted between Vertica and ODBC. This section explains several data types require special handling.

Vertica Data Types	C Data Type	ODBC C Typedef	C Type Identifier
BINARY , VARBINARY	char[]	SQL_BINARY	SQL_C_BINARY
LONG VARBINARY	char[]	SQL_LONGVARBINARY	SQL_C_BINARY
BOOLEAN	SQLSMALLINT	SQL_SMALLINT	SQL_C_SSHORT
CHAR , VARCHAR	char[]	SQL_CHAR	SQL_C_CHAR
LONG VARCHAR	char[]	SQL_LONGVARCHAR	SQL_C_CHAR
DATE	SQL_DATE_STRUCT	SQL_TYPE_DATE	SQL_C_TYPE_DATE
TIME	SQL_TIME_STRUCT	SQL_TYPE_TIME	SQL_C_TYPE_TIME
TIMESTAMP	SQL_TIMESTAMP_STRUCT	SQL_TYPE_TIMESTAMP	SQL_C_TYPE_TIMESTAMP
INTERVAL	SQL_INTERVAL_STRUCT	SQL_INTERVAL_DAY_TO_SECOND	SQL_C_INTERVAL_DAY_TO_SECOND
INTERVAL DAY TO SECOND	SQL_INTERVAL_STRUCT	SQL_INTERVAL_DAY_TO_SECOND	SQL_C_INTERVAL_DAY_TO_SECOND
INTERVAL YEAR TO MONTH	SQL_INTERVAL_STRUCT	SQL_INTERVAL_YEAR_TO_MONTH	SQL_C_INTERVAL_YEAR_TO_MONTH
DOUBLE PRECISION FLOAT	SQLREAL	SQL_REAL	SQL_C_FLOAT
INTEGER , BIGINT , SMALLINT	SQLBIGINT	SQL_BIGINT	SQL_C_SBIGINT
NUMERIC , DECIMAL , NUMBER , MONEY	SQL_NUMERIC_STRUCT	SQL_NUMERIC	SQL_C_NUMERIC
GEOMETRY	char[]	SQL_LONGVARBINARY	SQL_C_CHAR
GEOGRAPHY	char[]	SQL_LONGVARBINARY	SQL_C_CHAR
UUID	SQLGUID (see note below)	SQL_GUID	SQL_C_GUID

Notes

- The GEOMETRY and GEOGRAPHY data types are treated as LONG VARCHAR data by the ODBC driver.
- Vertica supports the standard interval data types supported by ODBC. See Interval Data Types in Microsoft's ODBC reference.
- Vertica version 9.0.0 introduced the UUID data type, including JDBC support for UUIDs. The Vertica ADO.NET, ODBC, and OLE DB clients added full support for UUIDs in version 9.0.1. Vertica maintains backwards compatibility with older supported client driver versions that do not support the UUID data type, as follows:

When an older client...	Vertica...
Queries tables with UUID columns	Translates the native UUID values to CHAR values.
Inserts data into a UUID column	Converts the CHAR value sent by the client into a native UUID value.
Queries a UUID column's metadata	Reports its data type as CHAR.

See also

- Data types
- Using LONG VARCHAR and LONG VARBINARY data types with ODBC
- Using GEOMETRY and GEOGRAPHY data types in ODBC
- SQL Data Types in the Microsoft ODBC reference documentation

ODBC header file

The Vertica ODBC driver provides a C header file named verticaodbc.h that defines several useful constants that you can use in your applications. These constants let you access and alter settings specific to Vertica.

This file's location depends on your client operating system:

- /opt/vertica/include on Linux and UNIX systems.
- C:\Program Files (x86)\Vertica\ODBC\include on Windows systems.

The constants defined in this file are listed below.

Parameter	Description
SQL_ATTR_VERTICA_RESULT_BUFFER_SIZE	Sets the size of the buffer used when retrieving results from the server. **Associated functions** : SQLSetConnectAttr() SQLGetConnectAttr()
SQL_ATTR_VERTICA_DIRECT_BATCH_INSERT	Deprecated, always set to 1. **Associated functions:** SQLSetConnectAttr() SQLSetStmtAttr() SQLGetConnectAttr() SQLGetStmtAttr()
SQL_ATTR_VERTICA_LOCALE	Changes the locale from en_US@collation=binary to the ICU locale specified. See Setting the locale and encoding for ODBC sessions for an example of using this parameter. **Associated functions** : SQLSetConnectAttr() SQLGetConnectAttr()

Connecting to the database

The first step in any ODBC application is to connect to the database. When you create the connection to a data source using ODBC, you use the name of the DSN that contains the details of the driver to use, the database host, and other basic information about connecting to the data source.

There are 4 steps your application needs to take to connect to a database:

1. Call SQLAllocHandle() to allocate a handle for the ODBC environment. This handle is used to create connection objects and to set application-wide settings.
2. Use the environment handle to set the version of ODBC that your application wants to use. This ensures that the data source knows which API your application will use to interact with it.
3. Allocate a database connection handle by calling SQLAllocHandle() . This handle represents a connection to a specific data source.
4. Use the SQLConnect() or SQLDriverConnect() functions to open the connection to the database.

> **Note**
> If you specify a locale either in the connection string or in the DSN, the call to the connection function returns SQL_SUCCESS_WITH_INFO on a successful connection, with messages about the state of the locale.

When creating the connection to the database, use SQLConnect() when the only options you need to set at connection time is the username and password. Use SQLDriverConnect() when you want to change connection options, such as the locale.

The following example demonstrates connecting to a database using a DSN named ExampleDB. After it creates the connection successfully, this example simply closes it.

```
// Demonstrate connecting to Vertica using ODBC.
// Standard i/o library
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// SQL include files that define data types and ODBC API
// functions
#include <sql.h>
#include <sqlext.h>
#include <sqltypes.h>
int main()
{
   SQLRETURN ret;  // Stores return value from ODBC API calls
   SQLHENV hdlEnv;  // Handle for the SQL environment object
   // Allocate an a SQL environment object
   ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
   if(!SQL_SUCCEEDED(ret)) {
      printf("Could not allocate a handle.\n");
      exit(EXIT_FAILURE);
   } else {
      printf("Allocated an environment handle.\n");
   }

   // Set the ODBC version we are going to use to
   // 3.
   ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
         (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
   if(!SQL_SUCCEEDED(ret)) {
      printf("Could not set application version to ODBC 3.\n");
      exit(EXIT_FAILURE);
   } else {
      printf("Set application version to ODBC 3.\n");
   }
   // Allocate a database handle.
   SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    if(!SQL_SUCCEEDED(ret)) {
       printf("Could not allocate database handle.\n");
       exit(EXIT_FAILURE);
    } else {
       printf("Allocated Database handle.\n");
    }
   // Connect to the database using
   // SQL Connect
   printf("Connecting to database.\n");
   const char *dsnName = "ExampleDB";
   const char* userID = "ExampleUser";
   const char* passwd = "password123";
   ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
      SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
      (SQLCHAR*)passwd, SQL_NTS);
   if(!SQL_SUCCEEDED(ret)) {
```

```
    printf("Could not connect to database.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Connected to database.\n");
}
// We're connected. You can do real
// work here

// When done, free all of the handles to close them
// in an orderly fashion.
printf("Disconnecting and freeing handles.\n");
ret = SQLDisconnect( hdlDbc );
if(!SQL_SUCCEEDED(ret)) {
    printf("Error disconnecting from database. Transaction still open?\n");
    exit(EXIT_FAILURE);
}

SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
exit(EXIT_SUCCESS);
```

Running the above code prints the following:

```
Allocated an environment handle.
Set application version to ODBC 3.
Allocated Database handle.
Connecting to database.
Connected to database.
Disconnecting and freeing handles.
```

See Setting the locale and encoding for ODBC sessions for an example of using SQLDriverConnect to connect to the database.

Notes

- If you use the DataDirect ® driver manager, you should always use the SQL_DRIVER_NOPROMPT value for the SQLDriverConnect function's DriverCompletion parameter (the final parameter in the function call) when connecting to Vertica. Vertica's ODBC driver on Linux and UNIX platforms does not contain a UI, and therefore cannot prompt users for a password.
- On Windows client platforms, the ODBC driver can prompt users for connection information. See Prompting windows users for missing connection properties for more information.
- If your database does not comply with your Vertica license agreement, your application receives a warning message in the return value of the SQLConnect() function. Always have your application examine this return value to see if it is SQL_SUCCESS_WITH_INFO. If it is, have your application extract and display the message to the user.

Load balancing

Native connection load balancing

Native connection load balancing helps spread the overhead caused by client connections on the hosts in the Vertica database. Both the server and the client must enable native connection load balancing. If enabled by both, then when the client initially connects to a host in the database, the host picks a host to handle the client connection from a list of the currently up hosts in the database, and informs the client which host it has chosen.

If the initially-contacted host does not choose itself to handle the connection, the client disconnects, then opens a second connection to the host selected by the first host. The connection process to this second host proceeds as usual—if SSL is enabled, then SSL negotiations begin, otherwise the client begins the authentication process. See About native connection load balancing for details.

To enable native load balancing on your client, set the ConnectionLoadBalance connection parameter to true either in the DSN entry or in the connection string. The following example demonstrates connecting to the database several times with native connection load balancing enabled, and fetching the name of the node handling the connection from the V_MONITOR. CURRENT_SESSION system table.

```
// Demonstrate enabling native load connection balancing.
// Standard i/o library
#include <stdlib.h>
#include <iostream>
#include <assert.h>
// Only needed for Windows clients
```

```
// Only needed for Windows clients
// #include <windows.h>
// SQL include files that define data types and ODBC API
// functions
#include <sql.h>
#include <sqlext.h>
#include <sqltypes.h>

using namespace std;
int main()
{
   SQLRETURN ret;   // Stores return value from ODBC API calls
   SQLHENV hdlEnv;  // Handle for the SQL environment object
   // Allocate an a SQL environment object
   ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
   assert(SQL_SUCCEEDED(ret));

   // Set the ODBC version we are going to use to
   // 3.
   ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
         (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
   assert(SQL_SUCCEEDED(ret));

   // Allocate a database handle.
   SQLHDBC hdlDbc;
   ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
   assert(SQL_SUCCEEDED(ret));

   // Connect four times. If load balancing is on, client should
   // connect to different nodes.
   for (int x=1; x <= 4; x++) {

      // Connect to the database using SQLDriverConnect. Set
      // ConnectionLoadBalance to 1 (true) to enable load
      // balancing.
      cout << endl << "Connection attempt #" << x << "... ";
      const char *connStr = "DSN=VMart;ConnectionLoadBalance=1;"
         "UID=ExampleUser;PWD=password123";


      ret = SQLDriverConnect(hdlDbc, NULL, (SQLCHAR*)connStr, SQL_NTS,
            NULL, 0, NULL, SQL_DRIVER_NOPROMPT );
      if(!SQL_SUCCEEDED(ret)) {
         cout << "failed. Exiting." << endl;
         exit(EXIT_FAILURE);
      } else {
         cout << "succeeded" << endl;
      }
      // We're connected. Query the v_monitor.current_session table to
      // find the name of the node we've connected to.

      // Set up a statement handle
      SQLHSTMT hdlStmt;
      SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
      assert(SQL_SUCCEEDED(ret));

      ret = SQLExecDirect( hdlStmt, (SQLCHAR*)"SELECT node_name FROM "
         "V_MONITOR.CURRENT_SESSION;", SQL_NTS );

      if(SQL_SUCCEEDED(ret)) {
         // Bind varible to column in result set.
         SQLTCHAR node_name[256];
```

```
    SQLGetData(hdlStmt, 1, SQL_C_TCHAR, node_name, sizeof(node_name), NULL);
    ret = SQLBindCol(hdlStmt, 1, SQL_C_TCHAR, (SQLPOINTER)node_name,
        sizeof(node_name), NULL);
    while(SQL_SUCCEEDED(ret = SQLFetchScroll(hdlStmt, SQL_FETCH_NEXT,1))) {
        // Print the bound variables, which now contain the values from the
        // fetched row.
        cout << "Connected to node " << node_name << endl;
    }
  }
  // Free statement handle
  SQLFreeHandle(SQL_HANDLE_STMT,hdlStmt);
  cout << "Disconnecting." << endl;
  ret = SQLDisconnect( hdlDbc );
  assert(SQL_SUCCEEDED(ret));
}
// When done, free all of the handles to close them
// in an orderly fashion.
cout << endl << "Freeing handles..." << endl;
SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
cout << "Done!" << endl;
exit(EXIT_SUCCESS);
```

Running the above example produces output similar to the following:

```
Connection attempt #1... succeeded
Connected to node v_vmart_node0001
Disconnecting.

Connection attempt #2... succeeded
Connected to node v_vmart_node0002
Disconnecting.

Connection attempt #3... succeeded
Connected to node v_vmart_node0003
Disconnecting.

Connection attempt #4... succeeded
Connected to node v_vmart_node0001
Disconnecting.

Freeing handles...
Done!
```

Hostname-based load balancing

You can also balance workloads by resolving a single hostname to multiple IP addresses. The ODBC client driver load balances by automatically resolving the hostname to one of the specified IP addresses at random.

For example, suppose the hostname verticahost.example.com has the following entries in etc/hosts :

```
192.0.2.0 verticahost.example.com
192.0.2.1 verticahost.example.com
192.0.2.2 verticahost.example.com
```

Specifying the hostname verticahost.example.com randomly resolves to one of the listed IP addresses.

Configuring TLS for ODBC Clients

You can configure TLS for ODBC clients by setting the DSN connection properties setting the DSN connection properties for the following. For details on these parameters, see ODBC DSN connection properties :

- SSLMode: Determines whether TLS is required and how the client should behave if the TLS connection attempt fails.

- SSLCertFile (SSL CA file in Windows): The absolute path of the client's public certificate file.
- SSLKeyFile (SSL cert file in Windows): The absolute path to the client's private key file.

SSLModes: Verify_ca and verify_full

You can use the SSLMode property values verify_ca and verify_full if you want the client to verify the server's information before establishing the connection. If any of these verifications fail, the connection fails:

- verify_ca : The client verifies that the server's certificate is from a trusted certificate authority (CA).
- verify_full : The client verifies both that the server's certificate is from a trusted CA and that the server's hostname matches the hostname on the certificate.

If verify_ca or verify_full are specified, the client requires the following to establish the connection:

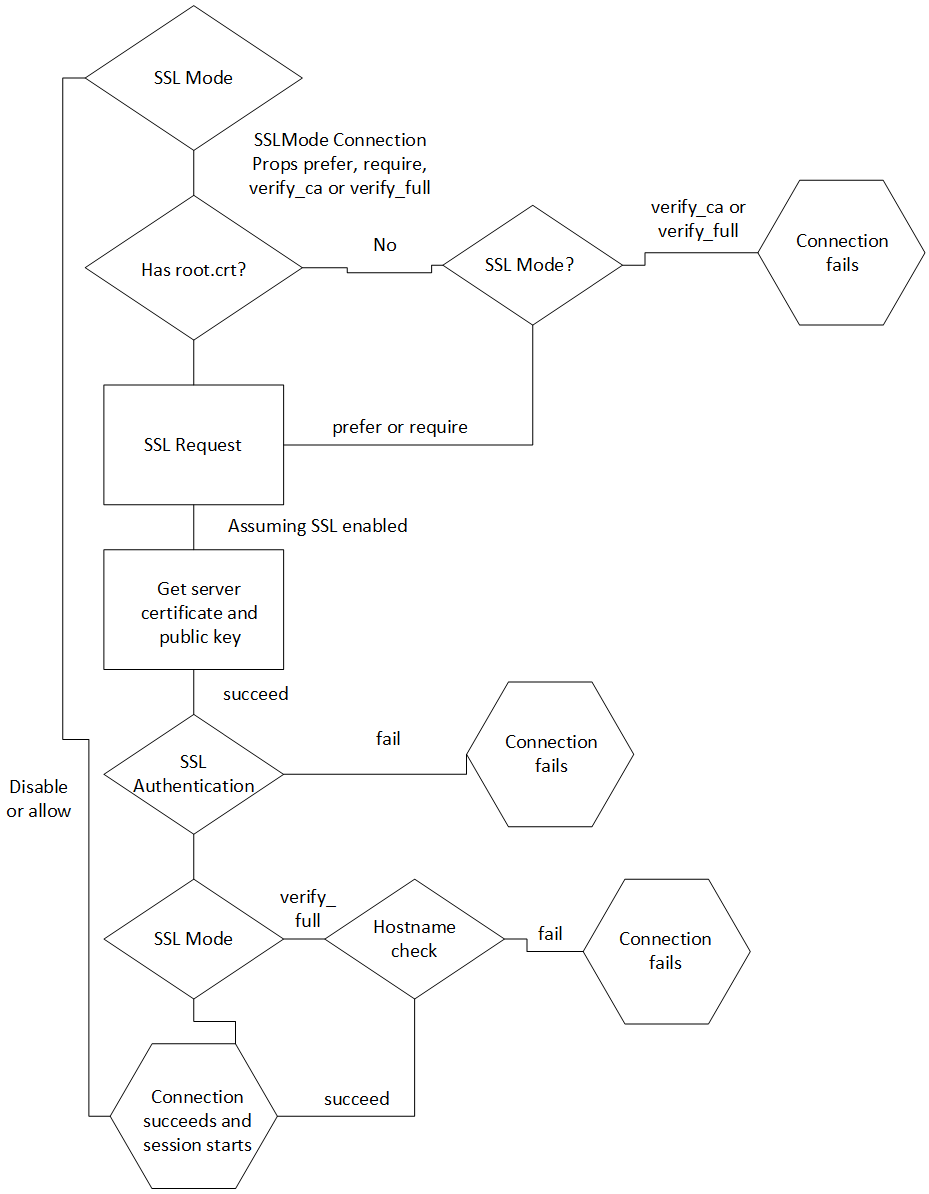
- The root.crt , which is the certificate of a CA trusted by both the server and the client.
- The server must have:
  - server.crt , a certificate signed by the trusted CA.
  - server.key , the server's private key.
- For verify_full , each server node must meet one of the following requirements:
  - Its hostname matches the common name specified in server.crt .
  - Its hostname or IP address appears in the Subject Alternative Name (SAN) field of server.crt .

TLS behavior flowchart

The following diagram shows an example flowchart for a client connecting with TLS.

In this example:

- If SSLMode is set to none or allow , the client connects without authentication.
- If SSLMode is set to verify_ca or verify_full and the client does not have root.crt , the connection fails.
- At the SSL authentication node, if the SSLMode connection is set to verify_full and the server hostname differs from the hostname specified by the client, authentication fails.



Connection failover

If a client application attempts to connect to a host in the Vertica cluster that is down, the connection attempt fails when using the default connection configuration. This failure usually returns an error to the user. The user must either wait until the host recovers and retry the connection or manually edit the connection settings to choose another host.

Due to Vertica Analytic Database's distributed architecture, you usually do not care which database host handles a client application's connection. You can use the client driver's connection failover feature to prevent the user from getting connection errors when the host specified in the connection settings is unreachable. The JDBC driver gives you several ways to let the client driver automatically attempt to connect to a different host if the one specified in the connection parameters is unreachable:

- Configure your DNS server to return multiple IP addresses for a host name. When you use this host name in the connection settings, the client attempts to connect to the first IP address from the DNS lookup. If the host at that IP address is unreachable, the client tries to connect to the second IP, and so on until it either manages to connect to a host or it runs out of IP addresses.
- Supply a list of backup hosts for the client driver to try if the primary host you specify in the connection parameters is unreachable.
- (JDBC only) Use driver-specific connection properties to manage timeouts before attempting to connect to the next node.

For all methods, the process of failover is transparent to the client application (other than specifying the list of backup hosts, if you choose to use the list method of failover). If the primary host is unreachable, the client driver automatically tries to connect to other hosts.

Failover only applies to the initial establishment of the client connection. If the connection breaks, the driver does not automatically try to reconnect to another host in the database.

Choosing a failover method

You usually choose to use one of the two failover methods. However, they do work together. If your DNS server returns multiple IP addresses and you supply a list of backup hosts, the client first tries all of the IPs returned by the DNS server, then the hosts in the backup list.

> **Note**
> If a host name in the backup host list resolves to multiple IP addresses, the client does not try all of them. It just tries the first IP address in the list.

The DNS method of failover centralizes the configuration client failover. As you add new nodes to your Vertica Analytic Database cluster, you can choose to add them to the failover list by editing the DNS server settings. All client systems that use the DNS server to connect to Vertica Analytic Database automatically use connection failover without having to change any settings. However, this method does require administrative access to the DNS server that all clients use to connect to the Vertica Analytic Database cluster. This may not be possible in your organization.

Using the backup server list is easier than editing the DNS server settings. However, it decentralizes the failover feature. You may need to update the application settings on each client system if you make changes to your Vertica Analytic Database cluster.

Using DNS failover

To use DNS failover, you need to change your DNS server's settings to map a single host name to multiple IP addresses of hosts in your Vertica Analytic Database cluster. You then have all client applications use this host name to connect to Vertica Analytic Database.

You can choose to have your DNS server return as many IP addresses for the host name as you want. In smaller clusters, you may choose to have it return the IP addresses of all of the hosts in your cluster. However, for larger clusters, you should consider choosing a subset of the hosts to return. Otherwise there can be a long delay as the client driver tries unsuccessfully to connect to each host in a database that is down.

Using the backup host list

To enable backup list-based connection failover, your client application has to specify at least one IP address or host name of a host in the BackupServerNode parameter. The host name or IP can optionally be followed by a colon and a port number. If not supplied, the driver defaults to the standard Vertica port number (5433). To list multiple hosts, separate them by a comma.

The following example demonstrates setting the BackupServerNode connection parameter to specify additional hosts for the connection attempt. The connection string intentionally has a non-existent node, so that the initial connection fails. The client driver has to resort to trying the backup hosts to establish a connection to Vertica.

```
// Demonstrate using connection failover.
// Standard i/o library
#include <stdlib.h>
#include <iostream>
#include <assert.h>

// Only needed for Windows clients
// #include <windows.hgt;

// SQL include files that define data types and ODBC API
// functions
#include <sql.h>
#include <sqlext.h>
#include <sqltypes.h>

using namespace std;

int main()
{
   SQLRETURN ret;  // Stores return value from ODBC API calls
   SQLHENV hdlEnv;  // Handle for the SQL environment object
   // Allocate an a SQL environment object
   ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
   assert(SQL_SUCCEEDED(ret));

   // Set the ODBC version we are going to use to
   // 3.
   ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
         (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
```

```
    assert(SQL_SUCCEEDED(ret));

    // Allocate a database handle.
    SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    assert(SQL_SUCCEEDED(ret));

/* DSN for this connection specifies a bad node, and good backup nodes:
[VMartBadNode]
Description=VMart Vertica Database
Driver=/opt/vertica/lib64/libverticaodbc.so
Database=VMart
Servername=badnode.example.com
BackupServerNode=v_vmart_node0002.example.com,v_vmart_node0003.example.com
*/

    // Connect to the database using SQLConnect
    cout << "Connecting to database." << endl;
    const char *dsnName = "VMartBadNode"; // Name of the DSN
    const char* userID = "ExampleUser"; // Username
    const char* passwd = "password123"; // password
    ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
        SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
        (SQLCHAR*)passwd, SQL_NTS);
    if(!SQL_SUCCEEDED(ret)) {
        cout << "Could not connect to database." << endl;
        exit(EXIT_FAILURE);
    } else {
        cout << "Connected to database." << endl;
    }
    // We're connected. Query the v_monitor.current_session table to
    // find the name of the node we've connected to.

    // Set up a statement handle
    SQLHSTMT hdlStmt;
    SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
    assert(SQL_SUCCEEDED(ret));

    ret = SQLExecDirect( hdlStmt, (SQLCHAR*)"SELECT node_name FROM "
        "v_monitor.current_session;", SQL_NTS );

    if(SQL_SUCCEEDED(ret)) {
        // Bind varible to column in result set.
        SQLTCHAR node_name[256];
        ret = SQLBindCol(hdlStmt, 1, SQL_C_TCHAR, (SQLPOINTER)node_name,
            sizeof(node_name), NULL);
        while(SQL_SUCCEEDED(ret = SQLFetchScroll(hdlStmt, SQL_FETCH_NEXT,1))) {
            // Print the bound variables, which now contain the values from the
            // fetched row.
            cout << "Connected to node " << node_name << endl;
        }
    }

    cout << "Disconnecting." << endl;
    ret = SQLDisconnect( hdlDbc );
    assert(SQL_SUCCEEDED(ret));

    // When done, free all of the handles to close them
    // in an orderly fashion.
    cout << endl << "Freeing handles..." << endl;
    SQLFreeHandle(SQL_HANDLE_STMT,hdlStmt);
```

```
SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
cout << "Done!" << endl;
exit(EXIT_SUCCESS);
```

When run, the example's output on the system console is similar to the following:

```
Connecting to database.
Connected to database.
Connected to node v_vmart_node0002
Disconnecting.

Freeing handles...
Done!
```

Notice that the connection was made to the first node in the backup list (node 2).

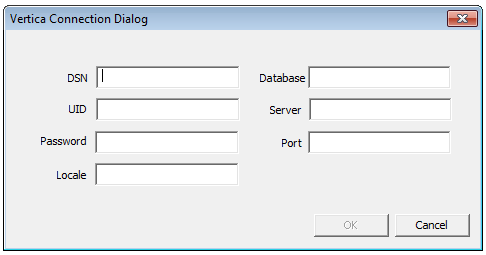
> **Note**
> When native connection load balancing is enabled, the additional servers specified in the BackupServerNode connection parameter are only used for the initial connection to a Vertica host. If host redirects the client to another host in the database cluster to handle its connection request, the second connection does not use the backup node list. This is rarely an issue, since native connection load balancing is aware of which nodes are currently up in the database. See Load balancing for more information.

Prompting windows users for missing connection properties

The Vertica Windows ODBC driver can prompt the user for connection information if required information is missing. The driver displays the Vertica Connection Dialog if the client application calls SQLDriverConnect to connect to Vertica and either of the following is true:

- The DriverCompletion property is set to SQL_DRIVER_PROMPT.
- The DriverCompletion property is set to SQL_DRIVER_COMPLETE or SQL_DRIVER_COMPLETE_REQUIRED and the connection string or DSN being used to connect is missing the server, database, or port information.

If either of the above conditions are true, the driver displays a Vertica Connection Dialog to the user to prompt for connection information.



The dialog has all of the property values supplied in the connection string or DSN filled in.

> **Note**
> Your connection string at least needs to specify Vertica as the driver, otherwise Windows will not know to use the Vertica ODBC driver to try to open the connection.

The required fields on the connection dialog are Database, UID, Server, and Port. Once these are filled in, the form enables the **OK** button.

If the user clicks **Cancel** on the dialog, the SQLDriverConnect function call returns SQL_NO_DATA immediately, without attempting to connect to Vertica. If the user supplies incomplete or incorrect information for the connection, the connection function returns SQL_ERROR after the connection attempt fails.

> **Note**
> If the DriverCompletion property of the SQLDriverConnect function call is SQL_DRIVER_NOPROMPT, the ODBC driver immediately returns a SQL_ERROR indicating that it cannot connect because not enough information has been supplied and the driver is not allowed to prompt the user for the missing information.

Prompting windows users for passwords

If the connection string or DSN supplied to the SQLDriverConnect function that client applications call to connect to Vertica lacks any of the required connection properties needed to connect, the Vertica's Windows ODBC driver opens a dialog box to prompt the user to enter the missing information (see Prompting windows users for missing connection properties). The user's password is not normally considered a required connection property because Vertica user accounts may not have a password. If the password property is missing, the ODBC driver still tries to connect to Vertica without supplying a password.

You can use the PromptOnNoPassword DSN parameter to force ODBC driver to treat the password as a required connection property. This parameter is useful if you do not want to store passwords in DSN entries. Passwords saved in DSN entries are insecure, since they are stored as clear text in the Windows registry and therefore visible to other users on the same system.

There are two other factors which also decide whether the ODBC driver displays the Vertica Connection Dialog. These are (in order of priority):

- The SQLDriverConnect function call's DriverCompletion parameter.
- Whether the DSN or connection string contain a password

The following table shows how the PromptOnNoPassword DSN parameter, the DriverCompletion parameter of the SQLDriverConnect function, and whether the DSN or connection string contains a password interact to control whether the Vertica Connection dialog appears.

PromptOnNoPassword Setting	DriverCompletion Value	DSN or Connection String Contains Password?	Vertica Connection Dialog Displays?	Notes
any value	SQL_DRIVER_PROMPT	any case	Yes	This DriverCompletion value forces the dialog to always appear, even if all required connection properties are supplied.
any value	SQL_DRIVER_NOPROMPT	any case	No	This DriverCompletion value always prevents the dialog from appearing.
any value	SQL_DRIVER_COMPLETE	Yes	No	Connection dialog displays if another required connection property is missing.
true	SQL_DRIVER_COMPLETE	No	Yes	
false (default)	SQL_DRIVER_COMPLETE	No	No	Connection dialog displays if another required connection property is missing.

The following example code demonstrates using the PromptOnNoPassword DSN parameter along with a system DSN in C++:

```
wstring connectString = L "DSN=VerticaDSN;PromptOnNoPassword=1;";
retcode = SQLDriverConnect(
   hdbc,
   0,
   (SQLWCHAR * ) connectString.c_str(),
   connectString.length(),
   OutConnStr,
   255, &
   amp; OutConnStrLen,
   SQL_DRIVER_COMPLETE);
```

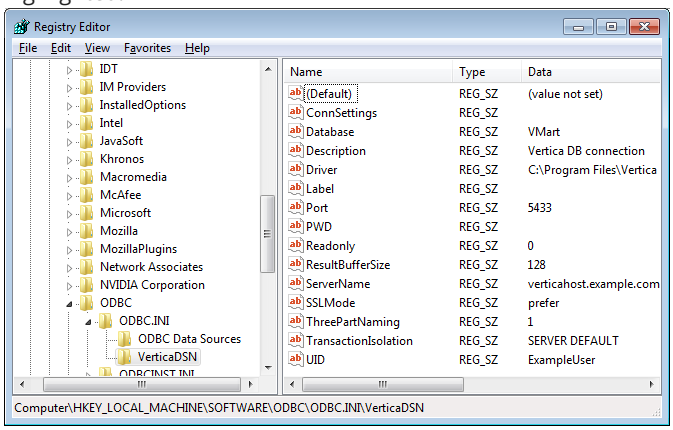
No password entry vs. empty passwords

There is a difference between not having a password property in the connection string or DSN and having an empty password. The PromptOnNoPassword DSN parameter only has an effect if the connection string or DSN does not have a PWD property (which holds the user's password). If it does, even if it is empty, PromptOnNoPassword will not prompt the Windows ODBC driver to display the Vertica Connection Dialog.

This difference can cause confusion if you are using a DSN to provide the properties for your connection. Once you enter a password for a DSN connection in the Windows ODBC Manager and save it, Windows adds a PWD property to the DSN definition in the registry. If you later delete the password, the PWD property remains in the DSN definition—value is just set to an empty string. The PWD property is created even if you just use the Test button on the ODBC Manager dialog to test the DSN and later clear it before saving the DSN.

Once the password has been set, the only way to remove the PWD property from the DSN definition is to delete it using the Windows Registry Editor:

1. On the Windows Start menu, click Run.
2. In the Run dialog, type regedit, then click OK.
3. In the Registry Editor window, click Edit > Find (or press Ctrl+F).
4. In the Find window, enter the name of the DSN whose PWD property you want to delete and click OK.
5. If find operation did not locate a folder under the ODBC.INI folder, click Edit > Find Next (or press F3) until the folder matching your DSN's name is highlighted.



6. Select the PWD entry and press Delete.
7. Click Yes to confirm deleting the value.

The DSN now does not have a PWD property and can trigger the connection dialog to appear when used along with PromptOnNoPassword=true and DriverConnect=SQL_DRIVER_COMPLETE.

Setting the locale and encoding for ODBC sessions

Vertica provides the following methods to set the locale and encoding for an ODBC session:

- Specify the locale for all connections made using the DSN:
  - On Linux and other UNIX-like platforms: Creating an ODBC DSN for Linux
  - On Windows platforms, set the locale in the ODBC DSN configuration editor's Locale field on the Server Settings tab. See Creating an ODBC DSN for windows clients for detailed information.
- Set the Locale connection parameter in the connection string in SQLDriverConnect() function. For example:

```
SQLDriverConnect(conn, NULL, (SQLCHAR*)"DSN=Vertica;Locale=en_GB@collation=binary", SQL_NTS, szConnOut, sizeof(szConnOut), &iAvailable, S
```

- Use SQLSetConnectAttr() to set the encoding and locale. In general, you should always set the encoding with this function as opposed to, for example, setting it in the DSN.
  - Pass the SQL_ATTR_VERTICA_LOCALE constant and the ICU string as the attribute value. For example:

```
-> SQLSetConnectAttr(hdlDbc, SQL_ATTR_VERTICA_LOCALE, (SQLCHAR*)newLocale,
     SQL_NTS);
```

  - Pass the SQL_ATTR_AP_WCHAR_TYPE constant and the encoding as the attribute value. For example:

```
-> rc = SQLSetConnectAttr (hdbc, SQL_ATTR_APP_WCHAR_TYPE, (void *)SQL_DD_CP_UTF16, SQL_IS_INTEGER);
```

Notes

- Having the client system use a non-Unicode locale (such as setting LANG=C on Linux platforms) and using a Unicode locale for the connection to Vertica can result in errors such as "(10170) String data right truncation on data from data source." If data received from Vertica isn't in UTF-8 format. The driver allocates string memory based on the system's locale setting, and non-UTF-8 data can trigger an overrun. You can avoid these errors by always using a Unicode locale on the client system.
  If you specify a locale either in the connection string or in the DSN, the call to the connection function returns SQL_SUCCESS_WITH_INFO on a successful connection, with messages about the state of the locale.
- ODBC applications can be in either ANSI or Unicode mode:
  - If Unicode, the encoding used by ODBC is UCS-2.
  - If ANSI, the data must be in single-byte ASCII, which is compatible with UTF-8 on the database server.

The ODBC driver converts UCS-2 to UTF-8 when passing to the Vertica server and converts data sent by the Vertica server from UTF-8 to UCS-2.

- If the end-user application is not already in UCS-2, the application is responsible for converting the input data to UCS-2, or unexpected results could occur. For example:
  - On non-UCS-2 data passed to ODBC APIs, when it is interpreted as UCS-2, it could result in an invalid UCS-2 symbol being passed to the APIs, resulting in errors.
  - Or the symbol provided in the alternate encoding could be a valid UCS-2 symbol; in this case, incorrect data is inserted into the database.

  ODBC applications should set the correct server session locale using SQLSetConnectAttr (if different from database-wide setting) in order to set the proper collation and string functions behavior on server.

The following example code demonstrates setting the locale using both the connection string and with the SQLSetConnectAttr() function.

```
// Standard i/o library
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// SQL include files that define data types and ODBC API
// functions
#include <sql.h>
#include <sqlext.h>
#include <sqltypes.h>
// Vertica-specific definitions. This include file is located as
// /opt/vertica/include on database hosts.
#include <verticaodbc.h>
int main()
{
  SQLRETURN ret;  // Stores return value from ODBC API calls
  SQLHENV hdlEnv;  // Handle for the SQL environment object
  // Allocate an a SQL environment object
  ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
  if(!SQL_SUCCEEDED(ret)) {
    printf("Could not allocate a handle.\n");
    exit(EXIT_FAILURE);
  } else {
    printf("Allocated an environment handle.\n");
  }
  // Set the ODBC version we are going to use to 3.
  ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
    (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
  if(!SQL_SUCCEEDED(ret)) {
    printf("Could not set application version to ODBC 3.\n");
    exit(EXIT_FAILURE);
  } else {
    printf("Set application version to ODBC 3.\n");
  }
  // Allocate a database handle.
  SQLHDBC hdlDbc;
  ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
  if(!SQL_SUCCEEDED(ret)) {
    printf("Could not allocate database handle.\n");
    exit(EXIT_FAILURE);
  } else {
    printf("Allocated Database handle.\n");
  }
  // Connect to the database using SQLDriverConnect
  printf("Connecting to database.\n");
  // Set the locale to English in Great Britain.
  const char *connStr = "DSN=ExampleDB;locale=en_GB;"
    "UID=dbadmin;PWD=password123";
  ret = SQLDriverConnect(hdlDbc, NULL, (SQLCHAR*)connStr, SQL_NTS,
          NULL, 0, NULL, SQL_DRIVER_NOPROMPT );
  if(!SQL_SUCCEEDED(ret)) {
```

```
    printf("Could not connect to database.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Connected to database.\n");
}
// Get the Locale
char locale[256];
SQLGetConnectAttr(hdlDbc, SQL_ATTR_VERTICA_LOCALE, locale, sizeof(locale),
    0);
printf("Locale is set to: %s\n", locale);
// Set the locale to a new value
const char* newLocale = "en_GB";
SQLSetConnectAttr(hdlDbc, SQL_ATTR_VERTICA_LOCALE, (SQLCHAR*)newLocale,
    SQL_NTS);

// Get the Locale again
SQLGetConnectAttr(hdlDbc, SQL_ATTR_VERTICA_LOCALE, locale, sizeof(locale),
    0);
printf("Locale is now set to: %s\n", locale);

// Set the encoding
SQLSetConnectAttr (hdbc, SQL_ATTR_APP_WCHAR_TYPE, (void *)SQL_DD_CP_UTF16,
    SQL_IS_INTEGER);

// When done, free all of the handles to close them
// in an orderly fashion.
printf("Disconnecting and freeing handles.\n");
ret = SQLDisconnect( hdlDbc );
if(!SQL_SUCCEEDED(ret)) {
    printf("Error disconnecting from database. Transaction still open?\n");
    exit(EXIT_FAILURE);
}
SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
exit(EXIT_SUCCESS);
```

AUTOCOMMIT and ODBC transactions

The AUTOCOMMIT connection attribute controls whether INSERT, ALTER, COPY and other data-manipulation statements are automatically committed after they complete. By default, AUTOCOMMIT is enabled—all statements are committed after they execute. This is often not the best setting to use, since it is less efficient. Also, you often want to control whether a set of statements are committed as a whole, rather than have each individual statement committed. For example, you may only want to commit a series of inserts if all of the inserts succeed. With AUTOCOMMIT disabled, you can roll back the transaction if one of the statements fail.

If AUTOCOMMIT is on, the results of statements are committed immediately after they are executed. You cannot roll back a statement executed in AUTOCOMMIT mode.

For example, when AUTOCOMMIT is on, the following single INSERT statement is automatically committed:

```
ret = SQLExecDirect(hdlStmt, (SQLCHAR*)"INSERT INTO customers VALUES(500,"
    "'Smith, Sam', '123-456-789');", SQL_NTS);
```

If AUTOCOMMIT is off, you need to manually commit the transaction after executing a statement. For example:

```
ret = SQLExecDirect(hdlStmt, (SQLCHAR*)"INSERT INTO customers VALUES(500,"
    "'Smith, Sam', '123-456-789');", SQL_NTS);
// Other inserts and data manipulations
// Commit the statements(s)
ret = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
```

The inserted row is only committed when you call SQLEndTran() . You can roll back the INSERT and other statements at any point before committing the transaction.

> **Note**
> Prepared statements cache the AUTOCOMMIT setting when you create them using SQLPrepare() . Later changing the connection's AUTOCOMMIT setting has no effect on the AUTOCOMMIT settings of previously created prepared statements. See Using prepared statements for details.

The following example demonstrates turning off AUTOCOMMIT, executing an insert, then manually committing the transaction.

```
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
int main()
{
  // Set up the ODBC environment
  SQLRETURN ret;
  SQLHENV hdlEnv;
  ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
  if(!SQL_SUCCEEDED(ret)) {
    printf("Could not allocate a handle.\n");
    exit(EXIT_FAILURE);
  } else {
    printf("Allocated an environment handle.\n");
  }
  // Tell ODBC that the application uses ODBC 3.
  ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
    (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
  if(!SQL_SUCCEEDED(ret)) {
    printf("Could not set application version to ODBC3.\n");
    exit(EXIT_FAILURE);
  } else {
    printf("Set application to ODBC 3.\n");
  }
  // Allocate a database handle.
  SQLHDBC hdlDbc;
  ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
  if(!SQL_SUCCEEDED(ret)) {
    printf("Could not allocate database handle.\n");
    exit(EXIT_FAILURE);
  } else {
    printf("Allocated Database handle.\n");
  }
  // Connect to the database
  printf("Connecting to database.\n");
  const char *dsnName = "ExampleDB";
  const char* userID = "dbadmin";
  const char* passwd = "password123";
  ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
    SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
    (SQLCHAR*)passwd, SQL_NTS);
  if(!SQL_SUCCEEDED(ret)) {
    printf("Could not connect to database.\n");
    exit(EXIT_FAILURE);
```

```
    exit(EXIT_FAILURE);
} else {
    printf("Connected to database.\n");
}
// Get the AUTOCOMMIT state
SQLINTEGER  autoCommitState;
SQLGetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, &autoCommitState, 0, NULL);
printf("Autocommit is set to: %d\n", autoCommitState);


// Disable AUTOCOMMIT
printf("Disabling autocommit.\n");
ret = SQLSetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF,
    SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not disable autocommit.\n");
    exit(EXIT_FAILURE);
}

// Get the AUTOCOMMIT state again
SQLGetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, &autoCommitState, 0, NULL);
printf("Autocommit is set to: %d\n", autoCommitState);

// Set up a statement handle
SQLHSTMT hdlStmt;
SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);


// Create a table to hold the data
SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers",
    SQL_NTS);
SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers "
    "(CustID int, CustName varchar(100), Phone_Number char(15));",
    SQL_NTS);


// Insert a single row.
ret = SQLExecDirect(hdlStmt, (SQLCHAR*)"INSERT INTO customers VALUES(500,"
    "'Smith, Sam', '123-456-789');", SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not perform single insert.\n");
} else {
    printf("Performed single insert.\n");
}


// Need to commit the transaction before closing, since autocommit is
// disabled. Otherwise SQLDisconnect returns an error.
printf("Committing transaction.\n");
ret =  SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
if(!SQL_SUCCEEDED(ret)) {
    printf("Error committing transaction.\n");
    exit(EXIT_FAILURE);
}

// Clean up
printf("Free handles.\n");
ret = SQLDisconnect(hdlDbc);
if(!SQL_SUCCEEDED(ret)) {
    printf("Error disconnecting from database. Transaction still open?\n");
    exit(EXIT_FAILURE);
}
```

```
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_SUCCESS);
}
```

Running the above code results in the following output:

```
Allocated an environment handle.
Set application to ODBC 3.
Allocated Database handle.
Connecting to database.
Connected to database.
Autocommit is set to: 1
Disabling autocommit.
Autocommit is set to: 0
Performed single insert.
Committing transaction.
Free handles.
```

> **Note**
> You can also disable AUTOCOMMIT in the ODBC connection string. See Setting DSN connection properties for more information.

Retrieving data

To retrieve data through ODBC, you execute a query that returns a result set (SELECT , for example), then retrieve the results using one of two methods:

- Use the SQLFetch() function to retrieve a row of the result set, then access column values in the row by calling SQLGetData() .
- Use the SQLBindColumn() function to bind a variable or array to a column in the result set, then call SQLExtendedFetch() or SQLFetchScroll() to read a row of the result set and insert its values into the variable or array.

In both methods you loop through the result set until you either reach the end (signaled by the SQL_NO_DATA return status) or encounter an error.

> **Note**
> Vertica supports one cursor per connection. Attempting to use more than one cursor per connection will result in an error. For example, you receive an error if you execute a statement while another statement has a result set open.

The following code example demonstrates retrieving data from Vertica by:

1. Connecting to the database.
2. Executing a SELECT statement that returns the IDs and names of all tables.
3. Binds two variables to the two columns in the result set.
4. Loops through the result set, printing the ids and name values.

```
// Demonstrate running a query and getting results by querying the tables
// system table for a list of all tables in the current schema.
// Some standard headers
#include <stdlib.h>
#include <sstream>
#include <iostream>
#include <assert.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>

// Use std namespace to make output easier
using namespace std;
```

```
// Helper function to print SQL error messages.
template <typename HandleT>
void reportError(int handleTypeEnum, HandleT hdl)
{
    // Get the status records.
    SQLSMALLINT    i, MsgLen;
    SQLRETURN ret2;
    SQLCHAR        SqlState[6], Msg[SQL_MAX_MESSAGE_LENGTH];
    SQLINTEGER     NativeError;
    i = 1;
    cout << endl;
    while ((ret2 = SQLGetDiagRec(handleTypeEnum, hdl, i, SqlState, &NativeError,
        Msg, sizeof(Msg), &MsgLen)) != SQL_NO_DATA) {
        cout << "error record #" << i++ << endl;
        cout << "sqlstate: " << SqlState << endl;
        cout << "detailed msg: " << Msg << endl;
        cout << "native error code: " << NativeError << endl;
    }
}

typedef struct {
    SQLHENV hdlEnv;
    SQLHDBC hdlDbc;
} DBConnection;

void connect(DBConnection *pConnInfo)
{
    // Set up the ODBC environment
    SQLRETURN ret;
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &pConnInfo->hdlEnv);
    assert(SQL_SUCCEEDED(ret));
    // Tell ODBC that the application uses ODBC 3.
    ret = SQLSetEnvAttr(pConnInfo->hdlEnv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER)SQL_OV_ODBC3, SQL_IS_UINTEGER);
    assert(SQL_SUCCEEDED(ret));

    // Allocate a database handle.
    ret = SQLAllocHandle(SQL_HANDLE_DBC, pConnInfo->hdlEnv, &pConnInfo->hdlDbc);
    assert(SQL_SUCCEEDED(ret));
    // Connect to the database
    cout << "Connecting to database:" << endl;
    const char* dsnName = "ExampleDB";
    const char* userID = "dbadmin";
    const char* passwd = "password123";
    ret = SQLConnect(pConnInfo->hdlDbc, (SQLCHAR*)dsnName,
        SQL_NTS, (SQLCHAR*)userID, SQL_NTS,
        (SQLCHAR*)passwd, SQL_NTS);
    if (!SQL_SUCCEEDED(ret)) {
        cout << "Could not connect to database" << endl;
        reportError<SQLHDBC>(SQL_HANDLE_DBC, pConnInfo->hdlDbc);
        exit(EXIT_FAILURE);
    }
    else {
        cout << "Connected to database." << endl;
    }
}

void disconnect(DBConnection *pConnInfo)
{
    SQLRETURN ret;
    // Clean up by shutting down the connection
```

```
    cout << "Free handles." << endl;
    ret = SQLDisconnect(pConnInfo->hdlDbc);
    if (!SQL_SUCCEEDED(ret)) {
        cout << "Error disconnecting. Transaction still open?" << endl;
        exit(EXIT_FAILURE);
    }
    SQLFreeHandle(SQL_HANDLE_DBC, pConnInfo->hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, pConnInfo->hdlEnv);
}

void executeQuery(SQLHDBC hdlDbc, SQLCHAR* pQuery)
{
    SQLRETURN ret;
    // Set up a statement handle
    SQLHSTMT hdlStmt;
    SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
    assert(SQL_SUCCEEDED(ret));

    // Execute a query to get the names and IDs of all tables in the schema
    // search p[ath (usually public).
    ret = SQLExecDirect(hdlStmt, pQuery, SQL_NTS);

    if (!SQL_SUCCEEDED(ret)) {
        // Report error an go no further if statement failed.
        cout << "Error executing statement." << endl;
        reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);
        exit(EXIT_FAILURE);
    }
    else {
        // Query succeeded, so bind two variables to the two colums in the
        // result set,
        cout << "Fetching results..." << endl;
        SQLBIGINT table_id;       // Holds the ID of the table.
        SQLTCHAR table_name[256]; // buffer to hold name of table
        ret = SQLBindCol(hdlStmt, 1, SQL_C_SBIGINT, (SQLPOINTER)&table_id,
            sizeof(table_id), NULL);
        ret = SQLBindCol(hdlStmt, 2, SQL_C_TCHAR, (SQLPOINTER)table_name,
            sizeof(table_name), NULL);

        // Loop through the results,
        while (SQL_SUCCEEDED(ret = SQLFetchScroll(hdlStmt, SQL_FETCH_NEXT, 1))) {
            // Print the bound variables, which now contain the values from the
            // fetched row.
            cout << table_id << " | " << table_name << endl;
        }


        // See if loop exited for reasons other than running out of data
        if (ret != SQL_NO_DATA) {
            // Exited for a reason other than no more data... report the error.
            reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);
        }
    }
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
}

int main()
{
    DBConnection conn;

    connect(&conn);
```

```
executeQuery(conn.hdlDbc,
    (SQLCHAR*)"SELECT table_id, table_name FROM tables ORDER BY table_name");
executeQuery(conn.hdlDbc,
    (SQLCHAR*)"SELECT table_id, table_name FROM tables ORDER BY table_id");
disconnect(&conn);
exit(EXIT_SUCCESS);
}
```

Running the example code in the vmart database produces output similar to this:

```
Connecting to database.
Connected to database.
Fetching results...
45035996273970908 | call_center_dimension
45035996273970836 | customer_dimension
45035996273972958 | customers
45035996273970848 | date_dimension
45035996273970856 | employee_dimension
45035996273970868 | inventory_fact
45035996273970904 | online_page_dimension
45035996273970912 | online_sales_fact
45035996273970840 | product_dimension
45035996273970844 | promotion_dimension
45035996273970860 | shipping_dimension
45035996273970876 | store_dimension
45035996273970894 | store_orders_fact
45035996273970880 | store_sales_fact
45035996273972806 | t
45035996273970852 | vendor_dimension
45035996273970864 | warehouse_dimension
Fetching results...
45035996273970836 | customer_dimension
45035996273970840 | product_dimension
45035996273970844 | promotion_dimension
45035996273970848 | date_dimension
45035996273970852 | vendor_dimension
45035996273970856 | employee_dimension
45035996273970860 | shipping_dimension
45035996273970864 | warehouse_dimension
45035996273970868 | inventory_fact
45035996273970876 | store_dimension
45035996273970880 | store_sales_fact
45035996273970894 | store_orders_fact
45035996273970904 | online_page_dimension
45035996273970908 | call_center_dimension
45035996273970912 | online_sales_fact
45035996273972806 | t
45035996273972958 | customers
Free handles.
```

Loading data

A primary task for many client applications is loading data into the Vertica database. There are several different ways to insert data using ODBC, which are covered by the topics in this section.

In this section

- Using a single row insert
- Using prepared statements
- Using batch inserts
- Using the COPY statement
- Streaming data from the client using COPY LOCAL

Using a single row insert

The easiest way to load data into Vertica is to run an INSERT SQL statement using the SQLExecuteDirect function. However this method is limited to inserting a single row of data.

```
ret = SQLExecDirect(hstmt, (SQLTCHAR*)"INSERT into Customers values"
    "(1,'abcda','efgh','1')", SQL_NTS);
```

Using prepared statements

Vertica supports using server-side prepared statements with both ODBC and JDBC. Prepared statements let you define a statement once, and then run it many times with different parameters. The statement you want to execute contains placeholders instead of parameters. When you execute the statement, you supply values for each placeholder.

Placeholders are represented by question marks (?) as in the following example query:

```
SELECT * FROM public.inventory_fact WHERE product_key = ?
```

Server-side prepared statements are useful for:

- Optimizing queries. Vertica only needs to parse the statement once.
- Preventing SQL injection attacks. A SQL injection attack occurs when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly run. Since a prepared statement is parsed separately from the input data, there is no chance the data can be accidentally executed by the database.
- Binding direct variables to return columns. By pointing to data structures, the code doesn't have to perform extra transformations.

The following example demonstrates a using a prepared statement for a single insert.

```
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
// Some constants for the size of the data to be inserted.
#define CUST_NAME_LEN 50
#define PHONE_NUM_LEN 15
#define NUM_ENTRIES 4
int main()
{
   // Set up the ODBC environment
   SQLRETURN ret;
   SQLHENV hdlEnv;
   ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
   if(!SQL_SUCCEEDED(ret)) {
      printf("Could not allocate a handle.\n");
      exit(EXIT_FAILURE);
   } else {
      printf("Allocated an environment handle.\n");
   }
   // Tell ODBC that the application uses ODBC 3.
   ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
      (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
   if(!SQL_SUCCEEDED(ret)) {
      printf("Could not set application version to ODBC3.\n");
      exit(EXIT_FAILURE);
   } else {
      printf("Set application to ODBC 3.\n");
   }
   // Allocate a database handle.
   SQLHDBC hdlDbc;
```

```
SQLHDBC hdlDbc;
ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
// Connect to the database
printf("Connecting to database.\n");
const char *dsnName = "ExampleDB";
const char* userID = "dbadmin";
const char* passwd = "password123";
ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
    SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
    (SQLCHAR*)passwd, SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not connect to database.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Connected to database.\n");
}

// Disable AUTOCOMMIT
printf("Disabling autocommit.\n");
ret = SQLSetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF,
    SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not disable autocommit.\n");
    exit(EXIT_FAILURE);
}


// Set up a statement handle
SQLHSTMT hdlStmt;
SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers",
    SQL_NTS);
SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers "
    "(CustID int, CustName varchar(100), Phone_Number char(15));",
    SQL_NTS);

// Set up a bunch of variables to be bound to the statement
// parameters.

// Create the prepared statement. This will insert data into the
// table we created above.
printf("Creating prepared statement\n");
ret = SQLPrepare (hdlStmt, (SQLTCHAR*)"INSERT INTO customers (CustID, "
    "CustName,  Phone_Number) VALUES(?,?,?)", SQL_NTS) ;
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not create prepared statement\n");
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_FAILURE);
} else {
    printf("Created prepared statement.\n");
}
SQLINTEGER custID = 1234;
SQLCHAR custName[100] = "Fein, Fredrick";
SQLVARCHAR phoneNum[15] = "555-123-6789";
SQLLEN strFieldLen = SQL_NTS;
SQLLEN custIDLen = 0;
// Bind the data arrays to the parameters in the prepared SQL
// statement
ret = SQLBindParameter(hdlStmt, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
    0, 0, &custID, 0, &custIDLen);
```

```
    0, 0, &custID, 0, &custIDLen);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind custID array\n");
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_FAILURE);
} else {
    printf("Bound custID to prepared statement\n");
}
// Bind CustNames
SQLBindParameter(hdlStmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
    50, 0, (SQLPOINTER)custName,  0, &strFieldLen);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind custNames\n");
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_FAILURE);
} else {
    printf("Bound custName to prepared statement\n");
}
// Bind phoneNums
SQLBindParameter(hdlStmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
    15, 0, (SQLPOINTER)phoneNum, 0, &strFieldLen);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind phoneNums\n");
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_FAILURE);
} else {
    printf("Bound phoneNum to prepared statement\n");
}
// Execute the prepared statement.
printf("Running prepared statement...");
ret = SQLExecute(hdlStmt);
if(!SQL_SUCCEEDED(ret)) {
    printf("not successful!\n");
}  else {
    printf("successful.\n");
}

// Done with batches, commit the transaction
printf("Committing transaction\n");
ret = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not commit transaction\n");
}  else {
    printf("Committed transaction\n");
}

// Clean up
printf("Free handles.\n");
ret = SQLDisconnect( hdlDbc );
if(!SQL_SUCCEEDED(ret)) {
    printf("Error disconnecting. Transaction still open?\n");
    exit(EXIT_FAILURE);
}
SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
```

```
    exit(EXIT_SUCCESS);
```

Using batch inserts

You use batch inserts to insert chunks of data into the database. By breaking the data into batches, you can monitor the progress of the load by receiving information about any rejected rows after each batch is loaded. To perform a batch load through ODBC, you typically use a prepared statement with the parameters bound to arrays that contain the data to be loaded. For each batch, you load a new set of data into the arrays then execute the prepared statement.

When you perform a batch load, Vertica uses a COPY statement to load the data. Each additional batch you load uses the same COPY statement. The statement remains open until you end the transaction, close the cursor for the statement, or execute a non-INSERT statement.

Using a single COPY statement for multiple batches improves batch loading efficiency by:

- reducing the overhead of inserting individual batches
- combining individual batches into larger ROS containers

> **Note**
> If the database connection has AUTOCOMMIT enabled, then the transaction is automatically committed after each batch insert statement which closes the COPY statement. Leaving AUTOCOMMIT enabled makes your batch load much less efficient, and can cause added overhead in your database as all of the smaller loads are consolidated.

Even though Vertica uses a single COPY statement to insert multiple batches within a transaction, you can locate which (if any) rows were rejected due to invalid row formats or data type issues after each batch is loaded. See Tracking load status (ODBC) for details.

> **Note**
> While you can find rejected rows during the batch load transaction, other types of errors (such as running out of disk space or a node shutdown that makes the database unsafe) are only reported when the COPY statement ends.

Since the batch loads share a COPY statement, errors in one batch can cause earlier batches in the same transaction to be rolled back.

Batch insert steps

The steps your application needs to take in order to perform an ODBC Batch Insert are:

1. Connect to the database.
2. Disable autocommit for the connection.
3. Create a prepared statement that inserts the data you want to load.
4. Bind the parameters of the prepared statement to arrays that will contain the data you want to load.
5. Populate the arrays with the data for your batches.
6. Execute the prepared statement.
7. Optionally, check the results of the batch load to find rejected rows.
8. Repeat the previous three steps until all of the data you want to load is loaded.
9. Commit the transaction.
10. Optionally, check the results of the entire batch transaction.

The following example code demonstrates a simplified version of the above steps.

```
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
int main()

    // Number of data rows to insert
```

```
// Number of data rows to insert
const int NUM_ENTRIES = 4;

// Set up the ODBC environment
SQLRETURN ret;
SQLHENV hdlEnv;
ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not allocate a handle.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Allocated an environment handle.\n");
}
// Tell ODBC that the application uses ODBC 3.
ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
    (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not set application version to ODBC3.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Set application to ODBC 3.\n");
}
// Allocate a database handle.
SQLHDBC hdlDbc;
ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not allocate database handle.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Allocated Database handle.\n");
}
// Connect to the database
printf("Connecting to database.\n");
const char *dsnName = "ExampleDB";
const char* userID = "dbadmin";
const char* passwd = "password123";
ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
    SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
    (SQLCHAR*)passwd, SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not connect to database.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Connected to database.\n");
}


// Disable AUTOCOMMIT
printf("Disabling autocommit.\n");
ret = SQLSetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF,
                        SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not disable autocommit.\n");
    exit(EXIT_FAILURE);
}

// Set up a statement handle
SQLHSTMT hdlStmt;
SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);

// Create a table to hold the data
SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers";
```

```
    SQL_NTS);
SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers "
    "(CustID int, CustName varchar(100), Phone_Number char(15));",
    SQL_NTS);

// Create the prepared statement. This will insert data into the
// table we created above.
printf("Creating prepared statement\n");
ret = SQLPrepare (hdlStmt, (SQLTCHAR*)"INSERT INTO customers (CustID, "
    "CustName,  Phone_Number) VALUES(?,?,?)", SQL_NTS) ;
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not create prepared statement\n");
    exit(EXIT_FAILURE);
} else {
    printf("Created prepared statement.\n");
}
// This is the data to be inserted into the database.
SQLCHAR custNames[][50] = { "Allen, Anna", "Brown, Bill", "Chu, Cindy",
    "Dodd, Don" };
SQLINTEGER custIDs[] = { 100, 101, 102, 103};
// year, month, day, hour, minute, second, fraction
// struct accepts fraction in billionths of a second, but Vertica supports millionths
SQL_TIMESTAMP_STRUCT accountCreationDates[] = {
    {1997, 4, 1, 12, 35, 29, 0},          // 1997-04-01 12:35:29
    {2002, 6, 13, 1,  0, 12, 1000},       // 2002-06-13 01:00:12.000001
    {2000, 9, 2, 2,  59, 37, 999000000},  // 2000-09-02 02:59:37.999
    {2009, 1, 25, 3,  7, 59, 999999000},  // 2009-01-25 03:07:59.999999
};
SQLCHAR phoneNums[][15] = {"1-617-555-1234", "1-781-555-1212",
    "1-508-555-4321", "1-617-555-4444"};
// Bind the data arrays to the parameters in the prepared SQL
// statement. First is the custID.
ret = SQLBindParameter(hdlStmt, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
    0, 0, (SQLPOINTER)custIDs, sizeof(SQLINTEGER) , NULL);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind custID array\n");
    exit(EXIT_FAILURE);
} else {
    printf("Bound CustIDs array to prepared statement\n");
}
// Bind the customer account creation date
// timestamp column size is safe at 23 + length of longest fractional component precision
// Max precision that vertica supports for timestamp second precision is 6 digits
ret = SQLBindParameter(stmt.hstmt, 2, SQL_PARAM_INPUT, SQL_C_TYPE_TIMESTAMP, SQL_TIMESTAMP, 29, 6,
                       (SQLPOINTER)accountCreationDates, sizeof(SQL_TIMESTAMP_STRUCT), NULL);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind account creation dates\n");
    exit(EXIT_FAILURE);
} else {
    printf("Bound account creation date array to prepared statement\n");
}
// Bind CustNames
ret = SQLBindParameter(hdlStmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
    50, 0, (SQLPOINTER)custNames, 50, NULL);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind custNames\n");
    exit(EXIT_FAILURE);
} else {
    printf("Bound CustNames array to prepared statement\n");
}
// Bind phoneNums
```

```
ret = SQLBindParameter(hdlStmt, 4, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
    15, 0, (SQLPOINTER)phoneNums, 15, NULL);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind phoneNums\n");
    exit(EXIT_FAILURE);
} else {
    printf("Bound phoneNums array to prepared statement\n");
}
// Tell the ODBC driver how many rows we have in the
// array.
ret = SQLSetStmtAttr( hdlStmt, SQL_ATTR_PARAMSET_SIZE,
    (SQLPOINTER)NUM_ENTRIES, 0 );
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind set parameter size\n");
    exit(EXIT_FAILURE);
} else {
    printf("Bound phoneNums array to prepared statement\n");
}

// Add multiple batches to the database. This just adds the same
// batch of data four times for simplicity's sake. Each call adds
// the 4 rows into the database.
for (int batchLoop=1; batchLoop<=5; batchLoop++) {
    // Execute the prepared statement, loading all of the data
    // in the arrays.
    printf("Adding Batch #%d...", batchLoop);
    ret = SQLExecute(hdlStmt);
    if(!SQL_SUCCEEDED(ret)) {
        printf("not successful\n");
    } else {
        printf("successful.\n");
    }
}
// Done with batches, commit the transaction
printf("Committing transaction\n");
ret = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not commit transaction\n");
} else {
    printf("Committed transaction\n");
}

// Clean up
printf("Free handles.\n");
ret = SQLDisconnect( hdlDbc );
if(!SQL_SUCCEEDED(ret)) {
    printf("Error disconnecting. Transaction still open?\n");
    exit(EXIT_FAILURE);
}
SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
exit(EXIT_SUCCESS);
```

The result of running the above code is shown below.

```
Allocated an environment handle.
Set application to ODBC 3.
Allocated Database handle.
Connecting to database.
Connected to database.
Creating prepared statement
Created prepared statement.
Bound CustIDs array to prepared statement
Bound CustNames array to prepared statement
Bound phoneNums array to prepared statement
Adding Batch #1...successful.
Adding Batch #2...successful.
Adding Batch #3...successful.
Adding Batch #4...successful.
Adding Batch #5...successful.
Committing transaction
Committed transaction
Free handles.
```

The resulting table looks like this:

```
=> SELECT * FROM customers;
 CustID |  CustName   |  Phone_Number
--------+-------------+----------------
    100 | Allen, Anna | 1-617-555-1234
    101 | Brown, Bill | 1-781-555-1212
    102 | Chu, Cindy  | 1-508-555-4321
    103 | Dodd, Don   | 1-617-555-4444
    100 | Allen, Anna | 1-617-555-1234
    101 | Brown, Bill | 1-781-555-1212
    102 | Chu, Cindy  | 1-508-555-4321
    103 | Dodd, Don   | 1-617-555-4444
    100 | Allen, Anna | 1-617-555-1234
    101 | Brown, Bill | 1-781-555-1212
    102 | Chu, Cindy  | 1-508-555-4321
    103 | Dodd, Don   | 1-617-555-4444
    100 | Allen, Anna | 1-617-555-1234
    101 | Brown, Bill | 1-781-555-1212
    102 | Chu, Cindy  | 1-508-555-4321
    103 | Dodd, Don   | 1-617-555-4444
    100 | Allen, Anna | 1-617-555-1234
    101 | Brown, Bill | 1-781-555-1212
    102 | Chu, Cindy  | 1-508-555-4321
    103 | Dodd, Don   | 1-617-555-4444
(20 rows)
```

> **Note**
> An input parameter bound with the SQL_C_NUMERIC data type uses the default numeric precision (37) and the default scale (0) instead of the precision and scale set by the SQL_NUMERIC_STRUCT input value. This behavior adheres to the ODBC standard. If you do not want to use the default precision and scale, use SQLSetDescField() or SQLSetDescRec() to change them in the statement's attributes.

In this section

- Tracking load status (ODBC)
- Error handling during batch loads

Tracking load status (ODBC)

After loading a batch of data, your client application can get the number of rows that were processed and find out whether each row was accepted or rejected.

Finding the number of accepted rows

To get the number of rows processed by a batch, you add an attribute named SQL_ATTR_PARAMS_PROCESSED_PTR to the statement object that points to a variable to receive the number rows:

```
SQLULEN rowsProcessed;
SQLSetStmtAttr(hdlStmt, SQL_ATTR_PARAMS_PROCESSED_PTR, &rowsProcessed, 0);
```

When your application calls SQLExecute() to insert the batch, the Vertica ODBC driver saves the number of rows that it processed (which is not necessarily the number of rows that were successfully inserted) in the variable you specified in the SQL_ATTR_PARAMS_PROCESSED_PTR statement attribute.

Finding the accepted and rejected rows

Your application can also set a statement attribute named SQL_ATTR_PARAM_STATUS_PTR that points to an array where the ODBC driver can store the result of inserting each row:

```
SQLUSMALLINT  rowResults[ NUM_ENTRIES ];
SQLSetStmtAttr(hdlStmt, SQL_ATTR_PARAM_STATUS_PTR, rowResults, 0);
```

This array must be at least as large as the number of rows being inserted in each batch.

When your application calls SQLExecute to insert a batch, the ODBC driver populates the array with values indicating whether each row was successfully inserted (SQL_PARAM_SUCCESS or SQL_PARAM_SUCCESS_WITH_INFO) or encountered an error (SQL_PARAM_ERROR).

The following example expands on the example shown in Using batch inserts to include reporting the number of rows processed and the status of each row inserted.

In this example, SQLGetDiagRec() is called several times to retrieve the failures for each bulk load. SQLGetDiagRec() returns up to 50 failures for any given operation:

```
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
// Helper function to print SQL error messages.
template <typename HandleT>
void reportError(int handleTypeEnum, HandleT hdl)
{
   // Get the status records.
   SQLSMALLINT   i, MsgLen;
   SQLRETURN ret2;
   SQLCHAR       SqlState[6], Msg[SQL_MAX_MESSAGE_LENGTH];
   SQLINTEGER    NativeError;
   i = 1;
   printf("\n");
   while ((ret2 = SQLGetDiagRec(handleTypeEnum, hdl, i, SqlState, &NativeError,
      Msg, sizeof(Msg), &MsgLen)) != SQL_NO_DATA) {
         printf("error record %d\n", i);
         printf("sqlstate: %s\n", SqlState);
         printf("detailed msg: %s\n", Msg);
         printf("native error code: %d\n\n", NativeError);
         i++;
   }
}
int main()
{
   // Number of data rows to insert
   const int NUM_ENTRIES = 4;
```

```
SQLRETURN ret;
SQLHENV hdlEnv;
ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not allocate a handle.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Allocated an environment handle.\n");
}
ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
    (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not set application version to ODBC3.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Set application to ODBC 3.\n");
}
SQLHDBC hdlDbc;
ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not allocate database handle.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Allocated Database handle.\n");
}
// Connect to the database
printf("Connecting to database.\n");
const char *dsnName = "ExampleDB";
const char* userID = "dbadmin";
const char* passwd = "password123";
ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
    SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
    (SQLCHAR*)passwd, SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not connect to database.\n");
    reportError<SQLHDBC>(SQL_HANDLE_DBC, hdlDbc);
    exit(EXIT_FAILURE);
} else {
    printf("Connected to database.\n");
}
// Set up a statement handle
SQLHSTMT hdlStmt;
SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers",
    SQL_NTS);
// Create a table into which we can store data
printf("Creating table.\n");
ret = SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers "
    "(CustID int, CustName varchar(50), Phone_Number char(15));",
    SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
    exit(EXIT_FAILURE);
} else {
    printf("Created table.\n");
}
// Create the prepared statement. This will insert data into the
// table we created above.
printf("Creating prepared statement\n");
ret = SQLPrepare (hdlStmt, (SQLTCHAR*)"INSERT INTO customers (CustID, "
    "CustName, Phone_Number) VALUES(?,?,?)", SQL_NTS);
```

```
    CustName, Phone_Number) VALUES(?,?,?)", SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
    exit(EXIT_FAILURE);
} else {
    printf("Created prepared statement.\n");
}
// This is the data to be inserted into the database.
char custNames[][50] = { "Allen, Anna", "Brown, Bill", "Chu, Cindy",
    "Dodd, Don" };
SQLINTEGER custIDs[] = { 100, 101, 102, 103};
char phoneNums[][15] = {"1-617-555-1234", "1-781-555-1212",
    "1-508-555-4321", "1-617-555-4444"};
// Bind the data arrays to the parameters in the prepared SQL
// statement
ret = SQLBindParameter(hdlStmt, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
    0, 0, (SQLPOINTER)custIDs, sizeof(SQLINTEGER) , NULL);
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
    exit(EXIT_FAILURE);
} else {
    printf("Bound CustIDs array to prepared statement\n");
}
// Bind CustNames
SQLBindParameter(hdlStmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
    50, 0, (SQLPOINTER)custNames, 50, NULL);
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
    exit(EXIT_FAILURE);
} else {
    printf("Bound CustNames array to prepared statement\n");
}
// Bind phoneNums
SQLBindParameter(hdlStmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
    15, 0, (SQLPOINTER)phoneNums, 15, NULL);
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
    exit(EXIT_FAILURE);
} else {
    printf("Bound phoneNums array to prepared statement\n");
}
// Set up a variable to recieve number of parameters processed.
SQLULEN rowsProcessed;
// Set a statement attribute to point to the variable
SQLSetStmtAttr(hdlStmt, SQL_ATTR_PARAMS_PROCESSED_PTR, &rowsProcessed, 0);
// Set up an array to hold the result of each row insert
SQLUSMALLINT   rowResults[ NUM_ENTRIES ];
// Set a statement attribute to point to the array
SQLSetStmtAttr(hdlStmt, SQL_ATTR_PARAM_STATUS_PTR, rowResults, 0);
// Tell the ODBC driver how many rows we have in the
// array.
SQLSetStmtAttr(hdlStmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)NUM_ENTRIES, 0);
// Add multiple batches to the database. This just adds the same
// batch of data over and over again for simplicity's sake.
for (int batchLoop=1; batchLoop<=5; batchLoop++) {
    // Execute the prepared statement, loading all of the data
    // in the arrays.
    printf("Adding Batch #%d...", batchLoop);
    ret = SQLExecute(hdlStmt);
    if(!SQL_SUCCEEDED(ret)) {
        reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
        exit(EXIT_FAILURE);
```

```
    exit(EXIT_FAILURE);
  }
  // Number of rows processed is in rowsProcessed
  printf("Params processed: %d\n", rowsProcessed);
  printf("Results of inserting each row:\n");
  int i;
  for (i = 0; i<NUM_ENTRIES; i++) {
    SQLUSMALLINT result = rowResults[i];
    switch(rowResults[i]) {
      case SQL_PARAM_SUCCESS:
      case SQL_PARAM_SUCCESS_WITH_INFO:
        printf("  Row %d inserted successsfully\n", i+1);
        break;
      case SQL_PARAM_ERROR:
        printf("  Row %d was not inserted due to an error.", i+1);
        break;
      default:
        printf("  Row %d had some issue with it: %d\n", i+1, result);
    }
  }
}
// Done with batches, commit the transaction
printf("Commit Transaction\n");
ret = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
if(!SQL_SUCCEEDED(ret)) {
  reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
}


// Clean up
printf("Free handles.\n");
ret = SQLDisconnect( hdlDbc );
if(!SQL_SUCCEEDED(ret)) {
  printf("Error disconnecting. Transaction still open?\n");
  exit(EXIT_FAILURE);
}
SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
exit(EXIT_SUCCESS);
```

Running the example code produces the following output:

```
Allocated an environment handle.Set application to ODBC 3.
Allocated Database handle.
Connecting to database.
Connected to database.
Creating table.
Created table.
Creating prepared statement
Created prepared statement.
Bound CustIDs array to prepared statement
Bound CustNames array to prepared statement
Bound phoneNums array to prepared statement
Adding Batch #1...Params processed: 4
Results of inserting each row:
  Row 1 inserted successfully
  Row 2 inserted successfully
  Row 3 inserted successfully
  Row 4 inserted successfully
Adding Batch #2...Params processed: 4
Results of inserting each row:
  Row 1 inserted successfully
  Row 2 inserted successfully
  Row 3 inserted successfully
  Row 4 inserted successfully
Adding Batch #3...Params processed: 4
Results of inserting each row:
  Row 1 inserted successfully
  Row 2 inserted successfully
  Row 3 inserted successfully
  Row 4 inserted successfully
Adding Batch #4...Params processed: 4
Results of inserting each row:
  Row 1 inserted successfully
  Row 2 inserted successfully
  Row 3 inserted successfully
  Row 4 inserted successfully
Adding Batch #5...Params processed: 4
Results of inserting each row:
  Row 1 inserted successfully
  Row 2 inserted successfully
  Row 3 inserted successfully
  Row 4 inserted successfully
Commit Transaction
Free handles.
```

Error handling during batch loads

When loading individual batches, you can find information on how many rows were accepted and what rows were rejected (see Tracking load status (ODBC) for details). Other errors, such as disk space errors, do not occur while inserting individual batches. This behavior is caused by having a single COPY statement perform the loading of multiple consecutive batches. Using the single COPY statement makes the batch load process perform much faster. It is only when the COPY statement closes that the batched data is committed and Vertica reports other types of errors.

Your bulk loading application should check for errors when the COPY statement closes. Normally, you force the COPY statement to close by calling the SQLEndTran() function to end the transaction. You can also force the COPY statement to close by closing the cursor using the SQLCloseCursor() function, or by setting the database connection's AutoCommit property to true before inserting the last batch in the load.

> **Note**
> The COPY statement also closes if you execute any non-insert statement. However having to deal with errors from the COPY statement in what might be an otherwise-unrelated query is not intuitive, and can lead to confusion and a harder to maintain application. You should explicitly end the COPY statement at the end of your batch load and handle any errors at that time.

Using the COPY statement

COPY lets you bulk load data from a file stored on a database node into the Vertica database. This method is the most efficient way to load data into Vertica because the file resides on the database server. You must be a superuser to use COPY to access the file system of the database node.

Important
In databases that were created in versions of Vertica ≤ 9.2, COPY supports the DIRECT option, which specifies to load data directly into ROS rather than WOS. Use this option when loading large (>100MB) files into the database; otherwise, the load is liable to fill the WOS. When this occurs, the Tuple Mover must perform a moveout operation on the WOS data. It is more efficient to directly load into ROS and avoid forcing a moveout.

In databases created in Vertica 9.3, Vertica ignores load options and hints and always uses a load method of DIRECT. Databases created in versions ≥ 10.0 no longer support WOS and moveout operations; all data is always loaded directly into ROS.

> **Note**
> The exceptions/rejections files are created on the client machine when the exceptions and rejected data modifiers are specified on the COPY command. Specify a local path and filename for these modifiers when executing a COPY query from the driver.

The following example demonstrates using the COPY command:

```
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
// Helper function to determine if an ODBC function call returned
// successfully.
bool notSuccess(SQLRETURN ret) {
    return (ret != SQL_SUCCESS && ret != SQL_SUCCESS_WITH_INFO);
}

int main()
{
    // Set up the ODBC environment
    SQLRETURN ret;
    SQLHENV hdlEnv;
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    if(notSuccess(ret)) {
        printf("Could not allocate a handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated an environment handle.\n");
    }
    // Tell ODBC that the application uses ODBC 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    if(notSuccess(ret)) {
        printf("Could not set application version to ODBC3.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Set application to ODBC 3.\n");
    }
    // Allocate a database handle.
    SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    // Connect to the database
    printf("Connecting to database.\n");
```

```
printf("Connecting to database\n");
const char *dsnName = "ExampleDB";

// Note: User MUST be a database superuser to be able to access files on the
// filesystem of the node.
const char* userID = "dbadmin";
const char* passwd = "password123";
ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
    SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
    (SQLCHAR*)passwd, SQL_NTS);
if(notSuccess(ret)) {
    printf("Could not connect to database.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Connected to database.\n");
}

// Disable AUTOCOMMIT
printf("Disabling autocommit.\n");
ret = SQLSetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, SQL_NTS);
if(notSuccess(ret)) {
    printf("Could not disable autocommit.\n");
    exit(EXIT_FAILURE);
}

// Set up a statement handle
SQLHSTMT hdlStmt;
SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
// Create table to hold the data
SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers",
    SQL_NTS);
SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers"
    "(Last_Name char(50) NOT NULL, First_Name char(50),Email char(50), "
    "Phone_Number char(15));",
    SQL_NTS);

// Run the copy command to load data.
ret=SQLExecDirect(hdlStmt, (SQLCHAR*)"COPY customers "
    "FROM '/data/customers.txt'",
    SQL_NTS);
if(notSuccess(ret)) {
    printf("Data was not successfully loaded.\n");
    exit(EXIT_FAILURE);
} else {
    // Get number of rows added.
    SQLLEN numRows;
    ret=SQLRowCount(hdlStmt, &numRows);
    printf("Successfully inserted %d rows.\n", numRows);

}

// Done with batches, commit the transaction
printf("Committing transaction\n");
ret = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
if(notSuccess(ret)) {
    printf("Could not commit transaction\n");
}  else {
    printf("Committed transaction\n");
}

// Clean up
printf("Free handles.\n");
```

```
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_SUCCESS);
}
```

The example prints the following when run:

```
Allocated an environment handle.
Set application to ODBC 3.
Connecting to database.
Connected to database.
Disabling autocommit.
Successfully inserted 10001 rows.
Committing transaction
Committed transaction
Free handles.
```

Streaming data from the client using COPY LOCAL

COPY LOCAL streams data from a client system file to your Vertica database. This statement works through the ODBC driver, which simplifies the task of transferring data files from the client to the server.

COPY LOCAL works transparently through the ODBC driver. When a client application executes a COPY LOCAL statement, the ODBC driver reads and streams the data file from the client to the server.

> **Note**
> COPY LOCAL must be the first statement in a query,otherwise Vertica returns an error.

This example demonstrates loading data from the client system using the COPY LOCAL statement:

```
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
int main()
{
    // Set up the ODBC environment
    SQLRETURN ret;
    SQLHENV hdlEnv;
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not allocate a handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated an environment handle.\n");
    }
    // Tell ODBC that the application uses ODBC 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not set application version to ODBC3.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Set application to ODBC 3.\n");
    }
```

```
}
// Allocate a database handle.
SQLHDBC hdlDbc;
ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not aalocate a database handle.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Set application to ODBC 3.\n");
}
// Connect to the database
printf("Connecting to database.\n");
const char *dsnName = "ExampleDB";
const char* userID = "dbadmin";
const char* passwd = "password123";
ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
    SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
    (SQLCHAR*)passwd, SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not connect to database.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Connected to database.\n");
}

// Set up a statement handle
SQLHSTMT hdlStmt;
SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);


// Create table to hold the data
SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers",
    SQL_NTS);
SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers"
    "(Last_Name char(50) NOT NULL, First_Name char(50),Email char(50), "
    "Phone_Number char(15));",
    SQL_NTS);

// Run the copy command to load data.
ret=SQLExecDirect(hdlStmt, (SQLCHAR*)"COPY customers "
    "FROM LOCAL '/home/dbadmin/customers.txt'",
    SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    printf("Data was not successfully loaded.\n");
    exit(EXIT_FAILURE);
} else {
    // Get number of rows added.
    SQLLEN numRows;
    ret=SQLRowCount(hdlStmt, &numRows);
    printf("Successfully inserted %d rows.\n", numRows);
}

// COPY commits automatically, unless it is told not to, so
// there is no need to commit the transaction.

// Clean up
printf("Free handles.\n");
ret = SQLDisconnect( hdlDbc );
if(!SQL_SUCCEEDED(ret)) {
    printf("Error disconnecting. Transaction still open?\n");
    exit(EXIT_FAILURE);
}
```

```
SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
exit(EXIT_SUCCESS);
```

This example is essentially the same as the example shown in Using the COPY statement , except it uses the COPY statement's LOCAL option to load data from the client system rather than from the file system of the database node.

> **Note**
> On Windows clients, the path you supply for the COPY LOCAL file is limited to 216 characters due to limitations in the Windows API.

C#

The Vertica driver for ADO.NET allows applications written in C# to read data from, update, and load data into Vertica databases. It provides a data adapter (Vertica Data Adapter) that facilitates reading data from a database into a data set, and then writing changed data from the data set back to the database. It also provides a data reader (VerticaDataReader) for reading data. The driver requires the .NET framework version 3.5+.

For more information about ADO.NET, see:

- Overview of ADO.NET
- .NET Framework Developer Guide

Prerequisites

You must install the ADO.NET client driver before creating C# client applications.

In this section

- ADO.NET data types
- Setting the locale for ADO.NET sessions
- Connecting to the database
- Querying the database using ADO.NET
- Canceling ADO.NET queries
- Handling messages
- Getting table metadata

ADO.NET data types

This table details the mapping between Vertica data types and .NET and ADO.NET data types.

.NET Framework Type	ADO.NET DbType	VerticaType	Vertica Data Type	VerticaDataReader getter
Boolean	Boolean	Bit	Boolean	GetBoolean()
byte[]	Binary	Binary VarBinary LongVarBinary	Binary VarBinary LongVarBinary	GetBytes() **Note** The limit for LongVarBinary is 32 Million bytes. If you attempt to insert more than the limit during a batch transfer for any one row, then they entire batch fails. Verify the size of the data before attempting to insert a LongVarBinary during a batch.

Datetime	DateTime	Date Time TimeStamp	Date Time TimeStamp	GetDateTime() **Note** The Time portion of the DateTime object for vertica dates is set to DateTime.MinValue. Previously, VerticaType.DateTime was used for all date/time types. VerticaType.DateTime still exists for backwards compatibility, but now there are more specific VerticaTypes for each type.
DateTimeOffset	DateTimeOffset	TimestampTZ TimeTZ	TimestampTZ TimeTZ	GetDateTimeOffset() **Note** The Date portion of the DateTime is set to DateTime.MinValue
Decimal	Decimal	Numeric	Numeric	GetDecimal()
Double	Double	Double	Double Precision	GetDouble() **Note** Vertica Double type uses a default precision of 53.
Int64	Int64	BigInt	Integer	GetInt64()
TimeSpan	Object	13 Interval Types	13 Interval Types	GetInterval() **Note** There are 13 VerticaType values for the 13 types of intervals. The specific VerticaType used determines the conversion rules that the driver applies. Year/Month intervals represented as 365/30 days
String	String	Varchar LongVarChar	Varchar LongVarChar	GetString()
String	StringFixedLengt	Char	Char	GetString()
Guid	Guid	UUID (see note below)	UUID	GetGuid()
Object	Object	N/A	N/A	GetValue()

UUID backwards compatibility

Vertica version 9.0.0 introduced the UUID data type, including JDBC support for UUIDs. The Vertica ADO.NET, ODBC, and OLE DB clients added full support for UUIDs in version 9.0.1. Vertica maintains backwards compatibility with older supported client driver versions that do not support the UUID data type, as follows:

When an older client...	**Vertica...**

Queries tables with UUID columns	Translates the native UUID values to CHAR values.
Inserts data into a UUID column	Converts the CHAR value sent by the client into a native UUID value.
Queries a UUID column's metadata	Reports its data type as CHAR.

Setting the locale for ADO.NET sessions

- ADO.NET applications use a UTF-16 character set encoding and are responsible for converting any non-UTF-16 encoded data to UTF-16. The same cautions as for ODBC apply if this encoding is violated.
- The ADO.NET driver converts UTF-16 data to UTF-8 when passing to the Vertica server and converts data sent by Vertica server from UTF-8 to UTF-16.
- ADO.NET applications should set the correct server session locale by executing the SET LOCALE TO command in order to get expected collation and string functions behavior on the server.
- If there is no default session locale at the database level, ADO.NET applications need to set the correct server session locale by executing the SET LOCALE TO command in order to get expected collation and string functions behavior on the server. See the SET LOCALE command.

Connecting to the database

In this section

- Using TLS: installing certificates on Windows
- Opening and closing the database connection (ADO.NET)
- ADO.NET connection properties
- Load balancing in ADO.NET
- ADO.NET connection failover

Using TLS: installing certificates on Windows

You can optionally secure communication between your ADO.NET application and Vertica using TLS. The Vertica ADO.NET driver uses the default Windows key store when looking for TLS certificates. This is the same key store that Internet Explorer uses.

Before you can use TLS on the client side, you must implement TLS on the server. See TLS protocol , perform those steps, then return to this topic to install the TLS certificate on Windows.

To use TLS for ADO.NET connections to Vertica:

- Import the server and client certificates into the Windows Key Store.
- If required by your certificates, import the public certificate of your Certifying Authority.

Import the server and client certificates into the windows key store:

1. Copy the server.crt file you generated when you enabled TLS on the server to your Windows Machine.
2. Double-click the certificate.
3. Let Windows determine the key type, and click **Install** .

Import the public certificate of your CA:

You must establish a chain of trust for the certificates. You may need to import the public certificate for your Certifying Authority (CA) (especially if it is a self-signed certificate).

1. Using the same certificate as above, double-click the certificate.
2. Select **Place all certificates in the following store** .
3. Click **Browse,** select **Trusted Root Certification Authorities** and click **Next** .
4. Click **Install** .

Enable SSL in your ADO.NET applications

In your connection string, be sure to enable SSL by setting the SSL property in VerticaConnectionStringBuilder to true, for example:

```
//configure connection properties    VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
   builder.Host = "192.168.17.10";
   builder.Database = "VMart";
   builder.User = "dbadmin";
   builder.SSL = true;
   //open the connection
   VerticaConnection _conn = new VerticaConnection(builder.ToString());
   _conn.Open();
```

Opening and closing the database connection (ADO.NET)

Before you can access data in Vertica through ADO.NET, you must create a connection to the database using the VerticaConnection class which is an implementation of System.Data.DbConnection. The VerticaConnection class takes a single argument that contains the connection properties as a string. You can manually create a string of property keywords to use as the argument, or you can use the VerticaConnectionStringBuilder class to build a connection string for you.

To download the ADO.NET driver, go to the Client Drivers Downloads page .

This topic details the following:

- Manually building a connection string and connecting to Vertica
- Using VerticaConnectionStringBuilder to create the connection string and connecting to Vertica
- Closing the connection

To manually create a connection string:

See ADO.NET connection properties for a list of available properties to use in your connection string. At a minimum, you need to specify the Host, Database, and User.

1. For each property, provide a value and append the properties and values one after the other, separated by a semicolon. Assign this string to a variable. For example:
   ```
   String connectString = "DATABASE=VMart;HOST=v_vmart_node0001;USER=dbadmin";
   ```
2. Build a Vertica connection object that specifies your connection string.
   ```
   VerticaConnection _conn = new VerticaConnection(connectString)
   ```
3. Open the connection.
   ```
   _conn.Open();
   ```
4. Create a command object and associate it with a connection. All VerticaCommand objects must be associated with a connection.
   ```
   VerticaCommand command = _conn.CreateCommand();
   ```

To use the VerticaConnectionStringBuilder class to create a connection string and open a connection:

1. Create a new object of the VerticaConnectionStringBuilder class.
   ```
   VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
   ```
2. Update your VerticaConnectionStringBuilder object with property values. See ADO.NET connection properties for a list of available properties to use in your connection string. At a minimum, you need to specify the Host, Database, and User.
   ```
   builder.Host = "v_vmart_node0001";
   builder.Database = "VMart";
   builder.User = "dbadmin";
   ```
3. Build a Vertica connection object that specifies your connection VerticaConnectionStringBuilder object as a string.
   ```
   VerticaConnection _conn = new VerticaConnection(builder.ToString());
   ```
4. Open the connection.
   ```
   _conn.Open();
   ```
5. Create a command object and associate it with a connection. All VerticaCommand objects must be associated with a connection.
   ```
   VerticaCommand command = _conn.CreateCommand;
   ```

> **Note**
> If your database is not in compliance with your Vertica license, the call to VerticaConnection.open() returns a warning message to the console and the log. See Managing licenses for more information.

To close the connection:

When you're finished with the database, close the connection. Failure to close the connection can deteriorate the performance and scalability of your application. It can also prevent other clients from obtaining locks.

```
_conn.Close();
```

Example usage:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
         builder.Host = "192.168.1.10";
         builder.Database = "VMart";
         builder.User = "dbadmin";
         VerticaConnection _conn = new VerticaConnection(builder.ToString());
         _conn.Open();
      //Perform some operations
         _conn.Close();
      }
   }
}
```

ADO.NET connection properties

To download the ADO.NET driver, go to the Client Drivers Downloads page .

You use connection properties to configure the connection between your ADO.NET client application and your Vertica database. The properties provide the basic information about the connections, such as the server name and port number, needed to connect to your database.

You can set a connection property in two ways:

- Include the property name and value as part of the connection string you pass to a VerticaConnection .
- Set the properties in a VerticaConnectionStringBuilder object, and then pass the object as a string to a VerticaConnection .

Property	Description	Default Value
Database	Name of the Vertica database to which you want to connect. For example, if you installed the example VMart database, the database is "VMart".	none
User	Name of the user to log into Vertica.	none
Port	Port on which Vertica is running.	5433

Host	The host name or IP address of the server on which Vertica is running. You can provide an IPv4 address, IPv6 address, or host name. In mixed IPv4/IPv6 networks, the DNS server configuration determines which IP version address is sent first. Use the PreferredAddressFamily option to force the connection to use either IPv4 or IPv6.	none
PreferredAddressFamily	The IP version to use if the client and server have both IPv4 and IPv6 addresses and you have provided a host name. Valid values are: • Ipv4—Connect to the server using IPv4. • Ipv6—Connect to the server using IPv6. • None—Use the IP address provided by the DNS server.	Vertica.Data.VerticaClient.AddressFamilyPreference.None
Password	The password associated with the user connecting to the server.	string.Empty
BinaryTransfer	Provides a Boolean value that, when set to true, uses binary transfer instead of text transfer. When set to false, the ADO.NET connection uses text transfer. Binary transfer provides faster performance in reading data from a server to an ADO.NET client. Binary transfer also requires less bandwidth than text transfer, although it sometimes uses more when transferring a large number of small values. Binary transfer mode is not backwards compatible to ADO.NET versions earlier than 3.8. If you are using an earlier version, set this value to false. The data output by both modes is identical with the following exceptions for certain data types: • FLOAT: Binary transfer has slightly better precision. • TIMESTAMPTZ: Binary transfer can fail to get the session time zone and default to the local time zone, while text transfer reliably uses the session time zone. • NUMERIC: Binary transfer is forcibly disabled for NUMERIC data by the server for Vertica 11.0.2+.	true
ConnSettings	SQL commands to run upon connection. Uses **%3B** for semicolons.	string.Empty
IsolationLevel	Sets the transaction isolation level for Vertica. See Transactions for a description of the different transaction levels. This value is either Serializable, ReadCommitted, or Unspecified. See Setting the transaction isolation level for an example of setting the isolation level using this keyword. **Note:** By default, this value is set to IsolationLevel.Unspecified, which means the connection uses the server's default transaction isolation level. Vertica's default isolation level is IsolationLevel.ReadCommitted.	System.Data. IsolationLevel.Unspecified

Label	A string to identify the session on the server.	string
DirectBatchInsert	Deprecated	true
ResultBufferSize	The size of the buffer to use when streaming results. A value of 0 means ResultBufferSize is turned off.	8192
ConnectionTimeout	Number seconds to wait for a connection. A value of 0 means no timeout.	0
ReadOnly	A Boolean value. If true, throw an exception on write attempts.	false
Pooling	A boolean value, whether to enable connection pooling. Connection pooling is useful for server applications because it allows the server to reuse connections. This saves resources and enhances the performance of executing commands on the database. It also reduces the amount of time a user must wait to establish a connection to the database	false
MinPoolSize	An integer that defines the minimum number of connections to pool. **Valid Values:** Cannot be greater than the number of connections that the server is configured to allow. Otherwise, an exception results. **Default:** 55	1
MaxPoolSize	An integer that defines the maximum number of connections to pool. **Valid Values:** Cannot be greater than the number of connections that the server is configured to allow. Otherwise, an exception results.	20
LoadBalanceTimeout	The amount of time, expressed in seconds, to timeout or remove unused pooled connections. **Disable: **Set to 0 (no timeouts) If you are using a cluster environment to load-balance the work, then pool is restricted to the servers in the cluster when the pool was created. If additional servers are added to the cluster, and the pool is not removed, then the new servers are never added to the connection pool unless LoadBalanceTimeout is set and exceeded or VerticaConnection.ClearAllPools() is called manually from an application. If you are using load balancing, then set this property to a value that considers when new servers are added to the cluster. However, do not set it so low that pools are frequently removed and rebuilt, doing so makes pooling ineffective.	0 (no timeout)
SSL	A Boolean value, indicating whether to use SSL for the connection.	false

IntegratedSecurity	Provides a Boolean value that, when set to true, uses the user's Windows credentials for authentication, instead of user/password in the connection string.	false
KerberosServiceName	Provides the service name portion of the Vertica Kerberos principal; for example: vertica/host@EXAMPLE.COM	vertica
KerberosHostname	Provides the instance or host name portion of the Vertica Kerberos principal; for example: verticaost@EXAMPLE.COM	Value specified in the servername connection string property

Load balancing in ADO.NET

Native connection load balancing

Native connection load balancing helps spread the overhead caused by client connections on the hosts in the Vertica database. Both the server and the client must enable native connection load balancing. If enabled by both, then when the client initially connects to a host in the database, the host picks a host to handle the client connection from a list of the currently up hosts in the database, and informs the client which host it has chosen.

If the initially-contacted host does not choose itself to handle the connection, the client disconnects, then opens a second connection to the host selected by the first host. The connection process to this second host proceeds as usual—if SSL is enabled, then SSL negotiations begin, otherwise the client begins the authentication process. See About native connection load balancing for details.

To enable native load balancing on your client, set the ConnectionLoadBalance connection parameter to true either in the connection string or using the ConnectionStringBuilder() . The following example demonstrates connecting to the database several times with native connection load balancing enabled, and fetching the name of the node handling the connection from the V_MONITOR. CURRENT_SESSION system table.

```
using System;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;

namespace ConsoleApplication1 {
   class Program {
      static void Main(string[] args) {
         VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
         builder.Host = "v_vmart_node0001.example.com";
         builder.Database = "VMart";
         builder.User = "dbadmin";
         // Enable native client load balancing in the client,
         // must also be enabled on the server!
         builder.ConnectionLoadBalance = true;

         // Connect 3 times to verify a new node is connected
         // for each connection.
         for (int i = 1; i <= 4; i++) {
            try {
               VerticaConnection _conn = new VerticaConnection(builder.ToString());
               _conn.Open();
               if (i == 1) {
                  // On the first connection, check the server policy for load balance
                  VerticaCommand sqlcom = _conn.CreateCommand();
                  sqlcom.CommandText = "SELECT LOAD_BALANCE_POLICY FROM V_CATALOG.DATABASES";
                  var returnValue = sqlcom.ExecuteScalar();
                  Console.WriteLine("Status of load balancy policy
          on server: " + returnValue.ToString() + "\n");
               }
               VerticaCommand command = _conn.CreateCommand();
               command.CommandText = "SELECT node_name FROM V_MONITOR.CURRENT_SESSION";
               VerticaDataReader dr = command.ExecuteReader();
               while (dr.Read()) {
                  Console.Write("Connect attempt #" + i + "... ");
                  Console.WriteLine("Connected to node " + dr[0]);
               }
               dr.Close();
               _conn.Close();
               Console.WriteLine("Disconnecting.\n");
            }
            catch(Exception e) {
               Console.WriteLine(e.Message);
            }
         }
      }
   }
}
```

Running the above example produces the following output:

```
Status of load balancing policy on server: roundrobin

Connect attempt #1... Connected to node v_vmart_node0001
Disconnecting.

Connect attempt #2... Connected to node v_vmart_node0002
Disconnecting.

Connect attempt #3... Connected to node v_vmart_node0003
Disconnecting.

Connect attempt #4... Connected to node v_vmart_node0001
Disconnecting.
```

Hostname-based load balancing

You can also balance workloads by resolving a single hostname to multiple IP addresses. The ADO.NET client driver load balances by automatically resolving the hostname to one of the specified IP addresses at random.

For example, suppose the hostname verticahost.example.com has the following entries in C:\Windows\System32\drivers\etc\hosts :

```
192.0.2.0 verticahost.example.com
192.0.2.1 verticahost.example.com
192.0.2.2 verticahost.example.com
```

Specifying the hostname verticahost.example.com randomly resolves to one of the listed IP addresses.

ADO.NET connection failover

If a client application attempts to connect to a host in the Vertica cluster that is down, the connection attempt fails when using the default connection configuration. This failure usually returns an error to the user. The user must either wait until the host recovers and retry the connection or manually edit the connection settings to choose another host.

Due to Vertica Analytic Database's distributed architecture, you usually do not care which database host handles a client application's connection. You can use the client driver's connection failover feature to prevent the user from getting connection errors when the host specified in the connection settings is unreachable. The JDBC driver gives you several ways to let the client driver automatically attempt to connect to a different host if the one specified in the connection parameters is unreachable:

- Configure your DNS server to return multiple IP addresses for a host name. When you use this host name in the connection settings, the client attempts to connect to the first IP address from the DNS lookup. If the host at that IP address is unreachable, the client tries to connect to the second IP, and so on until it either manages to connect to a host or it runs out of IP addresses.
- Supply a list of backup hosts for the client driver to try if the primary host you specify in the connection parameters is unreachable.
- (JDBC only) Use driver-specific connection properties to manage timeouts before attempting to connect to the next node.

For all methods, the process of failover is transparent to the client application (other than specifying the list of backup hosts, if you choose to use the list method of failover). If the primary host is unreachable, the client driver automatically tries to connect to other hosts.

Failover only applies to the initial establishment of the client connection. If the connection breaks, the driver does not automatically try to reconnect to another host in the database.

Choosing a failover method

You usually choose to use one of the two failover methods. However, they do work together. If your DNS server returns multiple IP addresses and you supply a list of backup hosts, the client first tries all of the IPs returned by the DNS server, then the hosts in the backup list.

> **Note**
> If a host name in the backup host list resolves to multiple IP addresses, the client does not try all of them. It just tries the first IP address in the list.

The DNS method of failover centralizes the configuration client failover. As you add new nodes to your Vertica Analytic Database cluster, you can choose to add them to the failover list by editing the DNS server settings. All client systems that use the DNS server to connect to Vertica Analytic Database automatically use connection failover without having to change any settings. However, this method does require administrative access to the DNS server that all clients use to connect to the Vertica Analytic Database cluster. This may not be possible in your organization.

Using the backup server list is easier than editing the DNS server settings. However, it decentralizes the failover feature. You may need to update the application settings on each client system if you make changes to your Vertica Analytic Database cluster.

Using DNS failover

To use DNS failover, you need to change your DNS server's settings to map a single host name to multiple IP addresses of hosts in your Vertica Analytic Database cluster. You then have all client applications use this host name to connect to Vertica Analytic Database.

You can choose to have your DNS server return as many IP addresses for the host name as you want. In smaller clusters, you may choose to have it return the IP addresses of all of the hosts in your cluster. However, for larger clusters, you should consider choosing a subset of the hosts to return. Otherwise there can be a long delay as the client driver tries unsuccessfully to connect to each host in a database that is down.

Using the backup host list

To enable backup list-based connection failover, your client application has to specify at least one IP address or host name of a host in the BackupServerNode parameter. The host name or IP can optionally be followed by a colon and a port number. If not supplied, the driver defaults to the standard Vertica port number (5433). To list multiple hosts, separate them by a comma.

The following example demonstrates setting the BackupServerNode connection parameter to specify additional hosts for the connection attempt. The connection string intentionally has a non-existent node, so that the initial connection fails. The client driver has to resort to trying the backup hosts to establish a connection to Vertica.

```
using System;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         VerticaConnectionStringBuilder builder =
      new VerticaConnectionStringBuilder();
         builder.Host = "not.a.real.host:5433";
         builder.Database = "VMart";
         builder.User = "dbadmin";
         builder.BackupServerNode =
      "another.broken.node:5433,v_vmart_node0002.example.com:5433";
         try
         {
            VerticaConnection _conn =
         new VerticaConnection(builder.ToString());
            _conn.Open();
            VerticaCommand sqlcom = _conn.CreateCommand();
            sqlcom.CommandText = "SELECT node_name FROM current_session";
            var returnValue = sqlcom.ExecuteScalar();
            Console.WriteLine("Connected to node: " +
         returnValue.ToString() + "\n");
            _conn.Close();
            Console.WriteLine("Disconnecting.\n");
         }
         catch (Exception e)
         {
            Console.WriteLine(e.Message);
         }
      }
   }
}
```

Notes

- When native connection load balancing is enabled, the additional servers specified in the BackupServerNode connection parameter are only used for the initial connection to a Vertica host. If host redirects the client to another host in the database cluster to handle its connection request, the second connection does not use the backup node list. This is rarely an issue, since native connection load balancing is aware of which nodes are currently up in the database. See Load balancing in ADO.NET .
- Connections to a host taken from the BackupServerNode list are not pooled for ADO.NET connections.

Querying the database using ADO.NET

This section describes how to create queries to do the following:

- Inserting data into the database
- Read data from the database
- Load data into the database

> **Note**
> The ExecuteNonQuery() method used to query the database returns an int32 with the number of rows affected by the query. The maximum size of an int32 type is a constant and is defined to be 2,147,483,547. If your query returns more results than the int32 max, then ADO.NET throws an exception because of the overflow of the int32 type. However the query is still processed by Vertica even when the reporting of the return value fails. This is a limitation in .NET, as ExecuteNonQuery() is part of the standard ADO.NET interface.

In this section

- Inserting data (ADO.NET)
- Reading data (ADO.Net)
- Loading data through ADO.Net

Inserting data (ADO.NET)

Inserting data can done using the VerticaCommand class. VerticaCommand is an implementation of DbCommand. It allows you to create and send a SQL statement to the database. Use the CommandText method to assign a SQL statement to the command and then execute the SQL by calling the ExecuteNonQuery method. The ExecuteNonQuery method is used for executing statements that do not return result sets.

To insert a single row of data:

1. Create a connection to the database .
2. Create a command object using the connection.
   ```
   VerticaCommand command = _conn.CreateCommand();
   ```
3. Insert data using an INSERT statement. The following is an example of a simple insert. Note that is does not contain a COMMIT statement because the Vertica ADO.NET driver operates in autocommit mode.
   ```
   command.CommandText =
       "INSERT into test values(2, 'username', 'email', 'password')";
   ```
4. Execute the query. The rowsAdded variable contains the number of rows added by the insert statement.
   ```
   Int32 rowsAdded = command.ExecuteNonQuery();
   ```

   The ExecuteNonQuery() method returns the number of rows affected by the command for UPDATE, INSERT, and DELETE statements. For all other types of statements it returns -1. If a rollback occurs then it is also set to -1.

Example usage:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
         builder.Host = "192.168.1.10";
         builder.Database = "VMart";
         builder.User = "dbadmin";
         VerticaConnection _conn = new VerticaConnection(builder.ToString());
         _conn.Open();
      VerticaCommand command = _conn.CreateCommand();
      command.CommandText =
            "INSERT into test values(2, 'username', 'email', 'password')";
      Int32 rowsAdded = command.ExecuteNonQuery();
      Console.WriteLine( rowsAdded + " rows added!");
         _conn.Close();
      }
   }
}
```

In this section

- Using parameters
- Creating and rolling back transactions

Using parameters

You can use parameters to execute similar SQL statements repeatedly and efficiently.

Using parameters

VerticaParameters are an extension of the System.Data.DbParameter base class in ADO.NET and are used to set parameters in commands sent to the server. Use Parameters in all queries (SELECT/INSERT/UPDATE/DELETE) for which the values in the WHERE clause are not static; that is for all queries that have a known set of columns, but whose filter criteria is set dynamically by an application or end user. Using parameters in this way greatly decreases the chances of a SQL injection issue that can occur when simply creating a SQL query from a number of variables.

Parameters require that a valid DbType, VerticaDbType, or System type be assigned to the parameter. See Data types and ADO.NET data types for a mapping of System, Vertica, and DbTypes.

To create a parameter placeholder, place either the at sign (@) or a colon (:) character in front of the parameter name in the actual query string. Do not insert any spaces between the placeholder indicator (@ or :) and the placeholder.

> **Note**
> The @ character is the preferred way to identify parameters. The colon (:) character is supported for backward compatibility.

For example, the following typical query uses the string 'MA' as a filter.

```
SELECT customer_name, customer_address, customer_city, customer_state
FROM customer_dimension WHERE customer_state = 'MA';
```

Instead, the query can be written to use a parameter. In the following example, the string MA is replaced by the parameter placeholder @STATE.

```
SELECT customer_name, customer_address, customer_city, customer_state
FROM customer_dimension WHERE customer_state = @STATE;
```

For example, the ADO.net code for the prior example would be written as:

```
VerticaCommand command = _conn.CreateCommand();
command.CommandText = “SELECT customer_name, customer_address, customer_city, customer_state
   FROM customer_dimension WHERE customer_state = @STATE”;
command.Parameters.Add(new VerticaParameter( “STATE”, VerticaType.VarChar));
command.Parameters["STATE"].Value = "MA";
```

> **Note**
> Although the VerticaCommand class supports a Prepare() method, you do not need to call the Prepare() method for parameterized statements because Vertica automatically prepares the statement for you.

Creating and rolling back transactions

Creating transactions

Transactions in Vertica are atomic, consistent, isolated, and durable. When you connect to a database using the Vertica ADO.NET Driver, the connection is in autocommit mode and each individual query is committed upon execution. You can collect multiple statements into a single transaction and commit them at the same time by using a transaction. You can also choose to rollback a transaction before it is committed if your code determines that a transaction should not commit.

Transactions use the VerticaTransaction object, which is an implementation of DbTransaction. You must associate the transaction with the VerticaCommand object.

The following code uses an explicit transaction to insert one row each into to tables of the VMart schema.

To create a transaction in Vertica using the ADO.NET driver:

1. Create a connection to the database .
2. Create a command object using the connection.
   ```
   VerticaCommand command = _conn.CreateCommand();
   ```
3. Start an explicit transaction, and associate the command with it.
   ```
   VerticaTransaction txn = _conn.BeginTransaction();
   command.Connection = _conn;
   command.Transaction = txn;
   ```
4. Execute the individual SQL statements to add rows.
   ```
   command.CommandText =
      "insert into product_dimension values( ... )";
   command.ExecuteNonQuery();
   command.CommandText =
      "insert into store_orders_fact values( ... )";
   ```
5. Commit the transaction.
   ```
   txn.Commit();
   ```

Rolling back transactions

If your code checks for errors, then you can catch the error and rollback the entire transaction.

```
VerticaTransaction txn = _conn.BeginTransaction();
VerticaCommand command = new
      VerticaCommand("insert into product_dimension values( 838929, 5, 'New item 5' )", _conn);
// execute the insert
command.ExecuteNonQuery();
command.CommandText = "insert into product_dimension values( 838929, 6, 'New item 6' )";
// try insert and catch any errors
bool error = false;
try
{
   command.ExecuteNonQuery();
}
catch (Exception e)
{
   Console.WriteLine(e.Message);
   error = true;
}
if (error)
{
   txn.Rollback();
   Console.WriteLine("Errors. Rolling Back.");
}
else
{
   txn.Commit();
   Console.WriteLine("Queries Successful. Committing.");
}
```

Commit and rollback example

This example details how you can commit or rollback queries during a transaction.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
         builder.Host = "192.168.1.10";
         builder.Database = "VMart";
         builder.User = "dbadmin";
         VerticaConnection _conn = new VerticaConnection(builder.ToString());
         _conn.Open();
         bool error = false;
            VerticaCommand command = _conn.CreateCommand();
            VerticaCommand command2 = _conn.CreateCommand();
            VerticaTransaction txn = _conn.BeginTransaction();
            command.Connection = _conn;
            command.Transaction = txn;
            command.CommandText =
            "insert into test values(1, 'test', 'test', 'test' )";
            Console.WriteLine(command.CommandText);
            try
            {
               command.ExecuteNonQuery();
            }
```

```
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                error = true;
            }
            command.CommandText =
            "insert into test values(2, 'ear', 'eye', 'nose', 'extra' )";
            Console.WriteLine(command.CommandText);
            try
            {
                command.ExecuteNonQuery();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                error = true;
            }
            if (error)
            {
                txn.Rollback();
                Console.WriteLine("Errors. Rolling Back.");
            }
            else
            {
                txn.Commit();
                Console.WriteLine("Queries Successful. Committing.");
            }
        _conn.Close();
    }
  }
}
```

The example displays the following output on the console:

```
insert into test values(1, 'test', 'test', 'test' )
insert into test values(2, 'ear', 'eye', 'nose', 'extra' )
[42601]ERROR: INSERT has more expressions than target columns
Errors. Rolling Back.
```

See also

- Setting the transaction isolation level

In this section

- Setting the transaction isolation level

Setting the transaction isolation level

You can set the transaction isolation level on a per-connection and per-transaction basis. See Transaction for an overview of the transaction isolation levels supported in Vertica. To set the default transaction isolation level for a connection, use the *IsolationLevel* keyword in the VerticaConnectionStringBuilder string (see Connection String Keywords for details). To set the isolation level for an individual transaction, pass the isolation level to the VerticaConnection.BeginTransaction() method call to start the transaction.

To set the isolation level on a connection-basis:

1. Use the VerticaConnectionStringBuilder to build the connection string.
2. Provide a value for the IsolationLevel builder string. It can take one of two values: IsolationLevel.ReadCommited (default) or IsolationLevel.Serializeable. For example:

```
VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
builder.Host = "192.168.1.100";
builder.Database = "VMart";
builder.User = "dbadmin";
builder.IsolationLevel = System.Data.IsolationLevel.Serializeable
VerticaConnection _conn1 = new VerticaConnection(builder.ToString());
_conn1.Open();
```

To set the isolation level on a transaction basis:

1. Set the IsolationLevel on the BeginTransaction method, for example

```
VerticaTransaction txn = _conn.BeginTransaction(IsolationLevel.Serializable);
```

Example usage:

The following example demonstrates:

- getting the connection's transaction isolation level.
- setting the connection's isolation level using connection property.
- setting the transaction isolation level for a new transaction.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
         builder.Host = "192.168.1.10";
         builder.Database = "VMart";
         builder.User = "dbadmin";
         VerticaConnection _conn1 = new VerticaConnection(builder.ToString());
          _conn1.Open();
         VerticaTransaction txn1 = _conn1.BeginTransaction();
         Console.WriteLine("\n Transaction 1 Transaction Isolation Level: " +
          txn1.IsolationLevel.ToString());
         txn1.Rollback();
         VerticaTransaction txn2 = _conn1.BeginTransaction(IsolationLevel.Serializable);
         Console.WriteLine("\n Transaction 2 Transaction Isolation Level: " +
          txn2.IsolationLevel.ToString());
         txn2.Rollback();
         VerticaTransaction txn3 = _conn1.BeginTransaction(IsolationLevel.ReadCommitted);
         Console.WriteLine("\n Transaction 3 Transaction Isolation Level: " +
          txn3.IsolationLevel.ToString());
         _conn1.Close();
      }
   }
}
```

When run, the example code prints the following to the system console:

```
Transaction 1 Transaction Isolation Level: ReadCommitted
Transaction 2 Transaction Isolation Level: Serializable
Transaction 3 Transaction Isolation Level: ReadCommitted
```

Reading data (ADO.Net)

To read data from the database use VerticaDataReader, an implementation of DbDataReader. This implementation is useful for moving large volumes of data quickly off the server where it can be run through analytic applications.

> **Note**
> A VerticaCommand cannot execute anything else while it has an open VerticaDataReader associated with it. To execute something else, close the data reader or use a different VerticaCommand object.

To read data from the database using VerticaDataReader:

1. Create a connection to the database .
2. Create a command object using the connection.
```
VerticaCommand command = _conn.CreateCommand();
```
3. Create a query. This query works with the example VMart database.
```
command.CommandText =
"SELECT fat_content, product_description " +
"FROM (SELECT DISTINCT fat_content, product_description" +
"      FROM product_dimension " +
"      WHERE department_description " +        "      IN ('Dairy') " +
"      ORDER BY fat_content) AS food " +
"LIMIT 10;";
```
4. Execute the reader to return the results from the query. The following command calls the ExecuteReader method of the VerticaCommand object to obtain the VerticaDataReader object.
```
VerticaDataReader dr = command.ExecuteReader();
```
5. Read the data. The data reader returns results in a sequential stream. Therefore, you must read data from tables row-by-row. The following example uses a while loop to accomplish this:
```
Console.WriteLine("\n\n Fat Content\t  Product Description");
   Console.WriteLine("------------\t  -------------------");
   int rows = 0;
   while (dr.Read())
   {
     Console.WriteLine("     " + dr[0] + "    \t  " + dr[1]);
     ++rows;
   }
   Console.WriteLine("------------\n  (" + rows + " rows)\n");
```
6. When you're finished, close the data reader to free up resources.
```
dr.Close();
```

Example usage:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
         builder.Host = "192.168.1.10";
         builder.Database = "VMart";
         builder.User = "dbadmin";
         VerticaConnection _conn = new VerticaConnection(builder.ToString());
         _conn.Open();
      VerticaCommand command = _conn.CreateCommand();
         command.CommandText =
            "SELECT fat_content, product_description " +
            "FROM (SELECT DISTINCT fat_content, product_description" +
            "      FROM product_dimension " +
            "      WHERE department_description " +
            "      IN ('Dairy') " +
            "      ORDER BY fat_content) AS food " +
            "LIMIT 10;";
       VerticaDataReader dr = command.ExecuteReader();

       Console.WriteLine("\n\n Fat Content\t  Product Description");
       Console.WriteLine("------------\t  --------------------");
       int rows = 0;
       while (dr.Read())
       {
            Console.WriteLine("     " + dr[0] + "    \t  " + dr[1]);
            ++rows;
       }
       Console.WriteLine("------------\n  (" + rows + " rows)\n");
           dr.Close();
         _conn.Close();
      }
   }
}
```

Loading data through ADO.Net

This section details the different ways that you can load data in Vertica using the ADO.NET client driver:

- Using the Vertica Data Adapter
- Using batch inserts and prepared statements
- Streaming data via ADO.NET

In this section

- Using the Vertica data adapter
- Using batch inserts and prepared statements
- Streaming data via ADO.NET

Using the Vertica data adapter

The Vertica data adapter (VerticaDataAdapter) enables a client to exchange data between a data set and a Vertica database. It is an implementation of DbDataAdapter. You can use VerticaDataAdapter to simply read data, or, for example, read data from a database into a data set, and then write changed data from the data set back to the database.

Batching updates

When using the Update() method to update a dataset, you can optionally use the UpdateBatchSize() method prior to calling Update() to reduce the number of times the client communicates with the server to perform the update. The default value of UpdateBatchSize is 1. If you have multiple rows.Add() commands for a data set, then you can change the batch size to an optimal size to speed up the operations your client must perform to complete the update.

Reading data from Vertica using the data adapter:

The following example details how to perform a select query on the VMart schema and load the result into a DataTable, then output the contents of the DataTable to the console.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
         builder.Host = "192.168.1.10";
         builder.Database = "VMart";
         builder.User = "dbadmin";
         VerticaConnection _conn = new VerticaConnection(builder.ToString());
         _conn.Open();

         // Try/Catch any exceptions
               try
         {
            using (_conn)
            {
               // Create the command
               VerticaCommand command = _conn.CreateCommand();
               command.CommandText = "select product_key, product_description " +
                  "from product_dimension where product_key < 10";

                  // Associate the command with the connection
                  command.Connection = _conn;

                  // Create the DataAdapter
                  VerticaDataAdapter adapter = new VerticaDataAdapter();
                  adapter.SelectCommand = command;

                  // Fill the DataTable
                  DataTable table = new DataTable();
                  adapter.Fill(table);

                  //  Display each row and column value.
                  int i = 1;
                  foreach (DataRow row in table.Rows)
                  {
                     foreach (DataColumn column in table.Columns)
                     {
                        Console.Write(row[column] + "\t");
                     }
                     Console.WriteLine();
                     i++;
                  }
```

```
                }
                Console.WriteLine(i + " rows returned.");
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        _conn.Close();
    }
  }
}
```

Reading data from Vertica into a data set and changing data:

The following example shows how to use a data adapter to read from and insert into a dimension table of the VMart schema.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
using Vertica.Data.VerticaClient
namespace ConsoleApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
      builder.Host = "192.168.1.10";
      builder.Database = "VMart";
      builder.User = "dbadmin";
      VerticaConnection _conn = new VerticaConnection(builder.ToString());
      _conn.Open();

        // Try/Catch any exceptions
        try
      {
        using (_conn)
        {

            //Create a data adapter object using the connection
            VerticaDataAdapter da = new VerticaDataAdapter();

            //Create a select statement that retrieves data from the table
            da.SelectCommand = new
          VerticaCommand("select * from product_dimension where product_key < 10",
          _conn);
            //Set up the insert command for the data adapter, and bind variables for some of the columns
            da.InsertCommand = new
          VerticaCommand("insert into product_dimension values( :key, :version, :desc )",
          _conn);
        da.InsertCommand.Parameters.Add(new VerticaParameter("key", VerticaType.BigInt));
        da.InsertCommand.Parameters.Add(new VerticaParameter("version", VerticaType.BigInt));
        da.InsertCommand.Parameters.Add(new VerticaParameter("desc", VerticaType.VarChar));
        da.InsertCommand.Parameters[0].SourceColumn = "product_key";
        da.InsertCommand.Parameters[1].SourceColumn = "product_version";
        da.InsertCommand.Parameters[2].SourceColumn = "product_description";
        da.TableMappings.Add("product_key", "product_key");
        da.TableMappings.Add("product_version", "product_version");
```

```
            da.TableMappings.Add("product_description", "product_description");

                //Create and fill a Data set for this dimension table, and get the resulting DataTable.
                DataSet ds = new DataSet();
            da.Fill(ds, 0, 0, "product_dimension");
            DataTable dt = ds.Tables[0];

                //Bind parameters and add two rows to the table.
                DataRow dr = dt.NewRow();
            dr["product_key"] = 838929;
            dr["product_version"] = 5;
            dr["product_description"] = "New item 5";
            dt.Rows.Add(dr);
            dr = dt.NewRow();
            dr["product_key"] = 838929;
            dr["product_version"] = 6;
            dr["product_description"] = "New item 6";
            dt.Rows.Add(dr);
            //Extract the changes for the added rows.
                DataSet ds2 = ds.GetChanges();

                //Send the modifications to the server.
                int updateCount = da.Update(ds2, "product_dimension");

               //Merge the changes into the original Data set, and mark it up to date.
                ds.Merge(ds2);
            ds.AcceptChanges();
            Console.WriteLine(updateCount + " updates made!");
          }
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        _conn.Close();
    }
  }
}
```

Using batch inserts and prepared statements

You can load data in batches using a prepared statement with parameters. You can also use transactions to rollback the batch load if any errors are encountered.

If you are loading large batches of data (more than 100MB), then consider using a direct batch insert.

The following example details using data contained in arrays, parameters, and a transaction to batch load data.

The test table used in the example is created with the command:

```
=> CREATE TABLE test (id INT, username VARCHAR(24), email VARCHAR(64), password VARCHAR(8));
```

Example batch insert using parameters and transactions

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
  class Program
```

```
{
    static void Main(string[] args)
    {
        VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
        builder.Host = "192.168.1.10";
        builder.Database = "VMart";
        builder.User = "dbadmin";
        VerticaConnection _conn = new VerticaConnection(builder.ToString());
        _conn.Open();
        // Create arrays for column data
            int[] ids = {1, 2, 3, 4};
        string[] usernames = {"user1", "user2", "user3", "user4"};
        string[] emails = { "user1@example.com", "user2@example.com","user3@example.com","user4@example.com" };
        string[] passwords = { "pass1", "pass2", "pass3", "pass4" };
        // create counters for accepted and rejected rows
            int rows = 0;
        int rejRows = 0;
        bool error = false;
        // Create the transaction
            VerticaTransaction txn = _conn.BeginTransaction();
        // Create the parameterized query and assign parameter types
            VerticaCommand command = _conn.CreateCommand();
        command.CommandText = "insert into TEST values (@id, @username, @email, @password)";
        command.Parameters.Add(new VerticaParameter("id", VerticaType.BigInt));
        command.Parameters.Add(new VerticaParameter("username", VerticaType.VarChar));
        command.Parameters.Add(new VerticaParameter("email", VerticaType.VarChar));
        command.Parameters.Add(new VerticaParameter("password", VerticaType.VarChar));
        // Prepare the statement
            command.Prepare();

            // Loop through the column arrays and insert the data
            for (int i = 0; i < ids.Length; i++)            {
          command.Parameters["id"].Value = ids[i];
          command.Parameters["username"].Value = usernames[i];
          command.Parameters["email"].Value = emails[i];
          command.Parameters["password"].Value = passwords[i];
          try
          {
              rows += command.ExecuteNonQuery();
          }
          catch (Exception e)
          {
              Console.WriteLine("\nInsert failed - \n  " + e.Message + "\n");
              ++rejRows;
              error = true;
          }
        }
        if (error)
        {
          // Roll back if errors
                Console.WriteLine("Errors. Rolling Back Transaction.");
          Console.WriteLine(rejRows + " rows rejected.");
          txn.Rollback();
        }
        else
        {
          // Commit if no errors
                Console.WriteLine("No Errors. Committing Transaction.");
          txn.Commit();
          Console.WriteLine("Inserted " + rows + " rows. ");
        }
```

```
        _conn.Close();
      }
   }
}
```

Streaming data via ADO.NET

There are two options to stream data from a file on the client to your Vertica database through ADO.NET:

- Use the VerticaCopyStream ADO.NET class to stream data in an object-oriented manner
- Execute a COPY LOCAL SQL statement to stream the data

The topics in this section explain how to use these options.

In this section

- Streaming from the client via VerticaCopyStream
- Using copy with ADO.NET

Streaming from the client via VerticaCopyStream

The VerticaCopyStream class lets you stream data from the client system to a Vertica database. It lets you use the SQL COPY statement directly without having to copy the data to a host in the database cluster first by substituting one or more data stream(s) for STDIN.

Notes:

- Use Transactions and disable auto commit on the copy command for better performance.
- Disable auto commit using the copy command with the 'no commit' modifier. You must explicitly disable commits. Enabling transactions does not disable autocommit when using VerticaCopyStream.
- The copy command used with VerticaCopyStream uses copy syntax.
- VerticaCopyStream.rejects is zeroed every time execute is called. If you want to capture the number of rejects, assign the value of VerticaCopyStream.rejects to another variable before calling execute again.
- You can add multiple streams using multiple AddStream() calls.

Example usage:

The following example demonstrates using VerticaCopyStream to copy a file stream into Vertica.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.IO;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         // Configure connection properties
               VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
            builder.Host = "192.168.1.10";
         builder.Database = "VMart";
         builder.User = "dbadmin";
         //open the connection
         VerticaConnection _conn = new VerticaConnection(builder.ToString());
         _conn.Open();
         try
         {
            using (_conn)
            {
               // Start a transaction
                     VerticaTransaction txn = _conn.BeginTransaction();
```

```
                // Create a table for this example
                VerticaCommand command = new VerticaCommand("DROP TABLE IF EXISTS copy_table", _conn);
        command.ExecuteNonQuery();
           command.CommandText = "CREATE TABLE copy_table (Last_Name char(50), "
                    + "First_Name char(50),Email char(50), "
                    + "Phone_Number char(15))";
           command.ExecuteNonQuery();
           // Create a new filestream from the data file
                string filename = "C:/customers.txt";
         Console.WriteLine("\n\nLoading File: " + filename);
           FileStream inputfile = File.OpenRead(filename);
           // Define the copy command
                string copy = "copy copy_table from stdin record terminator E'\n' delimiter '|'" + " enforcelength "
              + " no commit";
           // Create a new copy stream instance with the connection and copy statement
                VerticaCopyStream vcs = new VerticaCopyStream(_conn, copy);

                // Start the VerticaCopyStream process
                vcs.Start();
                // Add the file stream
                vcs.AddStream(inputfile, false);

                // Execute the copy
                vcs.Execute();

                // Finish stream and write out the list of inserted and rejected rows
                long rowsInserted = vcs.Finish();
        IList<long> rowsRejected = vcs.Rejects;
        // Does not work when rejected or exceptions defined
           Console.WriteLine("Number of Rows inserted: " + rowsInserted);
           Console.WriteLine("Number of Rows rejected: " + rowsRejected.Count);
           if (rowsRejected.Count > 0)
           {
              for (int i = 0; i < rowsRejected.Count; i++)
              {
                 Console.WriteLine("Rejected row #{0} is row {1}", i, rowsRejected[i]);
              }
           }

                // Commit the changes
                txn.Commit();
        }
        }
        catch (Exception e)
        {
           Console.WriteLine(e.Message);
        }


           //close the connection
           _conn.Close();
   }
   }
}
```

Using copy with ADO.NET

To use COPY with ADO.NET, just execute a COPY statement and the path to the source file on the client system. This method is simpler than using the VerticaCopyStream class. However, you may prefer using VerticaCopyStream if you have many files to copy to the database or if your data comes from a source other than a local file (streamed over a network connection, for example).

The following example code demonstrates using COPY to copy a file from the client to the database. It is the same as the code shown in Bulk Loading Using the COPY Statement and the path to the data file is on the client system, rather than on the server.

To load data that is stored on a database node, use a VerticaCommand object to create a COPY command:

1. Create a connection to the database through the node on which the data file is stored.
2. Create a command object using the connection.
   ```
   VerticaCommand command = _conn.CreateCommand();
   ```
3. Copy data. The following is an example of using the COPY command to load data. It uses the LOCAL modifier to copy a file local to the client issuing the command.
   ```
   command.CommandText = "copy lcopy_table from '/home/dbadmin/customers.txt'"
    + " record terminator E'\n' delimiter '|'"
    + " enforcelength ";

   Int32 insertedRows = command.ExecuteNonQuery();
   Console.WriteLine(insertedRows + " inserted.");
   ```

Example usage:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.IO;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         // Configure connection properties
         VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
    builder.Host = "192.168.1.10";
      builder.Database = "VMart";
      builder.User = "dbadmin";

               // Open the connection
                VerticaConnection _conn = new VerticaConnection(builder.ToString());
         _conn.Open();
         try
         {
            using (_conn)
            {

                     // Start a transaction
                     VerticaTransaction txn = _conn.BeginTransaction();

                     // Create a table for this example
                     VerticaCommand command = new VerticaCommand("DROP TABLE IF EXISTS lcopy_table", _conn);
               command.ExecuteNonQuery();
               command.CommandText = "CREATE TABLE IF NOT EXISTS lcopy_table (Last_Name char(50), "
                           + "First_Name char(50),Email char(50), "
                           + "Phone_Number char(15))";
               command.ExecuteNonQuery();
               // Define the copy command
                     command.CommandText = "copy lcopy_table from '/home/dbadmin/customers.txt'"
         + " record terminator E'\n' delimiter '|'"
                  + " enforcelength "
            + " no commit";
                     // Execute the copy
      Int32 insertedRows = command.ExecuteNonQuery();
Console.WriteLine(insertedRows + " inserted.");
                     // Commit the changes
                     txn.Commit();
            }
         }
         catch (Exception e)
         {
            Console.WriteLine("Exception: " + e.Message);
         }


                // Close the connection
                _conn.Close();
      }
   }
}
```

Canceling ADO.NET queries

You can cancel a running vsql query by calling the .Cancel() method of any Command object. The SampleCancelTests class demonstrates how to cancel a query after reading a specified number of rows. It implements the following methods:

- SampleCancelTest() executes the Setup() function to create a test table. Then, it calls RunQueryAndCancel() and RunSecondQuery() to demonstrate how to cancel a query after it reads a specified number of rows. Finally, it runs the Cleanup() function to drop the test table.
- Setup() creates a database for the example queries.
- Cleanup() drops the database.
- RunQueryAndCancel() reads exactly 100 rows from a query that returns more than 100 rows.
- RunSecondQuery() reads all rows from a query.

```
using System;
using Vertica.Data.VerticaClient;

class SampleCancelTests
{
    // Creates a database table, executes a query that cancels during a read loop,
    // executes a query that does not cancel, then drops the test database table.
    // connection: A connection to a Vertica database.

    public static void SampleCancelTest(VerticaConnection connection)
    {
        VerticaCommand command = connection.CreateCommand();

        Setup(command);

        try
        {
            Console.WriteLine("Running query that will cancel after reading 100 rows...");
            RunQueryAndCancel(command);
            Console.WriteLine("Running a second query...");
            RunSecondQuery(command);
            Console.WriteLine("Finished!");
        }
        finally
        {
            Cleanup(command);
        }
    }

    // Set up the database table for the example.
    // command: A Command object used to execute the query.
    private static void Setup(VerticaCommand command)
    {
        // Create table used for test.
        Console.WriteLine("Creating and loading table...");
        command.CommandText = "DROP TABLE IF EXISTS adocanceltest";
        command.ExecuteNonQuery();
        command.CommandText = "CREATE TABLE adocanceltest(id INTEGER, time TIMESTAMP)";
        command.ExecuteNonQuery();
        command.CommandText = @"INSERT INTO adocanceltest
        SELECT row_number() OVER(), slice_time
            FROM(
                SELECT slice_time FROM(
                SELECT '2021-01-01'::timestamp s UNION ALL SELECT '2022-01-01'::timestamp s
                ) sq TIMESERIES slice_time AS '1 second' OVER(ORDER BY s)
            ) sq2";
        command.ExecuteNonQuery();
    }
```

```
// Clean up the database after running the example.
// command: A Command object used to execute the query.
private static void Cleanup(VerticaCommand command)
{
    command.CommandText = "DROP TABLE IF EXISTS adocanceltest";
    command.ExecuteNonQuery();
}

// Execute a query that returns many rows and cancels after reading 100.
// command: A Command object used to execute the query.
private static void RunQueryAndCancel(VerticaCommand command)
{
    command.CommandText = "SELECT COUNT(id) from adocanceltest";
    int fullRowCount = Convert.ToInt32(command.ExecuteScalar());

    command.CommandText = "SELECT id, time FROM adocanceltest";
    VerticaDataReader dr = command.ExecuteReader();
    int nCount = 0;
    try
    {
        while (dr.Read())
        {
            nCount++;
            if (nCount == 100)
            {
                // After reaching 100 rows, cancel the command
                // Note that it is not necessary to read the remaining rows
                command.Cancel();
                return;
            }
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        dr.Close();
        // Verify that the cancel stopped the query
        Console.WriteLine((fullRowCount - nCount) + " rows out of " + fullRowCount + " discarded by cancel");
    }
}

// Execute a simple query and read all results.
// command: A Command object used to execute the query.
private static void RunSecondQuery(VerticaCommand command)
{
    command.CommandText = "SELECT 1 FROM dual";
    VerticaDataReader dr = command.ExecuteReader();
    try
    {
        while (dr.Read())
        {
            ;
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        Console.WriteLine("Warning: no exception should be thrown on query after cancel");
```

```
        }
        finally
        {
            dr.Close();
        }
    }
}
```

Handling messages

You can capture info and warning messages that Vertica provides to the ADO.NET driver by using the InfoMessage event on the VerticaConnection delegate class. This class captures messages that are not severe enough to force an exception to be triggered, but might still provide information that can benefit your application.

To use the VerticaInfoMessageEventHander class:

1. Create a method to handle the message sent from the even handler:
```
static void conn_InfoMessage(object sender, VerticaInfoMessageEventArgs e)
{
    Console.WriteLine(e.SqlState + ": " + e.Message);
}
```
2. Create a connection and register a new VerticaInfoMessageHandler delegate for the InfoMessage event:
```
_conn.InfoMessage += new VerticaInfoMessageEventHandler(conn_InfoMessage);
```
3. Execute your queries. If a message is generated, then the event handle function is run.
4. You can unsubscribe from the event with the following command:
```
_conn.InfoMessage -= new VerticaInfoMessageEventHandler(conn_InfoMessage);
```

Examples

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication {
 class Program {
  // define message handler to deal with messages
  static void conn_InfoMessage(object sender, VerticaInfoMessageEventArgs e) {
   Console.WriteLine(e.SqlState + ": " + e.Message);
  }
  static void Main(string[] args) {
   //configure connection properties
   VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
   builder.Host = "192.168.1.10";
   builder.Database = "VMart";
   builder.User = "dbadmin";

   //open the connection
   VerticaConnection _conn = new VerticaConnection(builder.ToString());
   _conn.Open();

   //create message handler instance by subscribing it to the InfoMessage event of the connection
   _conn.InfoMessage += new VerticaInfoMessageEventHandler(conn_InfoMessage);

   //create and execute the command
   VerticaCommand cmd = _conn.CreateCommand();
   cmd.CommandText = "drop table if exists fakeTable";
   cmd.ExecuteNonQuery();

   //close the connection
   _conn.Close();
  }
 }
}
```

This examples displays the following when run:

```
00000: Nothing was dropped
```

Getting table metadata

You can get the table metadata by using the GetSchema() method on a connection and loading the metadata into a DataTable:

- *database_name* , *schema_name* , and *table_name* can be set to null , a specific name, or use a LIKE pattern.
- *table_type* can be one of:
  - "SYSTEM TABLE"
  - "TABLE"
  - "GLOBAL TEMPORARY"
  - "LOCAL TEMPORARY"
  - "VIEW"
  - null
- If *table_type* is null , then the metadata for all metadata tables is returned.

For example:

```
DataTable table = _conn.GetSchema("Tables", new string[] { null, null, null, "SYSTEM TABLE" });
```

Examples

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         // configure connection properties
         VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
         builder.Host = "192.168.1.10";
         builder.Database = "VMart";
         builder.User = "dbadmin";

         // open the connection
         VerticaConnection _conn = new VerticaConnection(builder.ToString());
         _conn.Open();

         // create a new data table containing the schema
         // the last argument can be "SYSTEM TABLE", "TABLE", "GLOBAL TEMPORARY",
         // "LOCAL TEMPORARY", "VIEW", or null for all types
         DataTable table = _conn.GetSchema("Tables", new string[] { null, null, null, "SYSTEM TABLE" });

         // print out the schema
         foreach (DataRow row in table.Rows) {
            foreach (DataColumn col in table.Columns)
            {
               Console.WriteLine("{0} = {1}", col.ColumnName, row[col]);
            }
            Console.WriteLine("============================");
         }

         //close the connection
         _conn.Close();
      }
   }
}
```

Go

The open-source vertica-sql-go driver lets you interact with your database with Go. For details, see vertica-sql-go .

Java

The Vertica JDBC driver provides you with a standard JDBC API. If you have accessed other databases using JDBC, you should find accessing Vertica familiar. This section explains how to use the JDBC to connect your Java application to Vertica.

Prerequisites

You must install the JDBC client driver before creating Java client applications.

In this section

- JDBC feature support
- Creating and configuring a connection
- JDBC data types
- Executing queries through JDBC
- Canceling JDBC queries
- Loading data through JDBC

- Handling errors
- Routing JDBC queries directly to a single node

JDBC feature support

The Vertica JDBC driver complies with the JDBC 4.0 standards (although it does not implement all of the optional features in them). Your application can use the DatabaseMetaData class to determine if the driver supports a particular feature it wants to use. In addition, the driver implements the Wrapper interface, which lets your client code discover Vertica-specific extensions to the JDBC standard classes, such as VerticaConnection and VerticaStatement classes.

Some important facts to keep in mind when using the Vertica JDBC driver:

- Cursors are forward only and are not scrollable. Result sets cannot be updated.
- A connection supports executing a single statement at any time. If you want to execute multiple statements simultaneously, you must open multiple connections.
- CallableStatement is supported as of the version 12.0.0 of the client driver.

Multiple SQL statement support

The Vertica JDBC driver can execute strings containing multiple statements. For example:

```
stmt.executeUpdate("CREATE TABLE t(a INT);INSERT INTO t VALUES(10);");
```

Only the Statement interface supports executing strings containing multiple SQL statements. You cannot use multiple statement strings with PreparedStatement . COPY statements that copy a file from a host file system work in a multiple statement string. However, client COPY statements (COPY FROM STDIN) do not work.

Multiple batch conversion to COPY statements

The Vertica JDBC driver converts all batch inserts into Vertica COPY statements. If you turn off your JDBC connection's AutoCommit property, the JDBC driver uses a single COPY statement to load data from sequential batch inserts which can improve load performance by reducing overhead. See Batch inserts using JDBC prepared statements for details.

Multiple JDBC version support

The Vertica JDBC driver implements both JDBC 3.0 and JDBC 4.0 compliant interfaces. The interface that the driver returns to your application depends on the JVM version on which it is running. If your application is running on a 5.0 JVM, the driver supplies your application with JDBC 3.0 classes. If your application is running on a 6.0 or later JVM, the driver supplies it with JDBC 4.0 classes.

Multiple active result sets (MARS)

The Vertica JDBC driver supports Multiple active result sets (MARS) . MARS allows the execution of multiple queries on a single connection. While ResultBufferSize sends the results of a query directly to the client, MARS stores the results first on the server. Once query execution has finished and all of the results have been stored, you can make a retrieval request to the server to have rows returned to the client.

Creating and configuring a connection

Before your Java application can interact with Vertica, it must create a connection. Connecting to Vertica using JDBC is similar to connecting to most other databases.

Importing SQL packages

Before creating a connection, you must import the Java SQL packages. A simple way to do so is to import the entire package using a wildcard:

```
import java.sql.*;
```

You may also want to import the Properties class. You can use an instance of this class to pass connection properties when instantiating a connection, rather than encoding everything within the connection string:

```
import java.util.Properties;
```

If your application needs to run in a Java 5 JVM, it uses the older JDBC 3.0-compliant driver. This driver requires that you to manually load the Vertica JDBC driver using the Class.forName() method:

```
// Only required for old JDBC 3.0 driver
try {
   Class.forName("com.vertica.jdbc.Driver");
} catch (ClassNotFoundException e) {
   // Could not find the driver class. Likely an issue
   // with finding the .jar file.
   System.err.println("Could not find the JDBC driver class.");
   e.printStackTrace();
   return; // Exit. Cannot do anything further.
}
```

Your application may run in a Java 6 or later JVM. If so, then the JVM automatically loads the Vertica JDBC 4.0-compatible driver without requiring the call to Class.forName . However, making this call does not adversely affect the process. Thus, if you want your application to be compatible with both Java 5 and Java 6 (or later) JVMs, it can still call Class.forName .

Opening the connection

With SQL packages imported, you are ready to create your connection by calling the DriverManager.getConnection() method. You supply this method with at least the following information:

- The IP address or host name of a node in the database cluster.
  You can provide an IPv4 address, IPv6 address, or host name.
  In mixed IPv4/IPv6 networks, the DNS server configuration determines which IP version address is sent first. Use the PreferredAddressFamily option to force the connection to use either IPv4 or IPv6.
- Port number for the database
- Username of a database user account
- Password of the user (if the user has a password)

The first three parameters are always supplied as part of the *connection string,* a URL that tells the JDBC driver where to find the database. The format of the connection string is (*/databaseName* is optional):

```
jdbc:vertica://VerticaHost:portNumber/databaseName
```

The first portion of the connection string selects the Vertica JDBC driver, followed by the location of the database.

You can provide the last two parameters, username and password, to the JDBC driver, in one of three ways:

- As part of the connection string. The parameters are encoded similarly to URL parameters:
  ```
  "jdbc:vertica://VerticaHost:portNumber/databaseName?user=username&password=password"
  ```
- As separate parameters to DriverManager.getConnection() :
  ```
  Connection conn = DriverManager.getConnection(
      "jdbc:vertica://VerticaHost:portNumber/databaseName",
      "username", "password");
  ```
- In a Properties object:
  ```
  Properties myProp = new Properties();
  myProp.put("user", "username");
  myProp.put("password", "password");
  Connection conn = .getConnection(
    "jdbc:vertica://VerticaHost:portNumber/databaseName", myProp);
  ```

Of these three methods, the Properties object is the most flexible because it makes passing additional connection properties to the getConnection() method easy. See Connection Properties and Setting and getting connection property values for more information about the additional connection properties.

If there is any problem establishing a connection to the database, the getConnection() method throws a SQLException on one of its subclasses. To prevent an exception, enclose the method within a try-catch block, as shown in the following complete example of establishing a connection.

```
import java.sql.*;
import java.util.Properties;

public class VerySimpleVerticaJDBCExample {
    public static void main(String[] args) {
        /*
         * If your client needs to run under a Java 5 JVM, it will use the older
         * JDBC 3.0-compliant driver, which requires you manually load the
         * driver using Class.forname
         */
        /*
         * try { Class.forName("com.vertica.jdbc.Driver"); } catch
         * (ClassNotFoundException e) { // Could not find the driver class.
         * Likely an issue // with finding the .jar file.
         * System.err.println("Could not find the JDBC driver class.");
         * e.printStackTrace(); return; // Bail out. We cannot do anything
         * further. }
         */
        Properties myProp = new Properties();
        myProp.put("user", "dbadmin");
        myProp.put("password", "vertica");
        myProp.put("loginTimeout", "35");
        myProp.put("KeystorePath", "c:/keystore/keystore.jks");
      myProp.put("KeystorePassword", "keypwd");
      myProp.put("TrustStorePath", "c:/truststore/localstore.jks");
      myProp.put("TrustStorePassword", "trustpwd");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                    "jdbc:vertica://V_vmart_node0001.example.com:5433/vmart", myProp);
            System.out.println("Connected!");
            conn.close();
        } catch (SQLTransientConnectionException connException) {
            // There was a potentially temporary network error
            // Could automatically retry a number of times here, but
            // instead just report error and exit.
            System.out.print("Network connection issue: ");
            System.out.print(connException.getMessage());
            System.out.println(" Try again later!");
            return;
        } catch (SQLInvalidAuthorizationSpecException authException) {
            // Either the username or password was wrong
            System.out.print("Could not log into database: ");
            System.out.print(authException.getMessage());
            System.out.println(" Check the login credentials and try again.");
            return;
        } catch (SQLException e) {
            // Catch-all for other exceptions
            e.printStackTrace();
        }
    }
```

Creating a connection with a keystore and truststore

You can create secure connections with your JDBC client driver using a keystore and a truststore. For more information on security within Vertica, refer to Security and authentication .

For examples and instructions on how to generate (or import external) certificates in Vertica, see Generating TLS certificates and keys .

To view your keys and certificates in Vertica, see CERTIFICATES and CRYPTOGRAPHIC_KEYS .

1. Generate your own self-signed certificate or use an existing CA (certificate authority) certificate as the root CA. For information on this process, refer to the Schannel documentation .
2. **Optional** : Generate or import an intermediate CA certificate signed by your root CA. While not required, having an intermediate CA can be useful for testing and debugging your connection.
3. Generate and sign (or import) a server certificate for Vertica.
4. Use ALTER TLS CONFIGURATION to configure Vertica to use client/server TLS for new connections. For more information, see Configuring client-server TLS .

   For **Server Mode** (no client-certificate verification):

   ```
   => ALTER TLS CONFIGURATION server TLSMODE 'ENABLE';
   => ALTER TLS CONFIGURATION server CERTIFICATE server_cert;
   ```

   For **Mutual Mode** (client-certificate verification of varying strictness depending on the TLSMODE):

   ```
   => ALTER TLS CONFIGURATION server TLSMODE 'TRY_VERIFY';
   => ALTER TLS CONFIGURATION server CERTIFICATE server_cert ADD CA CERTIFICATES ca_cert;
   ```

5. Optionally, you can disable all non-SSL connections with CREATE AUTHENTICATION .

   ```
   => CREATE AUTHENTICATION no_tls METHOD 'reject' HOST NO TLS '0.0.0.0/0';
   => CREATE AUTHENTICATION no_tls METHOD 'reject' HOST NO TLS '::/128';
   ```

6. Generate and sign a certificate for your client using the same CA that signed your server certificate.
7. Convert your chain of pem certificates to a single pkcs 12 file.
8. Import the client key and chain into a keystore JKS file from your pkcs12 file. For information on using the keytool command interface, refer to the Java documentation .

   ```
   $ keytool -importkeystore -srckeystore -alias my_alias -srcstoretype PKCS12 -srcstorepass my_password -noprompt -deststorepass my_password -destke
   ```

9. Import the CA into a truststore JKS file.

   ```
   $ keytool -import -file certs/intermediate_ca.pem -alias my_alias -trustcacerts -keystore /tmp/truststore.jks -storepass my_truststore_password -noprompt
   ```

Usage considerations

- When you disconnect a user session, any uncommitted transactions are automatically rolled back.
- If your database is not compliant with your Vertica license terms, Vertica issues a SQLWarning when you establish the connection to the database. You can retrieve this warning using the Connection.getWarnings() method. See Managing licenses for more information about complying with your license terms.

In this section

- JDBC connection properties
- Setting and getting connection property values
- Configuring TLS for JDBC clients
- Setting and returning a client connection label
- Setting the locale for JDBC sessions
- Changing the transaction isolation level
- JDBC connection pools
- Load balancing in JDBC
- JDBC connection failover

JDBC connection properties

You use connection properties to configure the connection between your JDBC client application and your Vertica database. The properties provide the basic information about the connections, such as the server name and port number to use to connect to your database. They also let you tune the performance of your connection and enable logging.

You can set a connection property in one of the following ways:

- Include the property name and value as part of the connection string you pass to the method DriverManager.getConnection() .
- Set the properties in a Properties object, and then pass it to the method DriverManager.getConnection() .
- Use the method VerticaConnection.setProperty() . With this approach, you can change only those connection properties that remain changeable after the connection has been established.

Also, some standard JDBC connection properties have getters and setters on the Connection interface, such as Connection.setAutoCommit() .

Connection properties

The properties in the following table can only be set before you open the connection to the database. Two of them are required for every connection.

Property	Description
BinaryTransfer	Boolean value that determines which mode Vertica uses when connecting to a JDBC client: • **true** : binary transfer (default) • **false** : text transfer Binary transfer is generally more efficient at reading data from a server to a JDBC client and typically requires less bandwidth than text transfer. However, when transferring a large number of small values, binary transfer may use more bandwidth. The data output by both modes is identical with the following exceptions for certain data types: • FLOAT: Binary transfer has slightly better precision. • TIMESTAMPTZ: Binary transfer can fail to get the session time zone and default to the local time zone, while text transfer reliably uses the session time zone. • NUMERIC: Binary transfer is forcibly disabled for NUMERIC data by the server for Vertica 11.0.2+.
ConnSettings	A string containing SQL statements that the JDBC driver automatically runs after it connects to the database. You can use this property to set the locale or schema search path, or perform other configuration that the connection requires.
Label	Sets a label for the connection on the server. This value appears in the client_label column of the SESSIONS system table. **Default:** jdbc- *driver-version - random_number*
SSL	When set to true, use SSL to encrypt the connection to the server. Vertica must be configured to handle SSL connections before you can establish an SSL-encrypted connection to it. See TLS protocol . This property has been deprecated in favor of the TLSmode property. **Default:** false
TLSmode	TLSmode identifies the security level that Vertica applies to the JDBC connection. Vertica must be configured to handle TLS connections before you can establish an encrypted connection to it. See TLS protocol for details. Valid values are: • disable : JDBC connects using plain text and implements no security measures. • require : JDBC connects using TLS without verifying the CA certificate. • verify-ca : JDBC connects using TLS and confirms that the server certificate has been signed by the certificate authority. This setting is equivalent to the deprecated ssl=true property. • verify-full : JDBC connects using TLS, confirms that the server certificate has been signed by the certificate authority, and verifies that the host name matches the name provided in the server certificate. If this property and the SSL property are set, this property takes precedence. **Default:** disable
HostnameVerifier	If TLSmode is set to verify-full, this property the fully qualified domain name of the verifier that you want to confirm the host name.
Password	Required (for non-OAuth connections), the password to use to log into the database.
User	Required (for non-OAuth connections), the database user name to use to connect to the database.
ConnectionLoadBalance	A Boolean value indicating whether the client is willing to have its connection redirected to another host in the Vertica database. This setting has an effect only if the server has also enabled connection load balancing. See About native connection load balancing for more information about native connection load balancing. **Default:** false

BackupServerNode	A string containing the host name or IP address of one or more hosts in the database. If the connection to the host specified in the connection string times out, the client attempts to connect to any host named in this string.The host name or IP address can also include a colon followed by the port number for the database. If no port number is specified, the client uses the standard port number (5433) . Separate multiple host name or IP address entries with commas.
PreferredAddressFamily	The IP version to use if the client and server have both IPv4 and IPv6 addresses and you have provided a host name, one of the following: • ipv4 : Connect to the server using IPv4. • ipv6 : Connect to the server using IPv6. • none : Use the IP address provided by the DNS server. **Default:** none
KeyStorePath	The path to a .JKS file containing your private keys and their corresponding certificate chains. For information on creating a keystore, refer to documentation for your development environment. For information on creating a keystore, refer to the Java documentation .
KeyStorePassword	The password protecting the keystore file. If individual keys are also encrypted, the keystore file password must match the password for a key within the keystore.
TrustStorePath	The path to a .JKS truststore file containing certificates from authorities you trust.
TrustStorePassword	The password protecting the truststore file.

OAuth connection properties

The following connection properties pertain to OAuth in JDBC.

Property	**Description**
oauthaccesstoken	Required if oauthrefreshtoken is unspecified, an OAuth token that authorizes a user to the database.
oauthrefreshtoken	Required if oauthaccesstoken is unspecified, allows a user to refresh and obtain a new oauthaccesstoken when their old one expires. If you set this parameter, you must also set the following refresh properties in oauthjsonconfig: • oauthrefreshtoken • oauthdiscoveryurl or oauthtokenurl • oauthclientid • oauthclientsecret In cases where introspection fails (e.g. when the access token expires), Vertica responds to the request with an error. If introspection fails and OAuthRefreshToken is specified, the driver attempts to refresh and silently retrieve a new access token. Otherwise, the driver passes error to the client application.

oauthjsonconfig	A JSON string or file that lets you set the following:
	• oauthclientid : The client ID of the client application registered in the identity provider.
	• oauthclientsecret : The client secret of the client application registered in the identity provider.
	• oauthtokenurl : The endpoint to which token refresh requests are sent. The format for this depends on your provider. For examples, see the Keycloak and Okta documentation.
	If oauthdiscoveryurl is not set: this parameter is only required for IDP valdiation if oauthdiscoveryurl , but should also be set for JWT validation to support token refresh.
	• oauthauthurl : The authorization endpoint used for single sign-on. For examples, see the Keycloak and Okta documentation.
	• oauthdiscoveryurl : Also known as the OpenID Provider Configuration Document , this endpoint contains a list of all other endpoints supported by the IDP. If set, the other endpoints (such as oauthtokenurl and oauthauthurl) do not need to be specified. This parameter is only supported for Keycloak. For other identity providers like Okta, the endpoints must be set manually.
	If you set both oauthdiscoveryurl and another endpoint (like oauthtokenurl), oauthdiscoveryurl takes precedence. This parameter is only required for IDP valdiation , but should also be set for JWT validation to support token refresh.
	• oauthscope: The requested OAuth scopes, delimited with spaces. These scopes define the extent of access to the resource server (in this case, Vertica) granted to the client by the access token. For details, see the OAuth documentation .
	• oauthvalidatehostname: Boolean, whether to verify the subjectAltName of the identity provider host. If enabled, the IP address or hostname must be set as the subjectAltName in its certificate. Hostname verification is enabled by default.
	
	Unlike oauthaccesstoken or oauthrefreshtoken, which must be set programmatically by the client when they attempt to connect, the same oauthjsonconfig can be reused between connections to the database.
	
	For example, to set it as a connection property:
	`Properties myProp new Properties();`
	`String oauthrefreshparams = "{`
	`  \"oauthdiscoveryurl\": \"http://203.0.113.1:8443/realms/myrealm/.well-known/openid-configuration",`
	`  \"oauthclientid\": \"vertica\",`
	`  \"oauthclientsecret\": \"eba23135-834f-1341-aa34-bf9345713dfc\",`
	`  \"oauthscope\": \"offline_access openid\",`
	`  \"oauthvalidatehostname\": \"false\"`
	`}"`
	`myProp.put.setProperty("oauthjsonconfig", oauthrefreshparams);`

oauthtruststorepath	The path to a custom truststore. If unspecified, JDBC uses the default system truststore.
oauthtruststorepassword	The password to the truststore.

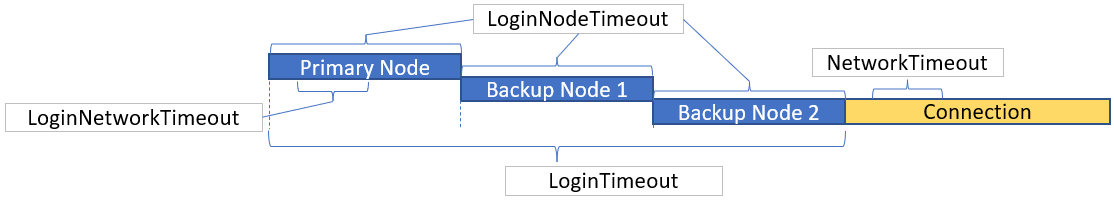
Timeout properties

With the following parameters, you can specify various timeouts for each step and the overall connection of JDBC to your Vertica database.

Property	**Description**
LoginTimeout	The number of seconds Vertica waits for the client to log in to the database before throwing a SQLException . **Default:** 0 (no timeout)

LoginNodeTimeout	The number of seconds the JDBC client waits before attempting to connect to the next node if the Vertica process is running, but does not respond. The "next" node is determined by the either the BackupServerNode connection property or DNS resolution. If you only provide a single IP address, the JDBC client returns an error. A timeout value of 0 instructs JDBC to wait indefinitely for an error/a successful connection rather than attempt to connect to another node. **Default:** 0 (no timeout)
LoginNetworkTimeout	The number of seconds the JDBC client has to establish a TCP connection to a Vertica node. A typical use case for this property is to let JDBC connect to another node if the system is down for maintenance and modifying the JDBC application's connection string is infeasible. **Default:** 0 (no timeout)
NetworkTimeout	The number of milliseconds for the server to reply to a request after the client has established a connection with the database. **Default:** 0

The relationship between these properties and the role they play when JDBC attempts to connect to a Vertica database is illustrated in the following diagram:



General properties

The following properties can be set after the connection is established. None of these properties are required.

Property	**Description**
AutoCommit	Controls whether the connection automatically commits transactions. Set this parameter to false to prevent the connection from automatically committing its transactions. You often want to do this when you are bulk loading multiple batches of data and you want the ability to roll back all of the loads if an error occurs. **Set After Connection:** Connection.setAutoCommit() **Default:** true
DirectBatchInsert	Deprecated, always set to true.
DisableCopyLocal	When set to true, disables file-based COPY LOCAL operations, including copying data from local files and using local files to store data and exceptions. You can use this property to prevent users from writing to and copying from files on a Vertica host, including an MC host. **Default:** false

MultipleActiveResultSets	Allows more than one active result set on a single connection via MultipleActiveResultSets (MARS). If both MultipleActiveResultSets and ResultBufferSize are turned on, MultipleActiveResultSets takes precedence. The connection does not provide an error, however ResultBufferSize is ignored. **Set After Connection:** VerticaConnection.setProperty() **Default:** false
ReadOnly	When set to true, makes the data connection read-only. Any queries attempting to update the database using a read-only connection cause a SQLException . **Set After Connection:** Connection.setReadOnly() **Default:** false
ResultBufferSize	Sets the size of the buffer the Vertica JDBC driver uses to temporarily store result sets. A value of 0 means ResultBufferSize is turned off. **Note:** This property was named maxLRSMemory in previous versions of the Vertica JDBC driver. **Set After Connection:** VerticaConnection.setProperty() **Default:** 8912 (8KB)
SearchPath	Sets the schema search path for the connection. This value is a string containing a comma-separated list of schema names. See Setting Search Paths for more information on the schema search path. **Set After Connection:** VerticaConnection.setProperty() **Default:** "$user", public, v_catalog, v_monitor, v_internal
ThreePartNaming	A Boolean value that controls how DatabaseMetaData reports the catalog name. When set to true, the database name is returned as the catalog name in the database metadata. When set to false, NULL is returned as the catalog name. Enable this option if your client software is set up to get the catalog name from the database metadata for use in a three-part name reference. **Set After Connection:** VerticaConnection.setProperty() **Default:** true
TransactionIsolation	Sets the isolation level of the transactions that use the connection. See Changing the transaction isolation level for details. **Note:** In previous versions of the Vertica JDBC driver, this property was only available using a getter and setter on the PGConnection object. You can now set it in the same way as other connection properties. **Set After Connection:** Connection.setTransactionIsolation() **Default:** TRANSACTION_READ_COMMITTED

Logging properties

The properties that control client logging must be set before the connection is opened. None of these properties are required, and none can be changed after the Connection object has been instantiated.

Property	Description
LogLevel	Sets the type of information logged by the JDBC driver. The value is set to one of the following values: • "DEBUG" • "ERROR" • "TRACE" • "WARNING" • "INFO" • "OFF" **Default:** "OFF"
LogNameSpace	Restricts logging to just messages generated by a specific packages. Valid values are: • com.vertica — All messages generated by the JDBC driver • com.vertica.jdbc — All messages generated by the top-level JDBC API • com.vertica.jdbc.kv — A ll messages generated by the JDBC KV API) • com.vertica.jdbc.core — Connection and statement settings • com.vertica.jdbc.io — Client/server protocol messages • com.vertica.jdbc.util — Miscellaneous utilities • com.vertica.jdbc.dataengine — Query execution and result set iteration • com.vertica.dataengine — Query execution and result set iteration
LogPath	The path for the log file. **Default:** The current working directory

Kerberos connection parameters

Use the following parameters to set the service and host name principals for client authentication using Kerberos.

Parameters	Description
JAASConfigName	Provides the name of the JAAS configuration that contains the JAAS Krb5LoginModule and its settings **Default:** verticajdbc
KerberosServiceName	Provides the service name portion of the Vertica Kerberos principal, for example: vertichost@EXAMPLE.COM **Default:** vertica
KerberosHostname	Provides the instance or host name portion of the Vertica Kerberos principal, for example: verticaosEXAMPLE.COM **Default:** Value specified in the servername connection string property

Routable connection API connection parameters

Use the following parameters to set properties to enable and configure the connection for Routable Connection lookups.

Parameters	Description
EnableRoutableQueries	Enables Routable Connection lookup. See Routing JDBC queries directly to a single node **Default:** false

FailOnMultiNodePlans	If the query plan requires more than one node, then the query fails. Only applicable when EnableRoutableQueries = true. **Default:** true
MetadataCacheLifetime	The time in seconds to keep projection metadata. Only applicable when EnableRoutableQueries = true. **Default:**
MaxPooledConnections	Cluster-wide maximum number of connections to keep in the VerticaRoutableConnection's internal pool. Only applicable when EnableRoutableQueries = true. **Default:** 20
MaxPooledConnections PerNode	Per-node maximum number of connections to keep in the VerticaRoutableConnection's internal pool. Only applicable when EnableRoutableQueries = true. **Default:** 5

> **Note**
> You can also use VerticaConnection.setProperty() method to set properties that have standard JDBC Connection setters, such as AutoCommit.

For information about manipulating these attributes, see Setting and getting connection property values .

Setting and getting connection property values

You can set a connection property in one of the following ways:

- Include the property name and value as part of the connection string you pass to the method DriverManager.getConnection() .
- Set the properties in a Properties object, and then pass it to the method DriverManager.getConnection() .
- Use the method VerticaConnection.setProperty() . With this approach, you can change only those connection properties that remain changeable after the connection has been established.

Also, some standard JDBC connection properties have getters and setters on the Connection interface, such as Connection.setAutoCommit() .

Setting properties when connecting

When creating a connection to Vertica, you can set connection properties by:

- Specifying them in the connection string.
- Modifying the Properties object passed to getConnection() .

Connection string properties

You can specify connection properties in the connection string with the same URL parameter format used for usernames and passwords. For example, the following string enables a TLS connection:

```
"jdbc:vertica://VerticaHost:5433/db?user=UserName&password=Password&TLSmode=require"
```

Setting a host name using the setProperty() method overrides the host name set in a connection string. If this occurs, Vertica might not be able to connect to a host. For example, using the connection string above, the following overrides the VerticaHost name:

```
Properties props = new Properties();
props.setProperty("dataSource", dataSourceURL);
props.setProperty("database", database);
props.setProperty("user", user);
props.setProperty("password", password);
ps.setProperty("jdbcDriver", jdbcDriver);
props.setProperty("hostName", "NonVertica_host");
```

However, if a new connection or override connection is needed, you can enter a valid host name in the hostname properties object.

The NonVertica_host hostname overrides VerticaHost name in the connection string. To avoid this issue, comment out the props.setProperty("hostName", "NonVertica_host"); line:

```
//props.setProperty("hostName", "NonVertica_host");
```

Properties object

To set connection properties with the Properties object passed to the getConnection() call:

1. Import the java.util.Properties class to instantiate a Properties object.
2. Use the put() method to add name-value pairs to the object.

```
Properties myProp = new Properties();
myProp.put("user", "ExampleUser");
myProp.put("password", "password123");
myProp.put("LoginTimeout", "35");
Connection conn;
try {
   conn = DriverManager.getConnection(
      "jdbc:vertica://VerticaHost:/ExampleDB", myProp);
} catch (SQLException e) {
   e.printStackTrace();
}
```

> **Note**
> The data type of all of the values you set in the Properties object are strings, regardless of the property value's data type.

Getting and setting properties after connecting

After you establish a connection with Vertica, you can use the VerticaConnection methods getProperty() and setProperty() to set the values of some connection properties, respectively.

The VerticaConnection.getProperty() method lets you get the value of some connection properties. Use this method to change the value for properties that can be set after you establish a connection with Vertica.

Because these methods are Vertica-specific, you must cast your Connection object to the VerticaConnection interface with one of the following methods:

- Import the Connection object into your client application.
- Use a fully-qualified reference: com.vertica.jdbc.VerticaConnection .

The following example demonstrates getting and setting the value of the ReadOnly property .

```
import java.sql.*;
import java.util.Properties;
import com.vertica.jdbc.*;

public class SetConnectionProperties {
   public static void main(String[] args) {
      // Note: If your application needs to run under Java 5, you need to
      // load the JDBC driver using Class.forName() here.
      Properties myProp = new Properties();
      myProp.put("user", "ExampleUser");
      myProp.put("password", "password123");
      // Set ReadOnly to true initially
      myProp.put("ReadOnly", "true");
      Connection conn;
      try {
         conn = DriverManager.getConnection(
                     "jdbc:vertica://VerticaHost:5433/ExampleDB",
                     myProp);
         // Show state of the ReadOnly property. This was set at the
         // time the connection was created.
         System.out.println("ReadOnly state: "
                     + ((VerticaConnection) conn).getProperty(
                           "ReadOnly"));

         // Change it and show it again
         ((VerticaConnection) conn).setProperty("ReadOnly", false);
         System.out.println("ReadOnly state is now: " +
                     ((VerticaConnection) conn).getProperty(
                           "ReadOnly"));
         conn.close();
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
```

When run, the example prints the following on the standard output:

```
ReadOnly state: true
ReadOnly state is now: false
```

Configuring TLS for JDBC clients

To configure TLS for JDBC clients:

- Configure Vertica for client-server TLS .
- Set the keystore and truststore properties.
- Set the TLSmode parameter.
- (Optional) Run the SSL debug utility to test your configuration.

Setting keystore/truststore properties

You can set the keystore and truststore properties in the following ways, each with their own pros and cons:

- At the driver level.
- At the JVM level.

Driver-level configuration

If you use tools like DbVizualizer with many connections, configure the keystore and truststore with the JDBC connection properties . This does, however, expose these values in the connection string:

- KeyStorePath
- KeyStorePassword
- TrustStorePath

- TrustStorePassword

For example:

```
Properties props = new Properties();
props.setProperty("KeyStorePath", keystorepath);
props.setProperty("KeyStorePassword", keystorepassword);
props.setProperty("TrustStorePath", truststorepath);
props.setProperty("TrustStorePassword", truststorepassword);
```

JVM-level configuration

Setting keystore and truststore parameters at the JVM level excludes them from the connection string, which may be more accommodating for environments with more stringent security requirements:

- javax.net.ssl.keyStore
- javax.net.ssl.trustStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.trustStorePassword

For example:

```
System.setProperty("javax.net.ssl.keyStore","clientKeyStore.key");
System.setProperty("javax.net.ssl.trustStore","clientTrustStore.key");
System.setProperty("javax.net.ssl.keyStorePassword","new_keystore_password")
System.setProperty("javax.net.ssl.trustStorePassword","new_truststore_password");
```

Set the TLSmode connection property

You can set the TLSmode connection property to determine how certificates are handled. TLSmode is disabled by default.

TLSmode identifies the security level that Vertica applies to the JDBC connection. Vertica must be configured to handle TLS connections before you can establish an encrypted connection to it. See TLS protocol for details. Valid values are:

- disable : JDBC connects using plain text and implements no security measures.
- require : JDBC connects using TLS without verifying the CA certificate.
- verify-ca : JDBC connects using TLS and confirms that the server certificate has been signed by the certificate authority. This setting is equivalent to the deprecated ssl=true property.
- verify-full : JDBC connects using TLS, confirms that the server certificate has been signed by the certificate authority, and verifies that the host name matches the name provided in the server certificate.

If this property and the SSL property are set, this property takes precedence.

For example, to configure JDBC to connect to the server with TLS without verifying the CA certificate, you can set the TLSmode property to 'require' with the method VerticaConnection.setProperty() :

```
Properties props = new Properties();
props.setProperty("TLSmode", "verify-full");
```

Run the SSL debug utility

After configuring TLS, you can run the following for a debugging utility:

```
$ java -Djavax.net.debug=ssl
```

You can use several debug specifiers (options) with the debug utility. The specifiers help narrow the scope of the debugging information that is returned. For example, you could specify one of the options that prints handshake messages or session activity.

For information on the debug utility and its options, see Debugging Utilities in the Oracle document, JSSE Reference Guide .

For information on interpreting debug information, refer to the Oracle document, Debugging SSL/TLS Connections .

Setting and returning a client connection label

The JDBC Client has a method to set and return the client connection label: getClientInfo() and setClientInfo(). You can use these methods with the SQL Functions GET_CLIENT_LABEL and SET_CLIENT_LABEL .

When you use these two methods, make sure you pass the string value APPLICATIONNAME to both the setter and getter methods.

Use setClientInfo() to create a client label, and use getClientInfo() to return the client label:

```
import java.sql.*;
import java.util.Properties;

public class ClientLabelJDBC {

   public static void main(String[] args) {
      Properties myProp = new Properties();
      myProp.put("user", "dbadmin");
      myProp.put("password", "");
      myProp.put("loginTimeout", "35");
      Connection conn;
      try {
         conn = DriverManager.getConnection(
               "jdbc:vertica://example.com:5433/mydb", myProp);
         System.out.println("Connected!");
         conn.setClientInfo("APPLICATIONNAME", "JDBC Client - Data Load");
         System.out.println("New Conn label: " + conn.getClientInfo("APPLICATIONNAME"));
         conn.close();
      } catch (SQLTransientConnectionException connException) {
         // There was a potentially temporary network error
         // Could automatically retry a number of times here, but
         // instead just report error and exit.
         System.out.print("Network connection issue: ");
         System.out.print(connException.getMessage());
         System.out.println(" Try again later!");
         return;
      } catch (SQLInvalidAuthorizationSpecException authException) {
         // Either the username or password was wrong
         System.out.print("Could not log into database: ");
         System.out.print(authException.getMessage());
         System.out.println(" Check the login credentials and try again.");
         return;
      } catch (SQLException e) {
         // Catch-all for other exceptions
         e.printStackTrace();
      }
   }
```

When you run this method, it prints the following result to the standard output:

```
Connected!
New Conn Label: JDBC Client - Data Load
```

Setting the locale for JDBC sessions

You set the locale for a connection while opening it by including a SET LOCALE statement in the ConnSettings property, or by executing a SET LOCALE statement at any time after opening the connection. Changing the locale of a Connection object affects all of the Statement objects you instantiated using it.

You can get the locale by executing a SHOW LOCALE query. The following example demonstrates setting the locale using ConnSettings and executing a statement, as well as getting the locale:

```
import java.sql.*;
import java.util.Properties;

public class GetAndSetLocale {
    public static void main(String[] args) {

    // If running under a Java 5 JVM, you need to load the JDBC driver
    // using Class.forname here

    Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");

        // Set Locale to true en_GB on connection. After the connection
        // is established, the JDBC driver runs the statements in the
        // ConnSettings property.
        myProp.put("ConnSettings", "SET LOCALE TO en_GB");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                        "jdbc:vertica://VerticaHost:5433/ExampleDB",
                        myProp);

            // Execute a query to get the locale. The results should
            // show "en_GB" as the locale, since it was set by the
            // conn settings property.
            Statement stmt = conn.createStatement();
            ResultSet rs = null;
            rs = stmt.executeQuery("SHOW LOCALE");
            System.out.print("Query reports that Locale is set to: ");
            while (rs.next()) {
                System.out.println(rs.getString(2).trim());
            }

            // Now execute a query to set locale.
            stmt.execute("SET LOCALE TO en_US");

            // Run query again to get locale.
            rs = stmt.executeQuery("SHOW LOCALE");
            System.out.print("Query now reports that Locale is set to: ");
            while (rs.next()) {
                System.out.println(rs.getString(2).trim());
            }
            // Clean up
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

Running the above example displays the following on the system console:

```
Query reports that Locale is set to: en_GB (LEN)
Query now reports that Locale is set to: en_US (LEN)
```

Notes:

- JDBC applications use a UTF-16 character set encoding and are responsible for converting any non-UTF-16 encoded data to UTF-16. Failing to convert the data can result in errors or the data being stored incorrectly.
- The JDBC driver converts UTF-16 data to UTF-8 when passing to the Vertica server and converts data sent by Vertica server from UTF-8 to UTF-16 .

Changing the transaction isolation level

Changing the transaction isolation level lets you choose how transactions prevent interference from other transactions. By default, the JDBC driver matches the transaction isolation level of the Vertica server. The Vertica default transaction isolation level is READ_COMMITTED , which means any changes made by a transaction cannot be read by any other transaction until after they are committed. This prevents a transaction from reading data inserted by another transaction that is later rolled back.

Vertica also supports the SERIALIZABLE transaction isolation level. This level locks tables to prevent queries from having the results of their WHERE clauses changed by other transactions. Locking tables can have a performance impact, since only one transaction is able to access the table at a time.

A transaction retains its isolation level until it completes, even if the session's isolation level changes during the transaction. Vertica internal processes (such as the Tuple Mover and refresh operations) and DDL operations always run at the SERIALIZABLE isolation level to ensure consistency.

You can change the transaction isolation level connection property after the connection has been established using the Connection object's setter (setTransactionIsolation()) and getter (getTransactionIsolation()). The value for transaction isolation property is an integer. The Connection interface defines constants to help you set the value in a more intuitive manner:

Constant	Value
Connection.TRANSACTION_READ_COMMITTED	2
Connection.TRANSACTION_SERIALIZABLE	8

Note
The Connection interface also defines several other transaction isolation constants (READ_UNCOMMITTED and REPEATABLE_READ). Since Vertica does not support these isolation levels, they are converted to READ_COMMITTED and SERIALIZABLE , respectively.

The following example demonstrates setting the transaction isolation level to SERIALIZABLE.

```
import java.sql.*;
import java.util.Properties;

public class SetTransactionIsolation {
  public static void main(String[] args) {
    Properties myProp = new Properties();
    myProp.put("user", "ExampleUser");
    myProp.put("password", "password123");
    Connection conn;
    try {
      conn = DriverManager.getConnection(
                "jdbc:vertica://VerticaHost:5433/ExampleDB",
                myProp);
      // Get default transaction isolation
      System.out.println("Transaction Isolation Level: "
                + conn.getTransactionIsolation());
      // Set transaction isolation to SERIALIZABLE
      conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
      // Get the transaction isolation again
      System.out.println("Transaction Isolation Level: "
                + conn.getTransactionIsolation());
      conn.close();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
```

Running the example results in the following being printed out to the console:

```
Transaction Isolation Level: 2Transaction Isolation Level: 8
```

JDBC connection pools

A pooling data source uses a collection of persistent connections in order to reduce the overhead of repeatedly opening network connections between the client and server. Opening a new connection for each request is more costly for both the server and the client than keeping a small pool of connections open constantly, ready to be used by new requests. When a request comes in, one of the pre-existing connections in the pool is assigned to it. Only if there are no free connections in the pool is a new connection created. Once the request is complete, the connection returns to the pool and waits to service another request.

The Vertica JDBC driver supports connection pooling as defined in the JDBC 4.0 standard. If you are using a J2EE-based application server in conjunction with Vertica, it should already have a built-in data pooling feature. All that is required is that the application server work with the PooledConnection interface implemented by Vertica's JDBC driver. An application server's pooling feature is usually well-tuned for the works loads that the server is designed to handle. See your application server's documentation for details on how to work with pooled connections. Normally, using pooled connections should be transparent in your code—you will just open connections and the application server will worry about the details of pooling them.

If you are not using an application server, or your application server does not offer connection pooling that is compatible with Vertica, you can use a third-party pooling library, such as the open-source c3p0 or DBCP libraries, to implement connection pooling.

> **Note**
> The Vertica Analytic Database client driver's native connection load balancing feature works with third-party connection pooling supplied by application servers and third-party pooling libraries. See Load balancing in JDBC for more information.

Load balancing in JDBC

Native connection load balancing

Native connection load balancing helps spread the overhead caused by client connections on the hosts in the Vertica database. Both the server and the client must enable native connection load balancing. If enabled by both, then when the client initially connects to a host in the database, the host picks a host to handle the client connection from a list of the currently up hosts in the database, and informs the client which host it has chosen.

If the initially-contacted host does not choose itself to handle the connection, the client disconnects, then opens a second connection to the host selected by the first host. The connection process to this second host proceeds as usual—if SSL is enabled, then SSL negotiations begin, otherwise the client begins the authentication process. See About native connection load balancing for details.

To enable native load balancing on your client, set the ConnectionLoadBalance connection parameter to true. The following example demonstrates:

- Connecting to the database several times with native connection load balancing enabled.
- Fetching the name of the node handling the connection from the V_MONITOR. CURRENT_SESSION system table.

```
import java.sql.*;
import java.util.Properties;
import java.sql.*;
import java.util.Properties;

public class JDBCLoadingBalanceExample {
    public static void main(String[] args) {
        Properties myProp = new Properties();
        myProp.put("user", "dbadmin");
        myProp.put("password", "example_password123");
        myProp.put("loginTimeout", "35");
        myProp.put("ConnectionLoadBalance", "1");
        Connection conn;

        for (int x = 1; x <= 4; x++) {
            try {
                System.out.print("Connect attempt #" + x + "...");
                conn = DriverManager.getConnection(
                    "jdbc:vertica://node01.example.com:5433/vmart", myProp);
                Statement stmt = conn.createStatement();
                // Set the load balance policy to round robin before testing the database's load balancing.
                stmt.execute("SELECT SET_LOAD_BALANCE_POLICY('ROUNDROBIN');");
                // Query system to table to see what node we are connected to. Assume a single row
                // in response set.
                ResultSet rs = stmt.executeQuery("SELECT node_name FROM v_monitor.current_session;");
                rs.next();
                System.out.println("Connected to node " + rs.getString(1).trim());
                conn.close();
            } catch (SQLTransientConnectionException connException) {
                // There was a potentially temporary network error
                // Could automatically retry a number of times here, but
                // instead just report error and exit.
                System.out.print("Network connection issue: ");
                System.out.print(connException.getMessage());
                System.out.println(" Try again later!");
                return;
            } catch (SQLInvalidAuthorizationSpecException authException) {
                // Either the username or password was wrong
                System.out.print("Could not log into database: ");
                System.out.print(authException.getMessage());
                System.out.println(" Check the login credentials and try again.");
                return;
            } catch (SQLException e) {
                // Catch-all for other exceptions
                e.printStackTrace();
            }
        }
    }
```

Running the previous example produces the following output:

```
Connect attempt #1...Connected to node v_vmart_node0002
Connect attempt #2...Connected to node v_vmart_node0003
Connect attempt #3...Connected to node v_vmart_node0001
Connect attempt #4...Connected to node v_vmart_node0002
```

Hostname-based load balancing

You can load balance workloads by resolving a single hostname to multiple IP addresses. When you specify the hostname for the DriverManager.getConnection() method, the hostname resolves to a random listed IP address from the each connection.

For example, the hostname verticahost.example.com has the following entries in etc/hosts :

```
192.0.2.0 verticahost.example.com
192.0.2.1 verticahost.example.com
192.0.2.2 verticahost.example.com
```

Specifying verticahost.example.com as the connection for DriverManager.getConnection() randomly resolves to one of the listed IP address.

JDBC connection failover

If a client application attempts to connect to a host in the Vertica cluster that is down, the connection attempt fails when using the default connection configuration. This failure usually returns an error to the user. The user must either wait until the host recovers and retry the connection or manually edit the connection settings to choose another host.

Due to Vertica Analytic Database's distributed architecture, you usually do not care which database host handles a client application's connection. You can use the client driver's connection failover feature to prevent the user from getting connection errors when the host specified in the connection settings is unreachable. The JDBC driver gives you several ways to let the client driver automatically attempt to connect to a different host if the one specified in the connection parameters is unreachable:

- Configure your DNS server to return multiple IP addresses for a host name. When you use this host name in the connection settings, the client attempts to connect to the first IP address from the DNS lookup. If the host at that IP address is unreachable, the client tries to connect to the second IP, and so on until it either manages to connect to a host or it runs out of IP addresses.
- Supply a list of backup hosts for the client driver to try if the primary host you specify in the connection parameters is unreachable.
- (JDBC only) Use driver-specific connection properties to manage timeouts before attempting to connect to the next node.

For all methods, the process of failover is transparent to the client application (other than specifying the list of backup hosts, if you choose to use the list method of failover). If the primary host is unreachable, the client driver automatically tries to connect to other hosts.

Failover only applies to the initial establishment of the client connection. If the connection breaks, the driver does not automatically try to reconnect to another host in the database.

Choosing a failover method

You usually choose to use one of the two failover methods. However, they do work together. If your DNS server returns multiple IP addresses and you supply a list of backup hosts, the client first tries all of the IPs returned by the DNS server, then the hosts in the backup list.

> **Note**
> If a host name in the backup host list resolves to multiple IP addresses, the client does not try all of them. It just tries the first IP address in the list.

The DNS method of failover centralizes the configuration client failover. As you add new nodes to your Vertica Analytic Database cluster, you can choose to add them to the failover list by editing the DNS server settings. All client systems that use the DNS server to connect to Vertica Analytic Database automatically use connection failover without having to change any settings. However, this method does require administrative access to the DNS server that all clients use to connect to the Vertica Analytic Database cluster. This may not be possible in your organization.

Using the backup server list is easier than editing the DNS server settings. However, it decentralizes the failover feature. You may need to update the application settings on each client system if you make changes to your Vertica Analytic Database cluster.

Using DNS failover

To use DNS failover, you need to change your DNS server's settings to map a single host name to multiple IP addresses of hosts in your Vertica Analytic Database cluster. You then have all client applications use this host name to connect to Vertica Analytic Database.

You can choose to have your DNS server return as many IP addresses for the host name as you want. In smaller clusters, you may choose to have it return the IP addresses of all of the hosts in your cluster. However, for larger clusters, you should consider choosing a subset of the hosts to return. Otherwise there can be a long delay as the client driver tries unsuccessfully to connect to each host in a database that is down.

Using the backup host list

To enable backup list-based connection failover, your client application has to specify at least one IP address or host name of a host in the BackupServerNode parameter. The host name or IP can optionally be followed by a colon and a port number. If not supplied, the driver defaults to the standard Vertica port number (5433). To list multiple hosts, separate them by a comma.

The following example demonstrates setting the BackupServerNode connection parameter to specify additional hosts for the connection attempt. The connection string intentionally has a non-existent node, so that the initial connection fails. The client driver has to resort to trying the backup hosts to establish a connection to Vertica.

```
import java.sql.*;
import java.util.Properties;

public class ConnectionFailoverExample {
  public static void main(String[] args) {
    // Assume using JDBC 4.0 driver on JVM 6+. No driver loading needed.
    Properties myProp = new Properties();
    myProp.put("user", "dbadmin");
    myProp.put("password", "vertica");
    // Set two backup hosts to be used if connecting to the first host
    // fails. All of these hosts will be tried in order until the connection
    // succeeds or all of the connections fail.
    myProp.put("BackupServerNode", "VerticaHost02,VerticaHost03");
    Connection conn;
    try {
      // The connection string is set to try to connect to a known
      // bad host (in this case, a host that never existed).
      // The database name is optional.
      conn = DriverManager.getConnection(
          "jdbc:vertica://BadVerticaHost:5433/vmart", myProp);
      System.out.println("Connected!");
      // Query system to table to see what node we are connected to.
      // Assume a single row in response set.
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(
          "SELECT node_name FROM v_monitor.current_session;");
      rs.next();
      System.out.println("Connected to node " + rs.getString(1).trim());
      // Done with connection.
      conn.close();
    } catch (SQLException e) {
      // Catch-all for other exceptions
      e.printStackTrace();
    }
  }
```

When run, the example outputs output similar to the following on the system console:

```
Connected!
Connected to node v_vmart_node0002
```

Notice that the connection was made to the first node in the backup list (node 2).

Specifying connection timeouts

LoginTimeout controls the timeout for JDBC to establish establish a TCP connection with a node and log in to Vertica.

LoginNodeTimeout controls the timeout for JDBC to log in to the Vertica database. After the specified timeout, JDBC attempts to connect to the "next" node, which is determined by either the connection property BackupServerNode or DNS resolution. This is useful if the node is up, but something is wrong with the Vertica process.

LoginNetworkTimeout controls the timeout for JDBC to establish a TCP connection to a Vertica node. If you do not set this connection property, if the node to which the JDBC client attempts to connect is down, the JDBC client will wait "indefinitely," but practically, the system default timeout of 70 seconds is used. A typical use case for LoginNetworkTimeout is to let JDBC connect to another node if the current Vertica node is down for maintenance and modifying the JDBC application's connection string is infeasible.

NetworkTimeout controls the timeout for Vertica to respond to a request from a client after it has established a connection and logged in to the database.

To set these parameters in a connection string:

```
# LoginTimeout is 30 seconds, LoginNodeTimeout is 10 seconds, LoginNetworkTimeout is 2 seconds, NetworkTimeout is 0.5 seconds
Connection conn = DriverManager.getConnection("jdbc:vertica://VerticaHost:5433/verticadb?user=dbadmin&loginTimeout=30&loginNodeTimeout=10"&loginN
```

To set these parameters as a connection property:

```
Properties myProp = new Properties();
myProp.put("user", "dbadmin");
myProp.put("loginTimeout", "30"); // overall connection timeout is 30 seconds to make sure it is not too small for failover
myProp.put("loginNodeTimeout", "10"); // JDBC waits 10 seconds before attempting to connect to the next node if the Vertica process is running but does not r
myProp.put("loginNetworkTimeout", "2"); // node connection timeout is 2 seconds
myProp.put("networkTimeout", "500"); // after the client has logged in, Vertica has 0.5 seconds to respond to each request
Connection conn = DriverManager.getConnection("jdbc:vertica://VerticaHost:5433/verticadb", myProp);
```

Interaction with load balancing

When native connection load balancing is enabled, the additional servers specified in the BackupServerNode connection parameter are only used for the initial connection to a Vertica host. If host redirects the client to another host in the database cluster to handle its connection request, the second connection does not use the backup node list. This is rarely an issue, since native connection load balancing is aware of which nodes are currently up in the database.

See Load balancing in JDBC for more information.

JDBC data types

The JDBC driver transparently converts most Vertica data types to the appropriate Java data type. In a few cases, a Vertica data type cannot be directly translated to a Java data type; these exceptions are explained in this section.

In this section

- The VerticaTypes class
- Numeric data alias conversion
- Using intervals with JDBC
- UUID values
- Complex types in JDBC
- Date types in JDBC

The VerticaTypes class

JDBC does not support all of the data types that Vertica supports. The Vertica JDBC client driver contains an additional class named VerticaTypes that helps you handle identifying these Vertica-specific data types. It contains constants that you can use in your code to specify Vertica data types. This class defines two different categories of data types:

- Vertica's 13 types of interval values. This class contains constant properties for each of these types. You can use these constants to select a specific interval type when instantiating members of the VerticaDayTimeInterval and VerticaYearMonthInterval classes:

  ```
  // Create a day to second interval.
  VerticaDayTimeInterval dayInt = new VerticaDayTimeInterval(
      VerticaTypes.INTERVAL_DAY_TO_SECOND, 10, 0, 5, 40, 0, 0, false);
  // Create a year to month interval.
  VerticaYearMonthInterval monthInt = new VerticaYearMonthInterval(
      VerticaTypes.INTERVAL_YEAR_TO_MONTH, 10, 6, false);
  ```

- Vertica UUID data type. One way you can use the VerticaTypes.UUID is to query a table's metadata to see if a column is a UUID. See UUID values for an example.

See the JDBC documentation for more information on this class.

Numeric data alias conversion

The Vertica server supports data type aliases for integer, float and numeric types. The JDBC driver reports these as its basic data types (BIGINT, DOUBLE PRECISION, and NUMERIC), as follows:

Vertica Server Types and Aliases	Vertica JDBC Type

INTEGER INT INT8 BIGINT SMALLINT TINYINT	BIGINT
DOUBLE PRECISION FLOAT5 FLOAT8 REAL	DOUBLE PRECISION
DECIMAL NUMERIC NUMBER MONEY	NUMERIC

If a client application retrieves the values into smaller data types, Vertica JDBC driver does not check for overflows. The following example demonstrates the results of this overflow.

```
import java.sql.*;
import java.util.Properties;

public class JDBCDataTypes {
    public static void main(String[] args) {
        // If running under a Java 5 JVM, use you need to load the JDBC driver
        // using Class.forname here

        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                    "jdbc:vertica://VerticaHost:5433/VMart",
                     myProp);
            Statement statement = conn.createStatement();
            // Create a table that will hold a row of different types of
            // numeric data.
            statement.executeUpdate(
                    "DROP TABLE IF EXISTS test_all_types cascade");
            statement.executeUpdate("CREATE TABLE test_all_types ("
                    + "c0 INTEGER, c1 TINYINT, c2 DECIMAL, "
                    + "c3 MONEY, c4 DOUBLE PRECISION, c5 REAL)");
            // Add a row of values to it.
            statement.executeUpdate("INSERT INTO test_all_types VALUES("
                    + "111111111111, 444, 555555555555.5555, "
                    + "77777777.77,  88888888888888888.88, "
                    + "10101010.10101010101010)");
            // Query the new table to get the row back as a result set.
            ResultSet rs = statement
```

```
                .executeQuery("SELECT * FROM test_all_types");
        // Get the metadata about the row, including its data type.
        ResultSetMetaData md = rs.getMetaData();
        // Loop should only run once...
        while (rs.next()) {
            // Print out the data type used to defined the column, followed
            // by the values retrieved using several different retrieval
            // methods.

            String[] vertTypes = new String[] {"INTEGER", "TINYINT",
                    "DECIMAL", "MONEY", "DOUBLE PRECISION", "REAL"};

            for (int x=1; x<7; x++) {
                System.out.println("\n\nColumn " + x + " (" + vertTypes[x-1]
                        + ")");
                System.out.println("\tgetColumnType()\t\t"
                        + md.getColumnType(x));
                System.out.println("\tgetColumnTypeName()\t"
                        + md.getColumnTypeName(x));
                System.out.println("\tgetShort()\t\t"
                        + rs.getShort(x));
                System.out.println("\tgetLong()\t\t" + rs.getLong(x));
                System.out.println("\tgetInt()\t\t" + rs.getInt(x));
                System.out.println("\tgetByte()\t\t" + rs.getByte(x));
            }
        }
        rs.close();
        statement.executeUpdate("drop table test_all_types cascade");
        statement.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

The above example prints the following on the console when run:

```
Column 1 (INTEGER)
    getColumnType()       -5
  getColumnTypeName()   BIGINT
  getShort()       455
  getLong()       111111111111
  getInt()       -558038585
  getByte()       -57
Column 2 (TINYINT)
  getColumnType()       -5
  getColumnTypeName()   BIGINT
  getShort()       444
  getLong()       444
  getInt()       444
  getByte()       -68
Column 3 (DECIMAL)
  getColumnType()       2
  getColumnTypeName()   NUMERIC
  getShort()       -1
  getLong()       55555555555
  getInt()       2147483647
  getByte()       -1
Column 4 (MONEY)
  getColumnType()       2
  getColumnTypeName()   NUMERIC
  getShort()       -13455
  getLong()       77777777
  getInt()       77777777
  getByte()       113
Column 5 (DOUBLE PRECISION)
  getColumnType()       8
  getColumnTypeName()   DOUBLE PRECISION
  getShort()       -1
  getLong()       88888888888888900
  getInt()       2147483647
  getByte()       -1
Column 6 (REAL)
  getColumnType()       8
  getColumnTypeName()   DOUBLE PRECISION
  getShort()       8466
  getLong()       10101010
  getInt()       10101010
  getByte()       18
```

Using intervals with JDBC

The JDBC standard does not contain a data type for intervals (the duration between two points in time). To handle Vertica's INTERVAL data type, you must use JDBC's database-specific object type.

When reading an interval value from a result set, use the ResultSet.getObject() method to retrieve the value, and then cast it to one of the Vertica interval classes: VerticaDayTimeInterval (which represents all ten types of day/time intervals) or VerticaYearMonthInterval (which represents all three types of year/month intervals).

> **Note**
> The units interval style is not supported. Do not use the SET INTERVALSTYLE statement to change the interval style in your client applications.

Using intervals in batch inserts

When inserting batches into tables that contain interval data, you must create instances of the VerticaDayTimeInterval or VerticaYearMonthInterval classes to hold the data you want to insert. You set values either when calling the class's constructor, or afterwards using setters. You then insert your interval values using the PreparedStatement.setObject() method. You can also use the .setString() method, passing it a string in " *DD* ** *HH* : *MM* : *SS* "

or " *YY - MM* " format.

The following example demonstrates inserting data into a table containing a day/time interval and a year/month interval:

```
import java.sql.*;
import java.util.Properties;
// You need to import the Vertica JDBC classes to be able to instantiate
// the interval classes.
import com.vertica.jdbc.*;

public class IntervalDemo {
    public static void main(String[] args) {
        // If running under a Java 5 JVM, use you need to load the JDBC driver
        // using Class.forname here
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                    "jdbc:vertica://VerticaHost:5433/VMart", myProp);
            // Create table for interval values
            Statement stmt = conn.createStatement();
            stmt.execute("DROP TABLE IF EXISTS interval_demo");
            stmt.executeUpdate("CREATE TABLE interval_demo("
                    + "DayInt INTERVAL DAY TO SECOND, "
                    + "MonthInt INTERVAL YEAR TO MONTH)");
            // Insert data into interval columns using
            // VerticaDayTimeInterval and VerticaYearMonthInterval
            // classes.
            PreparedStatement pstmt = conn.prepareStatement(
                    "INSERT INTO interval_demo VALUES(?,?)");
            // Create instances of the Vertica classes that represent
            // intervals.
            VerticaDayTimeInterval dayInt = new VerticaDayTimeInterval(10, 0,
                    5, 40, 0, 0, false);
            VerticaYearMonthInterval monthInt = new VerticaYearMonthInterval(
                    10, 6, false);
            // These objects can also be manipulated using setters.
            dayInt.setHour(7);
            // Add the interval values to the batch
            ((VerticaPreparedStatement) pstmt).setObject(1, dayInt);
            ((VerticaPreparedStatement) pstmt).setObject(2, monthInt);
            pstmt.addBatch();
            // Set another row from strings.
            // Set day interval in "days HH:MM:SS" format
            pstmt.setString(1, "10 10:10:10");
            // Set year to month value in "MM-YY" format
            pstmt.setString(2, "12-09");
            pstmt.addBatch();
            // Execute the batch to insert the values.
            try {
                pstmt.executeBatch();
            } catch (SQLException e) {
                System.out.println("Error message: " + e.getMessage());
            }
```

Reading interval values

You read an interval value from a result set using the ResultSet.getObject() method, and cast the object to the appropriate Vertica object class: VerticaDayTimeInterval for day/time intervals or VerticaYearMonthInterval for year/month intervals. This is easy to do if you know that the column contains an interval, and you know what type of interval it is. If your application cannot assume the structure of the data in the result set it reads in,

you can test whether a column contains a database-specific object type, and if so, determine whether the object belongs to either the VerticaDayTimeInterval or VerticaYearMonthInterval classes.

```
        // Retrieve the interval values inserted by previous demo.
        // Query the table to get the row back as a result set.
        ResultSet rs = stmt.executeQuery("SELECT * FROM interval_demo");
        // If you do not know the types of data contained in the result set,
        // you can read its metadata to determine the type, and use
        // additional information to determine the interval type.
        ResultSetMetaData md = rs.getMetaData();
        while (rs.next()) {
            for (int x = 1; x <= md.getColumnCount(); x++) {
                // Get data type from metadata
                int colDataType = md.getColumnType(x);
                // You can get the type in a string:
                System.out.println("Column " + x + " is a "
                        + md.getColumnTypeName(x));
                // Normally, you'd have a switch statement here to
                // handle all sorts of column types, but this example is
                // simplified to just handle database-specific types
                if (colDataType == Types.OTHER) {
                    // Column contains a database-specific type. Determine
                    // what type of interval it is. Assuming it is an
                    // interval...
                    Object columnVal = rs.getObject(x);
                    if (columnVal instanceof VerticaDayTimeInterval) {
                        // We know it is a date time interval
                        VerticaDayTimeInterval interval =
                                (VerticaDayTimeInterval) columnVal;
                        // You can use the getters to access the interval's
                        // data
                        System.out.print("Column " + x + "'s value is ");
                        System.out.print(interval.getDay() + " Days ");
                        System.out.print(interval.getHour() + " Hours ");
                        System.out.println(interval.getMinute()
                                + " Minutes");
                    } else if (columnVal instanceof VerticaYearMonthInterval) {
                        VerticaYearMonthInterval interval =
                                (VerticaYearMonthInterval) columnVal;
                        System.out.print("Column " + x + "'s value is ");
                        System.out.print(interval.getYear() + " Years ");
                        System.out.println(interval.getMonth() + " Months");
                    } else {
                        System.out.println("Not an interval.");
                    }
                }
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

The example prints the following to the console:

```
Column 1 is a INTERVAL DAY TO SECOND
Column 1's value is 10 Days 7 Hours 5 Minutes
Column 2 is a INTERVAL YEAR TO MONTH
Column 2's value is 10 Years 6 Months
Column 1 is a INTERVAL DAY TO SECOND
Column 1's value is 10 Days 10 Hours 10 Minutes
Column 2 is a INTERVAL YEAR TO MONTH
Column 2's value is 12 Years 9 Months
```

Another option is to use database metadata to find columns that contain intervals.

```
// Determine the interval data types by examining the database
// metadata.
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet dbMeta = dbmd.getColumns(null, null, "interval_demo", null);
int colcount = 0;
while (dbMeta.next()) {

  // Get the metadata type for a column.
  int javaType = dbMeta.getInt("DATA_TYPE");

  System.out.println("Column " + ++colcount + " Type name is " +
              dbMeta.getString("TYPE_NAME"));

  if(javaType == Types.OTHER) {
    // The SQL_DATETIME_SUB column in the metadata tells you
    // Specifically which subtype of interval you have.
    // The VerticaDayTimeInterval.isDayTimeInterval()
    // methods tells you if that value is a day time.
    //
    int intervalType = dbMeta.getInt("SQL_DATETIME_SUB");
    if(VerticaDayTimeInterval.isDayTimeInterval(intervalType)) {
      // Now you know it is one of the 10 day/time interval types.
      // When you select this column you can cast to
      // VerticaDayTimeInterval.
      // You can get more specific by checking intervalType
      // against each of the 10 constants directly, but
      // they all are represented by the same object.
      System.out.println("column " + colcount + " is a " +
                  "VerticaDayTimeInterval intervalType = "
                  + intervalType);
    } else if(VerticaYearMonthInterval.isYearMonthInterval(
                intervalType)) {
      //now you know it is one of the 3 year/month intervals,
      //and you can select the column and cast to
      // VerticaYearMonthInterval
      System.out.println("column " + colcount + " is a " +
                  "VerticaDayTimeInterval intervalType = "
                  + intervalType);
    } else {
      System.out.println("Not an interval type.");
    }
  }
```

UUID values

UUID is a core data type in Vertica. However, it is not a core Java data type. You must use the java.util.UUID class to represent UUID values in your Java code. The JDBC driver does not translate values from Vertica to non-core Java data types. Therefore, you must send UUID values to Vertica using generic object methods such as PreparedStatement.setObject() . You also use generic object methods (such as ResultSet.getObject()) to retrieve UUID values from Vertica. You then cast the retrieved objects as a member of the java.util.UUID class.

The following example code demonstrates inserting UUID values into and retrieving UUID values from Vertica.

```
package jdbc_uuid_example;

import java.sql.*;
import java.util.Properties;

public class VerticaUUIDExample {

    public static void main(String[] args) {

        Properties myProp = new Properties();
        myProp.put("user", "dbadmin");
        myProp.put("password", "");
        Connection conn;

        try {
            conn = DriverManager.getConnection("jdbc:vertica://doch01:5433/VMart",
                                               myProp);
            Statement stmt = conn.createStatement();

            // Create a table with a UUID column and a VARCHAR column.
            stmt.execute("DROP TABLE IF EXISTS UUID_TEST CASCADE;");
            stmt.execute("CREATE TABLE UUID_TEST (id UUID, description VARCHAR(25));");

            // Prepare a statement to insert a UUID and a string into the table.
            PreparedStatement ps = conn.prepareStatement("INSERT INTO UUID_TEST VALUES(?,?)");

            java.util.UUID uuid;  // Holds the UUID value.

            for (Integer x = 0; x < 10; x++) {
                // Generate a random uuid
                uuid = java.util.UUID.randomUUID();
                // Set the UUID value by calling setObject.
                ps.setObject(1, uuid);
                // Set the String value to indicate which UUID this is.
                ps.setString(2, "UUID #" + x);
                ps.execute();
            }

            // Query the uuid
            ResultSet rs = stmt.executeQuery("SELECT * FROM UUID_TEST ORDER BY description ASC");
            while (rs.next()) {
                // Cast the object from the result set as a UUID.
                uuid = (java.util.UUID) rs.getObject(1);
                System.out.println(rs.getString(2) + " : " +  uuid.toString());
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

The previous example prints output similar to the following:

```
UUID #0 : 67b6dcb6-c28c-4965-b9f7-5c830a04664d
UUID #1 : 485d3835-2887-4233-b003-392254fa97e0
UUID #2 : 81421f51-c803-473d-8cfc-2c184582a117
UUID #3 : bec8b86a-b650-47b0-852c-8229155332d9
UUID #4 : 8ae5e3ec-d143-4ef7-8901-24f6d0483abf
UUID #5 : 669696ce-5e86-4e87-b8d0-a937f5fc18d7
UUID #6 : 19609ec9-ec56-4444-9cfe-ad2b8de537dd
UUID #7 : 97182e1d-5c7e-4da1-9922-67e804fde173
UUID #8 : c76c3a2b-a9ef-4d65-b2fb-7c637f872b3c
UUID #9 : 3cbbcd26-c177-4277-b3df-bf4d9389f69d
```

Determining whether a column has a UUID data type

JDBC does not support the UUID data type. This limitation means you cannot use the usual ResultSetMetaData.getColumnType() method to determine column's data type is UUID. Calling this method on a UUID column returns Types.OTHER . This value is also to identify interval columns. You can use two ways to determine if a column contains UUIDs:

- Use ResultSetMetaData.getColumnTypeName() to get the name of the column's data type. For UUID columns, this method returns the value "Uuid" as a String .
- Query the table's metadata to get the SQL data type of the column. If this value is equal to VerticaTypes.UUID , the column's data type is UUID.

The following example demonstrates both of these techniques:

```
// This example assumes you already have a database connection
// and result set from a query on a table that may contain a UUID.

//  Get the metadata of the result set to get the column definitions
ResultSetMetaData meta = rs.getMetaData();
int colcount;
int maxcol = meta.getColumnCount();

System.out.println("Using column metadata:");
for (colcount = 1; colcount < maxcol; colcount++) {
// .getColumnType() always returns "OTHER" for UUID columns.
    if (meta.getColumnType(colcount) == Types.OTHER) {
        // To determine that it is a UUID column, test the name of the column type.
        if (meta.getColumnTypeName(colcount).equalsIgnoreCase("uuid")) {
            // It's a UUID column
            System.out.println("Column "+ colcount + " is UUID");
        }
    }
}

// You can also query the table's metadata to find its column types and compare
// it to the VerticaType.UUID constant to see if it is a UUID column.
System.out.println("Using table metadata:");
DatabaseMetaData dbmd = conn.getMetaData();
// Get the metdata for the previously-created test table.
ResultSet tableMeta = dbmd.getColumns(null, null, "UUID_TEST", null);
colcount = 0;
// Each row in the result set has metadata that describes a single column.
while (tableMeta.next()) {
    colcount++;
    // The SQL_DATA_TYPE column holds the Vertica database data type. You compare
    // this value to the VerticvaTypes.UUID constant to see if it is a UUID.
    if (tableMeta.getInt("SQL_DATA_TYPE") == VerticaTypes.UUID) {
        // Column is a UUID data type...
        System.out.println("Column " + colcount + " is a UUID column.");
    }
}
```

This example prints the following to the console if it is run after running the prior example:

```
Using column metadata:
Column 1 is UUID
Using table metadata:
Column 1 is a UUID column.
```

Complex types in JDBC

The results of a java.sql query are stored in a ResultSet . If the ResultSet contains a column of complex type , you can retrieve it with one of the following:

- For columns of type ARRAY , SET , or MAP , use getArray() , which returns a java.sql.Array .
- For columns of type ROW , use getObject() , which returns a java.sql.Struct .

Type conversion table

The objects java.sql.Array and java.sql.Struct each have their own API for accessing complex type data. In each case, the data is returned as java.lang.Object and will need to be type cast to a Java type. The exact Java type to expect depends on the Vertica type used in the complex type definition, as shown in this type conversion table:

java.sql Type	Vertica Type	Java Type
BIT	BOOL	java.lang.Boolean
BIGINT	INT	java.lang.Long
DOUBLE	FLOAT	java.lang.Double
CHAR	CHAR	java.lang.String
VARCHAR	VARCHAR	java.lang.String
LONGVARCHAR	LONGVARCHAR	java.lang.String
DATE	DATE	java.sql.Date
TIME	TIME	java.sql.Time
TIME	TIMETZ	java.sql.Time
TIMESTAMP	TIMESTAMP	java.sql.Timestamp
TIMESTAMP	TIMESTAMPTZ	com.vertica.dsi.dataengine.utilities.TimestampTz
getIntervalRange(oid, typmod)	INTERVAL	com.vertica.jdbc.VerticaDayTimeInterval
getIntervalRange(oid, typmod)	INTERVALYM	com.vertica.jdbc.VerticaYearMonthInterval
BINARY	BINARY	byte[]
VARBINARY	VARBINARY	byte[]
LONGVARBINARY	LONGVARBINARY	byte[]
NUMERIC	NUMERIC	java.math.BigDecimal
TYPE_SQL_GUID	UUID	java.util.UUID
ARRAY	ARRAY	java.lang.Object[]

ARRAY	SET	java.lang.Object[]
STRUCT	ROW	java.sql.Struct
ARRAY	MAP	java.lang.Object[]

ARRAY, SET, and MAP columns

For example, the following methods run queries that return an ARRAY of some Vertica type, which is then type cast to an array of its corresponding Java type by the JDBC driver when retrieved with getArray() . This particular example starts with ARRAY[INT] and ARRAY[FLOAT], so they are type cast to Long[] and Double[] , respectively, as determined by the type conversion table.

- getArrayResultSetExample() shows how the ARRAY can be processed as a java.sql.ResultSet . This example uses getResultSet() which returns the underlying array as another ResultSet . You can use this underlying ResultSet to:
  - Retrieve the parent ResultSet .
  - Treat it as an Object array or ResultSet .
- getArrayObjectExample() shows how the ARRAY can be processed as a native Java array. This example uses getArray() which returns the underlying array as an Object array rather than a ResultSet . This has the following implications:
  - You cannot use an underlying Object array to retrieve its parent array.
  - All underlying arrays are treated as Object arrays (rather than ResultSet s.

```
package com.vertica.jdbc.test.samples;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.Array;
import java.sql.Struct;


public class ComplexTypesArraySamples
{
  /**
   * Executes a query and gets a java.sql.Array from the ResultSet. It then uses the Array#getResultSet
   * method to get a ResultSet containing the contents of the array.
   * @param conn A Connection to a Vertica database
   * @throws SQLException
   */
  public static void getArrayResultSetExample (Connection conn) throws SQLException {
    Statement stmt = conn.createStatement();

    final String queryText = "SELECT ARRAY[ARRAY[1,2,3],ARRAY[4,5,6],ARRAY[7,8,9]]::ARRAY[ARRAY[INT]] as array";
    final String targetColumnName = "array";

    System.out.println ("queryText: " + queryText);
    ResultSet rs = stmt.executeQuery(queryText);
    int targetColumnId = rs.findColumn (targetColumnName);

    while (rs.next ()) {
      Array currentSqlArray = rs.getArray (targetColumnId);
      ResultSet level1ResultSet = currentSqlArray.getResultSet();
      if (level1ResultSet != null) {
        while (level1ResultSet.next ()) {
          // The first column of the result set holds the row index
          int i = level1ResultSet.getInt(1) - 1;
          Array level2SqlArray = level1ResultSet.getArray (2);
          Object level2Object = level2SqlArray.getArray ();
          // For this ARRAY[INT], the driver returns a Long[]
          assert (level2Object instanceof Long[]);
          Long [] level2Array = (Long [])level2Object;
```

```
            System.out.println (" level1Object [" + i + "]: " + level2SqlArray.toString () + " (" + level2SqlArray.getClass() + ")");

            for (int j = 0; j < level2Array.length; j++) {
              System.out.println (" Value [" + i + ", " + j + "]: " + level2Array[j] + " (" + level2Array[j].getClass() + ")");
            }
          }
        }
      }
    }

/**
 * Executes a query and gets a java.sql.Array from the ResultSet. It then uses the Array#getArray
 * method to get the contents of the array as a Java Object [].
 * @param conn A Connection to a Vertica database
 * @throws SQLException
 */
public static void getArrayObjectExample (Connection conn) throws SQLException {
    Statement stmt = conn.createStatement();

    final String queryText = "SELECT ARRAY[ARRAY[0.0,0.1,0.2],ARRAY[1.0,1.1,1.2],ARRAY[2.0,2.1,2.2]]::ARRAY[ARRAY[FLOAT]] as array";
    final String targetColumnName = "array";

    System.out.println ("queryText: " + queryText);
    ResultSet rs = stmt.executeQuery(queryText);
    int targetColumnId = rs.findColumn (targetColumnName);

    while (rs.next ()) {
      // Get the java.sql.Array from the result set
      Array currentSqlArray = rs.getArray (targetColumnId);
      // Get the internal Java Object implementing the array
      Object level1ArrayObject = currentSqlArray.getArray ();
      if (level1ArrayObject != null) {
        // All returned instances are Object[]
        assert (level1ArrayObject instanceof Object[]);
        Object [] level1Array = (Object [])level1ArrayObject;
        System.out.println ("Vertica driver returned a: " + level1Array.getClass());

        for (int i = 0; i < level1Array.length; i++) {
          Object level2Object = level1Array[i];
          // For this ARRAY[FLOAT], the driver returns a Double[]
          assert (level2Object instanceof Double[]);
          Double [] level2Array = (Double [])level2Object;
          for (int j = 0; j < level2Array.length; j++) {
            System.out.println (" Value [" + i + ", " + j + "]: " + level2Array[j] + " (" + level2Array[j].getClass() + ")");
          }
        }
      }
    }
}
```

The output of getArrayResultSetExample() shows that the Vertica column type ARRAY[INT] is type cast to Long[] :

```
queryText: SELECT ARRAY[ARRAY[1,2,3],ARRAY[4,5,6],ARRAY[7,8,9]]::ARRAY[ARRAY[INT]] as array
 level1Object [0]: [1,2,3] (class com.vertica.jdbc.jdbc42.S42Array)
 Value [0, 0]: 1 (class java.lang.Long)
 Value [0, 1]: 2 (class java.lang.Long)
 Value [0, 2]: 3 (class java.lang.Long)
 level1Object [1]: [4,5,6] (class com.vertica.jdbc.jdbc42.S42Array)
 Value [1, 0]: 4 (class java.lang.Long)
 Value [1, 1]: 5 (class java.lang.Long)
 Value [1, 2]: 6 (class java.lang.Long)
 level1Object [2]: [7,8,9] (class com.vertica.jdbc.jdbc42.S42Array)
 Value [2, 0]: 7 (class java.lang.Long)
 Value [2, 1]: 8 (class java.lang.Long)
 Value [2, 2]: 9 (class java.lang.Long)
```

The output of getArrayObjectExample() shows that the Vertica column type ARRAY[FLOAT] is type cast to Double[] :

```
queryText: SELECT ARRAY[ARRAY[0.0,0.1,0.2],ARRAY[1.0,1.1,1.2],ARRAY[2.0,2.1,2.2]]::ARRAY[ARRAY[FLOAT]] as array
Vertica driver returned a: class [Ljava.lang.Object;
 Value [0, 0]: 0.0 (class java.lang.Double)
 Value [0, 1]: 0.1 (class java.lang.Double)
 Value [0, 2]: 0.2 (class java.lang.Double)
 Value [1, 0]: 1.0 (class java.lang.Double)
 Value [1, 1]: 1.1 (class java.lang.Double)
 Value [1, 2]: 1.2 (class java.lang.Double)
 Value [2, 0]: 2.0 (class java.lang.Double)
 Value [2, 1]: 2.1 (class java.lang.Double)
 Value [2, 2]: 2.2 (class java.lang.Double)
```

ROW columns

Calling getObject() on a java.sql.ResultSet that contains a column of type ROW retrieves the column as a java.sql.Struct which contains an Object[] (itself retrievable with getAttributes()).

Each element of the Object[] represents an attribute from the struct, and each attribute has a corresponding Java type shown in the type conversion table above.

This example defines a ROW with the following attributes:

```
 Name    | Value        | Vertica Type | Java Type
-----------------------------------------------------------------
 name    | Amy          | VARCHAR      | String
 date    | '07/10/2021' | DATE         | java.sql.Date
 id      | 5            | INT          | java.lang.Long
 current | false        | BOOLEAN      | java.lang.Boolean
```

```
package com.vertica.jdbc.test.samples;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.Array;
import java.sql.Struct;


public class ComplexTypesSamples
{
    /**
     * Executes a query and gets a java.sql.Struct from the ResultSet. It then uses the Struct#getAttributes
     * method to get the contents of the struct as a Java Object [].
     * @param conn A Connection to a Vertica database
     * @throws SQLException
     */
    public static void getRowExample (Connection conn) throws SQLException {
        Statement stmt = conn.createStatement();

        final String queryText = "SELECT ROW('Amy', '07/10/2021'::Date, 5, false) as rowExample(name, date, id, current)";
        final String targetColumnName = "rowExample";

        System.out.println ("queryText: " + queryText);
        ResultSet rs = stmt.executeQuery(queryText);
        int targetColumnId = rs.findColumn (targetColumnName);

        while (rs.next ()) {
            // Get the java.sql.Array from the result set
            Object currentObject = rs.getObject (targetColumnId);
            assert (currentObject instanceof Struct);
            Struct rowStruct = (Struct)currentObject;

            Object[] attributes = rowStruct.getAttributes();

            // attributes.length should be 4 based on the queryText
            assert (attributes.length == 4);
            assert (attributes[0] instanceof String);
            assert (attributes[1] instanceof java.sql.Date);
            assert (attributes[2] instanceof java.lang.Long);
            assert (attributes[3] instanceof java.lang.Boolean);

            System.out.println ("attributes[0]: " + attributes[0] + " (" + attributes[0].getClass().getName() +")");
            System.out.println ("attributes[1]: " + attributes[1] + " (" + attributes[1].getClass().getName() +")");
            System.out.println ("attributes[2]: " + attributes[2] + " (" + attributes[2].getClass().getName() +")");
            System.out.println ("attributes[3]: " + attributes[3] + " (" + attributes[3].getClass().getName() +")");
        }
    }
```

The output of getRowExample() shows the attribute of each element and its corresponding Java type:

```
queryText: SELECT ROW('Amy', '07/10/2021'::Date, 5, false) as rowExample(name, date, id, current)
attributes[0]: Amy (java.lang.String)
attributes[1]: 2021-07-10 (java.sql.Date)
attributes[2]: 5 (java.lang.Long)
attributes[3]: false (java.lang.Boolean)
```

Date types in JDBC

Converting a date to a string

For the purposes of this page, a large date is defined as a date with a year that exceeds 9999.

If your database doesn't contain any large dates , then you can reliably call toString() to convert the dates to strings.

Otherwise, if your database contains large dates, you should use java.text.SimpleDateFormat and its format() method:

1. Define a String format with java.text.SimpleDateFormat . The number of characters in yyyy in the format defines the minimum number of characters to use in the date.
2. Call SimpleDateFormat.format() to convert the java.sql.Date object to a String.

Examples

For example, the following method returns a string when passed a java.sql.Date object as an argument. Here, the year part of the format, YYYY indicates that this format is compatible with all dates with at least four characters in its year.

```
#import java.sql.Date;

private String convertDate (Date date) {
   SimpleDateFormat dateFormat = new SimpleDateFormat ("yyyy-MM-dd");
   return dateFormat.format (date);
```

Executing queries through JDBC

To run a query through JDBC:

1. Connect with the Vertica database. See Creating and configuring a connection .
2. Run the query.

The method you use to run the query depends on the type of query you want to run:

- a DDL query that does not return a result set.
- a DDL query that returns a result set.
- a DML query

Executing DDL (data definition language) queries

To run DDL queries, such as CREATE TABLE and COPY , use the Statement.execute() method. You get an instance of this class by calling the createStatement method of your connection object.

The following example creates an instance of the Statement class and uses it to execute a CREATE TABLE and a COPY query:

```
Statement stmt = conn.createStatement();
stmt.execute("CREATE TABLE address_book (Last_Name char(50) default '',' +
   "First_Name char(50),Email char(50),Phone_Number char(50))");
stmt.execute("COPY address_book FROM 'address.dat' DELIMITER ',' NULL 'null'");
```

Executing queries that return result sets

Use the Statement class's executeQuery method to execute queries that return a result set, such as SELECT . To get the data from the result set, use methods such as getInt , getString , and getDouble to access column values depending upon the data types of columns in the result set. Use ResultSet.next to advance to the next row of the data set.

```
ResultSet rs = null;
rs = stmt.executeQuery("SELECT First_Name, Last_Name FROM address_book");
int x = 1;
while(rs.next()){
   System.out.println(x + ". " + rs.getString(1).trim() + " "
                  + rs.getString(2).trim());
   x++;
```

> **Note**
> The Vertica JDBC driver does not support scrollable cursors. You can only read forwards through the result set.

Executing DML (data manipulation language) queries using executeUpdate

Use the executeUpdate method for DML SQL queries that change data in the database, such as INSERT , UPDATE and DELETE which do not return a result set.

```
stmt.executeUpdate("INSERT INTO address_book " +
                   "VALUES ('Ben-Shachar', 'Tamar', 'tamarrow@example.com'," +
                   "'555-380-6466')");
stmt.executeUpdate("INSERT INTO address_book (First_Name, Email) " +
                   "VALUES ('Pete','pete@example.com')");
```

> **Note**
> The Vertica JDBC driver's Statement class supports executing multiple statements in the SQL string you pass to the execute method. The PreparedStatement class does not support using multiple statements in a single execution.

Executing stored procedures

You can create and execute stored procedures with CallableStatement s.

To create a stored procedure :

```
Statement st = conn.createStatement();

String createSimpleSp = "CREATE OR REPLACE PROCEDURE raiseInt(IN x INT) LANGUAGE PLvSQL AS $$ " +
 "BEGIN" +
   "RAISE INFO 'x = %', x;" +
"END;" +
"$$;";

st.execute(createSimpleSp);
```

To call a stored procedure:

```
String spCall = "CALL raiseInt (?)";
CallableStatement stmt = conn.prepareCall(spCall);
stmt.setInt(1, 42);
```

Stored procedures do not yet support OUT parameters . Instead, you can return and retrieve execution information with RAISE and getWarnings() respectively:

```
System.out.println(stmt.getWarnings().toString());
```

Canceling JDBC queries

You can cancel JDBC queries with the Statement.cancel() method.

The following example creates a table jdbccanceltest and runs two queries, canceling the first:

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.Array;
import java.sql.Struct;


public class CancelSamples
{
   /**
    * Sets up a large test table, queries its contents and cancels the query.
    * @param  conn A connection to a Vertica database
    * @throws SQLException
    */
```

```
public static void sampleCancelTest(Connection conn) throws SQLException
{
    setup(conn);
    try
    {
        runQueryAndCancel(conn);
        runSecondQuery(conn);
    }
    finally
    {
        cleanup(conn);
    }
}

// Set up table used in test.
private static void setup(Connection conn) throws SQLException
{
    System.out.println("Creating and loading table...");
    Statement stmt = conn.createStatement();
    String queryText = "DROP TABLE IF EXISTS jdbccanceltest";
    stmt.execute(queryText);

    queryText = "CREATE TABLE jdbccanceltest(id INTEGER, time TIMESTAMP)";
    stmt.execute(queryText);

    queryText = "INSERT INTO jdbccanceltest SELECT row_number() OVER(), slice_time "
              + "FROM(SELECT slice_time FROM("
              + "SELECT '2021-01-01'::timestamp s UNION ALL SELECT '2022-01-01'::timestamp s"
              + ") sq TIMESERIES slice_time AS '1 second' OVER(ORDER BY s)) sq2";
    stmt.execute(queryText);
}

/**
 * Execute a long-running query and cancel it.
 * @param  conn A connection to a Vertica database
 * @throws SQLException
 */
private static void runQueryAndCancel(Connection conn) throws SQLException
{
    System.out.println("Running and canceling query...");
    Statement stmt = conn.createStatement();
    String queryText = "select id, time from jdbccanceltest";
    ResultSet rs = stmt.executeQuery(queryText);

    int i=0;
    stmt.cancel();
    try
    {
        while (rs.next()) ;
        i++;
    }
    catch (SQLException e)
    {
        System.out.println("Query canceled after retrieving " + i + " rows");
        System.out.println(e.getMessage());
    }
}

/**
 * Run a simple query to demonstrate that it can be run after
 * the previous query was canceled.
```

```
     * @param conn A connection to a Vertica database
     * @throws SQLException
     */
    private static void runSecondQuery(Connection conn) throws SQLException
    {
        String queryText = "select 1 from dual";
        Statement stmt = conn.createStatement();
        try
        {
            ResultSet rs = stmt.executeQuery(queryText);
            while (rs.next()) ;
        }
        catch (SQLException e)
        {
            System.out.println(e.getMessage());
            System.out.println("warning: no exception should have been thrown on query after cancel");
        }
    }

    /**
     * Clean up table used in test.
     * @param conn A connetion to a Vertica database
     * @throws SQLException
     */
    private static void cleanup(Connection conn) throws SQLException
    {
        String queryText = "drop table if exists jdbccanceltest";
        Statement stmt = conn.createStatement();
        stmt.execute(queryText);
    }
```

Loading data through JDBC

You can use any of the following methods to load data via the JDBC interface:

- Executing a SQL INSERT statement to insert a single row directly.
- Batch loading data using a prepared statement.
- Bulk loading data from files or streams using COPY .

The following sections explain in detail how you load data using JDBC.

In this section

- Using a single row insert
- Batch inserts using JDBC prepared statements
- Bulk loading using the COPY statement
- Streaming data via JDBC

Using a single row insert

The simplest way to insert data into a table is to use the SQL INSERT statement. You can use this statement by instantiating a member of the Statement class, and use its executeUpdate() method to run your SQL statement.

The following code fragment demonstrates how you can create a Statement object and use it to insert data into a table named address_book:

```
Statement stmt = conn.createStatement();
stmt.executeUpdate("INSERT INTO address_book " +
                   "VALUES ('Smith', 'John', 'jsmith@example.com', " +
                   "'555-123-4567')");
```

This method has a few drawbacks: you need convert your data to string and escape any special characters in your data. A better way to insert data is to use prepared statements. See Batch inserts using JDBC prepared statements .

Batch inserts using JDBC prepared statements

You can load batches of data into Vertica using prepared INSERT statements—server-side statements that you set up once, and then call repeatedly. You instantiate a member of the PreparedStatement class with a SQL statement that contains question mark placeholders for data. For example:

```
PreparedStatement pstmt = conn.prepareStatement(
            "INSERT INTO customers(last, first, id) VALUES(?,?,?)");
```

You then set the parameters using data-type-specific methods on the PreparedStatement object, such as setString() and setInt() . Once your parameters are set, call the addBatch() method to add the row to the batch. When you have a complete batch of data ready, call the executeBatch() method to execute the insert batch.

Behind the scenes, the batch insert is converted into a COPY statement. When the connection's AutoCommit parameter is disabled, Vertica keeps the COPY statement open and uses it to load subsequent batches until the transaction is committed, the cursor is closed, or your application executes anything else (or executes any statement using another Statement or PreparedStatement object). Using a single COPY statement for multiple batch inserts makes loading data more efficient. If you are loading multiple batches, you should disable the AutoCommit property of the database to take advantage of this increased efficiency.

When performing batch inserts, experiment with various batch and row sizes to determine the settings that provide the best performance.

The following example demonstrates using a prepared statement to batch insert data.

```
import java.sql.*;
import java.util.Properties;

public class BatchInsertExample {
   public static void main(String[] args) {
      Properties myProp = new Properties();
      myProp.put("user", "ExampleUser");
      myProp.put("password", "password123");

    //Set streamingBatchInsert to True to enable streaming mode for batch inserts.
    //myProp.put("streamingBatchInsert", "True");

    Connection conn;
      try {
         conn = DriverManager.getConnection(
                  "jdbc:vertica://VerticaHost:5433/ExampleDB",
                  myProp);
         // establish connection and make a table for the data.
         Statement stmt = conn.createStatement();


         // Set AutoCommit to false to allow Vertica to reuse the same
         // COPY statement
         conn.setAutoCommit(false);


         // Drop table and recreate.
         stmt.execute("DROP TABLE IF EXISTS customers CASCADE");
         stmt.execute("CREATE TABLE customers (CustID int, Last_Name"
                  + " char(50), First_Name char(50),Email char(50), "
                  + "Phone_Number char(12))");
         // Some dummy data to insert.
         String[] firstNames = new String[] { "Anna", "Bill", "Cindy",
                  "Don", "Eric" };
         String[] lastNames = new String[] { "Allen", "Brown", "Chu",
                  "Dodd", "Estavez" };
         String[] emails = new String[] { "aang@example.com",
                  "b.brown@example.com", "cindy@example.com",
                  "d.d@example.com", "e.estavez@example.com" };
         String[] phoneNumbers = new String[] { "123-456-7890",
                  "555-444-3333", "555-867-5309",
```

```
                "555-555-1212", "781-555-0000" };
        // Create the prepared statement
        PreparedStatement pstmt = conn.prepareStatement(
                "INSERT INTO customers (CustID, Last_Name, " +
                "First_Name, Email, Phone_Number)" +
                " VALUES(?,?,?,?,?)");
        // Add rows to a batch in a loop. Each iteration adds a
        // new row.
        for (int i = 0; i < firstNames.length; i++) {
            // Add each parameter to the row.
            pstmt.setInt(1, i + 1);
            pstmt.setString(2, lastNames[i]);
            pstmt.setString(3, firstNames[i]);
            pstmt.setString(4, emails[i]);
            pstmt.setString(5, phoneNumbers[i]);
            // Add row to the batch.
            pstmt.addBatch();
        }

        try {
            // Batch is ready, execute it to insert the data
            pstmt.executeBatch();
        } catch (SQLException e) {
            System.out.println("Error message: " + e.getMessage());
            return; // Exit if there was an error
        }

        // Commit the transaction to close the COPY command
        conn.commit();


        // Print the resulting table.
        ResultSet rs = null;
        rs = stmt.executeQuery("SELECT CustID, First_Name, "
                    + "Last_Name FROM customers ORDER BY CustID");
        while (rs.next()) {
            System.out.println(rs.getInt(1) + " - "
                        + rs.getString(2).trim() + " "
                        + rs.getString(3).trim());
        }
        // Cleanup
        conn.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

The result of running the example code is:

```
1 - Anna Allen
2 - Bill Brown
3 - Cindy Chu
4 - Don Dodd
5 - Eric Estavez
```

Streaming batch inserts

By default, Vertica performs batch inserts by caching each row and inserting the cache when the user calls the executeBatch() method. Vertica also supports streaming batch inserts. A streaming batch insert adds a row to the database each time the user calls addBatch() . Streaming batch inserts improve database performance by allowing parallel processing and reducing memory demands.

Note
Once you begin a streaming batch insert, you cannot make other JDBC calls that require client-server communication until you have executed the batch or closed or rolled back the connection.

To enable streaming batch inserts, set the streamingBatchInsert property to True. The preceding code sample includes a line enabling streamingBatchInsert mode. Remove the // comment marks to enable this line and activate streaming batch inserts.

The following table explains the various batch insert methods and how their behavior differs between default batch insert mode and streaming batch insert mode.

Method	Default Batch Insert Behavior	Streaming Batch Insert Behavior
addBatch()	Adds a row to the row cache.	Inserts a row into the database.
executeBatch()	Adds the contents of the row cache to the database in a single action.	Sends an end-of-batch message to the server and returns an array of integers indicating the success or failure of each addBatch() attempt.
clearBatch()	Clears the row cache without inserting any rows.	Not supported. Triggers an exception if used when streaming batch inserts are enabled.

Notes

- Using the PreparedStatement.setFloat() method can cause rounding errors. If precision is important, use the .setDouble() method instead.
- The PreparedStatement object caches the connection's AutoCommit property when the statement is prepared. Later changes to the AutoCommit property have no effect on the prepared statement.

In this section

- Error handling during batch loads
- Identifying accepted and rejected rows (JDBC)
- Rolling back batch loads on the server

Error handling during batch loads

When loading individual batches, you can find how many rows were accepted and what rows were rejected (see Identifying Accepted and Rejected Rows for details). If you have disabled the AutoCommit connection setting, other errors (such as disk space errors, for example) do not occur while inserting individual batches. This behavior is caused by having a single SQL COPY statement perform the loading of multiple consecutive batches (which makes the load process more efficient). It is only when the COPY statement closes that the batched data is committed and Vertica reports other types of errors.

Therefore, your bulk loading application should be prepared to check for errors when the COPY statement closes. You can trigger the COPY statement to close by:

- ending the batch load transaction by calling Connection.commit()
- closing the statement using Statement.close()
- setting the connection's AutoCommit property to true before inserting the last batch in the load

Note
The COPY statement also closes if you execute any non-insert statement or execute any statement using a different Statement or PreparedStatement object. Ending the COPY statement using either of these methods can lead to confusion and a harder-to-maintain application, since you would need to handle batch load errors in a non-batch load statement. You should explicitly end the COPY statement at the end of your batch load and handle any errors at that time.

Identifying accepted and rejected rows (JDBC)

The return value of PreparedStatement.executeBatch is an integer array containing the success or failure status of inserting each row. A value 1 means the row was accepted and a value of -3 means that the row was rejected. In the case where an exception occurred during the batch execution, you can also get the array using BatchUpdateException.getUpdateCounts() .

The following example extends the example shown in Batch inserts using JDBC prepared statements to retrieve this array and display the results the batch load.

```
import java.sql.*;
import java.util.Arrays;
import java.util.Properties;

public class BatchInsertErrorHandlingExample {
  public static void main(String[] args) {
    Properties myProp = new Properties();
    myProp.put("user", "ExampleUser");
    myProp.put("password", "password123");
    Connection conn;

    // establish connection and make a table for the data.
    try {
      conn = DriverManager.getConnection(
                  "jdbc:vertica://VerticaHost:5433/ExampleDB",
                  myProp);


      // Disable auto commit
      conn.setAutoCommit(false);

      // Create a statement
      Statement stmt = conn.createStatement();
      // Drop table and recreate.
      stmt.execute("DROP TABLE IF EXISTS customers CASCADE");
      stmt.execute("CREATE TABLE customers (CustID int, Last_Name"
                  + " char(50), First_Name char(50),Email char(50), "
                  + "Phone_Number char(12))");

      // Some dummy data to insert. The one row won't insert because
      // the phone number is too long for the phone column.
      String[] firstNames = new String[] { "Anna", "Bill", "Cindy",
                  "Don", "Eric" };
      String[] lastNames = new String[] { "Allen", "Brown", "Chu",
                  "Dodd", "Estavez" };
      String[] emails = new String[] { "aang@example.com",
                  "b.brown@example.com", "cindy@example.com",
                  "d.d@example.com", "e.estavez@example.com" };
      String[] phoneNumbers = new String[] { "123-456-789",
                  "555-444-3333", "555-867-53093453453",
                  "555-555-1212", "781-555-0000" };

      // Create the prepared statement
      PreparedStatement pstmt = conn.prepareStatement(
                  "INSERT INTO customers (CustID, Last_Name, " +
                  "First_Name, Email, Phone_Number)" +
                  " VALUES(?,?,?,?,?)");

      // Add rows to a batch in a loop. Each iteration adds a
      // new row.
      for (int i = 0; i < firstNames.length; i++) {
        // Add each parameter to the row.
        pstmt.setInt(1, i + 1);
        pstmt.setString(2, lastNames[i]);
        pstmt.setString(3, firstNames[i]);
        pstmt.setString(4, emails[i]);
        pstmt.setString(5, phoneNumbers[i]);
        // Add row to the batch.
        pstmt.addBatch();
```

```
            pstmt.addBatch();
        }

        // Integer array to hold the results of inserting
        // the batch. Will contain an entry for each row,
        // indicating success or failure.
        int[] batchResults = null;

        try {
            // Batch is ready, execute it to insert the data
            batchResults = pstmt.executeBatch();
        } catch (BatchUpdateException e) {
            // We expect an exception here, since one of the
            // inserted phone numbers is too wide for its column. All of the
            // rest of the rows will be inserted.
            System.out.println("Error message: " + e.getMessage());

            // Batch results isn't set due to exception, but you
            // can get it from the exception object.
            //
            // In your own code, you shouldn't assume the a batch
            // exception occurred, since exceptions can be thrown
            // by the server for a variety of reasons.
            batchResults = e.getUpdateCounts();
        }
        // You should also be prepared to catch SQLExceptions in your own
        // application code, to handle dropped connections and other general
        // problems.

        // Commit the transaction
        conn.commit();


        // Print the array holding the results of the batch insertions.
        System.out.println("Return value from inserting batch: "
                    + Arrays.toString(batchResults));
        // Print the resulting table.
        ResultSet rs = null;
        rs = stmt.executeQuery("SELECT CustID, First_Name, "
                    + "Last_Name FROM customers ORDER BY CustID");
        while (rs.next()) {
            System.out.println(rs.getInt(1) + " - "
                        + rs.getString(2).trim() + " "
                        + rs.getString(3).trim());
        }

        // Cleanup
        conn.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Running the above example produces the following output on the console:

```
Error message: [Vertica][VJDBC](100172) One or more rows were rejected by the server.Return value from inserting batch: [1, 1, -3, 1, 1]
1 - Anna Allen
2 - Bill Brown
4 - Don Dodd
5 - Eric Estavez
```

Notice that the third row failed to insert because its phone number is too long for the Phone_Number column. All of the rest of the rows in the batch (including those after the error) were correctly inserted.

> **Note**
> It is more efficient for you to ensure that the data you are inserting is the correct data type and width for the table column you are inserting it into than to handle exceptions after the fact.

Rolling back batch loads on the server

Batch loads always insert all of their data, even if one or more rows is rejected. Only the rows that caused errors in a batch are not loaded. When the database connection's AutoCommit property is true, batches automatically commit their transactions when they complete, so once the batch finishes loading, the data is committed.

In some cases, you may want all of the data in a batch to be successfully inserted—none of the data should be committed if an error occurs. The best way to accomplish this is to turn off the database connection's AutoCommit property to prevent batches from automatically committing themselves. Then, if a batch encounters an error, you can roll back the transaction after catching the BatchUpdateException caused by the insertion error.

The following example demonstrates performing a rollback if any error occurs when loading a batch.

```
import java.sql.*;
import java.util.Arrays;
import java.util.Properties;

public class RollbackBatchOnError {
  public static void main(String[] args) {
    Properties myProp = new Properties();
    myProp.put("user", "ExampleUser");
    myProp.put("password", "password123");
    Connection conn;
    try {
      conn = DriverManager.getConnection(
                  "jdbc:vertica://VerticaHost:5433/ExampleDB",
                  myProp);
      // Disable auto-commit. This will allow you to roll back a
      // a batch load if there is an error.
      conn.setAutoCommit(false);
      // establish connection and make a table for the data.
      Statement stmt = conn.createStatement();
      // Drop table and recreate.
      stmt.execute("DROP TABLE IF EXISTS customers CASCADE");
      stmt.execute("CREATE TABLE customers (CustID int, Last_Name"
                  + " char(50), First_Name char(50),Email char(50), "
                  + "Phone_Number char(12))");

      // Some dummy data to insert. The one row won't insert because
      // the phone number is too long for the phone column.
      String[] firstNames = new String[] { "Anna", "Bill", "Cindy",
                  "Don", "Eric" };
      String[] lastNames = new String[] { "Allen", "Brown", "Chu",
                  "Dodd", "Estavez" };
      String[] emails = new String[] { "aang@example.com",
                  "b.brown@example.com", "cindy@example.com",
                  "d.d@example.com", "e.estavez@example.com" };
      String[] phoneNumbers = new String[] { "123-456-789",
                  "555-444-3333", "555-867-53094535", "555-555-1212",
                  "781-555-0000" };
      // Create the prepared statement
      PreparedStatement pstmt = conn.prepareStatement(
                  "INSERT INTO customers (CustID, Last_Name, " +
                  "First_Name, Email, Phone_Number) "+
```

```
                "VALUES(?,?,?,?,?)");
        // Add rows to a batch in a loop. Each iteration adds a
        // new row.
        for (int i = 0; i < firstNames.length; i++) {
            // Add each parameter to the row.
            pstmt.setInt(1, i + 1);
            pstmt.setString(2, lastNames[i]);
            pstmt.setString(3, firstNames[i]);
            pstmt.setString(4, emails[i]);
            pstmt.setString(5, phoneNumbers[i]);
            // Add row to the batch.
            pstmt.addBatch();
        }
        // Integer array to hold the results of inserting
        // the batch. Will contain an entry for each row,
        // indicating success or failure.
        int[] batchResults = null;
        try {
            // Batch is ready, execute it to insert the data
            batchResults = pstmt.executeBatch();
            // If we reach here, we inserted the batch without errors.
            // Commit it.
            System.out.println("Batch insert successful. Committing.");
            conn.commit();
        } catch (BatchUpdateException e) {
                System.out.println("Error message: " + e.getMessage());
                // Batch results isn't set due to exception, but you
                // can get it from the exception object.
                batchResults =  e.getUpdateCounts();
                // Roll back the batch transaction.
                System.out.println("Rolling back batch insertion");
                conn.rollback();
        }
        catch  (SQLException e) {
            // General SQL errors, such as connection issues, throw
            // SQLExceptions. Your application should do something more
            // than just print a stack trace.
            e.printStackTrace();
        }
        System.out.println("Return value from inserting batch: "
                    + Arrays.toString(batchResults));
        System.out.println("Customers table contains:");



        // Print the resulting table.
        ResultSet rs = null;
        rs = stmt.executeQuery("SELECT CustID, First_Name, "
                    + "Last_Name FROM customers ORDER BY CustID");
        while (rs.next()) {
            System.out.println(rs.getInt(1) + " - "
                        + rs.getString(2).trim() + " "
                        + rs.getString(3).trim());
        }

        // Cleanup
        conn.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Running the above example prints the following on the system console:

```
Error message: [Vertica][VJDBC](100172) One or more rows were rejected by the server.Rolling back batch insertion
Return value from inserting batch: [1, 1, -3, 1, 1]
Customers table contains:
```

The return values indicate whether each rows was successfully inserted. The value 1 means the row inserted without any issues, and a -3 indicates the row failed to insert.

The customers table is empty since the batch insert was rolled back due to the error caused by the third column.

Bulk loading using the COPY statement

One of the fastest ways to load large amounts of data into Vertica at once (bulk loading) is to use the COPY statement . This statement loads data from a file stored on a Vertica host (or in a data stream) into a table in the database. You can pass the COPY statement parameters that define the format of the data in the file, how the data is to be transformed as it is loaded, how to handle errors, and how the data should be loaded. See the COPY documentation for details.

Important
In databases that were created in versions of Vertica ≤ 9.2, COPY supports the DIRECT option, which specifies to load data directly into ROS rather than WOS. Use this option when loading large (>100MB) files into the database; otherwise, the load is liable to fill the WOS. When this occurs, the Tuple Mover must perform a moveout operation on the WOS data. It is more efficient to directly load into ROS and avoid forcing a moveout.

In databases created in Vertica 9.3, Vertica ignores load options and hints and always uses a load method of DIRECT. Databases created in versions ≥ 10.0 no longer support WOS and moveout operations; all data is always loaded directly into ROS.

Only a superuser can use COPY to copy a file stored on a host, so you must connect to the database with a superuser account. If you want to have a non-superuser user bulk-load data, you can use COPY to load from a stream on the host (such as STDIN) rather than a file or stream data from the client (see Streaming data via JDBC). You can also perform a standard batch insert using a prepared statement , which uses the COPY statement in the background to load the data.

> **Note**
> When using COPY parameter ON ANY NODE , confirm that the source file is identical on all nodes. Using different files can produce inconsistent results.

The following example demonstrates using the COPY statement through the JDBC to load a file name customers.txt into a new database table. This file must be stored on the database host to which your application connects—in this example, a host named VerticaHost.

```
import java.sql.*;
import java.util.Properties;
import com.vertica.jdbc.*;

public class COPYFromFile {
    public static void main(String[] args) {
        Properties myProp = new Properties();
        myProp.put("user", "ExampleAdmin"); // Must be superuser
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                        "jdbc:vertica://VerticaHost:5433/ExampleDB",myProp);
            // Disable AutoCommit
            conn.setAutoCommit(false);
            Statement stmt = conn.createStatement();
            // Create a table to hold data.
            stmt.execute("DROP TABLE IF EXISTS customers;");
            stmt.execute("CREATE TABLE IF NOT EXISTS customers (Last_Name char(50) "
                        + "NOT NULL, First_Name char(50),Email char(50), "
                        + "Phone_Number char(15))");

             // Use the COPY command to load data. Use ENFORCELENGTH to reject
            // strings too wide for their columns.
            boolean result = stmt.execute("COPY customers FROM "
                        + " '/data/customers.txt' ENFORCELENGTH");

            // Determine if execution returned a count value, or a full result
            // set.
            if (result) {
                System.out.println("Got result set");
            } else {
                // Count will usually return the count of rows inserted.
                System.out.println("Got count");
                int rowCount = stmt.getUpdateCount();
                System.out.println("Number of accepted rows = " + rowCount);
            }


            // Commit the data load
            conn.commit();
        } catch (SQLException e) {
            System.out.print("Error: ");
            System.out.println(e.toString());
        }
    }
```

The example prints the following out to the system console when run (assuming that the customers.txt file contained two million valid rows):

```
Number of accepted rows = 2000000
```

Streaming data via JDBC

There are two options to stream data from a file on the client to your Vertica database:

- Use the VerticaCopyStream class to stream data in an object-oriented manner - details on the class are available in the JDBC documentation .
- Execute a COPY LOCAL SQL statement to stream the data

The topics in this section explain how to use these options.

In this section

- Using VerticaCopyStream
- Using COPY LOCAL with JDBC

Using VerticaCopyStream

The VerticaCopyStream class lets you stream data from the client system to a Vertica database. It lets you use COPY directly without first copying the data to a host in the database cluster. Using COPY to load data from the host requires superuser privileges to access the host's file system. The COPY statement used to load data from a stream does not require superuser privileges, so your client can connect with any user account that has INSERT privileges on the target table.

To copy streams into the database:

1. Disable the database connections AutoCommit connection parameter.
2. Instantiate a VerticaCopyStreamObject , passing it at least the database connection objects and a string containing a COPY statement to load the data. This statement must copy data from the STDIN into your table. You can use any parameters that are appropriate for your data load.

   > **Note**
   > The VerticaCopyStreamObject constructor optionally takes a single InputStream object, or a List of InputStream objects. This option lets you pre-populate the list of streams to be copied into the database.

3. Call VerticaCopyStreamObject.start() to start the COPY statement and begin streaming the data in any streams you have already added to the VerticaCopyStreamObject .
4. Call VerticaCopyStreamObject.addStream() to add additional streams to the list of streams to send to the database. You can then call VerticaCopyStreamObject.execute() to stream them to the server.
5. Optionally, call VerticaCopyStreamObject.getRejects() to get a list of rejected rows from the last .execute() call. The list of rejects is reset by each call to .execute() or .finish() .

   > **Note**
   > If you used either the REJECTED DATA or EXCEPTIONS options in the COPY statement you passed to VerticaCopyStreamObject the object in step 2, .getRejects() returns an empty list. You can only use one method of tracking the rejected rows at a time.

6. When you are finished adding streams, call VerticaCopyStreamObject.finish() to send any remaining streams to the database and close the COPY statement.
7. Call Connection.commit() to commit the loaded data.

Getting rejected rows

The VerticaCopyStreamObject.getRejects() method returns a List containing the row numbers of rows that were rejected after the previous .execute() method call. Each call to .execute() clears the list of rejected rows, so you need to call .getRejects() after each call to .execute() . Since .start() and .finish() also call .execute() to send any pending streams to the server, you should also call .getRejects() after these methods as well.

The following example demonstrates loading the content of five text files stored on the client system into a table.

```
import java.io.File;
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.util.Iterator;
import java.util.List;
import java.util.Properties;
import com.vertica.jdbc.VerticaConnection;
import com.vertica.jdbc.VerticaCopyStream;

public class CopyMultipleStreamsExample {
    public static void main(String[] args) {
        // Note: If running on Java 5, you need to call Class.forName
        // to manually load the JDBC driver.
        // Set up the properties of the connection
```

```
Properties myProp = new Properties();
myProp.put("user", "ExampleUser"); // Must be superuser
myProp.put("password", "password123");
// When performing bulk loads, you should always disable the
// connection's AutoCommit property to ensure the loads happen as
// efficiently as possible by reusing the same COPY command and
// transaction.
myProp.put("AutoCommit", "false");
Connection conn;
try {
    conn = DriverManager.getConnection(
                "jdbc:vertica://VerticaHost:5433/ExampleDB", myProp);
    Statement stmt = conn.createStatement();

    // Create a table to receive the data
    stmt.execute("DROP TABLE IF EXISTS customers");
    stmt.execute("CREATE TABLE customers (Last_Name char(50), "
                 + "First_Name char(50),Email char(50), "
                 + "Phone_Number char(15))");

    // Prepare the query to insert from a stream. This query must use
    // the COPY statement to load data from STDIN. Unlike copying from
    // a file on the host, you do not need superuser privileges to
    // copy a stream. All your user account needs is INSERT privileges
    // on the target table.
    String copyQuery = "COPY customers FROM STDIN "
                 + "DELIMITER '|' ENFORCELENGTH";

    // Create an instance of the stream class. Pass in the
    // connection and the query string.
    VerticaCopyStream stream = new VerticaCopyStream(
                (VerticaConnection) conn, copyQuery);

    // Keep running count of the number of rejects
    int totalRejects = 0;

    // start() starts the stream process, and opens the COPY command.
    stream.start();

    // If you added streams to VerticaCopyStream before calling start(),
    // You should check for rejects here (see below). The start() method
    // calls execute() to send any pre-queued streams to the server
    // once the COPY statement has been created.

    // Simple for loop to load 5 text files named customers-1.txt to
    // customers-5.txt
    for (int loadNum = 1; loadNum <= 5; loadNum++) {
        // Prepare the input file stream. Read from a local file.
        String filename = "C:\\Data\\customers-" + loadNum + ".txt";
        System.out.println("\n\nLoading file: " + filename);
        File inputFile = new File(filename);
        FileInputStream inputStream = new FileInputStream(inputFile);

        // Add stream to the VerticaCopyStream
        stream.addStream(inputStream);

        // call execute() to load the newly added stream. You could
        // add many streams and call execute once to load them all.
        // Which method you choose depends mainly on whether you want
        // the ability to check the number of rejections as the load
        // progresses so you can stop if the number of rejects gets too
```

```
            // high. Also, high numbers of InputStreams could create a
            // resource issue on your client system.
            stream.execute();

            // Show any rejects from this execution of the stream load
            // getRejects() returns a List containing the
            // row numbers of rejected rows.
            List<Long> rejects = stream.getRejects();

            // The size of the list gives you the number of rejected rows.
            int numRejects = rejects.size();
            totalRejects += numRejects;
            System.out.println("Number of rows rejected in load #"
                               + loadNum + ": " + numRejects);

            // List all of the rows that were rejected.
            Iterator<Long> rejit = rejects.iterator();
            long linecount = 0;
            while (rejit.hasNext()) {
                System.out.print("Rejected row #" + ++linecount);
                System.out.println(" is row " + rejit.next());
            }
        }
        // Finish closes the COPY command. It returns the number of
        // rows inserted.
        long results = stream.finish();
        System.out.println("Finish returned " + results);

        // If you added any streams that hadn't been executed(),
        // you should also check for rejects here, since finish()
        // calls execute() to

        // You can also get the number of rows inserted using
        // getRowCount().
        System.out.println("Number of rows accepted: "
                           + stream.getRowCount());
        System.out.println("Total number of rows rejected: " + totalRejects);

        // Commit the loaded data
        conn.commit();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Running the above example on some sample data results in the following output:

```
Loading file: C:\Data\customers-1.txtNumber of rows rejected in load #1: 3
Rejected row #1 is row 3
Rejected row #2 is row 7
Rejected row #3 is row 51
Loading file: C:\Data\customers-2.txt
Number of rows rejected in load #2: 5Rejected row #1 is row 4143
Rejected row #2 is row 6132
Rejected row #3 is row 9998
Rejected row #4 is row 10000
Rejected row #5 is row 10050
Loading file: C:\Data\customers-3.txt
Number of rows rejected in load #3: 9
Rejected row #1 is row 14142
Rejected row #2 is row 16131
Rejected row #3 is row 19999
Rejected row #4 is row 20001
Rejected row #5 is row 20005
Rejected row #6 is row 20049
Rejected row #7 is row 20056
Rejected row #8 is row 20144
Rejected row #9 is row 20236
Loading file: C:\Data\customers-4.txt
Number of rows rejected in load #4: 8
Rejected row #1 is row 23774
Rejected row #2 is row 24141
Rejected row #3 is row 25906
Rejected row #4 is row 26130
Rejected row #5 is row 27317
Rejected row #6 is row 28121
Rejected row #7 is row 29321
Rejected row #8 is row 29998
Loading file: C:\Data\customers-5.txt
Number of rows rejected in load #5: 1
Rejected row #1 is row 39997
Finish returned 39995
Number of rows accepted: 39995
Total number of rows rejected: 26
```

> **Note**
> The above example shows a simple load process that targets one node in the Vertica cluster. It is more efficient to simultaneously load multiple streams to multiple database nodes. Doing so greatly improves performance because it spreads the processing for the load across the cluster.

Using COPY LOCAL with JDBC

To use COPY LOCAL with JDBC, just execute a COPY LOCAL statement with the path to the source file on the client system. This method is simpler than using the VerticaCopyStream class (details on the class are available in the JDBC documentation . However, you may prefer using VerticaCopyStream if you have many files to copy to the database or if your data comes from a source other than a file (streamed over a network connection, for example).

You can use COPY LOCAL in a multiple-statement query. However, you should always make it the first statement in the query. You should not use it multiple times in the same query.

The following example code demonstrates using COPY LOCAL to copy a file from the client to the database. It is the same as the code shown in Bulk loading using the COPY statement , except for the use of the LOCAL option in the COPY statement, and the path to the data file is on the client system, rather than on the server.

> **Note**
> The exceptions/rejections files are created on the client machine when the exceptions and rejected data modifiers are specified on the copy local command. Specify a local path and filename for these modifiers when executing a COPY LOCAL query from the driver.

```
import java.sql.*;
import java.util.Properties;

public class COPYLocal {
   public static void main(String[] args) {
      // Note: If using Java 5, you must call Class.forName to load the
      // JDBC driver.
      Properties myProp = new Properties();
      myProp.put("user", "ExampleUser"); // Do not need to superuser
      myProp.put("password", "password123");
      Connection conn;
      try {
         conn = DriverManager.getConnection(
                     "jdbc:vertica://VerticaHost:5433/ExampleDB",myProp);
         // Disable AutoCommit
         conn.setAutoCommit(false);
         Statement stmt = conn.createStatement();
         // Create a table to hold data.
         stmt.execute("DROP TABLE IF EXISTS customers;");
         stmt.execute("CREATE TABLE IF NOT EXISTS customers (Last_Name char(50) "
                     + "NOT NULL, First_Name char(50),Email char(50), "
                     + "Phone_Number char(15))");

         // Use the COPY command to load data. Load directly into ROS, since
         // this load could be over 100MB. Use ENFORCELENGTH to reject
         // strings too wide for their columns.
         boolean result = stmt.execute("COPY customers FROM LOCAL "
                     + " 'C:\\Data\\customers.txt' DIRECT ENFORCELENGTH");

         // Determine if execution returned a count value, or a full result
         // set.
         if (result) {
            System.out.println("Got result set");
         } else {
            // Count will usually return the count of rows inserted.
            System.out.println("Got count");
            int rowCount = stmt.getUpdateCount();
            System.out.println("Number of accepted rows = " + rowCount);
         }

         conn.close();
      } catch (SQLException e) {
         System.out.print("Error: ");
         System.out.println(e.toString());
      }
   }
```

The result of running this code appears below. In this case, the customers.txt file contains 10000 rows, seven of which get rejected because they contain data too wide to fit into their database columns.

```
Got countNumber of accepted rows = 9993
```

Handling errors

When the Vertica JDBC driver encounters an error, it throws a SQLException or one of its subclasses. The specific subclass it throws depends on the type of error that has occurred. Most of the JDBC method calls can result in several different types of errors, in response to which the JDBC driver throws a specific SQLException subclass. Your client application can choose how to react to the error based on the specific exception that the JDBC driver threw.

> **Note**
> The specific SQLException subclasses were introduced in the JDBC 4.0 standard. If your client application runs in a Java 5 JVM, it will use the older JDBC 3.0-compliant driver which lacks these subclasses. In that case, all errors throw a SQLException .

The hierarchy of SQLException subclasses is arranged to help your client application determine what actions it can take in response to an error condition. For example:

- The JDBC driver throws SQLTransientException subclasses when the cause of the error may be a temporary condition, such as a timeout error (SQLTimeoutException) or a connection issue (SQLTransientConnectionIssue). Your client application can choose to retry the operation without making any sort of attempt to remedy the error, since it may not reoccur.
- The JDBC driver throws SQLNonTransientException subclasses when the client needs to take some action before it could retry the operation. For example, executing a statement with a SQL syntax error results in the JDBC driver throwing the a SQLSyntaxErrorException (a subclass of SQLNonTransientException). Often, your client application just has to report these errors back to the user and have him or her resolve them. For example, if the user supplied your application with a SQL statement that triggered a SQLSyntaxErrorException , it could prompt the user to fix the SQL error.

See SQLState mapping to Java exception classes for a list Java exceptions thrown by the JDBC driver.

In this section

- SQLState mapping to Java exception classes

SQLState mapping to Java exception classes

SQLSTATE Class or Value	Description	Java Exception Class
Class 00	Successful Completion	SQLException
Class 01	Warning	SQLException
Class 02	No Data	SQLException
Class 03	SQL Statement Not Yet Complete	SQLException
Class 08	Client Connection Exception	SQLNonTransientConnectionException
Class 09	Triggered Action Exception	SQLException
Class 0A	Feature Not Supported	SQLFeatureNotSupportedException
Class 0B	Invalid Transaction Initiation	SQLException
Class 0F	Locator Exception	SQLException
Class 0L	Invalid Grantor	SQLException
Class 0P	Invalid Role Specification	SQLException
Class 20	Case Not Found	SQLException
Class 21	Cardinality Violation	SQLException
Class 22	Data Exception	SQLDataException
22V21	INVALID_EPOCH	SQLNonTransientException
Class 23	Integrity Constraint Violation	SQLIntegrityConstraintViolationException

Class 24	Invalid Cursor State	SQLException
Class 25	Invalid Transaction State	SQLTransactionRollbackException
Class 26	Invalid SQL Statement Name	SQLException
Class 27	Triggered Data Change Violation	SQLException
Class 28	Invalid Authorization Specification	SQLInvalidAuthorizationException
Class 2B	Dependent Privilege Descriptors Still Exist	SQLDataException
Class 2D	Invalid Transaction Termination	SQLException
Class 2F	SQL Routine Exception	SQLException
Class 34	Invalid Cursor Name	SQLException
Class 38	External Routine Exception	SQLException
Class 39	External Routine Invocation Exception	SQLException
Class 3B	Savepoint Exception	SQLException
Class 3D	Invalid Catalog Name	SQLException
Class 3F	Invalid Schema Name	SQLException
Class 40	Transaction Rollback	SQLTransactionRollbackException
Class 42	Syntax Error or Access Rule Violation	SQLClientSyntaxErrorException
Class 44	WITH CHECK OPTION Violation	SQLException
Class 53	Insufficient Resources	SQLTransientException
53300	TOO_MANY_CONNECTIONS	SQLTransientConnectionException
Class 54	Program Limit Exceeded	SQLNonTransientException
Class 55	Object Not In Prerequisite State	SQLNonTransientException
55V03	LOCK_NOT_AVAILABLE	SQLTransactionRollbackException
Class 57	Operator Intervention	SQLTransientException
57V01	ADMIN_SHUTDOWN	SQLNonTransientConnectionException
57V02	CRASH_SHUTDOWN	SQLNonTransientConnectionException
57V03	CANNOT_CONNECT_NOW	SQLNonTransientConnectionException
Class 58	System Error	SQLException
Class V0	PL/vSQL errors	SQLException
Class V1	Vertica-specific multi-node errors class	SQLException

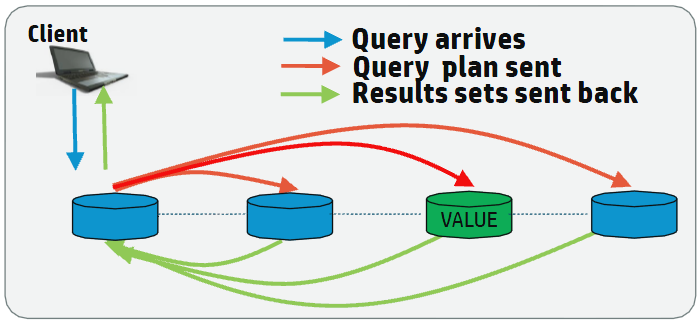
Class V2	Vertica-specific miscellaneous errors class	SQLException
V2000	AUTH_FAILED	SQLInvalidAuthorizationException
Class VC	Configuration File Error	SQLNonTransientException
Class VD	DB Designer errors	SQLNonTransientException
Class VP	User procedure errors	SQLNonTransientException
Class VX	Internal Error	SQLException

Routing JDBC queries directly to a single node

The JDBC driver has the ability to route queries directly to a single node using a special connection called a Routable Connection. This feature is ideal for high-volume "short" requests that return a small number of results that all exist on a single node. The common scenario for using this feature is to do high-volume lookups on data that is identified with a unique key. Routable queries typically provide lower latency and use less system resources than distributed queries. However, the data being queried must be segmented in such a way that the JDBC client can determine on which node the data resides.

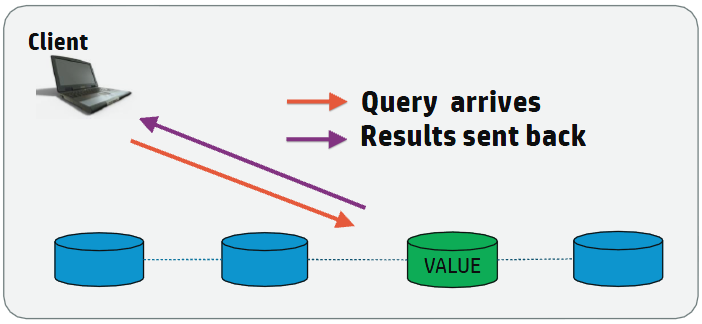
Vertica Typical Analytic Query

Typical analytic queries require dense computation on data across all nodes in the cluster and benefit from having all nodes involved in the planning and execution of the queries.



Vertica Routable Query API Query

For high-volume queries that return a single or a few rows of data, it is more efficient to execute the query on the single node that contains the data.



To effectively route a request to a single node, the client must determine the specific node on which the data resides. For the client to be able to determine the correct node, the table must be segmented by one or more columns. For example, if you segment a table on a Primary Key (PK) column, then the client can determine on which node the data resides based on the Primary Key and directly connect to that node to quickly fulfill the request.

The Routable Query API provides two classes for performing routable queries: VerticaRoutableExecutor and VGet. VerticaRoutableExecutor provides a more expressive SQL-based API while VGet provides a more structured API for programmatic access.

- The **VerticaRoutableExecutor** class allows you to use traditional SQL with a reduced feature set to query data on a single node.
  For joins, the table must be joined on a key column that exists in each table you are joining, and the tables must be segmented on that key. However, this is not true for unsegmented tables, which can always be joined (since all the data in an unsegmented table is available on all nodes).

- The **VGet** class does not use traditional SQL syntax. Instead, it uses a data structure that you build by defining predicates and predicate expressions and outputs and output expressions. This class is ideal for doing Key/Value type lookups on single tables.
  The data structure used for querying the table must provide a predicate for each segmented column defined in the projection for the table. You must provide, at a minimum, a predicate with a constant value for each segmented column. For example, an id with a value of 12234 if the table is segmented only on the id column. You can also specify additional predicates for the other, non-segmented, columns in the table. Predicates act like a SQL *WHERE* clause and multiple predicates/predicate expressions apply together with a SQL AND modifier. Predicates must be defined with a constant value. Predicate expressions can be used to refine the query and can contain any arbitrary SQL expressions (such as less than, greater than, and so on) for any of the non-segmented columns in the table.

Java documentation for all classes and methods in the JDBC Driver is available in the Vertica JDBC documentation .

> **Note**
> The JDBC Routable Query API is read-only and requires JDK 1.6 or greater.

In this section

- Creating tables and projections for use with the routable query API
- Creating a connection for routable queries
- Defining the query for routable queries using the VerticaRoutableExecutor class
- Defining the query for routable queries using the VGet class
- Routable query performance and troubleshooting
- Pre-segmenting data using VHash

Creating tables and projections for use with the routable query API

For routable queries, the client must determine the appropriate node to get the data. The client does this by comparing all projections available for the table, and determining the best projection to use to find the single node that contains data. You must create a projection segmented by the key column(s) on at least one table to take full advantage of the routable query API. Other tables that join to this table must either have an unsegmented projection, or a projection segmented as described below.

> **Note**
> Tables must be segmented by hash for routable queries. See Hash segmentation clause . Other segmentation types are not supported.

Creating tables for use with routable queries

To create a table that can be used with the routable query API, segment (by hash) the table on a uniformly distributed column. Typically, you segment on a primary key. For faster lookups, sort the projection on the same columns on which you segmented. For example, to create a table that is well suited to routable queries:

```
CREATE TABLE users (
id INT NOT NULL PRIMARY KEY,
username VARCHAR(32),
email VARCHAR(64),
business_unit VARCHAR(16))
ORDER BY id
SEGMENTED BY HASH(id)
ALL NODES;
```

This table is segmented based on the id column (and ordered by id to make lookups faster). To build a query for this table using the routable query API, you only need to provide a single predicate for the id column which returns a single row when queried.

However, you might add multiple columns to the segmentation clause. For example:

```
CREATE TABLE users2 (
   id INT NOT NULL PRIMARY KEY,
   username VARCHAR(32),
   email VARCHAR(64),
   business_unit VARCHAR(16))
ORDER BY id, business_unit
SEGMENTED BY HASH(id, business_unit)
ALL NODES;
```

In this case, you need to provide two predicates when querying the users2 table, as it is segmented on two columns, id and business_unit . However, if you know both id and business_unit when you perform the queries, then it is beneficial to segment on both columns, as it makes it easier for the client to determine that this projection is the best projection to use to determine the correct node.

Designing tables for single-node JOINs

If you plan to use the VerticaRoutableExecutor class and join tables during routable queries, then you must segment all tables being joined by the same segmentation key. Typically this key is a primary/foreign key on all the tables being joined. For example, the customer_key may be the primary key in a customers dimension table, and the same key is a foreign key in a sales fact table. Projections for a VerticaRoutableExecutor query using these tables must be segmented by hash on the customer key in each table.

If you want to join with small dimension tables, such as date dimensions, then it may be appropriate to make those tables unsegmented so that the date_dimension data exists on all nodes. It is important to note that when joining unsegmented tables, you still must specify a segmented table in the createRoutableExecutor() call.

Verifying existing projections for tables

If tables are already segmented by hash (for example, on an ID column), then you can determine what predicates are needed to query the table by using the Vertica function GET_PROJECTIONS to view that table's projections. For example:

```
=> SELECT GET_PROJECTIONS ('users');
...
Projection Name: [Segmented] [Seg Cols] [# of Buddies] [Buddy Projections] [Safe] [UptoDate] [Stats]
-------------------------------------------------------------------------------------------------------
public.users_b1 [Segmented: Yes] [Seg Cols: "public.users.id"] [K: 1] [public.users_b0] [Safe: Yes] [UptoDate: Yes] [Stats: RowCounts]
public.users_b0 [Segmented: Yes] [Seg Cols: "public.users.id"] [K: 1] [public.users_b1] [Safe: Yes] [UptoDate: Yes] [Stats: RowCounts]
```

For each projection, only the public.users.id column is specified, indicating your query predicate should include this column.

If the table is segmented on multiple columns, for example id and business_unit , then you need to provide both columns as predicates to the routable query.

Creating a connection for routable queries

The JDBC Routable Query API provides the VerticaRoutableConnection (details are available in the JDBC documentation interface to connect to a cluster and allow for Routable Queries. This interface provides advanced routing capabilities beyond those of a normal VerticaConnection. The VerticaRoutableConnection provides access to the VerticaRoutableExecutor and VGet classes. See Defining the query for routable queries using the VerticaRoutableExecutor class and Defining the query for routable queries using the VGet class respectively.

You enable access to this class by setting the EnableRoutableQueries JDBC connection property to true.

The VerticaRoutableConnection maintains an internal pool of connections and a cache of table metadata that is shared by all VerticaRoutableExecutor/VGet objects that are produced by the connection's createRoutableExecutor()/prepareGet() method. It is also a fully-fledged JDBC connection on its own and supports all the functionality that a VerticaConnection supports. When this connection is closed, all pooled connections managed by this VerticaRoutableConnection and all child objects are closed too. The connection pool and metadata is only used by child Routable Query operations.

Example:

You can create the connection using a JDBC DataSource:

```
com.vertica.jdbc.DataSource jdbcSettings = new com.vertica.jdbc.DataSource();
jdbcSettings.setDatabase("exampleDB");
jdbcSettings.setHost("v_vmart_node0001.example.com");
jdbcSettings.setUserID("dbadmin");
jdbcSettings.setPassword("password");
jdbcSettings.setEnableRoutableQueries(true);
jdbcSettings.setPort((short) 5433);

VerticaRoutableConnection conn;
conn = (VerticaRoutableConnection)jdbcSettings.getConnection();
```

You can also create the connection using a connection string and the DriverManager.getConnection() method:

```
String connectionString = "jdbc:vertica://v_vmart_node0001.example.com:5433/exampleDB?user=dbadmin&password=&EnableRoutableQueries=true";
VerticaRoutableConnection conn = (VerticaRoutableConnection) DriverManager.getConnection(connectionString);
```

Both methods result in a conn connection object that is identical.

> **Note**
> Avoid opening many VerticaRoutableConnection connections because this connection maintains its own private pool of connections which are not shared with other connections. Instead, your application should use a single connection and issue multiple queries through that connection.

In addition to the setEnableRoutableQueries property that the Routable Query API adds to the Vertica JDBC connection class, the API also adds additional properties. The complete list is below.

- EnableRoutableQueries : Enables Routable Query lookup capability. Default is false.
- FailOnMultiNodePlans : If the plan requires more than one node, and FailOnMultiNodePlans is true, then the query fails. If it is set to false then a warning is generated and the query continues. However, latency is greatly increased as the Routable Query must first determine the data is on multiple nodes, then a normal query is run using traditional (all node) execution and execution. Defaults to true. Note that this failure cannot occur on simple calls using only predicates and constant values.
- MetadataCacheLifetime : The time in seconds to keep projection metadata. The API caches metadata about the projection used for the query (such as projections). The cache is used on subsequent queries to reduce response time. The default is 300 seconds.
- MaxPooledConnections : Cluster-wide maximum number of connections to keep in the VerticaRoutableConnection's internal pool. Default 20.
- MaxPooledConnectionsPerNode : Per-node maximum number of connections to keep in the VerticaRoutableConnection's internal pool. Default 5.

Defining the query for routable queries using the VerticaRoutableExecutor class

Use the VerticaRoutableExecutor class to access table data directly from a single node. VerticaRoutableExecutor directly queries Vertica only on the node that has all the data needed for the query, avoiding the distributed planning and execution costs associated with Vertica query execution. You can use VerticaRoutableExecutor to join tables or use a GROUP BY clause, as these operations are not possible using VGet.

When using the VerticaRoutableExecutor class, the following rules apply:

- If joining tables, all tables being joined must be segmented (by hash) on the same set of columns referenced in the join predicate, unless the table to join is unsegmented.
- Multiple conditions in a join WHERE clause must be AND'd together. Using OR in the WHERE clause causes the query to degenerate to a multi-node plan. You can specify OR, IN list, or range conditions on columns *outside* the join condition if the data exists on the same node.
- You can only execute a single statement per request. Chained SQL statements are not permitted.
- Your query can be used in a driver-generated subquery to help determine whether the query can execute on a single node. Therefore, you cannot include the semi-colon at the end of the statement and you cannot include SQL comments using double-dashes (--), as these cause the driver-generated query to fail.

You create a VerticaRoutableExecutor by calling the createRoutableExecutor method on a connection object:

```
createRoutableExecutor( schema-name, table-name )
```

For example:

```
VerticaRoutableConnection conn;
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("customer_key", 1);
try {
  conn = (VerticaRoutableConnection)
    jdbcSettings.getConnection();
  String table = "customers";
  VerticaRoutableExecutor q = conn.createRoutableExecutor(null, table);
  ...
...
```

If *schema-name* is set to null, then the search path is used to find the table.

VerticaRoutableExecutor methods

VerticaRoutableExecutor has the following methods:

- execute
- close
- getWarnings

For details on this class, see the JDBC documentation .

Execute

```
execute( query-string, { column, value | map } )
```

Runs the query.

query-string	The query to execute

column, *value*	The column and value when the lookup is done on a single value. For example:
	`String column = "customer_key";`
	`Integer value = 1;`
	`ResultSet rs = q.execute(query, column, value)`

map	A Java map of the column names and corresponding values if the lookup is done on one or more columns. For example: ResultSet rs = q.execute(query, map); . The table must have at least one projection segmented by a set of columns that exactly match the columns in the map. Each column defined in the map can have only one value. For example:
	`Map<String, Object> map = new HashMap<String, Object>();`
	`map.put("customer_key", 1);`
	`map.put("another_key", 42);`
	`ResultSet rs = q.execute(query, map);`

The following requirements apply:

- The query to execute must use regular SQL that complies with the rules of the VerticaRoutableExecutor class. For example, you can add limits and sorts, or use aggregate functions, provided the data exists on a single node.
- The JDBC client uses the *column* / *value* or *map* arguments to determine on which node to execute the query. The content of the query must use the same values that you provide in the column/value or map arguments.
- The following data types cannot be used as column values: * INTERVAL * TIMETZ * TIMESTAMPTZ
  Also, if a table is segmented on any columns with the following data types then the table cannot be queried with the routable query API:

The driver does not verify the syntax of the query before it sends the query to the server. If your expression is incorrect, then the query fails.

Close

```
close()
```

Closes this VerticaRoutableExecutor by releasing resources used by this VerticaRoutableExecutor . It does not close the parent JDBC connection to Vertica.

getWarnings

```
getWarnings()
```

Retrieves the first warning reported by calls on this VerticaRoutableExecutor . Additional warnings are chained and can be accessed with the JDBC method getNextWarning() .

Example

The following example shows how to use VerticaRoutableExecutor to execute a query using both a JOIN clause and an aggregate function with a GROUP BY clause. The example also shows how to create a customer and sales table, and segment the tables so they can be joined using the VerticaRoutableExecutor class. This example uses the date_dimension table in the VMart schema to show how to join data on unsegmented tables.

1. Create the customers table to store customer details, and then create projections that are segmented on the table's customer_key column:

```
=> CREATE TABLE customers (customer_key INT, customer_name VARCHAR(128), customer_email VARCHAR(128));
=> CREATE PROJECTION cust_proj_b0 AS SELECT * FROM customers SEGMENTED BY HASH (customer_key) ALL NODES;
=> CREATE PROJECTION cust_proj_b1 AS SELECT * FROM customers SEGMENTED BY HASH (customer_key) ALL NODES OFFSET 1;
=> CREATE PROJECTION cust_proj_b2 AS SELECT * FROM customers SEGMENTED BY HASH (customer_key) ALL NODES OFFSET 2;
=> SELECT start_refresh();
```

2. Create the sales table, then create projections that are segmented on its customer_key column. Because the customer and sales tables are segmented on the same key, you can join them later with the VerticaRoutableExecutor routable query lookup.

```
=> CREATE TABLE sales (sale_key INT, customer_key INT, date_key INT, sales_amount FLOAT);
=> CREATE PROJECTION sales_proj_b0 AS SELECT * FROM sales SEGMENTED BY HASH (customer_key) ALL NODES;
=> CREATE PROJECTION sales_proj_b1 AS SELECT * FROM sales SEGMENTED BY HASH (customer_key) ALL NODES OFFSET 1;
=> CREATE PROJECTION sales_proj_b2 AS SELECT * FROM sales SEGMENTED BY HASH (customer_key) ALL NODES OFFSET 2;
=> SELECT start_refresh();
```

3. Add some sample data:

```
=> INSERT INTO customers VALUES (1, 'Fred', 'fred@example.com');
=> INSERT INTO customers VALUES (2, 'Sue', 'Sue@example.com');
=> INSERT INTO customers VALUES (3, 'Dave', 'Dave@example.com');
=> INSERT INTO customers VALUES (4, 'Ann', 'Ann@example.com');
=> INSERT INTO customers VALUES (5, 'Jamie', 'Jamie@example.com');
=> COMMIT;

=> INSERT INTO sales VALUES(1, 1, 1, '100.00');
=> INSERT INTO sales VALUES(2, 2, 2, '200.00');
=> INSERT INTO sales VALUES(3, 3, 3, '300.00');
=> INSERT INTO sales VALUES(4, 4, 4, '400.00');
=> INSERT INTO sales VALUES(5, 5, 5, '400.00');
=> INSERT INTO sales VALUES(6, 1, 15, '500.00');
=> INSERT INTO sales VALUES(7, 1, 15, '400.00');
=> INSERT INTO sales VALUES(8, 1, 35, '300.00');
=> INSERT INTO sales VALUES(9, 1, 35, '200.00');
=> COMMIT;
```

4. Create an unsegmented projection of the VMart date_dimension table for use in this example. Call the meta-function START_REFRESH to unsegment the existing data:

```
=> CREATE PROJECTION date_dim AS SELECT * FROM date_dimension UNSEGMENTED ALL NODES;
=> SELECT start_refresh();
```

Using the customer , sales , and date_dimension data, you can now create a routable query lookup that uses joins and a group by to query the customers table and return the total number of purchases per day for a given customer:

```
import java.sql.*;
import java.util.HashMap;
import java.util.Map;
import com.vertica.jdbc.kv.*;

public class verticaKV_doc {
   public static void main(String[] args) {
      com.vertica.jdbc.DataSource jdbcSettings
         = new com.vertica.jdbc.DataSource();
      jdbcSettings.setDatabase("VMart");
      jdbcSettings.setHost("vertica.example.com");
      jdbcSettings.setUserID("dbadmin");
      jdbcSettings.setPassword("password");
      jdbcSettings.setEnableRoutableQueries(true);
      jdbcSettings.setFailOnMultiNodePlans(true);
      jdbcSettings.setPort((short) 5433);
      VerticaRoutableConnection conn;
            Map<String, Object> map = new HashMap<String, Object>();
            map.put("customer_key", 1);
      try {
         conn = (VerticaRoutableConnection)
            jdbcSettings.getConnection();
         String table = "customers";
         VerticaRoutableExecutor q = conn.createRoutableExecutor(null, table);
         String query = "select d.date, SUM(s.sales_amount) as Total ";
            query += " from customers as c";
            query += " join sales as s ";
            query += " on s.customer_key = c.customer_key ";
            query += " join date_dimension as d ";
            query += " on d.date_key = s.date_key ";
            query += " where c.customer_key = " + map.get("customer_key");
            query += " group by (d.date) order by Total DESC";
         ResultSet rs = q.execute(query, map);
         while(rs.next()) {
            System.out.print("Date: " + rs.getString("date") + ":  ");
            System.out.println("Amount: " + rs.getString("Total"));
         }
         conn.close();
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
```

The example code produces output like this:

```
Date: 2012-01-15:  Amount: 900.0
Date: 2012-02-04:  Amount: 500.0
Date: 2012-01-01:  Amount: 100.0
```

> **Note**
> Your output might be different, because the VMart schema randomly generates dates in the date_dimension table.

Defining the query for routable queries using the VGet class

The VGet class is used to access table data directly from a single node when you do not need to join the data or use a group by clause. Like VerticaRoutableExecutor, VGet directly queries Vertica nodes that have the data needed for the query, avoiding the distributed planning and execution costs associated with a normal Vertica execution. However, VGet does not use SQL. Instead, you define predicates and values to perform key/value

type lookups on a single table. VGet is especially suited to key/value-type lookups on single tables.

You create a VGet by calling the prepareGet method on a connection object:

```
prepareGet( schema-name, { table-name | projection-name } )
```

For example:

```
VerticaRoutableConnection conn;
try {
   conn = (VerticaRoutableConnection)
      jdbcSettings.getConnection();
      System.out.println("Connected.");
      VGet get = conn.prepareGet("public", "users");
         ...
...
```

VGet operations span multiple JDBC connections (and multiple Vertica sessions) and do not honor the parent connection's transaction semantics. If consistency is required across multiple executions, the parent VerticaRoutableConnection's consistent read API can be used to guarantee all operations occur at the same epoch.

VGet is thread safe, but all methods are synchronized, so threads that share a VGet instance are never run in parallel. For better parallelism, each thread should have its own VGet instance. Different VGet instances that operate on the same table share pooled connections and metadata in a manner that enables a high degree of parallelism.

VGet methods

VGet has the following methods:

- VGet methods
  - addPredicate
  - addPredicateExpression
  - addOutputColumn
  - addOutputExpression
  - addSortColumn
  - setLimit
  - clearPredicates
  - clearOutputs
  - clearSortColumns
  - Execute
  - Close
  - getWarnings
- Example

By default, VGet fetches all columns of all rows that satisfy the logical AND of predicates passed via the addPredicate method. You can further customize the get operation with the following methods: addOutputColumn , addOutputExpression , addPredicateExpression , addSortColumn , and setLimit .

addPredicate

```
addPredicate(string, object)
```

Adds a predicate column and a constant value to the query. You must include a predicate for each column on which the table is segmented. The predicate acts as the query WHERE clause. Multiple addPredicate method calls are joined by AND modifiers. The VGet retains this value after each call to execute. To remove it, use clearPredicates .

The following data types cannot be used as column values. Also, if a table is segmented on any columns with these data types then the table cannot be queried with the Routable Query API:

- INTERVAL
- TIMETZ
- TIMESTAMPTZ

addPredicateExpression

addPredicateExpression(*string*)

Accepts arbitrary SQL expressions that operate on the table's columns as input to the query. Predicate expressions and predicates are joined by AND modifiers. You can use segmented columns in predicate expressions, but they must also be specified as a regular predicate with addPredicate . The VGet retains this value after each call to execute. To remove it, use clearPredicates .

The driver does not verify the syntax of the expression before it sends it to the server. If your expression is incorrect then the query fails.

addOutputColumn

```
addOutputColumn(string)
```

Adds a column to be included in the output. By default the query runs as SELECT * and you do not need to define any output columns to return the data. If you add output columns then you must add all the columns to be returned. The VGet retains this value after each call to execute. To remove it, use clearOutputs .

addOutputExpression

```
addOutputExpression(string)
```

Accepts arbitrary SQL expressions that operate on the table's columns as output. The VGet retains this value after each call to execute. To remove it, use ClearOutputs.

The following restrictions apply:

- The driver does not verify the syntax of the expression before it sends it to the server. If your expression is incorrect then the query fails.
- addOutputExpression is not supported when querying flex tables . If you use addOutputExpression on a flex table query, then a *SQLFeatureNotSupportedException* is thrown.

addSortColumn

```
addSortColumn(string, SortOrder)
```

Adds a sort order to an output column. The output column can be either the one returned by the default query (SELECT *) or one of the columns defined in addSortColumn or addOutputExpress. You can defined multiple sort columns.

setLimit

```
setLimit(int)
```

Sets a limit on the number of results returned. A limit of 0 is unlimited.

clearPredicates

```
clearPredicates()
```

Removes predicates that were added by addPredicate and addPredicateExpression .

clearOutputs

```
clearOutputs()
```

Removes outputs added by addOutputColumn and addOutputExpression .

clearSortColumns

```
clearSortColumns()
```

Removes sort columns previously added by addSortColumn .

Execute

```
execute()
```

Runs the query. Care must be taken to ensure that the predicate columns exist on the table and projection used by VGet, and that the expressions do not require multiple nodes to execute. If an expression is sufficiently complex as to require more than one node to execute, execute throws a SQLException if the FailOnMultiNodePlans connection property is true.

Close

```
close()
```

Closes this VGet by releasing resources used by this VGet. It does not close the parent JDBC connection to Vertica.

getWarnings

```
getWarnings()
```

Retrieves the first warning reported by calls on this VGet. Additional warnings are chained and can be accessed with the JDBC method getNextWarning.

Example

The following code queries the users table that is defined in Creating tables and projections for use with the routable query API . The table defines an id column that is segmented by hash.

```
import java.sql.*;
import com.vertica.jdbc.kv.*;

public class verticaKV2 {
  public static void main(String[] args) {
    com.vertica.jdbc.DataSource jdbcSettings
      = new com.vertica.jdbc.DataSource();
    jdbcSettings.setDatabase("exampleDB");
    jdbcSettings.setHost("v_vmart_node0001.example.com");
    jdbcSettings.setUserID("dbadmin");
    jdbcSettings.setPassword("password");
    jdbcSettings.setEnableRoutableQueries(true);
    jdbcSettings.setPort((short) 5433);

    VerticaRoutableConnection conn;
    try {
      conn = (VerticaRoutableConnection)
        jdbcSettings.getConnection();
        System.out.println("Connected.");
      VGet get = conn.prepareGet("public", "users");
      get.addPredicate("id", 5);
      ResultSet rs = get.execute();
      rs.next();
      System.out.println("ID: " +
        rs.getString("id"));
      System.out.println("Username: "
        + rs.getString("username"));
      System.out.println("Email: "
        + rs.getString("email"));
      System.out.println("Closing Connection.");
      conn.close();
    } catch (SQLException e) {
      System.out.println("Error! Stacktrace:");
      e.printStackTrace();
    }
  }
```

This code produces the following output:

```
Connected.
ID: 5
Username: userE
Email: usere@example.com
Closing Connection.
```

Routable query performance and troubleshooting

This topic details performance considerations and common issues you might encounter when using the routable query API.

Using resource pools with routable queries

Individual routable queries are serviced quickly since they directly access a single node and return only one or a few rows of data. However, by default, Vertica resource pools use an AUTO setting for the execution parallelism parameter. When set to AUTO, the setting is determined by the number of CPU cores available and generally results in multi-threaded execution of queries in the resource pool. It is not efficient to create parallel threads on

the server because routable query operations return data so quickly and routable query operations only use a single thread to find a row. To prevent the server from opening unneeded processing threads, you should create a specific resource pool for routable query clients. Consider the following settings for the resource pool you use for routable queries:

- Set execution parallelism to 1 to force single-threaded queries. This setting improves routable query performance.
- Use CPU affinity to limit the resource pool to a specific CPU or CPU set. The setting ensures that the routable queries have resources available to them, but it also prevents routable queries from significantly impacting performance on the system for other general queries.
- If you do not set a CPU affinity for the resource pool, consider setting the maximum concurrency value of the resource pool to a setting that ensures good performance for routable queries, but does not negatively impact the performance of general queries.

Performance considerations for routable query connections

Because a VerticaRoutableConnection opens an internal pool of connections, it is important to configure MaxPooledConnections and MaxPooledConnectionsPerNode appropriately for your cluster size and the amount of simultaneous client connections. It is possible to impact normal database connections if you are overloading the cluster with VerticaRoutableConnection s.

The initial connection to the initiator node discovers all other nodes in the cluster. The internal-pool connections are not opened until a VerticaRoutableExecutor or VGet query is sent. All VerticaRoutableExecutors/VGets in a connection object use connections from the internal pool and are limited by the MaxPooledConnections settings. Connections remain open until they are closed so a new connection can be opened elsewhere if the connection limit has been reached.

Troubleshooting routable queries

Routable query issues generally fall into two categories:

- Not providing enough predicates.
- Queries having to span multiple nodes.

Predicate Requirements

You must provide the same number of predicates that correspond to the columns of the table segmented by hash. To determine the segmented columns, call the Vertica function GET_PROJECTIONS . You must provide a predicate for each column displayed in the Seg Cols field.

For VGet, this means you must use addPredicate() to add each of the columns. For VerticaRoutableExecutor, this means you must provide all of the predicates and values in the map sent to execute() .

Multi-node Failures

It is possible to define the correct number of predicates, but still have a failure because multiple nodes contain the data. This failure occurs because the projection's data is not segmented in such a way that the data being queried is contained on a single node. Enable logging for the connection and view the logs to verify the projection being used. If the client is not picking the correct projection, then try to query the projection directly by specifying the projection instead of the table in the create/prepare statement, for example:

- Using VerticaRoutableExecutor:
```
conn.createRoutableExecutor(schema, table/projection);
```
- Using VGet:
```
conn.prepareGet("schema"."table/projection")
```

Additionally, you can use the EXPLAIN command in vsql to help determine if your query can run in single node. EXPLAIN can help you understand why the query is being run as single or multi-node.

Pre-segmenting data using VHash

The VHash class is an implementation of the Vertica hash function for use with JDBC client applications.

Hash segmentation in Vertica allows you to segment a projection based on a built-in hash function. The built-in hash function provides even data distribution across some or all nodes in a cluster, resulting in optimal query execution.

Suppose you have several million rows of values spread across thousands of CSV files. Assume that you already have a table segmented by hash. Before you load the values into your database, you probably want to know to which node a particular value loads. For this reason, using VHash can be particularly helpful, by allowing you to pre-segment your data before loading.

The following example shows the VHash class hashing the first column of a file named "testFile.csv". The name of the first column in this file is *meterId* .

Segment the data using VHash

This example demonstrates how you can read the testFile.csv file from the local file system and run a hash function on the meteterId column. Using the database metadata from a projection, you can then pre-segment the individual rows in the file based on the hash value of meterId.

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.UnsupportedEncodingException;
import java.util.*;
import java.io.IOException;
import java.sql.*;

import com.vertica.jdbc.kv.VHash;

public class VerticaKVDoc {

   final Map<String, FileOutputStream> files;
   final Map<String, List<Long>> nodeToHashList;
   String segmentationMetadata;
   List<String> lines;

   public static void main(String[] args) throws Exception {
      try {
         Class.forName("com.vertica.jdbc.Driver");
      } catch (ClassNotFoundException e) {
         System.err.println("Could not find the JDBC driver class.");
         e.printStackTrace();
         return;
      }

      Properties myProp = new Properties();
      myProp.put("user", "username");
      myProp.put("password", "password");

      VerticaKVDoc ex = new VerticaKVDoc();

      // Read in the data from a CSV file.
      ex.readLinesFromFile("C:\\testFile.csv");

      try (Connection conn = DriverManager.getConnection(
            "jdbc:vertica://VerticaHost:portNumber/databaseName", myProp)) {

      // Compute the hashes and create FileOutputStreams.
      ex.prepareForHashing(conn);

      }

      // Write to files.
      ex.writeLinesToFiles();
   }

   public VerticaKVDoc() {
      files = new HashMap<String, FileOutputStream>();
      nodeToHashList = new HashMap<String, List<Long>>();
   }

   public void prepareForHashing(Connection conn) throws SQLException,
         FileNotFoundException {

      // Send a query to Vertica to return the projection segments.
      try (ResultSet rs = conn.createStatement().executeQuery(
            "SELECT get_projection_segments('public.projectionName')")) {
```

```
            "SELECT get_projection_segments('public.projectionName')")) {
        rs.next();
        segmentationMetadata = rs.getString(1);
    }

    // Initialize the data files.
    try (ResultSet rs = conn.createStatement().executeQuery(
            "SELECT node_name FROM nodes")) {
        while (rs.next()) {
            String node = rs.getString(1);
            files.put(node, new FileOutputStream(node + ".csv"));
        }
    }
}

public void writeLinesToFiles() throws UnsupportedEncodingException,
        IOException {
    for (String line : lines) {

        long hashedValue = VHash.hashLong(getMeterIdFromLine(line));

        // Write the row data to that node's data file.
        String node = VHash.getNodeFor(segmentationMetadata, hashedValue);

        FileOutputStream fos = files.get(node);
        fos.write(line.getBytes("UTF-8"));
    }
}

private long getMeterIdFromLine(String line) {

    // In our file, "meterId" is the name of the first column in the file.
    return Long.parseLong(line.split(",")[0]);
}

public void readLinesFromFile(String filename) throws IOException {
    lines = new ArrayList<String>();
    String line;
    try (BufferedReader reader = new BufferedReader(
            new FileReader(filename))) {
        while ((line = reader.readLine()) != null) {
            lines.add(line);
        }
    }
}
```

JavaScript

The open-source vertica-nodejs client driver lets you interact with your database with JavaScript. For details, see the vertica-nodejs package on npm .

Perl

Perl scripts can interact with Vertica using the Perl DBI module along with the DBD::ODBC database driver to interface to the Vertica ODBC driver.

Prerequisites

You must configure a Perl development environment before creating Perl client applications.

In this section

- Configuring a Perl development environment
- Connecting to Vertica using Perl

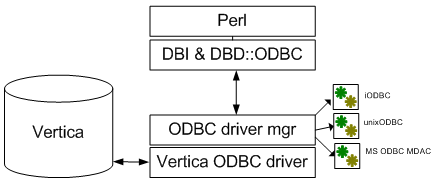
- Executing statements using Perl
- Batch loading data using Perl
- Using COPY LOCAL to load data in Perl
- Querying using Perl
- Conversions between Perl and Vertica data types
- Perl unicode support

Configuring a Perl development environment

Perl has a Database Interface module (DBI) that creates a standard interface for Perl scripts to interact with databases. The interface module relies on Database Driver modules (DBDs) to handle all of the database-specific communication tasks. The result is an interface that provides a consistent way for Perl scripts to interact with many different types of databases.

> **Note**
> With Perl ODBC clients, Vertica allows a forked process (a child process) to drop the parent connection to the Vertica server when the child process completes and exits. Vertica allows this behavior regardless of the setting of the Perl DBI AutoInactiveDestroy attribute.
>
> To change the default setting so that Vertica honors the setting of the Perl DBI AutoInactiveDestroy attribute, add the parameter CleanupInForkChild to your vertica.ini file, and set its value to 1. When the Perl DBI AutoInactiveDestroy attribute is set to 1, and the Vertica parameter CleanupInForkChild is set to 1, Vertica does not drop the parent connection upon child process completion.

Perl scripts can interact with Vertica using the Perl DBI module along with the DBD::ODBC database driver to interface to the Vertica ODBC driver. See the CPAN pages for Perl's DBI and DBD::ODBC modules for detailed documentation.



A Perl development environment depends on the Vertica ODBC driver and the DBI and DBD::ODBC modules.

1. Install and configure ODBC .
2. Verify that Perl is installed with the following command. If this command does not return version information, you must install Perl . For version support, see Perl driver requirements .
   ```
   $ perl -v
   ```
3. Install compatible versions of the Perl modules DBI and DBD::ODBC . Installation methods vary between environments. For details on installing Perl modules, see the cpan documentation .
4. Run the following commands to verify that DBI and DBD::ODBC are installed. If installed, these commands should return nothing. Otherwise, they return an error:
   ```
   $ perl -e "use DBI;"
   $ perl -e "use DBD::ODBC;"
   ```

Listing DSNs and verifying the installation

Another way to verify your installation is with the following Perl script. This script verifies if DBI and DBD::ODBC are installed and prints your ODBC DSN, if any:

```
#!/usr/bin/perl
use strict;
# Attempt to load the DBI module in an eval using require. Prevents
# script from erroring out if DBI is not installed.
eval
{
    require DBI;
    DBI->import();
};
if ($@) {
    # The eval failed, so DBI must not be installed
    print "DBI module is not installed\n";
} else {
    # Eval was successful, so DBI is installed
    print "DBI Module is installed\n";
    # List the drivers that DBI knows about.
    my @drivers = DBI->available_drivers;
    print "Available Drivers: \n";
    foreach my $driver (@drivers) {
        print "\t$driver\n";
    }
    # See if DBD::ODBC is installed by searching driver array.
    if (grep {/ODBC/i} @drivers) {
        print "\nDBD::ODBC is installed.\n";
        # List the ODBC data sources (DSNs) defined on the system
        print "Defined ODBC Data Sources:\n";
        my @dsns = DBI->data_sources('ODBC');
        foreach my $dsn (@dsns) {
            print "\t$dsn\n";
        }
    } else {
        print "DBD::ODBC is not installed\n";
    }
}
```

If your system is properly configured, the output should resemble the following:

```
DBI Module is installed
Available Drivers:
        ADO
        DBM
        ExampleP
        File
        Gofer
        ODBC
        Pg
        Proxy
        SQLite
        Sponge
        mysql
DBD::ODBC is installed.
Defined ODBC Data Sources:
        dbi:ODBC:dBASE Files
        dbi:ODBC:Excel Files
        dbi:ODBC:MS Access Database
        dbi:ODBC:VerticaDSN
```

Connecting to Vertica using Perl

You use the Perl DBI module's connect function to connect to Vertica. This function takes a required data source string argument and optional arguments for the username, password, and connection attributes.

The data source string must start with "dbi:ODBC:", which tells the DBI module to use the DBD::ODBC driver to connect to Vertica. The remainder of the string is interpreted by the DBD::ODBC driver. It usually contains the name of a DSN that contains the connection information needed to connect to your Vertica database. For example, to tell the DBD::ODBC driver to use the DSN named VerticaDSN, you use the data source string:

```
"dbi:ODBC:VerticaDSN"
```

The username and password parameters are optional. However, if you do not supply them (or just the username for a passwordless account) and they are not set in the DSN, attempting to connect always fails.

The connect function returns a database handle if it connects to Vertica. If it does not, it returns undef . In that case, you can access the DBI module's error string property ($DBI::errstr) to get the error message.

> **Note**
> By default, the DBI module prints an error message to STDERR whenever it encounters an error. If you prefer to display your own error messages or handle errors in some other manner, you may want to disable these automatic messages by setting DBI's PrintError connection attribute to false. See Setting Perl DBI connection attributes for details. Otherwise, users may see two error messages: the one that DBI prints automatically, and the one that your script prints on its own.

The following example connects to Vertica with a DSN named VerticaDSN. The call to connect supplies a username and password. After connecting, it calls the database handle's disconnect function, which closes the connection:

```
#!/usr/bin/perl -w
use strict;
use DBI;
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123");
unless (defined $dbh) {
   # Conection failed.
   die "Failed to connect: $DBI::errstr";
}
print "Connected!\n";
$dbh->disconnect();
```

In this section

- Setting ODBC connection parameters in Perl
- Setting Perl DBI connection attributes
- Connecting from Perl without a DSN

Setting ODBC connection parameters in Perl

To set ODBC connection parameters, replace the DSN name with a semicolon delimited list of parameter name and value pairs in the source data string. Use the DSN parameter to tell DBD::ODBC which DSN to use, then add in other the other ODBC parameters you want to set. For example, the following code connects using a DSN named VerticaDSN and sets the connection's locale to en_GB.

```
#!/usr/bin/perl -w
use strict;
use DBI;
# Instead of just using the DSN name, use name and value pairs.
my $dbh = DBI->connect("dbi:ODBC:DSN=VerticaDSN;Locale=en_GB@collation=binary","ExampleUser","password123");
unless (defined $dbh) {
   # Conection failed.
   die "Failed to connect: $DBI::errstr";
}
print "Connected!\n";
$dbh->disconnect();
```

See ODBC DSN connection properties for a list of the connection parameters you can set in the source data string.

Setting Perl DBI connection attributes

The Perl DBI module has attributes that you can use to control the behavior of its database connection. These attributes are similar to the ODBC connection parameters (in several cases, they duplicate each other's functionality). The DBI connection attributes are a cross-platform way of controlling the behavior of the database connection.

You can set the DBI connection attributes when establishing a connection by passing the DBI connect function a hash containing attribute and value pairs. For example, to set the DBI connection attribute AutoCommit to false, you would use:

```
# Create a hash that holds attributes for the connection
my $attr = {AutoCommit => 0};
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
   $attr);
```

See the DBI documentation's Database Handle Attributes section for a full description of the attributes you can set on the database connection.

After your script has connected, it can access and modify the connection attributes through the database handle by using it as a hash reference. For example:

```
print "The AutoCommit attribute is: " . $dbh->{AutoCommit} . "\n";
```

The following example demonstrates setting two connection attributes:

- RaiseError controls whether the DBI driver generates a Perl error if it encounters a database error. Usually, you set this to true (1) if you want your Perl script to exit if there is a database error.
- AutoCommit controls whether statements automatically commit their transactions when they complete. DBI defaults to Vertica's default AutoCommit value of true. Always set AutoCommit to false (0) when bulk loading data to increase database efficiency.

```
#!/usr/bin/perl
use strict;
use DBI;
# Create a hash that holds attributes for the connection
 my $attr = {
              RaiseError => 1, # Make database errors fatal to script
              AutoCommit => 0, # Prevent statements from committing
                               # their transactions.
           };
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
   $attr);

if (defined $dbh->err) {
   # Connection failed.
   die "Failed to connect: $DBI::errstr";
}
print "Connected!\n";
# The database handle lets you access the connection attributes directly:
print "The AutoCommit attribute is: " . $dbh->{AutoCommit} . "\n";
print "The RaiseError attribute is: " . $dbh->{RaiseError} . "\n";
# And you can change values, too...
$dbh->{AutoCommit} = 1;
print "The AutoCommit attribute is now: " . $dbh->{AutoCommit} . "\n";
$dbh->disconnect();
```

The example outputs the following when run:

```
Connected!The AutoCommit attribute is: 0
The RaiseError attribute is: 1
The AutoCommit attribute is now: 1
```

Connecting from Perl without a DSN

If you do not want to set up a Data Source Name (DSN) for your database, you can supply all of the information Perl's DBD::ODBC driver requires to connect to your Vertica database in the data source string. This source string must the DRIVER= parameter that tells DBD::ODBC which driver library to use in order to connect. The value for this parameter is the name assigned to the driver by the client system's driver manager:

- On Windows, the name assigned to the Vertica ODBC driver by the driver manager is Vertica.
- On Linux and other UNIX-like operating systems, the Vertica ODBC driver's name is assigned in the system's odbcinst.ini file. For example, if your /etc/odbcinst.ini contains the following:

  ```
  [Vertica]
  Description = Vertica ODBC Driver
  Driver = /opt/vertica/lib64/libverticaodbc.so
  ```

  you would use the name Vertica. See Creating an ODBC DSN for Linux for more information about the odbcinst.ini file.

You can take advantage of Perl's variable expansion within strings to use variables for most of the connection properties as the following example demonstrates.

```
#!/usr/bin/perl
use strict;
use DBI;
my $server='VerticaHost';
my $port = '5433';
my $database = 'VMart';
my $user = 'ExampleUser';
my $password = 'password123';
# Connect without a DSN by supplying all of the information for the connection.
# The DRIVER value on UNIX platforms depends on the entry in the odbcinst.ini
# file.
my $dbh = DBI->connect("dbi:ODBC:DRIVER={Vertica};Server=$server;" .
      "Port=$port;Database=$database;UID=$user;PWD=$password")
      or die "Could not connect to database: " . DBI::errstr;
print "Connected!\n";
$dbh->disconnect();
```

> **Note**
> Surrounding the driver name with braces ({ and }) in the source string is optional.

Executing statements using Perl

Once your Perl script has connected to Vertica (see Connecting to Vertica Using Perl), it can execute simple statements that return a value rather than a result set by using the Perl DBI module's do function. You usually use this function to execute DDL statements or data loading statements such as COPY (see Using COPY LOCAL to load data in Perl).

```
#!/usr/bin/perl
use strict;
use DBI;
# Disable autocommit
 my $attr = {AutoCommit => 0};
# Open a connection using a DSN.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
   $attr);
unless (defined $dbh) {
   # Conection failed.
   die "Failed to connect: $DBI::errstr";
}
# You can use the do function to perform DDL commands.
# Drop any existing table.
$dbh->do("DROP TABLE IF EXISTS TEST CASCADE;");
# Create a table to hold data.
$dbh->do("CREATE TABLE TEST( \
            C_ID  INT, \
            C_FP  FLOAT,\
            C_VARCHAR VARCHAR(100),\
            C_DATE DATE, C_TIME TIME,\
            C_TS TIMESTAMP,\
            C_BOOL BOOL)");
# Commit changes and exit.
$dbh->commit();
$dbh->disconnect();
```

Note

The do function returns the number of rows that were affected by the statement (or -1 if the count of rows doesn't apply or is unavailable). Usually, the only time you need to consult this value is after you deleted a number of rows or if you used a bulk load command such as COPY . You use other DBI functions instead of do to perform batch inserts and selects (see Batch loading data using Perl and Querying using Perl for details).

Batch loading data using Perl

To load large batches of data into Vertica using Perl:

1. Set DBI's AutoCommit connection attribute to false to improve the batch load speed. See Setting Perl DBI connection attributes for an example of disabling AutoCommit.
2. Call the database handle's prepare function to prepare a SQL INSERT statement that contains placeholders for the data values you want to insert. For example:

   ```
   # Prepare an INSERT statement for the test table
   $sth = $dbh->prepare("INSERT into test values(?,?,?,?,?,?,?)");
   ```

   The prepare function returns a statement handle that you will use to insert the data.
3. Assign data to the placeholders. There are several ways to do this. The easiest is to populate an array with a value for each placeholder in your INSERT statement.
4. Call the statement handle's execute function to insert a row of data into Vertica. The return value of this function call lets you know whether Vertica accepted or rejected the row.
5. Repeat steps 3 and 4 until you have loaded all of the data you need to load.
6. Call the database handle's commit function to commit the data you inserted.

The following example demonstrates inserting a small batch of data by populating an array of arrays with data, then looping through it and inserting each row.

```
#!/usr/bin/perl
use strict;
use DBI;
# Create a hash reference that holds a hash of parameters for the
# connection.
```

```
 my $attr = {AutoCommit => 0, # Turn off autocommit
           PrintError => 0   # Turn off automatic error printing.
                        # This is handled manually.
           };
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
    $attr);
if (defined DBI::err) {
    # Conection failed.
    die "Failed to connect: $DBI::errstr";
}
print "Connection AutoCommit state is: " . $dbh->{AutoCommit} . "\n";
# Create table to hold inserted data
$dbh->do("DROP TABLE IF EXISTS TEST CASCADE;") or die "Could not drop table";
$dbh->do("CREATE TABLE TEST( \
              C_ID  INT, \
              C_FP  FLOAT,\
              C_VARCHAR VARCHAR(100),\
              C_DATE DATE, C_TIME TIME,\
              C_TS TIMESTAMP,\
              C_BOOL BOOL)") or die "Could not create table";
# Populate an array of arrays with values. One of these rows contains
# data that will not be sucessfully inserted. Another contains an
# undef value, which gets inserted into the database as a NULL.
my @data = (
               [1,1.111,'Hello World!','2001-01-01','01:01:01'
                  ,'2001-01-01 01:01:01','t'],
               [2,2.22222,'How are you?','2002-02-02','02:02:02'
                  ,'2002-02-02 02:02:02','f'],
               ['bad value',2.22222,'How are you?','2002-02-02','02:02:02'
                  ,'2002-02-02 02:02:02','f'],
               [4,4.22222,undef,'2002-02-02','02:02:02'
                  ,'2002-02-02 02:02:02','f'],
            );
# Create a prepared statement to use parameters for inserting values.
my $sth = $dbh->prepare_cached("INSERT into test values(?,?,?,?,?,?,?)");
my $rowcount = 0; # Count # of rows
# Loop through the arrays to insert values
foreach my $tuple (@data) {
    $rowcount++;
    # Insert the row
    my $retval = $sth->execute(@$tuple);

    # See if the row was successfully inserted.
    if ($retval == 1) {
        # Value of 1 means the row was inserted (1 row was affected by insert)
        print "Row $rowcount successfully inserted\n";
    } else {
        print "Inserting row $rowcount failed";
        # Error message is not set on some platforms/versions of DBUI. Check to
        # ensure a message exists to avoid getting an unitialized var warning.
        if ($sth->err()) {
              print ": " . $sth->errstr();
        }
        print "\n";
    }
}
# Commit changes. With AutoCommit off, you need to use commit for batched
# data to actually be committed into the database. If your Perl script exits
# without committing its data, Vertica rolls back the transaction and the
# data is not committed.
```

```
$dbh->commit();
$dbh->disconnect();
```

The previous example displays the following when successfully run:

```
Connection AutoCommit state is: 0
Row 1 successfully inserted
Row 2 successfully inserted
Inserting row 3 failed with error 01000 [Vertica][VerticaDSII] (20) An
error occurred during query execution: Row rejected by server; see
server log for details (SQL-01000)
Row 4 successfully inserted
```

Note that one of the rows was not inserted because it contained a string value that could not be stored in an integer column. See Conversions between Perl and Vertica Data Types for details of data type handling in Perl scripts that communicate with Vertica.

Using COPY LOCAL to load data in Perl

You can use COPY LOCAL to load delimited files on your client system—for example, a file with comma-separated values—into Vertica. Rather than use Perl to read, parse, and then batch insert the file data, COPY LOCAL directly loads the file data from the local file system into Vertica. When execution completes, COPY LOCAL returns the number of rows that it successfully inserted.

> **Note**
> COPY LOCAL must be the first statement in a query,otherwise Vertica returns an error.

The following example uses COPY LOCAL to load into Vertica local file data.txt , which is located in the same directory as the Perl file.

```
#!/usr/bin/perl
use strict;
use DBI;
# Filesystem path handling module
use File::Spec;
# Create a hash reference that holds a hash of parameters for the
# connection.
 my $attr = {AutoCommit => 0}; # Turn off AutoCommit
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
    $attr) or die "Failed to connect: $DBI::errstr";
print "Connected!\n";
# Drop any existing table.
$dbh->do("DROP TABLE IF EXISTS Customers CASCADE;");
# Create a table to hold data.
$dbh->do("CREATE TABLE Customers( \
            ID  INT, \
            FirstName  VARCHAR(100),\
            LastName   VARCHAR(100),\
            Email      VARCHAR(100),\
            Birthday   DATE)");
# Find the absolute path to the data file located in the current working
# directory and named data.txt
my $currDir = File::Spec->rel2abs(File::Spec->curdir());
my $dataFile = File::Spec->catfile($currDir, 'data.txt');
print "Loading file $dataFile\n";
# Load local file using copy local. Return value is the # of rows affected
# which equates to the number of rows inserted.
my $rows = $dbh->do("COPY Customers FROM LOCAL '$dataFile' DIRECT")
    or die $dbh->errstr;
print "Copied $rows rows into database.\n";
$dbh->commit();
# Prepare a query to get the first 15 rows of the results
my $sth = $dbh->prepare("SELECT * FROM Customers WHERE ID < 15 \
                         ORDER BY ID");

$sth->execute() or die "Error querying table: " . $dbh->errstr;
my @row; # Pre-declare variable to hold result row used in format statement.
# Use Perl formats to pretty print the output. Declare the heading for the
# form.
format STDOUT_TOP =
ID  First          Last         EMail                    Birthday
==  =====          ====         =====                    ========
.
# The Perl write statement will output a formatted line with values from the
# @row array. See http://perldoc.perl.org/perlform.html for details.
format STDOUT =
@>  @<<<<<<<<<<<<<  @<<<<<<<<<<<  @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<  @<<<<<<<<<<
@row
.
# Loop through result rows while we have them
while (@row = $sth->fetchrow_array()) {
        write; # Format command does the work of extracting the columns from
               # the @row array and writing them out to STDOUT.
}
# Call commit to prevent Perl from complaining about uncommitted transactions
# when disconnecting
$dbh->commit();
$dbh->disconnect();
```

data.txt is a text file with a row of data on each line. The columns are delimited by pipe (|) characters. This is the default COPY delimiter for command accepts, which simplifies the COPY LOCAL statement.

Here is an example of the file content:

```
1|Georgia|Gomez|Rhiannon@magna.us|1937-10-03
2|Abdul|Alexander|Kathleen@ipsum.gov|1941-03-10
3|Nigel|Contreras|Tanner@et.com|1955-06-01
4|Gray|Holt|Thomas@Integer.us|1945-12-06
5|Candace|Bullock|Scott@vitae.gov|1932-05-27
6|Matthew|Dotson|Keith@Cras.com|1956-09-30
7|Haviva|Hopper|Morgan@porttitor.edu|1975-05-10
8|Stewart|Sweeney|Rhonda@lectus.us|2003-06-20
9|Allen|Rogers|Alexander@enim.gov|2006-06-17
10|Trevor|Dillon|Eagan@id.org|1988-11-27
11|Leroy|Ashley|Carter@turpis.edu|1958-07-25
12|Elmo|Malone|Carla@enim.edu|1978-08-29
13|Laurel|Ball|Zelenia@Integer.us|1989-09-20
14|Zeus|Phillips|Branden@blandit.gov|1996-08-08
15|Alexis|Mclean|Flavia@Suspendisse.org|2008-01-07
```

The example code produces the following output when run on a large sample file:

```
Connected!
Loading file /home/dbadmin/Perl/data.txt
Copied 1000000 rows into database.
ID  First          Last         EMail                    Birthday
==  =====          ====         =====                    ========
 1  Georgia        Gomez        Rhiannon@magna.us          1937-10-03
 2  Abdul          Alexander    Kathleen@ipsum.gov         1941-03-10
 3  Nigel          Contreras    Tanner@et.com              1955-06-01
 4  Gray           Holt         Thomas@Integer.us          1945-12-06
 5  Candace        Bullock      Scott@vitae.gov            1932-05-27
 6  Matthew        Dotson       Keith@Cras.com             1956-09-30
 7  Haviva         Hopper       Morgan@porttitor.edu       1975-05-10
 8  Stewart        Sweeney      Rhonda@lectus.us           2003-06-20
 9  Allen          Rogers       Alexander@enim.gov         2006-06-17
10  Trevor         Dillon       Eagan@id.org               1988-11-27
11  Leroy          Ashley       Carter@turpis.edu          1958-07-25
12  Elmo           Malone       Carla@enim.edu             1978-08-29
13  Laurel         Ball         Zelenia@Integer.us         1989-09-20
14  Zeus           Phillips     Branden@blandit.gov        1996-08-08
```

> **Note**
> Loading a single, large data file into Vertica through a single data connection is less efficient than loading a number of smaller files onto multiple nodes in parallel. Loading onto multiple nodes prevents any one node from becoming a bottleneck.

Querying using Perl

To query Vertica using Perl:

1. Prepare a query statement using the Perl DBI module's prepare function. This function returns a statement handle that you use to execute the query and get the result set.
2. Execute the prepared statement by calling the execute function on the statement handle.
3. Retrieve the results of the query from the statement handle using one of several methods, such as calling the statement handle's fetchrow_array function to retrieve a row of data, or fetchall_array to get an array of arrays containing the entire result set (not a good idea if your result set may be very large!).

The following example demonstrates querying the table created by the example shown in Batch loading data using Perl . It executes a query to retrieve all of the content of the table, then repeatedly calls the statement handle's fetchrow_array function to get rows of data in an array. It repeats this process until fetchrow_array returns undef, which means that there are no more rows to be read.

```
#!/usr/bin/perl
use strict;
use DBI;
my $attr = {RaiseError => 1 }; # Make errors fatal to the Perl script.
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
                       $attr);
# Prepare a query to get the content of the table
my $sth = $dbh->prepare("SELECT * FROM TEST ORDER BY C_ID ASC");
# Execute the query by calling execute on the statement handle
$sth->execute();
# Loop through result rows while we have them, getting each row as an array
while (my @row = $sth->fetchrow_array()) {
   # The @row array contains the column values for this row of data
   # Loop through the column values
   foreach my $column (@row) {
      if (!defined $column) {
         # NULLs are signaled by undefs. Set to NULL for clarity
         $column = "NULL";
      }
      print "$column\t"; # Output the column separated by a tab
   }
   print "\n";
}
$dbh->disconnect();
```

The example prints the following when run:

```
1    1.111    Hello World!    2001-01-01    01:01:01    2001-01-01 01:01:01    1
2    2.22222    How are you?    2002-02-02    02:02:02    2002-02-02 02:02:02    0
4    4.22222    NULL    2002-02-02    02:02:02    2002-02-02 02:02:02    0
```

Binding variables to column values

Another method of retrieving the query results is to bind variables to columns in the result set using the statement handle's bind_columns function. You may find this method convenient if you need to perform extensive processing on the returned data, since your code can use variables rather than array references to access the data. The following example demonstrates binding variables to the result set, rather than looping through the row and column values.

```
#!/usr/bin/perl
use strict;
use DBI;
my $attr = {RaiseError => 1 }; # Make SQL errors fatal to the Perl script.
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN32","ExampleUser","password123",
                       $attr);
# Prepare a query to get the content of the table
my $sth = $dbh->prepare("SELECT * FROM TEST ORDER BY C_ID ASC");
$sth->execute();
# Create a set of variables to bind to the column values.
my ($C_ID, $C_FP, $C_VARCHAR, $C_DATE, $C_TIME, $C_TS, $C_BOOL);
# Bind the variable references to the columns in the result set.
$sth->bind_columns(\$C_ID, \$C_FP, \$C_VARCHAR, \$C_DATE, \$C_TIME,
                   \$C_TS, \$C_BOOL);

# Now, calling fetch() to get a row of data updates the values of the bound
# variables. Continue calling fetch until it returns undefined.
while ($sth->fetch()) {
   # Note, you should always check that values are defined before using them,
   # since NULL values are translated into Perl as undefined. For this
   # example, just check the VARCHAR column for undefined values.
   if (!defined $C_VARCHAR) {
      $C_VARCHAR = "NULL";
   }
   # Just print values separated by tabs.
   print "$C_ID\t$C_FP\t$C_VARCHAR\t$C_DATE\t$C_TIME\t$C_TS\t$C_BOOL\n";
}
$dbh->disconnect();
```

The output of this example is identical to the output of the previous example.

Preparing, querying, and returning a single row

If you expect a single row as the result of a query (for example, when you execute a COUNT (*) query), you can use the DBI module's selectrow_array function to combine executing a statement and retrieving an array as a result.

The following example shows using selectrow_array to execute and get the results of the SHOW LOCALE statement. It also demonstrates changing the locale using the do function.

```
#!/usr/bin/perl
use strict;
use DBI;
my $attr = {RaiseError => 1 }; # Make SQL errors fatal to the Perl script.
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
                       $attr);
# Demonstrate setting/getting locale.
# Use selectrow_array to combine preparing a statement, executing it, and
# getting an array as a result.
my @localerv = $dbh->selectrow_array("SHOW LOCALE;");
# The locale name is the 2nd column (array index 1) in the result set.
print "Locale: $localerv[1]\n";
# Use do() to execute a SQL statement to set the locale.
$dbh->do("SET LOCALE TO en_GB");
# Get the locale again.
@localerv = $dbh->selectrow_array("SHOW LOCALE;");
print "Locale is now: $localerv[1]\n";
$dbh->disconnect();
```

The result of running the example is:

```
Locale: en_US@collation=binary (LEN_KBINARY)
Locale is now: en_GB (LEN)
```

Executing queries and ResultBufferSize settings

When you call the execute() function on a prepared statement, the client library retrieves results up to the size of the result buffer. The result buffer size is set using ODBC's ResultBufferSize setting.

Vertica does not allow multiple active queries per connection. However, you can simulate multiple active queries by setting the result buffer to be large enough to accommodate the entire results from the first query. To ensure that the ODBC client driver's buffer is large enough to store result set for first query you can set ResultBufferSize to 0. Setting this parameter to 0 makes the result buffer size unlimited. The ODBC driver allocates enough memory to read the entire result set. With the entire result set from the first query stored in the result set buffer, the database connection is free to perform another query. Your client can execute this second query even though it has not processed the entire result set from the first query.

However, if you set the ResultBufferSize to 0, you may find that your calls to execute() result in the operating system killing your Perl client script. The operating system may terminate your script if the ODBC driver allocates too much memory to store a large result set.

A workaround for this behavior is limit the number of rows returned by your query. Then you can set the ResultBufferSize to a value that accommodates this limited result set. For example, you can estimate the amount of memory needed to store a single row of your query result. Then use the LIMIT and OFFSET clauses to get a specific number of rows that will fit into the space you allocated using ResultBufferSize. If the results of your query is able to fit within the limited result set buffer, you can then perform additional queries with the same database connection. This solution makes your code more complex as you will need to perform multiple queries to get the entire result set. Also, it is not appropriate in cases where you need to operate on an entire result set at once, rather than just a portion of it at a time.

A better solution is to use separate database connections for each query you want to perform. The overhead of the additional database connection is small compared to the resources needed to process large data sets.

Conversions between Perl and Vertica data types

Perl is a loosely-typed programming language that does not assign specific data types to values. It converts between string and numeric values based on the operations being performed on the values. For this reason, Perl has little problem extracting most string and numeric data types from Vertica. All interval data types (DATE, TIMESTAMP, etc.) are converted to strings. You can use several different date and time handling Perl modules to manipulate these values in your scripts.

Vertica NULL values translate to Perl's undefined (undef) value. When reading data from columns that can contain NULL values, you should always test whether a value is defined before using it.

When inserting data into Vertica, Perl's DBI module attempts to coerce the data into the correct format. By default, it assumes column values are VARCHAR unless it can determine that they are some other data type. If given a string value to insert into a column that has an integer or numeric data type, DBI attempts to convert the string's contents to the correct data type. If the entire string can be converted to a value of the appropriate data type, it inserts the value into the column. If not, inserting the row of data fails.

DBI transparently converts integer values into numeric or float values when inserting into column of FLOAT, NUMERIC, or similar data types. It converts numeric or floating values to integers only when there would be no loss of precision (the value to the right of the decimal point is 0). For example, it can insert the value 3.0 into an INTEGER column since there is no loss of precision when converting the value to an integer. It cannot insert 3.1 into an INTEGER column, since that would result in a loss of precision. It returns an error instead of truncating the value to 3.

The following example demonstrates some of the conversions that the DBI module performs when inserting data into Vertica.

```
#!/usr/bin/perl
use strict;
use DBI;
# Create a hash reference that holds a hash of parameters for the
# connection.
 my $attr = {AutoCommit => 0, # Turn off autocommit
             PrintError => 0   # Turn off print error. Manually handled
             };
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
    $attr);
if (defined DBI::err) {
    # Conection failed.
    die "Failed to connect: $DBI::errstr";
}
print "Connection AutoCommit state is: " . $dbh->{AutoCommit} . "\n";
```

```
# Create table to hold inserted data
$dbh->do("DROP TABLE IF EXISTS TEST CASCADE;");
$dbh->do("CREATE TABLE TEST( \
            C_ID  INT, \
            C_FP  FLOAT,\
            C_VARCHAR VARCHAR(100),\
            C_DATE DATE, C_TIME TIME,\
            C_TS TIMESTAMP,\
            C_BOOL BOOL)");
# Populate an array of arrays with values.
my @data = (
             # Start with matching data types
             [1,1.111,'Matching datatypes','2001-01-01','01:01:01'
                ,'2001-01-01 01:01:01','t'],
             # Force floats -> int and int -> float.
             [2.0,2,"Ints <-> floats",'2002-02-02','02:02:02'
                ,'2002-02-02 02:02:02',1],
             # Float -> int *only* works when there is no loss of precision.
             # this row will fail to insert:
             [3.1,3,"float -> int with trunc?",'2003-03-03','03:03:03'
                ,'2003-03-03 03:03:03',1],
             # String values are converted into numbers
             ["4","4.4","Strings -> numbers", '2004-04-04','04:04:04',
                ,'2004-04-04 04:04:04',0],
             # String -> numbers only works if the entire string can be
             # converted into a number
             ["5 and a half","5.5","Strings -> numbers", '2005-05-05',
                '05:05:05', ,'2005-05-05 05:05:05',0],
             # Number are converted into string values automatically,
             # assuming they fit into the column width.
             [6,6.6,3.14159, '2006-06-06','06:06:06',
                ,'2006-06-06 06:06:06',0],
             # There are some variations in the accepted date strings
             [7,7.7,'Date/time formats', '07/07/2007','07:07:07',
                ,'07-07-2007 07:07:07',1],
           );
# Create a prepared statement to use parameters for inserting values.
my $sth = $dbh->prepare_cached("INSERT into test values(?,?,?,?,?,?,?)");
my $rowcount = 0; # Count # of rows
# Loop through the arrays to insert values
foreach my $tuple (@data) {
   $rowcount++;
   # Insert the row
   my $retval = $sth->execute(@$tuple);

   # See if the row was successfully inserted.
   if ($retval == 1) {
      # Value of 1 means the row was inserted (1 row was affected by insert)
      print "Row $rowcount successfully inserted\n";
   } else {
      print "Inserting row $rowcount failed with error " .
            $sth->state . " " . $sth->errstr . "\n";
   }
}
# Commit the data
$dbh->commit();
# Prepare a query to get the content of the table
$sth = $dbh->prepare("SELECT * FROM TEST ORDER BY C_ID ASC");
$sth->execute() or die "Error: " . $dbh->errstr;
my @row; # Need to pre-declare to use in the format statement.
# Use Perl formats to pretty print the output.
```

```
format STDOUT_TOP =
Int  Float       VarChar       Date      Time      Timestamp     Bool
===  =====  ==================  ==========  ========  ================  ====
.
format STDOUT =
@>>  @<<<<  @<<<<<<<<<<<<<<<<<  @<<<<<<<<<  @<<<<<<<  @<<<<<<<<<<<<<<<  @<<<<
@row
.
# Loop through result rows while we have them
while (@row = $sth->fetchrow_array()) {
        write; # Format command does the work of extracting the columns from
               # the array.
}
# Commit to stop Perl complaining about in-progress transactions.
$dbh->commit();
$dbh->disconnect();
```

The example produces the following output when run:

```
Connection AutoCommit state is: 0
Row 1 successfully inserted
Row 2 successfully inserted
Inserting row 3 failed with error 01000 [Vertica][VerticaDSII] (20) An error
occurred during query execution: Row rejected by server; see server log for
details (SQL-01000)
Row 4 successfully inserted
Inserting row 5 failed with error 01000 [Vertica][VerticaDSII] (20) An error
occurred during query execution: Row rejected by server; see server log for
details (SQL-01000)
Row 6 successfully inserted
Row 7 successfully inserted
Int  Float       VarChar       Date      Time      Timestamp     Bool
===  =====  ==================  ==========  ========  ================  ====
 1  1.111  Matching datatypes  2001-01-01 01:01:01 2001-01-01 01:01 1
 2  2      Ints <-> floats     2002-02-02 02:02:02 2002-02-02 02:02 1
 4  4.4    Strings -> numbers  2004-04-04 04:04:04 2004-04-04 04:04 0
 6  6.6    3.14159             2006-06-06 06:06:06 2006-06-06 06:06 0
 7  7.7    Date/time formats   2007-07-07 07:07:07 2007-07-07 07:07 1
```

Perl unicode support

Perl supports Unicode data with some caveats. See the perlunicode and the perlunitut (Perl Unicode tutorial) manual pages for details. (Be sure to see the copies of these manual pages included with the version of Perl installed on your client system, as the support for Unicode has changed in recent versions of Perl.) Perl DBI and DBD::ODBC also support Unicode, however DBD::ODBC must be compiled with Unicode support. See the DBD::ODBC documentation for details. You can check the DBD::ODBC-specific connection attribute named odbc_has_unicode to see if Unicode support is enabled in the driver.

The following example Perl script demonstrates directly inserting UTF-8 strings into Vertica and then reading them back. The example writes a text file with the output, since there are may problems displaying Unicode characters in terminal windows or consoles.

```
#!/usr/bin/perl
use strict;
use DBI;
# Open a connection using a DSN.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123");
unless (defined $dbh) {
   # Conection failed.
   die "Failed to connect: $DBI::errstr";
}
# Output to a file. Displaying Unicode characters to a console or terminal
# window has many problems. This outputs a UTF-8 text file that can
# be handled by many Unicode-aware text editors:
open OUTFILE, '>:utf8', "unicodeout.txt";
# See if the DBD::ODBC driver was compiled with Unicode support. If this returns
# 1, your Perl script will get get strings from the driver with the UTF-8
# flag set on them, ensuring that Perl handles them correctly.
print OUTFILE "Was DBD::ODBC compiled with Unicode support? " .
   $dbh->{odbc_has_unicode} . "\n";

# Create a table to hold VARCHARs
$dbh->do("DROP TABLE IF EXISTS TEST CASCADE;");

# Create a table to hold data. Remember that the width of the VARCHAR column
# is the number of bytes set aside to store strings, which often does not equal
# the number of characters it can hold when it comes to Unicode!
$dbh->do("CREATE TABLE test( C_VARCHAR VARCHAR(100) )");
print OUTFILE "Inserting data...\n";
# Use Do to perform simple inserts
$dbh->do("INSERT INTO test VALUES('Hello')");
# This string contains several non-latin accented characters and symbols, encoded
# with Unicode escape notation. They are converted by Perl into UTF-8 characters
$dbh->do("INSERT INTO test VALUES('My favorite band is " .
   "\N{U+00DC}ml\N{U+00E4}\N{U+00FC}t \N{U+00D6}v\N{U+00EB}rk\N{U+00EF}ll" .
   " \N{U+263A}')");
# Some Chinese (Simplified) characters. This again uses escape sequence
# that Perl translates into UTF-8 characters.
$dbh->do("INSERT INTO test VALUES('\x{4F60}\x{597D}')");
print OUTFILE "Getting data...\n";
# Prepare a query to get the content of the table
my $sth = $dbh->prepare_cached("SELECT * FROM test");
# Execute the query by calling execute on the statement handle
$sth->execute();
# Loop through result rows while we have them
while (my @row = $sth->fetchrow_array()) {
   # Loop through the column values
   foreach my $column (@row) {
      print OUTFILE "$column\t";
   }
   print OUTFILE "\n";
}
close OUTFILE;
$dbh->disconnect();
```

Viewing the unicodeout.txt file in a UTF-8-capable text editor or viewer displays:

```
Was DBD::ODBC compiled with Unicode support? 1
Inserting data...
Getting data...
My favorite band is Ümläüt Övërkïll ☺
你好
Hello
```

Note
Terminal windows and consoles often have problems properly displaying Unicode characters. That is why the example writes the output to a text file. With some text editors, you may need to manually set the encoding of the text file to UTF-8 in order for the characters to properly appear (and the font used to display text must have a full Unicode character set). If the character still do not show up, it may be that your version of DBD::ODBC was not compiled with UTF-8 support.

See also

- Locale and UTF-8 support
- ODBC driver settings

Python

The Vertica Python drivers provide an interface for Python client applications to interact with the database.

Prerequisites

You must configure a Python development environment before creating Python client applications.

Note
Both the Vertica Python client and Vertica ODBC driver (that pyodbc interacts with) do not support the native Vertica UUID data type. Values retrieved using these drivers from a UUID column are converted to strings. When your client queries the metadata for a UUID column, the drivers report its data type as a string. Convert any UUID values that you want to insert into a UUID column to strings. Vertica automatically converts these values into the native UUID data type before inserting them into a table.

In this section

- Configuring the ODBC run-time environment on Linux
- Querying the database with pyodbc

Configuring the ODBC run-time environment on Linux

To configure the ODBC run-time environment on Linux:

1. Create the odbc.ini file if it does not already exist.
2. Add the ODBC driver directory to the LD_LIBRARY_PATH system environment variable:
   ```
   export LD_LIBRARY_PATH=/path-to-vertica-odbc-driver:$LD_LIBRARY_PATH
   ```

   Important
   If you skip Step 2, the ODBC manager cannot find the driver in order to load it.

These steps are relevant only for unixODBC and iODBC. See their respective documentation for details on odbc.ini .

See also

- unixODBC Web site
- iODBC Web site

Querying the database with pyodbc

The example session below uses pyodbc with the Vertica ODBC driver to connect Python to the Vertica database.

Note
SQLFetchScroll and SQLFetch functions cannot be mixed together in iODBC code. When using pyodbc with the iODBC driver manager, skip cannot be used with the fetchall, fetchone, and fetchmany functions.

Example script

The following example script shows how to query Vertica using Python 3, pyodbc, and an ODBC DSN.

```
import pyodbc
cnxn = pyodbc.connect("DSN=VerticaDSN", ansi=True)
cursor = cnxn.cursor()
# create table
cursor.execute("CREATE TABLE TEST("
   "C_ID  INT,"
   "C_FP  FLOAT,"
   "C_VARCHAR VARCHAR(100),"
   "C_DATE DATE, C_TIME TIME,"
   "C_TS TIMESTAMP,"
   "C_BOOL BOOL)")
cursor.execute("INSERT INTO test VALUES(1,1.1,'abcdefg1234567890','1901-01-01','23:12:34','1901-01-01 09:00:09','t')")
cursor.execute("INSERT INTO test VALUES(2,3.4,'zxcasdqwe09876543','1991-11-11','00:00:01','1981-12-31 19:19:19','f')")
cursor.execute("SELECT * FROM TEST")
rows = cursor.fetchall()
for row in rows:
   print(row, end='\n')
cursor.execute("DROP TABLE TEST CASCADE")
cursor.close()
cnxn.close()
```

The resulting output displays:

```
(2, 3.4, 'zxcasdqwe09876543', datetime.date(1991, 11, 11), datetime.time(0, 0, 1), datetime.datetime(1981, 12, 31, 19, 19, 19), False)
(1, 1.1, 'abcdefg1234567890', datetime.date(1901, 1, 1), datetime.time(23, 12, 34), datetime.datetime(1901, 1, 1, 9, 0, 9), True)
```

Notes

SQLPrimaryKeys returns the table name in the primary (pk_name) column for unnamed primary constraints. For example:

- Unnamed primary key:
  ```
  CREATE TABLE schema.test(c INT PRIMARY KEY);

  SQLPrimaryKeys
  "TABLE_CAT", "TABLE_SCHEM", "TABLE_NAME", "COLUMN_NAME", "KEY_SEQ", "PK_NAME" <Null>, "SCHEMA", "TEST", "C", 1, "TEST"
  ```
- Named primary key:
  ```
  CREATE TABLE schema.test(c INT CONSTRAINT pk_1 PRIMARY KEY);

  SQLPrimaryKeys
  "TABLE_CAT", "TABLE_SCHEM", "TABLE_NAME", "COLUMN_NAME", "KEY_SEQ", "PK_NAME" <Null>, "SCHEMA", "TEST", "C", 1, "PK_1"
  ```

OpenText recommends that you name your constraints.

See also

- Loading data

PHP

> **Note**
> You must configure a PHP development environment before creating PHP client applications.

Setup

You must read C/C++ before connecting to Vertica through PHP. The following example ODBC configuration entries detail the typical settings required for PHP ODBC connections. The driver location assumes you have copied the Vertica drivers to /usr/lib64 .

Example odbc.ini

```
[ODBC Data Sources]
VerticaDSNunixodbc = exampledb
VerticaDNSiodbc = exampledb2
[VerticaDSNunixodbc]
Description = VerticaDSN Unix ODBC driver
Driver = /usr/lib64/libverticaodbc.so
Database = Telecom
Servername = localhost
UserName = dbadmin
Password =
Port = 5433
[VerticaDSNiodbc]
Description = VerticaDSN iODBC driver
Driver = /usr/lib64/libverticaodbc.so
Database = Telecom
Servername = localhost
UserName = dbadmin
Password =
Port = 5433
```

Example odbcinst.ini

```
# Vertica
[VerticaDSNunixodbc]
Description = VerticaDSN Unix ODBC driver
Driver = /usr/lib64/libverticaodbc.so
[VerticaDNSiodbc]
Description = VerticaDSN iODBC driver
Driver = /usr/lib64/libverticaodbc.so
[ODBC]
Threading = 1
```

Verify the Vertica UnixODBC or iODBC library

Verify the Vertica UnixODBC library can load all dependant libraries with the following command (assuming you have copies the libraries to /usr/lib64):

For example:

```
ldd /usr/lib64/libverticaodbc.so
```

You must resolve any "not found" libraries before continuing.

Test your ODBC connection

Test your ODBC connection with the following.

```
isql -v VerticaDSN
```

In this section

- Configuring a PHP development environment
- PHP unicode support
- Querying the database using PHP

Configuring a PHP development environment

To configure a PHP development environment:

1. Install and configure ODBC .
2. Install PHP.
3. Install the PDO and ODBC PHP extensions. On Linux, these are available as the following packages:
    - php-odbc
    - php-pdo

PHP unicode support

PHP does not offer native Unicode support. PHP only supports a 256-character set. However, PHP provides the UTF-8 functions utf8_encode() and utf8_decode() to provide some basic Unicode functionality.

See the PHP manual for strings for more details about PHP and Unicode.

Querying the database using PHP

The example script below details the use of PHP ODBC functions to connect to the Vertica Analytics Platform.

```
<?php
# Turn on error reporting
error_reporting(E_ERROR | E_WARNING | E_PARSE | E_NOTICE);
# A simple function to trap errors from queries
function odbc_exec_echo($conn, $sql) {
    if(!$rs = odbc_exec($conn,$sql)) {
        echo "<br/>Failed to execute SQL: $sql<br/>" . odbc_errormsg($conn);
    } else {
        echo "<br/>Success: " . $sql;
    }
    return $rs;
}
# Connect to the Database
$dsn = "VerticaDSNunixodbc";
$conn = odbc_connect($dsn,'','') or die ("<br/>CONNECTION ERROR");
echo "<p>Connected with DSN: $dsn</p>";
# Create a table
$sql = "CREATE TABLE TEST(
        C_ID INT,
        C_FP FLOAT,
        C_VARCHAR VARCHAR(100),
        C_DATE DATE, C_TIME TIME,
        C_TS TIMESTAMP,
        C_BOOL BOOL)";
$result = odbc_exec_echo($conn, $sql);
# Insert data into the table with a standard SQL statement
$sql = "INSERT into test values(1,1.1,'abcdefg1234567890','1901-01-01','23:12:34
','1901-01-01 09:00:09','t')";
$result = odbc_exec_echo($conn, $sql);
# Insert data into the table with odbc_prepare and odbc_execute
$values = array(2,2.28,'abcdefg1234567890','1901-01-01','23:12:34','1901-01-01 0
9:00:09','t');
$statement = odbc_prepare($conn,"INSERT into test values(?,?,?,?,?,?,?)");
if(!$result = odbc_execute($statement, $values)) {
            echo "<br/>odbc_execute Failed!";
} else {
            echo "<br/>Success: odbc_execute.";
}
# Get the data from the table and display it
$sql = "SELECT * FROM TEST";
if($result = odbc_exec_echo($conn, $sql)) {
    echo "<pre>";
    while($row = odbc_fetch_array($result) ) {
            print_r($row);
    }
    echo "</pre>";
}
# Drop the table and projection
$sql = "DROP TABLE TEST CASCADE";
$result = odbc_exec_echo($conn, $sql);
# Close the ODBC connection
odbc_close($conn);
?>
```

Example output

The following is the example output from the script.

```
Success: CREATE TABLE TEST( C_ID INT, C_FP FLOAT, C_VARCHAR VARCHAR(100), C_DATE DATE, C_TIME TIME, C_TS TIMESTAMP, C_BOOL BOOL)
Success: INSERT into test values(1,1.1,'abcdefg1234567890','1901-01-01','23:12:34 ','1901-01-01 09:00:09','t')
Success: odbc_execute.
Success: SELECT * FROM TEST
Array
(
    [C_ID] => 1
    [C_FP] => 1.1
    [C_VARCHAR] => abcdefg1234567890
    [C_DATE] => 1901-01-01
    [C_TIME] => 23:12:34
    [C_TS] => 1901-01-01 09:00:09
    [C_BOOL] => 1
)
Array
(
    [C_ID] => 2
    [C_FP] => 2.28
    [C_VARCHAR] => abcdefg1234567890
    [C_DATE] => 1901-01-01
    [C_TIME] => 23:12:34
    [C_TS] => 1901-01-01 23:12:34
    [C_BOOL] => 1
)
Success: DROP TABLE TEST CASCADE
```

Managing query execution between the client and Vertica

The following topics describe techniques that help you manage query execution between your client and your Vertica database.

In this section

- ResultBufferSize
- Multiple active result sets (MARS)

ResultBufferSize

By default, Vertica uses the ResultBufferSize parameter to determine the maximum size (in bytes) of a result set that a client can retrieve from a server. When ResultBufferSize is enabled, Vertica sends rows of data directly to the client making the query. The number of rows returned to the client at each fetch of data depends on the size (in bytes) of the ResultBufferSize parameter.

Sometimes, the size of the result set requested by the client is greater than what the ResultBufferSize parameter allows. In such cases, Vertica retrieves only a portion of the result set at a time. Each fetch of data returns the amount of data equal to the size set by the ResultBufferSize parameter. Ultimately, as the client iterates over the individual fetches of data, the entire result set is returned.

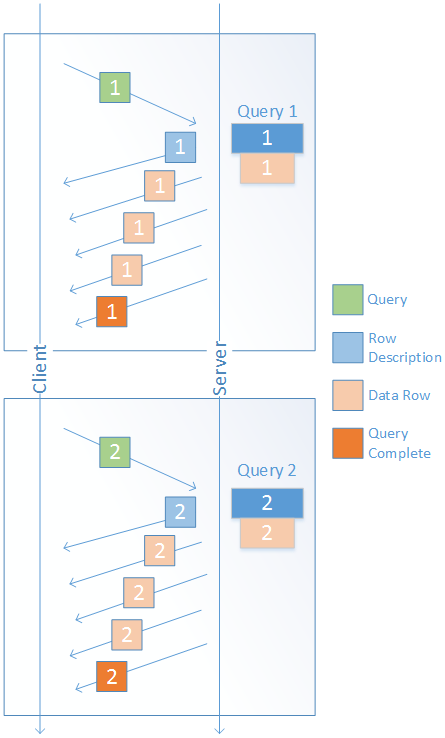
Benefits of ResultBufferSize

If you are concerned with the effect of your queries on network latency, ResultBufferSize may provide an advantage over MARS. MARS requires that the client wait until all rows of data are written to the server before the client can retrieve the data. This delay may cause latency issues for your network while waiting for the results to be stored.

In addition, MARS requires that you send two separate requests to return rows of data. The first request performs the query execution which stores the result set on the server. The second request retrieves the data rows that are stored on the server. With ResultBufferSize, you only need to send one request. This request both executes and retrieves the data rows of interest.

Query execution with ResultBufferSize

The following graphic shows how Vertica returns rows of data from a database to the client with ResultBufferSize enabled:



The query execution performs the following steps:

1. The client sends a query, such as a SELECT statement, to the server. In the preceding graphic, the first query is named Query 1.
2. The server receives the client's request and begins to send both a description of the result set and the requested rows of data back to the client.
3. After all possible rows are returned to the client, the execution is complete. The size of the data set returned equals either that of the data that was requested or the maximum amount of data that ResultBufferSize parameter can retrieve. If the ResultBufferSize maximum size is not yet reached, Vertica can execute Query 2.

The server can accept Query 2 and perform the same steps that it did for Query 1. If the results for Query 1 had reached the maximum ResultBufferSize allowable, Vertica could not execute Query 2 until the client freed the results from Query 1.

After Query 2 runs, you cannot view the results you retrieved for Query 1, unless you execute Query 1 again.

Setting an unlimited buffer size

Setting ResultBufferSize to 0 tells the client driver to use an unlimited result set buffer. With this setting, the client library allocates as much memory as it needs to read the entire result set of a query. You may choose to set ResultBufferSize to 0 of you want to simulate having multiple active queries over a single database connection at the same time. With an unlimited buffer size, your client can run a query and have its entire result set stored in memory. This ends the first query, so your client can execute a second query before it fully processes the results of the first query.

A drawback of this method is that your query may consume too much memory if your queries return large result sets. This over-allocation of memory can result in the operating system terminating your client. Due to this risk, consider using multiple database connections instead of trying to reuse a single connection for multiple queries. The overhead of multiple database connections is small compared to the overall amount of resources required to process a large data set.

Multiple active result sets (MARS)

You can only enable MARS when you connect to Vertica using a JDBC client connection. MARS allows the execution of multiple queries on a single connection. While ResultBufferSize sends the results of a query directly to the client, MARS stores the results first on the server. Once query execution has finished and all of the results have been stored, you can make a retrieval request to the server to have rows returned to the client.

MARS is set at the session level and must be enabled for every new session. When MARS is enabled, ResultBufferSize is disabled. No error is returned, however the ResultBufferSize parameter is ignored.

Benefits of MARS

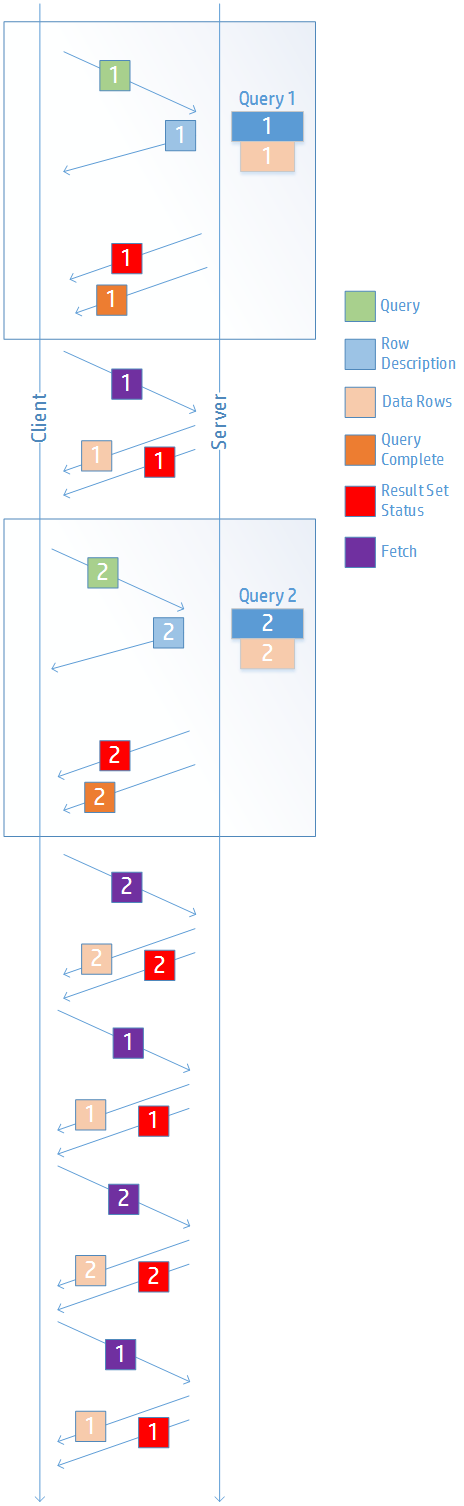
In comparison with ResultBufferSize, MARS enables you to store multiple result sets from different queries at the same time. You can also send new queries before all of the results of a previous result set have been returned to the client. This allows applications to decouple query execution from result retrieval so that, on a single connection, you can process different results at the same time.

When you enable ResultBufferSize, you must wait until all result sets have been returned to the client before a new query can be executed.

Another benefit of MARS is that it allows you to free up query resources faster than ResultBufferSize allows. While a query is running, resources are held by that query session. When ResultBufferSize is enabled, a client that is performing slowly might read a single row of a result set and then have to stop to retrieve the next row. This prevents the query from finishing quickly and, therefore, prevents the resources used from being freed up for other applications. With MARS, the speed of the client is irrelevant to the reading of rows. As soon as the results are written to the MARS storage, the resources are freed and the speed at which the client retrieves rows no longer matters.

Query execution with MARS

The following graphic demonstrates how multiple queries to the server are handled when MARS is enabled:



Query 1:

1. Query 1 is sent to the server.
2. Query 1's row description and the status of its result set are returned to the client. However, no results are returned to the client at this time.
3. Query 1 completes and its results are saved on the server.
    1. You can now send commands to retrieve the rows of Query 1's result set. These rows are stored on the server. Retrieved rows are sent to the client along with the status of the result set. By keeping track of the status of the result set, Vertica is able to keep track of which rows have been retrieved from the server.

4. Now that Query 1 has successfully completed, and its result sets are being stored on the server, Query 2 can be executed.

Query 2:

1. Query 2 is sent to the server.
2. Query 2's row description and the status of its result set are returned to the client. However, no results are returned to the client at this time.
3. Query 2 completes and its results are stored on the server. Both Query 1 and Query 2 now have result sets stored on the server.
4. You can now send retrieval requests to both Query 1 and Query 2's result sets that are stored on the server. Whenever a retrieval request is made for rows from Query 1, the request is sent and rows and the result set status are sent to the client. The same occurs for Query 2.

Once all rows have been read by the client, the MARS storage on the server closes the active results session. The MARS storage on the server is then freed to store more data. The MARS storage also closes and frees once your session is finished.

Enabling and disabling MARS

You can enable and disable MARS in two different ways:

1. To enable MARS using the JDBC client connection properties, see JDBC connection properties .
2. To enable MARS using the SET SESSION command, see SET SESSION MULTIPLEACTIVERESULTSETS .

See also

- SESSION_MARS_STORE
- CLOSE_RESULTSET
- CLOSE_ALL_RESULTSETS

Management API

The Management API is a REST API that you can use to view and manage Vertica databases with scripts or applications that accept REST and JSON. The response format for all requests is JSON.

In this section

- cURL
- General API information
- Rest APIs for the agent
- Rest APIs for the Management Console

cURL

cURL is a command-line tool and application library used to transfer data to or from a server. All API requests sent to a Vertica server must be made with HTTPS .

There are four HTTP requests that can be passed using cURL to call API methods:

- GET: Retrieves data.
- PUT: Updates data.
- POST: Creates new data.
- DELETE: Deletes data.

Syntax

```
curl https://<NODE>:5444/
```

Options

The following is a truncated list of options. For a complete list, see the cURL documentation .

-h --help	Lists all available options.
-H --header	Specifies custom headers. This is useful for sending a request that requires a Vertica API key. **Example:** `$ curl -H "VerticaApiKey: ValidAPIKey" https://<NODE>:5444/`

-k --insecure	Connects with TLS without validating the database's server certificate. **Example:** `$ curl -k https://<NODE>:5444/`
-X --request	Specifies a request type, one of the following: - GET (default) - PUT - POST - DELETE **Example:** `$ curl -X REQUEST https://<NODE>:5444/`

General API information

These API calls can interact with either standard Vertica nodes or Management Console nodes.

GET /	Returns the agent-specific information useful for version checking and service discovery.
GET api	Returns a list of api objects and properties.

In this section

- GET /
- GET api

GET /

Returns API version information and a list of links to child resources for the Management API.

Resource URL

```
https://<NODE>:5444/
```

Authentication

Not required.

Parameters

None.

Example request

GET	https://<NODE>:5444/

Response:


```
{
   "body": {
      "mime-types": [
         "default",
         "application/vertica.database.configuration.json-v2",
         "application/json",
         "application/vertica.nodes.json-v2",
         "default",
         "application/json",
         "default",
         "application/json",
         "application/vertica.jobs.json-v2",
         "default",
```

```
            "application/vertica.hosts.json-v2",
            "application/json",
            "default",
            "application/vertica.hosts.json-v2",
            "application/json",
            "default",
            "application/json",
            "application/vertica.host.json-v2",
            "default",
            "application/vertica.hosts.json-v2",
            "application/json",
            "application/vertica.nodes.json-v2",
            "default",
            "application/json",
            "default",
            "application/json",
            "application/vertica.database.json-v2",
            "default",
            "application/vertica.hosts.json-v2",
            "application/json",
            "default",
            "application/vertica.hosts.json-v2",
            "application/json",
            "default",
            "application/json",
            "application/vertica.databases.json-v2",
            "application/vertica.nodes.json-v2",
            "default",
            "application/json",
            "application/vertica.agent.json-v2",
            "default",
            "application/json",
            "default",
            "application/vertica.users.json-v2",
            "application/json"
        ],
        "version": "7.1.0"
    },
    "href": "/",
    "links": [
        "/databases",
        "/hosts",
        "/nodes",
        "/licenses",
        "/webhooks",
        "/backups",
        "/restore",
        "/jobs"
    ],
    "mime-type": "application/vertica.agent.json-v2"
}
```


GET api

Lists all Management API commands, with a brief description of each one and its parameters.

Resource URL

https://*node-ip-address*:5444/api

Authentication

None

Example

```
$ curl -k https://10.20.100.247:5444/api
[
  {
    "route": "/",
    "method": "GET",
    "description": "Returns the agent specific information useful for version checking and service discovery",
    "accepts": {},
    "params": []
  },
  {
    "route": "/api",
    "method": "GET",
    "description": "build the list of cluster objects and properties and return it as a JSON formatted array",
    "accepts": {},
    "params": []
  },
  {
    "route": "/backups",
    "method": "GET",
    "description": "list all the backups that have been created for all vbr configuration files ( *.ini ) that are located in the /opt/vertica/config directory.",
    "accepts": {},
    "params": []
  },
  {
    "route": "/backups/:config_script_base",
    "method": "POST",
    "description": "create a new backup as defined by the given vbr configuration script base (filename minus the .ini extenstion)",
    "accepts": {},
    "params": []
  },
  {
    "route": "/backups/:config_script_base/:archive_id",
    "method": "GET",
    "description": "get the  detail for a specific backup archive",
    "accepts": {},
    "params": []
  },
  {
    "route": "/backups/:config_script_base/:archive_id",
    "method": "DELETE",
    "description": "delete a backup based on the config ini file script",
    "accepts": {},
    "params": []
  },
  {
    "route": "/databases",
    "method": "GET",
    "description": "build the list of databases, their properties, and current status (from cache) and return it as a JSON formatted array",
    "accepts": {},
    "params": []
  },
  {
    "route": "/databases",
    "method": "POST",
    "description": "Create a new database by supplying a valid set of parameters",
    "accepts": {},
    "params": [
      "name    : name of the database to create",
      "passwd  : password used by the database administrative user",
      "only    : optional list of hostnames to include in database",
```

```
      "exclude : optional list of hostnames to exclude from the database",
      "catalog : directory used for the vertica catalog",
      "data    : directory used for the initial vertica storage location",
      "port    : port the database will listen on (default 5433)",
      "restart_policy : (optional) set restart policy",
      "force_cleanup_on_failure : (optional) Force removal of existing directories on failure of command",
      "force_removal_at_creation : (optional)  Force removal of existing directories before creating the database",
      "communal_storage_url : (optional) communal storage location for the database",
      "num_shards : (optional) number of shared for databases with communal storage",
      "depot_path : (optional, but if specified requires depot_size) path to a directory where files from communal storage can be locally cached",
      "depot_size : (optional, required by depot_path) size of the depot. Examples: (\"10G\", \"2000M\", \"1T\", \"250K\")",
      "aws_access_key_id: (optional)",
      "aws_secret_access_key : (optional)",
      "configuration_parameters : (optional) A string that is a serialized python-literal dictionary of configuration parameters set at bootstrap.
      '{\"kerberosservicename\":\"verticakerb\"}'"]
},
{
   "route": "/databases/:database_name",
   "method": "GET",
   "description": "Retrieve the database properties structure",
   "accepts": {},
   "params": []
},
{
   "route": "/databases/:database_name",
   "method": "PUT",
   "description": "Control / alter a database values using the PUT http method",
   "accepts": {},
   "params": ["action : value one of start|stop|rebalance|wla"]
},
{
   "route": "/databases/:database_name",
   "method": "DELETE",
   "description": "Delete an existing database",
   "accepts": {},
   "params": []
},
{
   "route": "/databases/:database_name/configuration",
   "method": "GET",
   "description": "retrieve the current parameters from the database. if its running return 503 Service Unavailable",
   "accepts": {},
   "params": [
      "user_id  : vertica database username",
      "passwd   : vertica database password"]
},
{
   "route": "/databases/:database_name/configuration",
   "method": "PUT",
   "description": "set a list of  parameters in the database. if its not running return 503 Service Unavailable",
   "accepts": {},
   "params": [
      "user_id   : vertica database username",
      "passwd    : vertica database password",
      "parameter : value  vertica parameter/key combo"]
},
...
{
   "route": "/webhooks/subscribe",
   "method": "POST",
   "description": "post a request with a callback url to subscribe to events from this agent.  Returns a subscription_id that can be used to unsubscribe from
```

```
the service.  @returns  subscription_id",
    "accepts": {},
    "params": ["url : full url to the callback resource"]
  }
]
```

Rest APIs for the agent

These API calls interact with standard Vertica nodes.

Backup and restore

GET backups	Returns all the backups that have been created for all vbr configuration files (*.ini) that are located in the /opt/vertica/config directory.
POST backups/:config_script_base	Creates a new backup as defined by the given vbr configuration script base (filename without the .ini extension).
GET backups/:config_script_base/:archive_id	Returns details for a specific backup archive.
POST restore/:archive_id	Restores a backup.

Databases

GET databases	Returns a list of databases, their properties, and current status.
POST databases	Creates a new database by supplying a valid set of parameters.
GET databases/:database_name	Returns details about a specific database.
PUT databases/:database_name	Starts, stops, rebalances, or runs Workload Analyzer on a database.
DELETE databases/:database_name	Deletes an existing database.
GET databases/:database_name/configuration	Returns the current configuration parameters from the database.
PUT databases/:database_name/configuration	Sets one or more configuration parameters in the database.
GET databases/:database_name/hosts	Returns hosts details for a specific database.
POST databases/:database_name/hosts	Adds a new host to the database.
DELETE databases/:database_name/hosts/:host_id	Removes a host from the database.
POST databases/:database_name/hosts/:host_id/process	Starts the database process on a specific host.
DELETE databases/:database_name/hosts/:host_id/process	Stops the database on a specific host.
POST databases/:database_name/hosts/:host_id/replace_with/:host_id_new	Replaces a host with a standby host in the database.
GET databases/:database_name/license	Returns the Vertica license that the specified database is using.
GET databases/:database_name/licenses	Returns all the feature licenses that the specified database is using.
GET databases/:database_name/nodes	Returns a list of nodes for the specified database.
GET databases/:database_name/nodes/:node_id	Returns details on a specific node for the specified database.
POST databases/:database_name/process	Starts the specified database.

GET databases/:database_name/process	Returns the state of the database as either UP or DOWN.
DELETE databases/:database_name/process	Stops the specified database on all hosts.
POST databases/:database_name/rebalance/process	Rebalances the specified database. This option can have a long run time.
GET databases/:database_name/status [broken]	Retrieves the database properties structure.
POST databases/:database_name/Workload Analyzer/process	Runs the analyze workload action against the specified database.This option can have a long run time.

Hosts

GET hosts	Returns a list of hosts in this cluster.
GET hosts/:hostid	Returns details for a specific host in this cluster.

Jobs

GET jobs	Returns a list of jobs the agent is tracking, along with their current status and exit codes.
GET jobs/:id	Returns the details (the saved output) for a specific job.

Licenses

POST licenses	Uploads and applies a new license to this cluster.
GET licenses	Returns the license field that databases created on this cluster use.

Nodes

GET nodes	Returns a list of nodes in this cluster.
GET nodes/:nodeid	Returns details for a specific node in this cluster.

Webhooks

GET webhooks	Returns a list of active webhooks.
POST webhooks/subscribe	Creates a new webhook.
DELETE webhooks/:subscriber_id	Deletes an existing webhook.

In this section

- VerticaAPIKey
- Backup and restore
- Databases
- Hosts
- Jobs
- Licenses
- Nodes
- Webhooks

VerticaAPIKey

The Management API requires an authentication key, named VerticaAPIKEY, to access some API resources. You can manage API keys by using the **apikeymgr** command-line tool.

```
usage: apikeymgr [-h] [--user REQUESTOR] [--app APPLICATION] [--delete]
                 [--create] [--update] [--migrate]
                 [--secure {restricted,normal,admin}] [--list]

API key management tool

optional arguments:
  -h, --help            show this help message and exit
  --user REQUESTOR      The name of the person requesting the key
  --app APPLICATION     The name of the application that will use the key
  --delete              Delete the key for the given R & A
  --create              Create a key for the given R & A
  --update              Update a key for the given R & A
  --migrate             migrate the keyset to the latest format
  --secure {restricted,normal,admin}
                        Set the keys security level
  --list                List all the keys known
```

Example request

To create a new VerticaAPIKEY for the dbadmin user with admin access, enter the following:

```
$ apikeymgr --user dbadmin --app vertica --create --secure admin
```

Response:

```
Requestor  : dbadmin
Application: vertica
API Key    : ValidAPIKey
Synchronizing cluster...
```

Backup and restore

You can use these API calls to perform backup and restore tasks for your database.

GET backups	Returns all the backups that have been created for all vbr configuration files (*.ini) that are located in the /opt/vertica/config directory.
POST backups/:config_script_base	Creates a new backup as defined by the given vbr configuration script base (filename without the .ini extension).
GET backups/:config_script_base/:archive_id	Returns details for a specific backup archive.
POST restore/:archive_id	Restores a backup.

In this section

- GET backups
- POST backups/:config_script_base
- GET backups/:config_script_base/:archive_id
- POST restore/:archive_id

GET backups

Returns a list of all backups created for vbr configuration (*.ini) files that reside in /opt/vertica/config and provides details about each backup.

Resource URL

```
https://<NODE>:5444/backups
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/backups

Response:


```
{
   "data": [
      {
         "backups": [
            {
               "archive_id": "v_vdb_bk_snapshot_20190305_174428",
               "version": "v9.2.1-20190305",
               "href": "/backups/fullbk/v_vdb_bk_snapshot_20190305_174428",
               "exclude_patterns": "",
               "backup_type": "full",
               "include_patterns": "",
               "epoch": "16",
               "objects": "",
               "hosts": "v_vdb_bk_node0001(10.20.91.240), v_vdb_bk_node0002(10.20.91.241), v_vdb_bk_node0003(10.20.91.242),
v_vdb_bk_node0004(10.20.91.243), v_vdb_bk_node0005(10.20.91.244)"
            },
            {
               "archive_id": "v_vdb_bk_snapshot_20190305_174025",
               "version": "v9.2.1-20190305",
               "href": "/backups/fullbk/v_vdb_bk_snapshot_20190305_174025",
               "exclude_patterns": "",
               "backup_type": "full",
               "include_patterns": "",
               "epoch": "16",
               "objects": "",
               "hosts": "v_vdb_bk_node0001(10.20.91.240), v_vdb_bk_node0002(10.20.91.241), v_vdb_bk_node0003(10.20.91.242),
v_vdb_bk_node0004(10.20.91.243), v_vdb_bk_node0005(10.20.91.244)"
            }
         ],
         "config_file": "/opt/vertica/config/fullbk.ini",
         "config_script_base": "fullbk",
         "num_backups": 2
      }
   ],
   "href": "/backups",
   "mime-type": "application/vertica.databases.json-v2"
}
```


POST backups/:config_script_base

Creates a new backup job for the backup defined in the vbr configuration script :config_script_base . The vbr configuration script must reside in /opt/vertica/configuration . The :config_script_base value does not include the .ini filename extention.

To determine valid :config_script_base values, see GET backups .

Returns a job ID that you can use to determine the status of the job.

Resource URL

https://<NODE>:5444/backups/:config_script_base

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

POST	https://<NODE>:5444/backups/backup3

Response:

```
{
   "id": "CreateBackup-VMart-1404750602.03",
   "url": "/jobs/CreateBackup-VMart-1404750602.03"
}
```

GET backups/:config_script_base/:archive_id

Returns details on a specific backup. You must provide the :config_script_base . This value is the name of a vbr config file (without the .ini filename extension) that resides in /opt/vertica/config . The :archive_id is the value of the *backup* field that the GET backups command returns.

Resource URL

```
https://<NODE>:5444/backups/:config_script_base/:archive_id
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/backups/fullbk/v_vdb_bk_snapshot_20190304_204814

Response:

```
{
   "archive_id": "v_vdb_bk_snapshot_20190304_204814",
   "config_file": "/opt/vertica/config/fullbk.ini",
   "objects": "",
   "href": "/backups/fullbk/v_vdb_bk_snapshot_20190304_204814",
   "exclude_patterns": "",
   "epoch": "16",
   "include_patterns": "",
   "backup_type": "full",
   "version": "v9.2.1-20190304",
   "hosts": "v_vdb_bk_node0001(10.20.91.240),
         v_vdb_bk_node0002(10.20.91.241),
         v_vdb_bk_node0003(10.20.91.242),
         v_vdb_bk_node0004(10.20.91.243),
         v_vdb_bk_node0005(10.20.91.244)"
}
```

POST restore/:archive_id

Creates a new restore job to restore the database from the backup archive identified by :archive_id . The :archive_id is the value of a *backup* field that the GET backups command returns.

Returns a job ID that you can use to determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/restore/:archive_id
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

POST	https://<NODE>:5444/restore/backup3_20140707_132904

Response:

```
{
   "id": "RestoreBackup-VMart-1404760113.71",
   "url": "/jobs/RestoreBackup-VMart-1404760113.71"
}
```

Databases

You can use these API calls to interact with your database.

GET databases	Returns a list of databases, their properties, and current status.
POST databases	Creates a new database by supplying a valid set of parameters.
GET databases/:database_name	Returns details about a specific database.
PUT databases/:database_name	Starts, stops, rebalances, or runs Workload Analyzer on a database.
DELETE databases/:database_name	Deletes an existing database.
GET databases/:database_name/configuration	Returns the current configuration parameters from the database.
PUT databases/:database_name/configuration	Sets one or more configuration parameters in the database.
GET databases/:database_name/hosts	Returns hosts details for a specific database.
POST databases/:database_name/hosts	Adds a new host to the database.
DELETE databases/:database_name/hosts/:host_id	Removes a host from the database.
POST databases/:database_name/hosts/:host_id/process	Starts the database process on a specific host.
DELETE databases/:database_name/hosts/:host_id/process	Stops the database on a specific host.
POST databases/:database_name/hosts/:host_id/replace_with/:host_id_new	Replaces a host with a standby host in the database.
GET databases/:database_name/license	Returns the Vertica license that the specified database is using.
GET databases/:database_name/licenses	Returns all the feature licenses that the specified database is using.
GET databases/:database_name/nodes	Returns a list of nodes for the specified database.
GET databases/:database_name/nodes/:node_id	Returns details on a specific node for the specified database.
POST databases/:database_name/process	Starts the specified database.
GET databases/:database_name/process	Returns the state of the database as either UP or DOWN.

DELETE databases/:database_name/process	Stops the specified database on all hosts.
POST databases/:database_name/rebalance/process	Rebalances the specified database. This option can have a long run time.
GET databases/:database_name/status [broken]	Retrieves the database properties structure.
POST databases/:database_name/Workload Analyzer/process	Runs the analyze workload action against the specified database.This option can have a long run time.

In this section

- GET databases
- POST databases
- GET databases/:database_name
- PUT databases/:database_name
- DELETE databases/:database_name
- GET databases/:database_name/configuration
- PUT databases/:database_name/configuration
- GET databases/:database_name/hosts
- POST databases/:database_name/hosts
- DELETE databases/:database_name/hosts/:host_id
- POST databases/:database_name/hosts/:host_id/process
- GET databases/:database_name/license
- GET databases/:database_name/licenses
- DELETE databases/:database_name/hosts/:host_id/process
- POST databases/:database_name/hosts/:host_id/replace_with/:host_id_new
- GET databases/:database_name/nodes
- GET databases/:database_name/nodes/:node_id
- POST databases/:database_name/process
- GET databases/:database_name/process
- DELETE databases/:database_name/process
- POST databases/:database_name/rebalance/process
- POST databases/:database_name/Workload analyzer/process

GET databases

Returns a list of databases, their current status, and database properties.

Resource URL

```
https://<NODE>:5444/databases
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/databases

An example of the full request using cURL:

```
curl -H "VerticaApiKey: ValidAPIKey" https://<NODE>:5444/databases
```

Response:

```
{
   "body": [
      {
         "href": "/databases/VMart",
         "mime-type": [
            "application/vertica.database.json-v2"
         ],
         "name": "VMart",
         "port": "5433",
         "status": "UP"
      },
      {
         "href": "/databases/testDB",
         "mime-type": [
            "application/vertica.database.json-v2"
         ],
         "name": "testDB",
         "port": "5433",
         "status": "DOWN"
      }
   ],
   "href": "/databases",
   "links": [
      "/:database_name"
   ],
   "mime-type": "application/vertica.databases.json-v2"
}
```


POST databases

Creates a job to create a new database with the provided parameters.

Important
You must stop any running databases on the nodes on which you want to create the new database. If you do not, database creation fails.

Returns a job ID that can be used to determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/databases
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **admin** level security.

Parameters

name	Name of the database to create.
passwd	Password for the new database.
only	Optional list of hostnames to include in the database. By default, all nodes in the cluster are added to the database.
exclude	Optional list of hostnames to exclude from the database.
catalog	Path of the catalog directory.
data	Path of the data directory.
port	Port where the database listens for client connections. Default is 5433.

Example request

POST	https://:5444/databases?passwd=*db_password*&name=*db_name*& catalog=%2F*path*%2F*to*%2F*catalog*&data=%2F*path*%2F*to*%2F*data_directory*

Response:

```
{
   "jobid": "CreateDatabase-testDB-2014-07-07 15:49:53.219445",
   "resource": "/jobs/CreateDatabase-testDB-2014-07-07 15:49:53.219445",
   "userid": "dbadmin"
}
```

GET databases/:database_name

Returns details about a specific database. The :database_name is the value of the *name* field that the GET databases command returns.

Resource URL

```
https://<NODE>:5444/databases/:database_name
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/databases/VMart

Response:

```
{
   "body": {
      "database_id": "VMart",
      "id": "VMart",
      "nodes": "v_vmart_node0001,v_vmart_node0002,v_vmart_node0003",
      "nodes_new": [
         {
            "catalog_base": "/home/dbadmin",
            "data_base": "/home/dbadmin",
            "host": "10.20.100.247",
            "id": "v_vmart_node0001"
         },
         {
            "catalog_base": "/home/dbadmin",
            "data_base": "/home/dbadmin",
            "host": "10.20.100.248",
            "id": "v_vmart_node0002"
         },
         {
            "catalog_base": "/home/dbadmin",
            "data_base": "/home/dbadmin",
            "host": "10.20.100.249",
            "id": "v_vmart_node0003"
         }
      ],
      "path": "/home/dbadmin/VMart",
      "port": "5433",
      "restartpolicy": "ksafe",
      "status": "UP"
   },
   "href": "/databases/VMart",
   "links": [
      "/configuration",
      "/hosts",
      "/license",
      "/nodes",
      "/process",
      "/rebalance/process",
      "/status",
      "/Workload Analyzer/process"
   ],
   "mime-type": "application/vertica.database.json-v2"
}
```


PUT databases/:database_name

Creates a job to run the action specified by the *action* parameter against the database identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Returns a job ID that you can use to determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/databases/:database_name
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **normal** level security or higher.

Parameters

user_id	A database username.

passwd	A password for the username.
action	Can be one of the following values: • start — Start the database. • stop — Stop the database. • rebalance — Rebalance the database. • Workload Analyzer — Run Work Load Analyzer against the database.

Example request

PUT	https:// :5444/databases/testDB?user_id= *username* &passwd= *username_password* &action=stop

Response:

```
{
   "id": "StopDatabase-testDB-2014-07-20 13:28:49.321744",
   "url": "/jobs/StopDatabase-testDB-2014-07-20 13:28:49.321744"
}
```

DELETE databases/:database_name

Creates a job to delete (drop) an existing database on the cluster. To perform this operation, you must first stop the database. The :database_name is the value of the *name* field that the GET databases command returns.

Returns a job ID that you can use to determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/databases/:database_name
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **admin** level security.

Parameters

None.

Example request

DELETE	https://<NODE>:5444/databases/TestDB

Response:

```
{
   "id": "DropDatabase-TestDB-2014-07-18 12:50:33.332383",
   "url": "/jobs/DropDatabase-TestDB-2014-07-18 12:50:33.332383"
}
```

GET databases/:database_name/configuration

Returns a list of configuration parameters for the database identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Resource URL

```
https://<NODE>:5444/databases/:database_name/configuration
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

user_id	A database username.
passwd	The password for the username.

Example request

GET	https:// :5444/databases/testDB/configuration?user_id= *username* &passwd= *username_password*

Response:

This API call returns over 100 configuration parameters.. The following response is a small subset of the total amount returned.


```
[
  {
    "node_name": "ALL",
    "parameter_name": "ACDAlgorithmForSynopsisVersion1",
    "current_value": "1",
    "restart_value": "1",
    "database_value": "1",
    "default_value": "1",
    "current_level": "DEFAULT",
    "restart_level": "DEFAULT",
    "is_mismatch": "f",
    "groups": "",
    "allowed_levels": "SESSION, DATABASE",
    "superuser_visible_only": "f",
    "change_under_support_guidance": "t",
    "change_requires_restart": "f",
    "description": "Algorithm used to interpret synopsis version 1 for approximate count distinct"
  },
  {
    "node_name": "ALL",
    "parameter_name": "ACDLinearCountThreshold",
    "current_value": "-1.000000",
    "restart_value": "-1.000000",
    "database_value": "-1.000000",
    "default_value": "-1.000000",
    "current_level": "DEFAULT",
    "restart_level": "DEFAULT",
    "is_mismatch": "f",
    "groups": "",
    "allowed_levels": "SESSION, DATABASE",
    "superuser_visible_only": "f",
    "change_under_support_guidance": "t",
    "change_requires_restart": "f",
    "description": "If positive, will overwrite the default linear counting threshold in approximate count distinct"
  },
  {
    "node_name": "ALL",
    "parameter_name": "ACDSynopsisVersion",
    "current_value": "2",
    "restart_value": "2",
    "database_value": "2",
    "default_value": "2",
    "current_level": "DEFAULT",
    "restart_level": "DEFAULT",
    "is_mismatch": "f",
    "groups": "",
    "allowed_levels": "SESSION, DATABASE",
    "superuser_visible_only": "f",
    "change_under_support_guidance": "t",
```

```
    "change_requires_restart": "f",
    "description": "Default synopsis version to be generated by approximate count distinct"
  },
  {
    "node_name": "ALL",
    "parameter_name": "AHMBackupManagement",
    "current_value": "0",
    "restart_value": "0",
    "database_value": "0",
    "default_value": "0",
    "current_level": "DEFAULT",
    "restart_level": "DEFAULT",
    "is_mismatch": "f",
    "groups": "",
    "allowed_levels": "NODE, DATABASE",
    "superuser_visible_only": "f",
    "change_under_support_guidance": "t",
    "change_requires_restart": "f",
    "description": "Consider backup epochs when setting new AHM"
  },
  {
    "node_name": "ALL",
    "parameter_name": "ARCCommitPercentage",
    "current_value": "3.000000",
    "restart_value": "3.000000",
    "database_value": "3.000000",
    "default_value": "3.000000",
    "current_level": "DEFAULT",
    "restart_level": "DEFAULT",
    "is_mismatch": "f",
    "groups": "",
    "allowed_levels": "DATABASE",
    "superuser_visible_only": "f",
    "change_under_support_guidance": "t",
    "change_requires_restart": "f",
    "description": "ARC will commit only if the change is more than the percentage specified"
  },
  {
    "node_name": "ALL",
    "parameter_name": "AWSCAFile",
    "current_value": "",
    "restart_value": "",
    "database_value": "",
    "default_value": "",
    "current_level": "DEFAULT",
    "restart_level": "DEFAULT",
    "is_mismatch": "f",
    "groups": "",
    "allowed_levels": "DATABASE",
    "superuser_visible_only": "f",
    "change_under_support_guidance": "f",
    "change_requires_restart": "f",
    "description": "Overrides the default CA file"
  },
  ...
]
```


PUT databases/:database_name/configuration

Sets one or more configuration parameters for the database identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Returns the parameter name, the requested value, and the result of the attempted change (Success or Failed).

Resource URL

```
https://<NODE>:5444/databases/:database_name/configuration
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **admin** level security.

Parameters

user_id	A database username.
passwd	The password for the username.
parameter_name	A parameter name and value combination for the parameter to be changed. Values must be URL encoded. You can include multiple name/value pairs to set multiple parameters with a single API call.

Example request

PUT	https://:5444/databases/testDB/configuration?user_id=*username*&passwd=*username_password* &JavaBinaryForUDx=%2F*usr*%2F*bin*%2F*java*&TransactionIsolationLevel=SERIALIZABLE

Response:

```
[
   {
      "key": "JavaBinaryForUDx",
      "result": "Success",
      "value": "/usr/bin/java"
   },
   {
      "key": "TransactionIsolationLevel",
      "result": "Success",
      "value": "SERIALIZABLE"
   }
]
```

GET databases/:database_name/hosts

Returns the hostname/IP address, node name, and UP/DOWN status of each host associated with the database identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Resource URL

```
https://<NODE>:5444/databases/:database_name/hosts
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/databases/VMart/hosts

Response:

```
{
   "body": [
      {
         "hostname": "10.20.100.247",
         "nodename": "v_vmart_node0001",
         "status": "UP",
         "ts": "2014-07-18T13:12:31.904191"
      },
      {
         "hostname": "10.20.100.248",
         "nodename": "v_vmart_node0002",
         "status": "UP",
         "ts": "2014-07-18T13:12:31.904209"
      },
      {
         "hostname": "10.20.100.249",
         "nodename": "v_vmart_node0003",
         "status": "UP",
         "ts": "2014-07-18T13:12:31.904215"
      }
   ],
   "href": "/databases/VMart/hosts",
   "links": [],
   "mime-type": "application/vertica.hosts.json-v2"
}
```


POST databases/:database_name/hosts

Creates a job to add a host to the database identified by :database_name . This host must already be part of the cluster. The :database_name is the value of the *name* field that the GET databases command returns.

Returns a job ID that you can use to determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/databases/:database_name/hosts
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **admin** level security.

Parameters

user_id	A database username.
passwd	The password for the username.
hostname	The hostname to add to the database. This host must already be part of the cluster.

Example request

POST	https:// :5444/databases/testDB/hosts?hostname=192.168.232.181&user_id= *username* &passwd= *username_password*

Response:


```
{
   "id": "AddHostToDatabase-testDB-2014-07-20 12:24:04.088812",
   "url": "/jobs/AddHostToDatabase-testDB-2014-07-20 12:24:04.088812"
}
```


DELETE databases/:database_name/hosts/:host_id

Creates a job to remove the host identified by :host_id from the database identified by :database_name . The :database_name is the value of the *name*

field that the GET databases command returns. The :host_id is the value of the *host* field returned by GET databases/:database_name .

Returns a job ID that you can use to determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/databases/:database_name/hosts/:host_id
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **admin** level security.

Parameters

user_id	A database username.
passwd	A password for the username.

Example request

DELETE	https:// :5444/databases/testDB/hosts/192.168.232.181?user_id= *username* &passwd= *username_password*

Response:

```
{
   "id": "RemoveHostFromDatabase-testDB-2014-07-20 13:41:15.646235",
   "url": "/jobs/RemoveHostFromDatabase-testDB-2014-07-20 13:41:15.646235"
}
```

POST databases/:database_name/hosts/:host_id/process

Creates a job to start the vertica process for the database identified by :database_name on the host identified by :host_id . The :database_name is the value of the *name* field that the GET databases command returns. The :host_id is the value of the *host* field returned by GET databases/:database_name .

Returns a job ID that you can use to determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/databases/:database_name/hosts/:host_id/process
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

POST	https://<NODE>:5444/databases/testDB/hosts/192.168.232.181/process

Response:

```
{
   "id": "StartDatabase-testDB-2014-07-20 13:14:03.968340",
   "url": "/jobs/StartDatabase-testDB-2014-07-20 13:14:03.968340"
}
```

GET databases/:database_name/license

Returns details about the database license being used by the database identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Resource URL

https://<NODE>:5444/:database_name/license

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

user_id	A database username.
passwd	The password for the username.

Example request

GET	https:// :5444/VMart/license?user_id= *username* &passwd= *username_password*

Response:


```
{
   "body": {
      "details": {
         "assigned_to": "Vertica Systems, Inc.",
         "grace_period": 0,
         "is_ce": false,
         "is_unlimited": false,
         "name": "vertica",
         "not_after": "Perpetual",
         "not_before": "2007-08-03"
      },
      "last_audit": {
         "audit_date": "2014-07-18 13:49:22.530105-04",
         "database_size_bytes": "814060522",
         "license_size_bytes": "536870912000",
         "usage_percent": "0.00151630588248372"
      }
   },
   "href": "/databases/VMart/license",
   "links": [],
   "mime-type": "application/vertica.license.json-v2"
}
```


GET databases/:database_name/licenses

Returns details about all license being used by the database identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Resource URL

https://<NODE>:5444/:database_name/licenses

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

user_id	A database username.
passwd	The password for the username.

Example request

GET	https:// :5444/VMart/licenses?user_id= *username* &passwd= *username_password*

Response:


```
{
   "body": [
      {
         "details": {
            "assigned_to": "Vertica Systems, Inc.",
            "audit_date": "2014-07-19 21:35:25.111312",
            "is_ce": "False",
            "name": "vertica",
            "node_restriction": "",
            "not_after": "Perpetual",
            "not_before": "2007-08-03",
            "size": "500GB"
         },
         "last_audit": {
            "audit_date": "2014-07-19 21:35:26.318378-04",
            "database_size_bytes": "819066288",
            "license_size_bytes": "536870912000",
            "usage_percent": "0.00152562984824181"
         }
      },
      {
         "details": {
            "assigned_to": "Vertica Systems, Inc., FlexTable",
            "audit_date": "2014-07-19 21:35:25.111312",
            "is_ce": "False",
            "name": "com.vertica.flextable",
            "node_restriction": "",
            "not_after": "Perpetual",
            "not_before": "2007-08-03",
            "size": "500GB"
         },
         "last_audit": {
            "audit_date": "2014-07-19 21:35:25.111312",
            "database_size_bytes": 0,
            "license_size_bytes": 536870912000,
            "usage_percent": 0
         }
      }
   ],
   "href": "/databases/VMart/licenses",
   "links": [],
   "mime-type": "application/vertica.features.json-v2"
}
```


DELETE databases/:database_name/hosts/:host_id/process

Creates a job to stop the vertica process for the database identified by :database_name on the host identified by :host_id . The :database_name is the value of the *name* field that the GET databases command returns. The :host_id is the value of the *host* field returned by GET databases/:database_name .

Returns a job ID that can be used to determine the status of the job. See GET jobs .

> **Note**
> If stopping the database on the hosts causes the database to no longer be k-safe, then the all database nodes may shut down.

Resource URL

```
https://<NODE>:5444/databases/:database_name/hosts/:host_id/process
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

DELETE	https://<NODE>:5444/databases/testDB/hosts/192.168.232.181/process

Response:

```
{
   "id": "StopDatabase-testDB-2014-07-20 13:02:08.453547",
   "url": "/jobs/StopDatabase-testDB-2014-07-20 13:02:08.453547"
}
```

POST databases/:database_name/hosts/:host_id/replace_with/:host_id_new

Creates a job to replace the host identified by hosts/:host_id with the host identified by replace_with/:host_id . Vertica performs these operations for the database identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns. The :host_id is the value of the *host* field as returned by GET databases/:database_name . You can find valid replacement hosts using GET hosts . The replacement host cannot already be part of the database. You must stop the vertica process on the host being replaced.

Returns a job ID that you can use to determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/databases/:database_name/hosts/:host_id/replace_with/:host_id_new
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **admin** level security.

Parameters

user_id	A database username.
passwd	A password for the username.

Example request

POST	https:// :5444/databases/testDB/hosts/192.168.232.180/replace_with/192.168.232.181?user_id= *username* &passwd= *username_password*

Response:

```
{
   "id": "ReplaceNode-testDB-2014-07-20 13:50:28.423509",
   "url": "/jobs/ReplaceNode-testDB-2014-07-20 13:50:28.423509"
}
```

GET databases/:database_name/nodes

Returns a comma-separated list of node IDs for the database identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Resource URL

```
https://<NODE>:5444/:database_name/nodes
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/VMart/nodes

Response:

```
[
   {
      "database_id": "VMart",
      "node_id": "v_vmart_node0001,v_vmart_node0002,v_vmart_node0003",
      "status": "Unknown"
   }
]
```

GET databases/:database_name/nodes/:node_id

Returns details about the node identified by :node_id . The :node_id is one of the node IDs returned by GET databases/:database_name/nodes .

Resource URL

```
https://<NODE>:5444/:database_name/nodes/:node_id
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/databases/VMart/nodes/v_vmart_node0001

Response:

```
{
   "db": "VMart",
   "host": "10.20.100.247",
   "name": "v_vmart_node0001",
   "state": "UP"
}
```

POST databases/:database_name/process

Creates a job to start the database identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Returns a job ID that can be used to determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/databases/:database_name/process
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

epoch	Start the database from this epoch.
include	Include only these hosts when starting the database. Use a comma-separated list of hostnames.

Example request

POST	https://<NODE>:5444/databases/:testDB/process

An example of the full request using cURL:

```
curl -d "epoch=epoch_number&include=host1,host2" -X POST -H "VerticaApiKey: ValidAPIKey" https://<NODE>:5444/:testDB/process
```

Response:

```
{
   "id": "StartDatabase-testDB-2014-07-20 12:41:46.061408",
   "url": "/jobs/StartDatabase-testDB-2014-07-20 12:41:46.061408"
}
```

GET databases/:database_name/process

Returns a state of UP or DOWN for the database identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Resource URL

```
https://<NODE>:5444/databases/:database_name/process
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/databases/VMart/process

Response:

```
{
   "state": "UP"
}
```

DELETE databases/:database_name/process

Creates a job to stop the database identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Returns a job ID that you can useto determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/databases/:database_name/process
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

user_id	A database username.
passwd	The password for the username.

Example request

DELETE	https:// :5444/databases/testDB/process?user_id= *username* &passwd= *username_password*

An example of the full request using cURL:

```
curl -X DELETE -H "VerticaApiKey: ValidAPIKey" https://<NODE>:5444/:testDB/process?user_id=dbadmin"&"passwd=vertica
```

Response:

```
{
   "id": "StopDatabase-testDB-2014-07-20 12:46:04.406637",
   "url": "/jobs/StopDatabase-testDB-2014-07-20 12:46:04.406637"
}
```

POST databases/:database_name/rebalance/process

Creates a job to run a rebalance on the database identified by host identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Returns a job ID that you can use to determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/databases/:database_name/rebalance/process
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

user_id	A database username.
passwd	A password for the username.

Example request

POST	https:// :5444/databases/testDB/rebalance/process?user_id= *username* &passwd= *username_password*

Response:

```
{
   "id": "RebalanceData-testDB-2014-07-20 21:42:45.731038",
   "url": "/jobs/RebalanceData-testDB-2014-07-20 21:42:45.731038"
}
```

POST databases/:database_name/Workload analyzer/process

Creates a job to run Workload Analyzer on the database identified by host identified by :database_name . The :database_name is the value of the *name* field that the GET databases command returns.

Returns a job ID that you can use to determine the status of the job. See GET jobs .

Resource URL

```
https://<NODE>:5444/databases/:database_name/Workload Analyzer/process
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

user_id	A database username.
passwd	A password for the username.

Example request

POST	https:// :5444/databases/testDB/Workload Analyzer/process?user_id= *username* &passwd= *username_password*

Response:

```
{
   "id": "AnalyzeWorkLoad-testDB-2014-07-20 21:48:27.972989",
   "url": "/jobs/AnalyzeWorkLoad-testDB-2014-07-20 21:48:27.972989"
}
```

Hosts

You can use these API calls to get information on the hosts in your cluster.

GET hosts	Returns a list of hosts in this cluster.
GET hosts/:hostid	Returns details for a specific host in this cluster.

In this section

- GET hosts
- GET hosts/:hostid

GET hosts

Returns a list of the hosts in the cluster and the hardware, software, and network details about each host.

Resource URL

```
https://<NODE>:5444/hosts
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/hosts

Response:

```
{
   "body": [
      {
         "cpu_info": {
            "cpu_type": " Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz",
            "number_of_cpus": 2
         },
         "host_id": "10.20.100.247",
         "hostname": "v_vmart_node0001.example.com",
         "max_user_proc": "3833",
         "nics": [
```

```
        {
            "broadcast": "10.20.100.255",
            "ipaddr": "10.20.100.247",
            "name": "eth0",
            "netmask": "255.255.255.0",
            "speed": "unknown"
        },
        {
            "broadcast": "255.255.255.255",
            "ipaddr": "127.0.0.1",
            "name": "lo",
            "netmask": "255.0.0.0",
            "speed": "locallink"
        }
    ],
    "total_memory": 3833,
    "vertica": {
        "arch": "x86_64",
        "brand": "vertica",
        "release": "20140716",
        "version": "12.0.x0"
    }
},
{
    "cpu_info": {
        "cpu_type": " Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz",
        "number_of_cpus": 2
    },
    "host_id": "10.20.100.248",
    "hostname": "v_vmart_node0002.example.com",
    "max_user_proc": "3833",
    "nics": [
        {
            "broadcast": "10.20.100.255",
            "ipaddr": "10.20.100.248",
            "name": "eth0",
            "netmask": "255.255.255.0",
            "speed": "unknown"
        },
        {
            "broadcast": "255.255.255.255",
            "ipaddr": "127.0.0.1",
            "name": "lo",
            "netmask": "255.0.0.0",
            "speed": "locallink"
        }
    ],
    "total_memory": 3833,
    "vertica": {
        "arch": "x86_64",
        "brand": "vertica",
        "release": "20140716",
        "version": "12.0.x0"
    }
},
{
    "cpu_info": {
        "cpu_type": " Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz",
        "number_of_cpus": 2
    },
    "host_id": "10.20.100.249",
    "hostname": "v_vmart_node0003.example.com"
```

```
        "hostname": "v_vmart_node0003.example.com",
        "max_user_proc": "3833",
        "nics": [
            {
                "broadcast": "10.20.100.255",
                "ipaddr": "10.20.100.249",
                "name": "eth0",
                "netmask": "255.255.255.0",
                "speed": "unknown"
            },
            {
                "broadcast": "255.255.255.255",
                "ipaddr": "127.0.0.1",
                "name": "lo",
                "netmask": "255.0.0.0",
                "speed": "locallink"
            }
        ],
        "total_memory": 3833,
        "vertica": {
            "arch": "x86_64",
            "brand": "vertica",
            "release": "20140716",
            "version": "12.0.x0"
        }
    }
  ],
  "href": "/hosts",
  "links": [
    "/:hostid"
  ],
  "mime-type": "application/vertica.hosts.json-v2"
}
```


GET hosts/:hostid

Returns hardware, software, and network details about the host identified by :host_id . You can find :host_id for each host using GET hosts .

Resource URL

```
https://<NODE>:5444/hosts/:hostid
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/hosts/:10.20.100.247

Response:

```
{
   "body": {
      "cpu_info": {
         "cpu_type": " Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz",
         "number_of_cpus": 2
      },
      "hostname": "v_vmart_node0001.example.com",
      "max_user_proc": "3833",
      "nics": [
         {
            "broadcast": "10.20.100.255",
            "ipaddr": "10.20.100.247",
            "name": "eth0",
            "netmask": "255.255.255.0",
            "speed": "unknown"
         },
         {
            "broadcast": "255.255.255.255",
            "ipaddr": "127.0.0.1",
            "name": "lo",
            "netmask": "255.0.0.0",
            "speed": "locallink"
         }
      ],
      "total_memory": 3833,
      "vertica": {
         "arch": "x86_64",
         "brand": "vertica",
         "release": "20140716",
         "version": "12.0.x0"
      }
   },
   "href": "/hosts/10.20.100.247",
   "links": [],
   "mime-type": "application/vertica.host.json-v2"
}
```


Jobs

You can use these API calls to get information on your database's jobs.

GET jobs	Returns a list of jobs the agent is tracking, along with their current status and exit codes.
GET jobs/:id	Returns the details (the saved output) for a specific job.

In this section

- GET jobs
- GET jobs/:id

GET jobs

Returns a list of jobs being tracked by the agent and job details.

Jobs always start immediately. The is_running field is a Boolean value. If is_running is false, then the job is complete.

The exit_code details the status of the job. The exit_code is different for certain types of jobs:

- For Backup jobs:
  - 0 indicates success.
  - Any other number indicates a failure.
- For all other jobs:

- -9 indicates success.
- Any other number indicates a failure.

You can see details about failures in /opt/vertica/log/agentStdMsg.log .

Resource URL

```
https://<NODE>:5444/jobs
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/jobs

Response:


```
{
   "body": [
      {
         "exit_code": 0,
         "id": "CreateBackup-VMart-1405012447.75",
         "is_running": false,
         "status": "unused",
         "ts": "1405012461.18"
      },
      {
         "exit_code": 1,
         "id": "CreateBackup-VMart-1405012454.88",
         "is_running": false,
         "status": "unused",
         "ts": "1405012455.18"
      }
   ],
   "href": "/jobs",
   "links": [
      "/:jobid"
   ],
   "mime-type": "application/vertica.jobs.json-v2"
}
```


GET jobs/:id

Gets the details for a specific job with the provided :id . You can determine the list of job :ids using GET jobs .

Details for a specific job are the same as the details provided for all jobs by GET jobs .

> **Note**
> You must URL encode the :id as some IDs may contain spaces or other special characters.

Resource URL

```
https://<NODE>:5444/jobs/:id
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/jobs/CreateBackup-VMart-1405012454.88

Licenses

You can use these API calls to manage licenses for your database.

POST licenses	Uploads and applies a new license to this cluster.
GET licenses	Returns the license field that databases created on this cluster use.

In this section

- POST licenses
- GET licenses

POST licenses

Uploads and applies a license file to this cluster.

You must provide the license file as an HTTP POST form upload, identified by the name *license* . For example, you can use cURL:

```
curl -k --request POST -H "VerticaApiKey:ValidAPIKey" \
https://v_vmart_node0001:5444/licenses --form "license=@vlicense.dat"
```

Resource URL

```
https://<NODE>:5444/licenses
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **admin** level security.

Parameters

None.

Example request

POST	https://<NODE>:5444/licenses

Response:

There is no HTTP body response for successful uploads. A successful upload returns an HTTP 200/OK header.

GET licenses

Returns any license files that are used by this cluster when creating databases. License files must reside in /opt/vertica/config/share .

Resource URL

```
https://<NODE>:5444/licenses
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/licenses

Response:


```
{
   "body": [
      {
         "comment": "Vertica license is valid",
         "end": "Perpetual",
         "grace": "0",
         "size": "1TB CE Nodes 3",
         "start": "2011-11-22",
         "status": true,
         "vendor": "Vertica Community Edition"
      }
   ],
   "href": "/license",
   "links": [],
   "mime-type": "application/vertica.license.json-v2"
}
```


Nodes

You can use these API calls to retrieve information on the nodes in your cluster.

GET nodes	Returns a list of nodes in this cluster.
GET nodes/:nodeid	Returns details for a specific node in this cluster.

In this section

- GET nodes
- GET nodes/:nodeid

GET nodes

Returns a list of nodes associated with this cluster.

Resource URL

```
https://<NODE>:5444/nodes
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/nodes

Response:

```
{
   "body": [
      "node0001",
      "node0002",
      "node0003",
      "v_testdb_node0001",
      "v_testdb_node0002",
      "v_testdb_node0003",
      "v_vmart_node0001",
      "v_vmart_node0002",
      "v_vmart_node0003"
   ],
   "href": "/nodes",
   "links": [
      "/:nodeid"
   ],
   "mime-type": "application/vertica.nodes.json-v2"
}
```


GET nodes/:nodeid

Returns details about the node identified by :node_id . You can find the :node_id for each node using GET nodes .

In the body field, the following information is detailed in comma-separated format:

- Node Name
- Host Address
- Catalog Directory
- Data Directory

Resource URL

```
https://<NODE>:5444/nodes/:node_id
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/nodes/v_vmart_node0001

Response:


```
{
   "body": [
      "v_vmart_node0001",
      "10.20.100.247,/home/dbadmin,/home/dbadmin"
   ],
   "href": "/nodes/v_vmart_node0001",
   "links": [],
   "mime-type": "application/vertica.node.json-v2"
}
```


Webhooks

You can use these API calls to obtain information on, create, or delete webhooks.

GET webhooks	Returns a list of active webhooks.

POST webhooks/subscribe	Creates a new webhook.
DELETE webhooks/:subscriber_id	Deletes an existing webhook.

In this section

- GET webhooks
- POST webhooks/subscribe
- DELETE webhooks/:subscriber_id

GET webhooks

Returns a list of active webhooks for this cluster.

Resource URL

```
https://<NODE>:5444/webhooks
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

GET	https://<NODE>:5444/webhooks

Response:


```
{
   "body": [
      {
         "host": "192.168.232.1",
         "id": "79c1c8a18be02804b3d2f48ea6462909",
         "port": 80,
         "timestamp": "2014-07-20 22:54:09.829642",
         "url": "/gettest.htm"
      },
      {
         "host": "192.168.232.1",
         "id": "9c32cb0f3d2f9a7cb10835f1732fd4a7",
         "port": 80,
         "timestamp": "2014-07-20 22:54:09.829707",
         "url": "/getwebhook.php"
      }
   ],
   "href": "/webhooks",
   "links": [
      "/subscribe",
      "/:subscriber_id"
   ],
   "mime-type": "application/vertica.webhooks.json-v2"
}
```


POST webhooks/subscribe

Creates a subscription for a webhook.

Resource URL

```
https://<NODE>:5444/webhooks/subscribe
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

url	A URL to an application that accepts JSON messages from this cluster.

Example request

POST	https:// :5444//webhooks/subscribe?url=http%3A%2F%2F *example.com* %2F *getwebhook.php*

Response:

The response is not JSON encoded. The only text response is the ID of the webhook subscription. Additionally, an HTTP 200/OK header indicates success.

```
79c1c8a18be02804b3d2f48ea6462909
```

DELETE webhooks/:subscriber_id

Deletes the webhook identified by :subscriber_id . The :subscriber_id is the value of the *id* field that the GET webhooks command returns.

Resource URL

```
https://<NODE>:5444/webhooks/:subscriber_id
```

Authentication

Requires a VerticaAPIKey in the request header.

The API key must have **restricted** level security or higher.

Parameters

None.

Example request

DELETE	https://<NODE>:5444/webhooks/79c1c8a18be02804b3d2f48ea6462909

Response:

There is no HTTP body response for successful deletes. A successful delete returns an HTTP 200/OK header.

Rest APIs for the Management Console

These API calls interact with Management Console nodes.

Alerts

GET alerts	Returns alerts for the current user.

Time information

GET mcTimeInfo	Returns the current time for the MC server and the timezone of the location where the MC server is located.

In this section

- MC-User-ApiKey
- GET alerts
- GET mcTimeInfo
- Thresholds category filter
- Database name category filter
- Combining sub-category filters with category filters

MC-User-ApiKey

The MC-User-ApiKey is a user-specific key used with Management Console. Users must have an MC-User-ApiKey to interact with MC using the Rest API. All users with roles other than None automatically receive an MC-User-ApiKey.

This key grants users the same rights through the API that they have available through their MC roles. To interact with the MC, users pass the key in the request header for the API.

View the MC-User-ApiKey

If you are the database administrator, you can view the MC-User-ApiKey for all users. Individual users can view their own keys.

1. Connect to MC and go to MC Settings > User Management.
2. Select the user to view and click Edit. The user's key appears in the User API Key field.

GET alerts

Returns a list of MC alerts, their current status, and database properties.

Resource URL

```
https://<MC_NODE>:5450/webui/api/alerts
```

Authentication

Requires an MC-User-Apikey in the request header.

Filter parameters

types	The type of alert to retrieve. Valid values are: • info • notice • warning • error • critical • alert • emergency
category	For information, see Thresholds category filter .
db_name	For information, see Database name category filter .
limit	The maximum number of alerts to retrieve. If the limit is lower than the number of existing alerts, Vertica retrieves the most recent alerts. Used with the type parameter, Vertica retrieves up to the limit for each type. For example, for a limit of five and types of critical and emergency, you could receive up to ten total alerts.
time_from	The timestamp start point from which to retrieve alerts. You can use this parameter in combination with the time_to parameter to retrieve alerts for a specific time range. Values must be passed in the following format: yyyy-MM-ddTHH:mm . If you provide only the time_from parameter, and omit the time_to parameter, the response contains all alerts generated from the time_from parameter to the current time.
time_to	The timestamp end point from which to retrieve alerts. You can use this parameter in combination with the time_from parameter to retrieve alerts for a specific time range. Values must be passed in the following format: yyyy-MM-ddTHH:mm . If you provide only the time_to parameter, and omit the time_from parameter , the response contains all alerts generated from the earliest possible time to the time passed in time_to .

Example request

GET	https://<MC_NODE>:5450/webui/api/alerts?types=critical

Request alerts using cURL

This example shows how you can request alerts using cURL. In this example, the limit parameter is set to '2' and the types parameters is set to info and notice :

```
curl -H "MC-User-ApiKey: ValidUserKey" https://<MC_NODE>:5450/webui/api/alerts?limit=2&types=info,notice
```

Response:


```
[
  {
    "alerts":[
      {
        "id":5502,
        "markedRead":false,
        "eventTypeCode":0,
        "create_time":"2016-02-02 05:12:10.0",
        "updated_time":"2016-02-02 15:50:20.511",
        "severity":"warning",
        "status":1,
        "nodeName":"v_vmart_node0001",
        "databaseName":"VMart",
        "databaseId":1,
        "clusterName":"1449695416208_cluster",
        "description":"Warning: Low disk space detected (73% in use)",
        "summary":"Low Disk Space",
        "internal":false,
        "count":3830
      },
      {
        "id":5501,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2016-02-02 05:12:02.31",
        "updated_time":"2016-02-02 05:12:02.31",
        "severity":"notice",
        "status":1,
        "databaseName":"VMart",
        "databaseId":1,
        "clusterName":"1449695416208_cluster",
        "description":"Analyze Workload operation started on Database",
        "summary":"Analyze Workload operation started on Database",
        "internal":false,
        "count":1
      }
    ],
    "total_alerts":190,
    "request_query":"limit=2",
    "request_time":"2016-02-02 15:50:26 -0500"
  }
]
```


Request alerts within a time range

These examples show various ways in which you can request the same alert as in the preceding example, but within specified time ranges.

Request the alert within a specific time range, using the time_from and time_to parameters:

```
curl -H "MC-User-ApiKey: ValidUserKey" https://<MC_NODE>:5450/webui/api/alerts?types=info,notice&time_from=2016-01-01T12:12&time_to=2016-02-01T12:12
```

Request the alert from a specific start time to the present using the time_from parameter:

```
curl -H "MC-User-ApiKey: ValidUserKey" https://<MC_NODE>:5450/webui/api/alerts?types=info,notice&time_from=2016-01-01T12:12
```

Request the alert to a specific end point using the time_to parameter. When you use the time_to parameter without the time_from parameter, the time_from parameter defaults to the oldest alerts your MC contains:

```
curl -H "MC-User-ApiKey: ValidUserKey" https://<MC_NODE>:5450/webui/api/alerts?types=info,notice&time_to=2016-01-01T12:12
```

GET mcTimeInfo

Returns the current time for the MC server and the timezone where the MC server is located.

Resource URL

```
https://<MC_NODE>:5450/webui/api/mcTimeInfo
```

Authentication

Requires an MC-User-Apikey in the request header.

Parameters

None.

Example request

GET	https://<MC_NODE>:5450/webui/api/mcTimeInfo

This example shows how you can request MC time information using cURL:

```
curl -H "MC-User-ApiKey: ValidUserKey" https://<MC_NODE>:5450/webui/api/mcTimeInfo
```

Response:


```
{"mc_current_time":"Tue, 2000-01-01 01:02:03 -0500","mc_timezone":"US/Eastern"}
```


Thresholds category filter

Returns a list of alerts related to threshold settings in MC.

Resource URL

```
https://<MC_NODE>:5450/webui/api/alerts?category=thresholds
```

Authentication

Requires an MC-User-Apikey in the request header.

Example request

GET	https://<MC_NODE>:5450/webui/api/alerts?category=thresholds

This example shows how you can request alerts on thresholds using cURL:

```
curl -H "MC-User-ApiKey: ValidUserKey" https://<MC_NODE>:5450/webui/api/alerts?category=thresholds
```

Response:


```
[
  {
    "alerts":[
      {
        "id":33,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-10 10:28:41.332",
        "updated_time":"2015-11-10 10:28:41.332",
        "severity":"warning",
        "status":1,
        "databaseName":"mydb",
        "databaseId":1,
        "clusterName":"1446668057043_cluster",
        "description":" Database: mydb Lower than threshold Node Disk I/O 10 %  v_mydb_node0002 ;1.6%  v_mydb_node0002 ;1.4%  v_mydb_node0002 ;2.3%  v_mydb_node0002 ;1.13%  v_mydb_node0002 ;1.39%  v_mydb_node0001 ;3.78%  v_mydb_node0003 ;1.79%  ",
        "summary":"Threshold : Node Disk I/O < 10 %",
```

```
        "internal":false,
        "count":1
      },
      {
        "id":32,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-10 10:28:40.975",
        "updated_time":"2015-11-10 10:28:40.975",
        "severity":"warning",
        "status":1,
        "databaseName":"mydb",
        "databaseId":1,
        "clusterName":"1446668057043_cluster",
        "description":" Database: mydb Lower than threshold Node Memory 10 %   v_mydb_node0002 ;5.47%  v_mydb_node0002 ;5.47%
v_mydb_node0002 ;5.47%  v_mydb_node0002 ;5.47%  v_mydb_node0002 ;5.48%  v_mydb_node0003 ;4.53%  ",
        "summary":"Threshold : Node Memory < 10 %",
        "internal":false,
        "count":1
      },
      {
        "id":31,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-10 10:28:40.044",
        "updated_time":"2015-11-10 10:28:40.044",
        "severity":"warning",
        "status":1,
        "databaseName":"mydb",
        "databaseId":1,
        "clusterName":"1446668057043_cluster",
        "description":" Database: mydb Lower than threshold Node CPU 10 %   v_mydb_node0002 ;1.4%  v_mydb_node0002 ;1.64%  v_mydb_node0002
;1.45%  v_mydb_node0002 ;2.49%  ",
        "summary":"Threshold : Node CPU < 10 %",
        "internal":false,
        "count":1
      },
      {
        "id":30,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-10 10:28:34.562",
        "updated_time":"2015-11-10 10:28:34.562",
        "severity":"warning",
        "status":1,
        "databaseName":"mydb",
        "databaseId":1,
        "clusterName":"1446668057043_cluster",
        "description":" Database: mydb Exceed threshold Node Disk Usage 60 %   v_mydb_node0001 ;86.41%  ",
        "summary":"Threshold : Node Disk Usage > 60 %",
        "internal":false,
        "count":1
      }
    ],
    "total_alerts":4,
    "request_query":"category=thresholds",
    "request_time":"2015-11-10 10:29:17.129"
  }
]
```


See also

- Combining sub-category filters with category filters

Database name category filter

Returns a list of MC alerts for a specific database.

Resource URL

```
https://<MC_NODE>:5450/webui/api/alerts?db_name=
```

Authentication

Requires an MC-User-Apikey in the request header.

Example request

GET	https://<MC_NODE>:5450/webui/api/alerts?db_name= *database_name*

This example shows how you can view alerts on a specific database using cURL:

```
curl -H "MC-User-ApiKey: ValidUserKey" https://<MC_NODE>:5450/webui/api/alerts?db_name="mydb"
```

Response:


```
[
  {
    "alerts":[
      {
        "id":9,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-05 15:10:53.391",
        "updated_time":"2015-11-05 15:10:53.391",
        "severity":"notice",
        "status":1,
        "databaseName":"mydb",
        "databaseId":1,
        "clusterName":"1446668057043_cluster",
        "description":"Workload analyzed successfully",
        "summary":"Analyze Workload operation has succeeded on Database",
        "internal":false,
        "count":1
      },
      {
        "id":8,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-05 15:10:31.16",
        "updated_time":"2015-11-05 15:10:31.16",
        "severity":"notice",
        "status":1,
        "databaseName":"mydb",
        "databaseId":1,
        "clusterName":"1446668057043_cluster",
        "description":"Analyze Workload operation started on Database",
        "summary":"Analyze Workload operation started on Database",
        "internal":false,
        "count":1
      },
      {
        "id":7,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-05 00:15:00.204",
        "updated_time":"2015-11-05 00:15:00.204",
```

```
    "severity":"alert",
    "status":1,
    "databaseName":"mydb",
    "databaseId":1,
    "clusterName":"1446668057043_cluster",
    "description":"Workload analyzed successfully",
    "summary":"Analyze Workload operation has succeeded on Database",
    "internal":false,
    "count":1
  },
  {
    "id":6,
    "markedRead":false,
    "eventTypeCode":2,
    "create_time":"2015-11-04 15:14:59.344",
    "updated_time":"2015-11-04 15:14:59.344",
    "severity":"notice",
    "status":1,
    "databaseName":"mydb",
    "databaseId":1,
    "clusterName":"1446668057043_cluster",
    "description":"Workload analyzed successfully",
    "summary":"Analyze Workload operation has succeeded on Database",
    "internal":false,
    "count":1
  },
  {
    "id":5,
    "markedRead":false,
    "eventTypeCode":2,
    "create_time":"2015-11-04 15:14:38.925",
    "updated_time":"2015-11-04 15:14:38.925",
    "severity":"notice",
    "status":1,
    "databaseName":"mydb",
    "databaseId":1,
    "clusterName":"1446668057043_cluster",
    "description":"Analyze Workload operation started on Database",
    "summary":"Analyze Workload operation started on Database",
    "internal":false,
    "count":1
  },
  {
    "id":4,
    "markedRead":false,
    "eventTypeCode":0,
    "create_time":"2015-11-04 15:14:33.0",
    "updated_time":"2015-11-05 16:26:17.978",
    "severity":"notice",
    "status":1,
    "nodeName":"v_mydb_node0001",
    "databaseName":"lmydb",
    "databaseId":1,
    "clusterName":"1446668057043_cluster",
    "description":"Workload analyzed successfully",
    "summary":"Analyze Workload operation has succeeded on Database",
    "internal":false,
    "count":1
  },
  {
    "id":3,
```

```
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-04 15:14:32.806",
        "updated_time":"2015-11-04 15:14:32.806",
        "severity":"info",
        "status":1,
        "hostIp":"10.20.100.64",
        "nodeName":"v_mydb_node0003",
        "databaseName":"mydb",
        "databaseId":1,
        "clusterName":"1446668057043_cluster",
        "description":"Agent status is UP on IP 127.0.0.1",
        "summary":"Agent status is UP on IP 127.0.0.1",
        "internal":false,
        "count":1
      },
      {
        "id":2,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-04 15:14:32.541",
        "updated_time":"2015-11-04 15:14:32.541",
        "severity":"info",
        "status":1,
        "hostIp":"10.20.100.63",
        "nodeName":"v_mydb_node0002",
        "databaseName":"mydb",
        "databaseId":1,
        "clusterName":"1446668057043_cluster",
        "description":"Agent status is UP on IP 127.0.0.1",
        "summary":"Agent status is UP on IP 127.0.0.1",
        "internal":false,
        "count":1
      },
      {
        "id":1,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-04 15:14:32.364",
        "updated_time":"2015-11-04 15:14:32.364",
        "severity":"info",
        "status":1,
        "hostIp":"10.20.100.62",
        "nodeName":"v_mydb_node0001",
        "databaseName":"mydb",
        "databaseId":1,
        "clusterName":"1446668057043_cluster",
        "description":"Agent status is UP on IP 127.0.0.1",
        "summary":"Agent status is UP on IP 127.0.0.1",
        "internal":false,
        "count":1
      }
    ],
    "total_alerts":9,
    "request_query":"db_name=mydb",
    "request_time":"2015-11-05 16:26:21.679"
  }
]
```


Combining sub-category filters with category filters

You can combine category filters with sub-category filters, to obtain alert messages for specific thresholds you set in MC. You can also use sub-category filters to obtain information about alerts on specific resource pools in your database.

Sub-category filters

You can use the following sub-category filters with the category filters. Sub-category filters are case sensitive and must be lowercase.

Sub-Category Filter	**Alerts Related to Threshold Value Set For:**
THRESHOLD_NODE_CPU	Node CPU
THRESHOLD_NODE_MEMORY	Node Memory
THRESHOLD_NODE_DISK_USAGE	Node Disk Usage
THRESHOLD_NODE_DISKIO	Node Disk I/O
THRESHOLD_NODE_CPUIO	Node CPU I/O Wait
THRESHOLD_NODE_REBOOTRATE	Node Reboot Rate
THRESHOLD_NETIO	Network I/O Error
THRESHOLD_QUERY_QUEUED	Queued Query Number
THRESHOLD_QUERY_FAILED	Failed Query Number
THRESHOLD_QUERY_SPILLED	Spilled Query Number
THRESHOLD_QUERY_RETRIED	Retried Query Number
THRESHOLD_QUERY_RUNTIME	Query Running Time

Resource pool-specific sub-category filters

To retrieve alerts for a specific resource pool, you can use sub-category filters in combination with the following category filters:

- thresholds
- rp_name

If you use these sub-category filters without the RP_NAME filter, the query retrieves alerts for all resource pools in your database.

Sub-Category Filter	**Alerts Related to Threshold Value Set For:**
THRESHOLD_RP_QUERY_MAX_TIME	Queries reaching the maximum allowed execution time.
THRESHOLD_RP_QUERY_RESOURCE_REJECT	The number of queries with resource rejections.
THRESHOLD_RP_QUERY_QUEUE_TIME	The number of queries that ended because of queue time exceeding a limit.
THRESHOLD_RP_QUERY_RUN_TIME	The number of queries that ended because of run time exceeding a limit.
THRESHOLD_RP_MEMORY	The minimum allowed resource pool size.
THRESHOLD_RP_MAX_MEMORY	The maximum allowed resource pool size.

Authentication

Requires an MC-User-Apikey in the request header.

Example request

GET	https://<MC_NODE>:5450/webui/api/alerts?category=thresholds&subcategory= *<subcategory_filter>*

Combine the thresholds category filter with a sub-category filter

This example shows how you can request alerts using cURL with the thresholds category filter and a sub-category filter. You apply the following filters:

- THRESHOLDS
- THRESHOLD_NODE_CPU

```
curl -H "MC-User-ApiKey: ValidUserKey" https://<MC_NODE>:5450/webui/api/alerts?category=thresholds&subcategory=threshold_node_cpu
```

Response:


```
[
  {
    "alerts":[
      {
        "id":11749,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-05 11:04:43.997",
        "updated_time":"2015-11-05 11:04:43.997",
        "severity":"warning",
        "status":1,
        "databaseName":"mydb",
        "databaseId":105,
        "clusterName":"1443122180317_cluster",
        "description":" Database: mydb Lower than threshold Node CPU 10 %   v_mydb_node0002 ;1.03%  v_mydb_node0003 ;0.9%  v_mydb_node0001 ;1.36%  ",
        "summary":"Threshold : Node CPU < 10 %",
        "internal":false,
        "count":1
      },
      {
        "id":11744,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-05 10:59:46.107",
        "updated_time":"2015-11-05 10:59:46.107",
        "severity":"warning",
        "status":1,
        "databaseName":"mydb2",
        "databaseId":106,
        "clusterName":"1443552354071_cluster",
        "description":" Database: mydb2 Lower than threshold Node CPU 10 %   v_mydb2_node0002 ;0.83%  v_mydb2_node0001 ;1.14%  ",
        "summary":"Threshold : Node CPU < 10 %",
        "internal":false,
        "count":1
      }
    ],
    "total_alerts":2,
    "request_query":"category=thresholds&subcategory=threshold_node_cpu",
    "request_time":"2015-11-05 11:05:28.116"
  }
]
```


Request an alert on a specific resource pool

This example shows how you can request alerts using cURL on a specific resource pool. The name of the resource pool is resourcepool1 . You apply the following filters:

- THRESHOLDS
- RP_NAME

- THRESHOLD_RP_QUERY_RUN_TIME

```
curl -H "MC-User-ApiKey: ValidUserKey" https://<MC_NODE>:5450/webui/api/alerts?
category=thresholds&subcategory=threshold_rp_query_run_time&rp_name=resourcepool1
```

Response:


```
[
  {
    "alerts":[
      {
        "id":6525,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-05 14:25:36.797",
        "updated_time":"2015-11-05 14:25:36.797",
        "severity":"warning",
        "status":1,
        "databaseName":"mydb",
        "databaseId":106,
        "clusterName":"1443552354071_cluster",
        "description":" Resource Pool: resourcepool1  Threshold Name: Ended Query with Run Time Exceeding Limit  Time Interval: 14:20:36 to 14:25:36
Threshold Value: 0 min(s)  Actual Value: 2186 query(s) ",
        "summary":"Resource Pool: resourcepool1; Threshold : Ended Query with Run Time Exceeding Limit > 0 min(s)",
        "internal":false,
        "count":1
      },
      {
        "id":6517,
        "markedRead":false,
        "eventTypeCode":2,
        "create_time":"2015-11-05 14:20:39.541",
        "updated_time":"2015-11-05 14:20:39.541",
        "severity":"warning",
        "status":1,
        "databaseName":"mydb",
        "databaseId":106,
        "clusterName":"1443552354071_cluster",
        "description":" Resource Pool: resourcepool1  Threshold Name: Ended Query with Run Time Exceeding Limit  Time Interval: 14:15:39 to 14:20:39
Threshold Value: 0 min(s)  Actual Value: 2259 query(s) ",
        "summary":"Resource Pool: resourcepool1; Threshold : Ended Query with Run Time Exceeding Limit > 0 min(s)",
        "internal":false,
        "count":1
      }
    ],
    "total_alerts":14,
    "request_query":"category=thresholds&subcategory=threshold_rp_query_run_time&rp_name=resourcepool1",
    "request_time":"2015-11-05 11:07:43.988"
  }
]
```


Connect with an SSH tunnel

You can set up an SSH tunnel to connect to Vertica through a proxy server. This can be useful in cases where the client or Vertica server is on a private network.

Server on private network

If the Vertica server is on a private network, run ssh -R to configure remote port forwarding on the Vertica server host. For example, to let the client connect to Vertica through a proxy hosted on proxy.example.com:9595 :

1. On the proxy server, add GatewayPorts yes to /etc/ssh/sshd_config .

2. On the proxy server, restart the SSH service:

```
$ sudo systemctl restart ssh
```

3. On the Vertica server host, run:

```
$ ssh -N -R 9595:localhost:5433 user@proxy.example.com
```

4. On the client host, run the following to connect to Vertica through the proxy server:

```
$ vsql -h proxy.example.com -p 9595
```

Client on private network

If the client machine is on a private network, run ssh -L on the client to configure local port forwarding. For example, to let the client use localhost:9595 to connect to Vertica hosted on vertica.example.com:5433 through a proxy on proxy.example.com :

```
$ ssh -N -L 9595:vertica.example.com:5433 user@proxy.example.com
```

You can then connect to Vertica from the client:

```
$ vsql -p 9595
```

Extending Vertica

You can extend Vertica to perform new operations or handle new types of data. There are several types of extensions:

- **External procedures:** external scripts or programs that are installed on a host in your database cluster.
- **User-defined SQL functions:** Frequently-used SQL expressions, which help you simplify and standardize your SQL scripts.
- **Stored Procedures:** SQL procedures that are stored in the database (as opposed to external procedures). Stored procedures can communicate and interact with your database directly to perform maintenance, execute queries, and update tables.
- **User-defined extensions (UDxs):** functions or data-load steps written in the C++, Python, Java, and R programming languages. They are useful when the type of data processing you want to perform is difficult or slow using SQL. User-defined extensions explains how to use them and Developing user-defined extensions (UDxs) explains how to create them.

In this section

- External procedures
- User-defined SQL functions
- Stored procedures
- User-defined extensions
- Developing user-defined extensions (UDxs)

External procedures

Enterprise Mode only

An external procedure is a script or executable program on a host in your database cluster that you can call from within Vertica. External procedures cannot communicate back to Vertica.

To implement an external procedure:

1. Create an external procedure executable file. See Requirements for external procedures .
2. Enable the set-user-ID(SUID), user execute, and group execute attributes for the file. Either the file must be readable by the dbadmin or the file owner's password must be given with the Administration tools install_procedure command.
3. Install the external procedure executable file .
4. Create the external procedure in Vertica .

After a procedure is created in Vertica, you can execute or drop it, but you cannot alter it.

In this section

- Requirements for external procedures
- Installing external procedure executable files
- Creating external procedures
- Executing external procedures
- Dropping external procedures

Requirements for external procedures

Enterprise Mode only

External procedures have requirements regarding their attributes, where you store them, and how you handle their output. You should also be cognizant of their resource usage.

Procedure file attributes

The procedure file cannot be owned by root. It must have the set-user-ID (SUID), user execute, and group execute attributes set. If it is not readable by the Linux database administrator user, then the owner's password will have to be specified when installing the procedure.

Handling procedure output

Vertica does not provide a facility for handling procedure output. Therefore, you must make your own arrangements for handling procedure output, which should include writing error, logging, and program information directly to files that you manage.

Handling resource usage

The Vertica resource manager is unaware of resources used by external procedures. Additionally, Vertica is intended to be the only major process running on your system. If your external procedure is resource intensive, it could affect the performance and stability of Vertica. Consider the types of external procedures you create and when you run them. For example, you might run a resource-intensive procedure during off hours.

Sample procedure file

```
#!/bin/bash
echo "hello planet argument: $1" >> /tmp/myprocedure.log
```

Installing external procedure executable files

Enterprise Mode only

To install an external procedure, use the Administration Tools through either the menu or the command line.

Menu

1. Run the Administration tools .
   ```
   $ /opt/vertica/bin/adminTools
   ```
2. On the AdminTools **Main Menu** , click **Configuration Menu** , and then click **OK** .
3. On the **Configuration Menu** , click **Install External Procedure** and then click **OK** .
4. Select the database on which you want to install the external procedure.
5. Either select the file to install or manually type the complete file path, and then click **OK** .
6. If you are not the superuser, you are prompted to enter your password and click **OK** .
   The Administration Tools automatically create the *database-name* /procedures directory on each node in the database and installs the external procedure in these directories for you.
7. Click **OK** in the dialog that indicates that the installation was successful.

Command line

If you use the command line, be sure to specify the full path to the procedure file and the password of the Linux user who owns the procedure file. For example:

```
$ admintools -t install_procedure -d vmartdb -f /scratch/helloworld.sh -p ownerpassword
Installing external procedure...
External procedure installed
```

After you have installed an external procedure, you need to make Vertica aware of it. To do so, use the CREATE PROCEDURE statement, but review Creating external procedures first.

Creating external procedures

Enterprise Mode only

After you install an external procedure, you must make Vertica aware of it with CREATE PROCEDURE (external) .

Only superusers can create an external procedure, and by default, only they have execute privileges. However, superusers can grant users and roles EXECUTE privilege on the stored procedure.

After you create a procedure, its metadata is stored in system table USER_PROCEDURES . Users can see only those procedures that they have been granted the privilege to execute.

Example

The following example creates a procedure named helloplanet for external procedure file helloplanet.sh . This file accepts one VARCHAR argument. The sample code is provided in Requirements for external procedures .

```
=> CREATE PROCEDURE helloplanet(arg1 VARCHAR) AS 'helloplanet.sh' LANGUAGE 'external'
  USER 'dbadmin';
```

The next example creates a procedure named proctest for the script copy_vertica_database.sh . This script copies a database from one cluster to another; it is included in the server RPM located in directory /opt/vertica/scripts .

```
=> CREATE PROCEDURE proctest(shosts VARCHAR, thosts VARCHAR, dbdir VARCHAR)
  AS 'copy_vertica_database.sh' LANGUAGE 'external' USER 'dbadmin';
```

Overloading external procedures

You can create multiple external procedures with the same name if they have different signatures—that is, accept a different set of arguments. For example, you can overload the helloplanet external procedure to also accept an integer value:

```
=> CREATE PROCEDURE helloplanet(arg1 INT) AS 'helloplanet.sh' LANGUAGE 'external'
  USER 'dbadmin';
```

After executing this statement, the database catalog stores two external procedures named helloplanet —one that accepts a VARCHAR argument and one that accepts an integer. When you call the external procedure, Vertica evaluates the arguments in the procedure call to determine which procedure to call.

See also

- CREATE PROCEDURE (external)
- GRANT (procedure)

Executing external procedures

Enterprise Mode only

After you define a procedure using the CREATE PROCEDURE (external) statement, you can use it as a meta command in a SELECT statement. Vertica does not support using procedures in more complex statements or in expressions.

The following example runs a procedure named helloplanet :

```
=> SELECT helloplanet('earthlings');
 helloplanet
-------------
           0
(1 row)
```

The following example runs a procedure named proctest . This procedure references the copy_vertica_database.sh script that copies a database from one cluster to another. It is installed by the server RPM in the /opt/vertica/scripts directory.

```
=> SELECT proctest(
   '-s qa01',
   '-t rbench1',
   '-D /scratch_b/qa/PROC_TEST' );
```

> **Note**
> External procedures have no direct access to database data. Use ODBC or JDBC for this purpose.

Procedures are executed on the initiating node. Vertica runs the procedure by forking and executing the program. Each procedure argument is passed to the executable file as a string. The parent fork process waits until the child process ends.

If the child process exits with status 0, Vertica reports the operation took place by returning one row as shown in the helloplanet example. If the child process exits with any other status, Vertica reports an error like the following:

```
ERROR 7112: Procedure reported: Procedure execution error: exit status = code
```

To stop execution, cancel the process by sending a cancel command (for example, CTRL+C) through the client. If the procedure program exits with an error, an error message with the exit status is returned.

Permissions

To execute an external procedure, the user needs:

- EXECUTE privilege on procedure
- USAGE privilege on schema that contains the procedure

Dropping external procedures

Enterprise Mode only

Only a superuser can drop an external procedure. To drop the definition for an external procedure from Vertica, use the DROP PROCEDURE (external) statement. Only the reference to the procedure is removed. The external file remains in the <database>/procedures directory on each node in the database.

> **Note**
> The definition Vertica uses for a procedure cannot be altered; it can only be dropped.

Example

```
=> DROP PROCEDURE helloplanet(arg1 varchar);
```

See also

- DROP PROCEDURE (external)

User-defined SQL functions

User-defined SQL functions let you define and store commonly-used SQL expressions as a function. User-defined SQL functions are useful for executing complex queries and combining Vertica built-in functions. You simply call the function name you assigned in your query.

A user-defined SQL function can be used anywhere in a query where an ordinary SQL expression can be used, except in a table partition clause or the projection segmentation clause.

For syntax and parameters for the commands and system table discussed in this section, see the following topics:

- CREATE FUNCTION
- ALTER FUNCTION (scalar)
- DROP FUNCTION
- GRANT (user defined extension)
- REVOKE (user defined extension)
- V_CATALOG.USER_FUNCTIONS

In this section

- Creating user-defined SQL functions
- Altering and dropping user-defined SQL functions
- Managing access to SQL functions
- Viewing information about user-defined SQL functions
- Migrating built-in SQL functions

Creating user-defined SQL functions

A user-defined SQL function can be used anywhere in a query where an ordinary SQL expression can be used, except in the table partition clause or the projection segmentation clause.

To create a SQL function, the user must have CREATE privileges on the schema. To use a SQL function, the user must have USAGE privileges on the schema and EXECUTE privileges on the defined function.

This following statement creates a SQL function called myzeroifnull that accepts an INTEGER argument and returns an INTEGER result.

```
=> CREATE FUNCTION myzeroifnull(x INT) RETURN INT
  AS BEGIN
    RETURN (CASE WHEN (x IS NOT NULL) THEN x ELSE 0 END);
  END;
```

You can use the new SQL function (myzeroifnull) anywhere you use an ordinary SQL expression. For example, create a simple table:

```
=> CREATE TABLE tabwnulls(col1 INT);
=> INSERT INTO tabwnulls VALUES(1);
=> INSERT INTO tabwnulls VALUES(NULL);
=> INSERT INTO tabwnulls VALUES(0);
=> SELECT * FROM tabwnulls;
 a
---
 1
 0
(3 rows)
```

Use the myzeroifnull function in a SELECT statement, where the function calls col1 from table tabwnulls:

```
=> SELECT myzeroifnull(col1) FROM tabwnulls;
 myzeroifnull
--------------
            1
            0
            0
(3 rows)
```

Use the myzeroifnull function in the GROUP BY clause:

```
=> SELECT COUNT(*) FROM tabwnulls GROUP BY myzeroifnull(col1);
 count
-------
     2
     1
(2 rows)
```

If you want to change a user-defined SQL function's body, use the CREATE OR REPLACE syntax. The following command modifies the CASE expression:

```
=> CREATE OR REPLACE FUNCTION myzeroifnull(x INT) RETURN INT
  AS BEGIN
    RETURN (CASE WHEN (x IS NULL) THEN 0 ELSE x END);
  END;
```

To see how this information is stored in the Vertica catalog, see Viewing Information About SQL Functions .

See also

- CREATE FUNCTION (SQL)
- USER_FUNCTIONS

Altering and dropping user-defined SQL functions

Vertica allows multiple functions to share the same name with different argument types. Therefore, if you try to alter or drop a SQL function without specifying the argument data type, the system returns an error message to prevent you from dropping the wrong function:

```
=> DROP FUNCTION myzeroifnull();
ROLLBACK:  Function with specified name and parameters does not exist: myzeroifnull
```

> **Note**
> Only a superuser or owner can alter or drop a SQL Function.

Altering a user-defined SQL function

The ALTER FUNCTION (scalar) command lets you assign a new name to a user-defined function, as well as move it to a different schema.

In the previous topic, you created a SQL function called myzeroifnull . The following command renames the myzeroifnull function to zerowhennull :

```
=> ALTER FUNCTION myzeroifnull(x INT) RENAME TO zerowhennull;
ALTER FUNCTION
```

This next command moves the renamed function into a new schema called macros :

```
=> ALTER FUNCTION zerowhennull(x INT) SET SCHEMA macros;
ALTER FUNCTION
```

Dropping a SQL function

The DROP FUNCTION command drops a SQL function from the Vertica catalog.

Like with ALTER FUNCTION, you must specify the argument data type or the system returns the following error message:

```
=> DROP FUNCTION zerowhennull();
ROLLBACK:  Function with specified name and parameters does not exist: zerowhennull
```

Specify the argument type:

```
=> DROP FUNCTION macros.zerowhennull(x INT);
DROP FUNCTION
```

Vertica does not check for dependencies, so if you drop a SQL function where other objects reference it (such as views or other SQL Functions), Vertica returns an error when those objects are used, not when the function is dropped.

> **Tip**
> To view a list of all user-defined SQL functions on which you have EXECUTE privileges, (which also returns their argument types), query the V_CATALOG.USER_FUNCTIONS system table.

See also

- ALTER FUNCTION (scalar)
- DROP FUNCTION

Managing access to SQL functions

Before a user can execute a user-defined SQL function, he or she must have USAGE privileges on the schema and EXECUTE privileges on the defined function. Only the superuser or owner can grant/revoke EXECUTE usage on a function.

To grant EXECUTE privileges to user Fred on the myzeroifnull function:

```
=> GRANT EXECUTE ON FUNCTION myzeroifnull (x INT) TO Fred;
```

To revoke EXECUTE privileges from user Fred on the myzeroifnull function:

```
=> REVOKE EXECUTE ON FUNCTION myzeroifnull (x INT) FROM Fred;
```

See also

- GRANT (user defined extension)
- REVOKE (user defined extension)

Viewing information about user-defined SQL functions

You can access information about user-defined SQL functions on which you have EXECUTE privileges. This information is available in system table USER_FUNCTIONS and from the vsql meta-command \df .

To view all user-defined SQL functions on which you have EXECUTE privileges, query USER_FUNCTIONS:

```
=> SELECT * FROM USER_FUNCTIONS;
-[ RECORD 1 ]----------+-----------------------------------------------------
schema_name            | public
function_name          | myzeroifnull
function_return_type   | Integer
function_argument_type | x Integer
function_definition    | RETURN CASE WHEN (x IS NOT NULL) THEN x ELSE 0 END
volatility             | immutable
is_strict              | f
```

If you want to change the body of a user-defined SQL function, use the CREATE OR REPLACE syntax. The following command modifies the CASE expression:

```
=> CREATE OR REPLACE FUNCTION myzeroifnull(x INT) RETURN INT
  AS BEGIN
    RETURN (CASE WHEN (x IS NULL) THEN 0 ELSE x END);
  END;
```

Now when you query USER_FUNCTIONS, you can see the changes in the function_definition column:

```
=> SELECT * FROM USER_FUNCTIONS;
-[ RECORD 1 ]----------+-----------------------------------------------------
schema_name            | public
function_name          | myzeroifnull
function_return_type   | Integer
function_argument_type | x Integer
function_definition    | RETURN CASE WHEN (x IS NULL) THEN 0 ELSE x END
volatility             | immutable
is_strict              | f
```

If you use CREATE OR REPLACE syntax to change only the argument name or argument type (or both), the system maintains both versions of the function. For example, the following command tells the function to accept and return a numeric data type instead of an integer for the myzeroifnull function:

```
=> CREATE OR REPLACE FUNCTION myzeroifnull(z NUMERIC) RETURN NUMERIC
  AS BEGIN
    RETURN (CASE WHEN (z IS NULL) THEN 0 ELSE z END);
  END;
```

Now query the USER_FUNCTIONS table, and you can see the second instance of myzeroifnull in Record 2, as well as the changes in the function_return_type , function_argument_type , and function_definition columns.

> **Note**
> Record 1 still holds the original definition for the myzeroifnull function:

```
=> SELECT * FROM USER_FUNCTIONS;
-[ RECORD 1 ]----------+-----------------------------------------------------------
schema_name            | public
function_name          | myzeroifnull
function_return_type   | Integer
function_argument_type | x Integer
function_definition    | RETURN CASE WHEN (x IS NULL) THEN 0 ELSE x END
volatility             | immutable
is_strict              | f
-[ RECORD 2 ]----------+-----------------------------------------------------------
schema_name            | public
function_name          | myzeroifnull
function_return_type   | Numeric
function_argument_type | z Numeric
function_definition    | RETURN (CASE WHEN (z IS NULL) THEN (0) ELSE z END)::numeric
volatility             | immutable
is_strict              | f
```

Because Vertica allows functions to share the same name with different argument types, you must specify the argument type when you alter or drop a function. If you do not, the system returns an error message:

```
=> DROP FUNCTION myzeroifnull();
ROLLBACK:  Function with specified name and parameters does not exist: myzeroifnull
```

Migrating built-in SQL functions

If you have built-in SQL functions from another RDBMS that do not map to a Vertica-supported function, you can migrate them into your Vertica database by using a user-defined SQL function.

The example scripts below show how to create user-defined functions for the following DB2 built-in functions:

- UCASE()
- LCASE()
- LOCATE()
- POSSTR()

UCASE()

This script creates a user-defined SQL function for the UCASE() function:

```
=> CREATE OR REPLACE FUNCTION UCASE (x VARCHAR)
  RETURN VARCHAR
  AS BEGIN
  RETURN UPPER(x);
  END;
```

LCASE()

This script creates a user-defined SQL function for the LCASE() function:

```
=> CREATE OR REPLACE FUNCTION LCASE (x VARCHAR)
  RETURN VARCHAR
  AS BEGIN
  RETURN LOWER(x);
  END;
```

LOCATE()

This script creates a user-defined SQL function for the LOCATE() function:

```
=> CREATE OR REPLACE FUNCTION LOCATE(a VARCHAR, b VARCHAR)
  RETURN INT
  AS BEGIN
  RETURN POSITION(a IN b);
  END;
```

POSSTR()

This script creates a user-defined SQL function for the POSSTR() function:

```
=> CREATE OR REPLACE FUNCTION POSSTR(a VARCHAR, b VARCHAR)
  RETURN INT
  AS BEGIN
  RETURN POSITION(b IN a);
  END;
```

Stored procedures

You can condense complex database tasks and routines into stored procedures. Unlike external procedures , stored procedures live and can be executed from inside your database; this lets them communicate and interact with your database directly to perform maintenance, execute queries, and update tables.

Best practices

Many other databases are optimized for online transaction processing (OLTP), which focuses on frequent transactions. In contrast, Vertica is optimized for online analytical processing (OLAP), which instead focuses on storing and analyzing large amounts of data and delivering the fastest responses to the most complex queries on that data.

This architecture difference means that the recommended use cases and best practices for stored procedures in Vertica differ slightly from stored procedures in other databases.

While stored procedures in OLTP-oriented databases are often used to perform small transactions, stored procedures in OLAP-oriented databases like Vertica should instead be used to enhance analytical workloads. Vertica can handle isolated transactions, but frequent small transactions can potentially hinder performance.

Some recommended use cases for stored procedures in Vertica include information lifecycle management (ILM) activities such as extract, transform, and load (ETL), and data preparation for tasks like machine learning. For example:

- Swapping partitions according to age
- Exporting data at end-of-life and dropping the partitions
- Saving inputs, outputs, and metadata from a machine learning model—who ran the model, the version of the model, how many times the model was run, and who received the results

Stored procedures in Vertica can also operate on objects that require higher privileges than that of the caller. An optional parameter allows procedures to run using the privileges of the definer, allowing callers to perform sensitive operations in a controlled way.

Viewing stored procedures

To view existing stored procedures, see USER_PROCEDURES .

```
=> SELECT * FROM USER_PROCEDURES;
 procedure_name |  owner  | language | security | procedure_arguments | schema_name
----------------+---------+----------+----------+---------------------+-------------
 raiseXY        | dbadmin | PL/vSQL  | INVOKER  | x int, y varchar    | public
 raiseXY        | dbadmin | PL/vSQL  | INVOKER  | x int, y int        | public
(2 rows)
```

To view the the source code for stored procedures, export them with EXPORT_OBJECTS .

To export a particular implementation, specify either the types or both the names and types of its formal parameters. The following example specifies the types:

```
=> SELECT EXPORT_OBJECTS('','raiseXY(int, int)');
   EXPORT_OBJECTS
----------------------

CREATE PROCEDURE public.raiseXY(x int, y int)
LANGUAGE 'PL/vSQL'
SECURITY INVOKER
AS '
BEGIN
RAISE NOTICE "x = %", x;
RAISE NOTICE "y = %", y;
-- some processing statements
END
';

SELECT MARK_DESIGN_KSAFE(0);

(1 row)
```

To export all implementations of the overloaded stored procedure raiseXY, export its parent schema:

```
=> SELECT EXPORT_OBJECTS('','public');
   EXPORT_OBJECTS
----------------------


...

CREATE PROCEDURE public.raiseXY(x int, y varchar)
LANGUAGE 'PL/vSQL'
SECURITY INVOKER
AS '
BEGIN
RAISE NOTICE "x = %", x;
RAISE NOTICE "y = %", y;
-- some processing statements
END
';

CREATE PROCEDURE public.raiseXY(x int, y int)
LANGUAGE 'PL/vSQL'
SECURITY INVOKER
AS '
BEGIN
RAISE NOTICE "x = %", x;
RAISE NOTICE "y = %", y;
-- some processing statements
END
';

SELECT MARK_DESIGN_KSAFE(0);

(1 row)
```

Known issues and workarounds

- You cannot use PERFORM CREATE FUNCTION to create a SQL macro.
  Workaround
  Use EXECUTE to make SQL macros inside a stored procedure:

```
CREATE PROCEDURE procedure_name()
LANGUAGE PLvSQL AS $$
BEGIN
   EXECUTE 'macro';
end;
$$;
```

where *macro* is the creation statement for a SQL macro. For example, this procedure creates the argmax macro:

```
=> CREATE PROCEDURE make_argmax() LANGUAGE PLvSQL AS $$
BEGIN
   EXECUTE
      'CREATE FUNCTION
      argmax(x int) RETURN int AS
      BEGIN
         RETURN (CASE WHEN (x IS NOT NULL) THEN x ELSE 0 END);
      END';
END;
$$;
```

- Non-error exceptions in embedded SQL statements are not reported.
- DECIMAL, NUMERIC, NUMBER, MONEY, and UUID data types cannot yet be used for arguments.
- Cursors should capture the variable context at declaration time, but they currently capture the variable context at open time.
- DML queries on tables with key constraints cannot yet return a value.

  Workaround

  Rather than:

```
DO $$
DECLARE
   y int;
BEGIN
   y := UPDATE tbl WHERE col1 = 3 SET col2 = 4;
END;
$$
```

  Check the result of the DML query with SELECT:

```
DO $$
DECLARE
   y int;
BEGIN
   y := SELECT COUNT(*) FROM tbl WHERE col1 = 3;
   PERFORM UPDATE tbl SET col2 = 4 WHERE col1 = 3;
END;
$$;
```

In this section

- PL/vSQL
- Parameter modes
- Executing stored procedures
- Altering stored procedures
- Stored procedures: use cases and examples

PL/vSQL

PL/vSQL is a powerful and expressive procedural language for creating reusable procedures, manipulating data, and simplifying otherwise complex database routines.

Vertica PL/vSQL is largely compatible with PostgreSQL PL/pgSQL, with minor semantic differences. For details on migrating your PostgreSQL PL/pgSQL stored procedures to Vertica, see the PL/pgSQL to PL/vSQL migration guide .

For real-world, practical examples of PL/vSQL usage, see Stored procedures: use cases and examples .

In this section

- Supported types
- Scope and structure

- Embedded SQL
- Control flow
- Errors and diagnostics
- Cursors
- PL/pgSQL to PL/vSQL migration guide

Supported types

Vertica PL/vSQL supports non-complex data types . The following types are supported as variables only and not as arguments:

- DECIMAL
- NUMERIC
- NUMBER
- MONEY
- UUID

Scope and structure

PL/vSQL uses block scope, where a block has the following structure:

```
[ <<label>> ]
[ DECLARE
   declarations ]
BEGIN
   statements
   ...
END [ label ];
```

Declarations

Variable *declarations* in the DECLARE block are structured as:

```
variable_name [ CONSTANT ] data_type [ NOT NULL ] [:= { expression | statement } ];
```

variable_name	Variable names must meet the following requirements: • Must be SQL identifiers • Cannot be reserved keywords • Cannot duplicate those in previous declarations inside the same block.
CONSTANT	Defines the variable as a constant (immutable). You can only set a constant variable's value during initialization.
data_type	The variable data type. PL/vSQL supports non-complex data types , with the following exceptions: • GEOMETRY • GEOGRAPHY You can optionally reference a particular column's data type: *variable_name table_name.column_name%TYPE ;*
NOT NULL	Specifies that the variable cannot hold a NULL value. If declared with NOT NULL, the variable must be initialized (otherwise, throws ERRCODE_SYNTAX_ERROR) and cannot be assigned NULL (otherwise, throws ERRCODE_WRONG_OBJECT_TYPE).

:= *expression*	Initializes a variable with *expression* or statement.
	
	If the variable is declared with NOT NULL , *expression* is required.
	
	Variable declarations in a given block execute sequentially, so old declarations can be referenced by newer ones. For example:
	
	`DECLARE`
	`  x int := 3;`
	`  y int := x;`
	
	Default (uninitialized): NULL

Aliases

Aliases are alternate names for the same variable. An alias of a variable is not a copy, and changes made to either reference affect the same underlying variable.

```
new_name ALIAS FOR variable;
```

Here, the identifier y is now an alias for variable x , and changes to y are reflected in x .

```
DO $$
DECLARE
  x int := 3;
  y ALIAS FOR x;
BEGIN
  y := 5; -- since y refers to x, x = 5
  RAISE INFO 'x = %, y = %', x, y;
END;
$$;

INFO 2005:  x = 5, y = 5
```

BEGIN and nested blocks

BEGIN contains *statements* . A *statement* is defined as a line or block of PL/vSQL.

Variables declared in inner blocks shadow those declared in outer blocks. To unambiguously specify a variable in a particular block, you can name the block with a *label* (case-insensitive), and then reference the variable declared in that block with:

```
label.variable_name
```

For example, specifying the variable x from inside the inner block implicitly refers to inner_block.x rather than outer_block.x because of shadowing:

```

<<outer_block>>
DECLARE
  x int;
BEGIN
  <<inner_block>>
  DECLARE
    x int;
  BEGIN
    x := 1000; -- implicitly specifies x in inner_block because of shadowing
    OUTER_BLOCK.x := 0; -- specifies x in outer_block; labels are case-insensitive
  END inner_block;
END outer_block;
```

NULL statement

The NULL statement does nothing. This can be useful as a placeholder statement or a way to show that a code block is intentionally empty. For example:

```
DO $$
BEGIN
   NULL;
END;
$$
```

Comments

Comments have the following syntax. You cannot nest comments.

```
-- single-line comment

/* multi-line
comment
*/
```

Embedded SQL

You can embed and execute SQL statements and expressions from within stored procedures.

Assignment

To save the value of an expression or returned value, you can assign it to a variable:

```
variable_name := expression;
variable_name := statement;
```

For example, this procedure assigns 3 into i and 'message' into v .

```
=> CREATE PROCEDURE performless_assignment() LANGUAGE PLvSQL AS $$
DECLARE
   i int;
   v varchar;
BEGIN
   i := SELECT 3;
   v := 'message';
END;
$$;
```

This type of assignment will fail if the query returns no rows or more than one row. For returns of multiple rows, use LIMIT or truncating assignment :

```
=> SELECT * FROM t1;
 b
---
 t
 f
 f
(3 rows)


=> CREATE PROCEDURE more_than_one_row() LANGUAGE PLvSQL as $$
DECLARE
   x boolean;
BEGIN
   x := SELECT * FROM t1;
END;
$$;
CREATE PROCEDURE

=> CALL more_than_one_row();
ERROR 10332:  Query returned multiple rows where 1 was expected
```

Truncating assignment

Truncating assignment stores in a variable the first row returned by a query. Row order is nondeterministic unless you specify an ORDER BY clause :

```
variable_name <- expression;
variable_name <- statement;
```

The following procedure takes the first row of the results returned by the specified query and assigns it to x :

```
=> CREATE PROCEDURE truncating_assignment() LANGUAGE PLvSQL AS $$
DECLARE
   x boolean;
BEGIN
   x <- SELECT * FROM t1 ORDER BY b DESC; -- x is now assigned the first row returned by the SELECT query
END;
$$;
```

PERFORM

The PERFORM keyword runs a SQL statement or expression and discards the returned result.

```
PERFORM statement;
PERFORM expression;
```

For example, this procedure inserts a value into a table. INSERT returns the number of rows inserted, so you must pair it with PERFORM.

```
=> DO $$
BEGIN
   PERFORM INSERT INTO coordinates VALUES(1,2,3);
END;
$$;
```

> **Note**
> If a SQL statement has no return value or you don't assign the return value to a variable, you must use PERFORM.

EXECUTE

EXECUTE allows you to dynamically construct a SQL query during execution:

```
EXECUTE command_expression [ USING expression [, ... ] ];
```

command_expression is a SQL expression that can reference PL/vSQL variables and evaluates to a string literal. The string literal is executed as a SQL statement, and $1, $2, ... are substituted with the corresponding * expression *s.

Constructing your query with PL/vSQL variables can be dangerous and expose your system to SQL injection, so wrap them with QUOTE_IDENT , QUOTE_LITERAL , and QUOTE_NULLABLE .

The following procedure constructs a query with a WHERE clause:

```
DO $$
BEGIN
   EXECUTE 'SELECT * FROM t1 WHERE x = $1' USING 10; -- becomes WHERE x = 10
END;
$$;
```

The following procedure creates a user with a password from the username and password arguments. Because the constructed CREATE USER statement uses variables, use the functions QUOTE_IDENT and QUOTE_LITERAL, concatenating them with ||.

```
=> CREATE PROCEDURE create_user(username varchar, password varchar) LANGUAGE PLvSQL AS $$
BEGIN
   EXECUTE 'CREATE USER ' || QUOTE_IDENT(username) || ' IDENTIFIED BY ' || QUOTE_LITERAL(password);
END;
$$;
```

EXECUTE is a SQL statement, so you can assign it to a variable or pair it with PERFORM:

```
variable_name:= EXECUTE command_expression;
PERFORM EXECUTE command_expression;
```

FOUND (special variable)

The special boolean variable FOUND is initialized as false and assigned true or false based on whether:

- A statement (but not expression) returns results with non-zero number of rows, or
- A FOR loop iterates at least once

You can use FOUND to distinguish between a NULL and 0-row return.

Special variables exist between the scope of a procedure's argument and the outermost block of its definition. This means that:

- Special variables shadow procedure arguments
- Variables declared in the body of the stored procedure will shadow the special variable

The following procedure demonstrates how FOUND changes. Before the SELECT statement, FOUND is false; after the SELECT statement, FOUND is true.

```
=> DO $$
BEGIN
   RAISE NOTICE 'Before SELECT, FOUND = %', FOUND;
   PERFORM SELECT 1; -- SELECT returns 1
   RAISE NOTICE 'After SELECT, FOUND = %', FOUND;
END;
$$;

NOTICE 2005:  Before SELECT, FOUND = f
NOTICE 2005:  After SELECT, FOUND = t
```

Similarly, UPDATE, DELETE, and INSERT return the number of rows affected. In the next example, UPDATE doesn't change any rows, but returns the value 0 to indicate that no rows were affected, so FOUND is set to true:

```
=> SELECT * t1;
 a  |  b
-----+-----
 100 | abc
(1 row)

DO $$
BEGIN
   PERFORM UPDATE t1 SET a=200 WHERE b='efg'; -- no rows affected since b doesn't contain 'efg'
   RAISE INFO 'FOUND = %', FOUND;
END;
$$;

INFO 2005:  FOUND = t
```

FOUND starts as false and is set to true if the loop iterates at least once:

```
=> DO $$
BEGIN
   RAISE NOTICE 'FOUND = %', FOUND;
   FOR i IN RANGE 1..1 LOOP -- RANGE is inclusive, so iterates once
      RAISE NOTICE 'i = %', i;
   END LOOP;
   RAISE NOTICE 'FOUND = %', FOUND;
END;
$$;

NOTICE 2005:  FOUND = f
NOTICE 2005:  FOUND = t

DO $$
BEGIN
   RAISE NOTICE 'FOUND = %', FOUND;
   FOR i IN RANGE 1..0 LOOP
      RAISE NOTICE 'i = %', i;
   END LOOP;
   RAISE NOTICE 'FOUND = %', FOUND;
END;
$$;

NOTICE 2005:  FOUND = f
NOTICE 2005:  FOUND = f
```

Control flow

Control flow constructs give you control over how many times and under what conditions a block of statements should run.

Conditionals

IF/ELSIF/ELSE

IF/ELSIF/ELSE statements let you perform different actions based on a specified condition.

```
IF condition_1 THEN
  statement_1;
[ ELSIF condition_2 THEN
  statement_2 ]
...
[ ELSE
  statement_n; ]
END IF;
```

Vertica successively evaluates each condition as a boolean until it finds one that's true, then executes the block of statements and exits the IF statement. If no conditions are true, it executes the ELSE block, if one exists.

```
IF i = 3 THEN...
ELSIF 0 THEN...
ELSIF true THEN...
ELSIF x <= 4 OR x >= 10 THEN...
ELSIF y = 'this' AND z = 'THAT' THEN...
```

For example, this procedure demonstrates a simple IF...ELSE branch. Because b is declared to be true, Vertica executes the first branch.

```
=> DO LANGUAGE PLvSQL $$
DECLARE
   b bool := true;
BEGIN
   IF b THEN
      RAISE NOTICE 'true branch';
   ELSE
      RAISE NOTICE 'false branch';
   END IF;
END;
$$;

NOTICE 2005:  true branch
```

CASE

CASE expressions are often more readable than IF...ELSE chains. After executing a CASE expression's branch, control jumps to the statement after the enclosing END CASE.

PL/vSQL CASE expressions are more flexible and powerful than SQL case expressions , but the latter are more efficient; you should favor SQL case expressions when possible.

```
CASE [ search_expression ]
  WHEN expression_1 [, expression_2, ...] THEN
     when_statements
 [ ... ]
 [ ELSE
     else_statements ]
END CASE;
```

search_expression is evaluated once and then compared with *expression_n* in each branch from top to bottom. If *search_expression* and a given *expression_n* are equal, then Vertica executes the WHEN block for *expression_n* and exits the CASE block. If no matching expression is found, the ELSE branch is executed, if one exists.

Case expressions must have either a matching case or an ELSE branch, otherwise Vertica throws a CASE_NOT_FOUND error.

If you omit *search_expression* , its value defaults to true .

For example, this procedure plays the game FizzBuzz, printing Fizz if the argument is divisible by 3, Buzz if the argument is divisible by 5, FizzBuzz if the if the argument is divisible by 3 and 5.

```
=> CREATE PROCEDURE fizzbuzz(IN x int) LANGUAGE PLvSQL AS $$
DECLARE
   fizz int := x % 3;
   buzz int := x % 5;
BEGIN
   CASE fizz
      WHEN 0 THEN -- if fizz = 0, execute WHEN block
         CASE buzz
            WHEN 0 THEN -- if buzz = 0, execute WHEN block
               RAISE INFO 'FizzBuzz';
            ELSE -- if buzz != 0, execute WHEN block
               RAISE INFO 'Fizz';
         END CASE;
      ELSE -- if fizz != 0, execute ELSE block
         CASE buzz
            WHEN 0 THEN
               RAISE INFO 'Buzz';
            ELSE
               RAISE INFO '';
         END CASE;
   END CASE;
END;
$$;

=> CALL fizzbuzz(3);
INFO 2005:  Fizz

=> CALL fizzbuzz(5);
INFO 2005:  Buzz

=> CALL fizzbuzz(15);
INFO 2005:  FizzBuzz
```

Loops

Loops repeatedly execute a block of code until a given condition is satisfied.

WHILE

A WHILE loop checks a given condition and, if the condition is true, it executes the loop body, after which the condition is checked again: if true, the loop body executes again; if false, control jumps to the end of the loop body.

```
[ <<label>> ]
WHILE condition LOOP
  statements;
END LOOP;
```

For example, this procedure computes the factorial of the argument:

```
=> CREATE PROCEDURE factorialSP(input int) LANGUAGE PLvSQL AS $$
DECLARE
   i int := 1;
   output int := 1;
BEGIN
   WHILE i <= input loop
      output := output * i;
      i := i + 1;
   END LOOP;
   RAISE INFO '%! = %', input, output;
END;
$$;

=> CALL factorialSP(5);
INFO 2005:  5! = 120
```

LOOP

This type of loop is equivalent to WHILE true and only terminates if it encounters a RETURN or EXIT statement, or if an exception is thrown.

```
[ <<label>> ]
LOOP
  statements;
END LOOP;
```

For example, this procedure prints the integers from counter up to upper_bound , inclusive:

```
DO $$
DECLARE
   counter int := 1;
   upper_bound int := 3;
BEGIN
   LOOP
      RAISE INFO '%', counter;
      IF counter >= upper_bound THEN
         RETURN;
      END IF;
      counter := counter + 1;
   END LOOP;
END;
$$;

INFO 2005:  1
INFO 2005:  2
INFO 2005:  3
```

FOR

FOR loops iterate over a collection, which can be an integral range, query, or cursor.

If a FOR loop iterates at least once, the special FOUND variable is set to true after the loop ends. Otherwise, FOUND is set to false.

The FOUND variable can be useful for distinguishing between a NULL and 0-row return, or creating an IF branch if a LOOP didn't run.

FOR (RANGE)

A FOR (RANGE) loop iterates over a range of integers specified by the expressions *left* and *right* .

```
[ <<label>> ]
FOR loop_counter IN RANGE [ REVERSE ] left..right [ BY step ] LOOP
   statements
END LOOP [ label ];
```

loop_counter :

- does not have to be declared and is initialized with the value of *left*
- is only available within the scope of the FOR loop

loop_counter iterates from *left* to *right* (inclusive), incrementing by *step* at the end of each iteration.

The REVERSE option instead iterates from *right* to *left* (inclusive), decrementing by *step* .

For example, here is a standard ascending FOR loop with *step* = 1:

```
=> DO $$
BEGIN
   FOR i IN RANGE 1..4 LOOP -- loop_counter i does not have to be declared
      RAISE NOTICE 'i = %', i;
   END LOOP;
   RAISE NOTICE 'after loop: i = %', i; -- fails
END;
$$;

NOTICE 2005:  i = 1
NOTICE 2005:  i = 2
NOTICE 2005:  i = 3
NOTICE 2005:  i = 4
ERROR 2624:  Column "i" does not exist -- loop_counter i is only available inside the FOR loop
```

Here, the *loop_counter* i starts at 4 and decrements by 2 at the end of each iteration:

```
=> DO $$
BEGIN
   FOR i IN RANGE REVERSE 4..0 BY 2 LOOP
      RAISE NOTICE 'i = %', i;
   END LOOP;
END;
$$;

NOTICE 2005:  i = 4
NOTICE 2005:  i = 2
NOTICE 2005:  i = 0
```

FOR (query)

A FOR (QUERY) loop iterates over the results of a query.

```
[ <<label>> ]
FOR target IN QUERY statement LOOP
   statements
END LOOP [ label ];
```

You can include an ORDER BY clause in the query to make the ordering deterministic.

Unlike FOR (RANGE) loops, you must declare the *target* variables. The values of these variables persist after the loop ends.

For example, suppose given the table tuple :

```
=> SELECT * FROM tuples ORDER BY x ASC;
 x | y | z
---+---+---
 1 | 2 | 3
 4 | 5 | 6
 7 | 8 | 9
(3 rows)
```

This procedure retrieves the tuples in each row and stores them in the variables a , b , and c , and prints them after each iteration:

```
=>
=> DO $$
DECLARE
   a int; -- target variables must be declared
   b int;
   c int;
   i int := 1;
BEGIN
   FOR a,b,c IN QUERY SELECT * FROM tuples ORDER BY x ASC LOOP
      RAISE NOTICE 'iteration %: a = %, b = %, c = %', i,a,b,c;
      i := i + 1;
   END LOOP;
   RAISE NOTICE 'after loop: a = %, b = %, c = %', a,b,c;
END;
$$;

NOTICE 2005:  iteration 1: a = 1, b = 2, c = 3
NOTICE 2005:  iteration 2: a = 4, b = 5, c = 6
NOTICE 2005:  iteration 3: a = 7, b = 8, c = 9
NOTICE 2005:  after loop: a = 7, b = 8, c = 9
```

You can also use a query constructed dynamically with EXECUTE :

```
[ <<label>> ]
FOR target IN EXECUTE 'statement' [ USING expression [, ... ] ] LOOP
   statements
END LOOP [ label ];
```

The following procedure uses EXECUTE to construct a FOR (QUERY) loop and stores the results of that SELECT statement in the variables x and y . The result set of a statement like this has only one row, so it only iterates once.

```
=> SELECT 'first string', 'second string';
  ?column?   |   ?column?
-------------+---------------
 first string | second string
(1 row)

=> DO $$
DECLARE
   x varchar; -- target variables must be declared
   y varchar;
BEGIN
   -- substitute the placeholders $1 and $2 with the strings
   FOR x, y IN EXECUTE 'SELECT $1, $2' USING 'first string', 'second string' LOOP
      RAISE NOTICE '%', x;
      RAISE NOTICE '%', y;
   END LOOP;
END;
$$;

NOTICE 2005:  first string
NOTICE 2005:  second string
```

FOR (cursor)

A FOR (CURSOR) loop iterates over a bound, unopened cursor , executing some set of *statements* for each iteration.

```
[ <<label>> ]
FOR loop_variable [, ...] IN CURSOR bound_unopened_cursor [ ( [ arg_name := ] arg_value [, ...] ) ] LOOP
   statements
END LOOP [ label ];
```

This type of FOR loop opens the cursor at start of the loop and closes at the end.

For example, this procedure creates a cursor c . The procedure passes 6 as an argument to the cursor, so the cursor only retrieves rows where the y-coordinate is 6, storing the coordinates in the variables x_ , y_ , and z_ and printing them at the end of each iteration:

```
=> SELECT * FROM coordinates;
 x  | y | z
----+---+----
 14 | 6 | 19
  1 | 6 |  2
 10 | 6 | 39
 10 | 2 | 1
  7 | 1 | 10
 67 | 1 | 77
(6 rows)

DO $$
DECLARE
   c CURSOR (key int) FOR SELECT * FROM coordinates WHERE y=key;
   x_ int;
   y_ int;
   z_ int;
BEGIN
   FOR x_,y_,z_ IN CURSOR c(6) LOOP
      RAISE NOTICE 'cursor returned %,%,% FOUND=%', x_,y_,z_,FOUND;
   END LOOP;
   RAISE NOTICE 'after loop: %,%,% FOUND=%', x_,y_,z_,FOUND;
END;
$$;

NOTICE 2005:  cursor returned 14,6,19 FOUND=f -- FOUND is only set after the loop ends
NOTICE 2005:  cursor returned 1,6,2 FOUND=f
NOTICE 2005:  after loop: 10,6,39 FOUND=t -- x_, y_, and z_ retain their values, FOUND is now true because the FOR loop iterated at least once
```

Manipulating loops

RETURN

You can exit the entire procedure (and therefore the loop) with RETURN. RETURN is an optional statement and can be added to signal to readers the end of a procedure.

```
RETURN;
```

EXIT

Similar to a break or labeled break in other programming languages, EXIT statements let you exit a loop early, optionally specifying:

- *loop_label* : the name of the loop to exit from
- *condition* : if the *condition* is true , execute the EXIT statement

```
EXIT [ loop_label ] [ WHEN condition ];
```

CONTINUE

CONTINUE skips to the next iteration of the loop without executing statements that follow the CONTINUE itself. You can specify a particular loop with *loop_label* :

```
CONTINUE [loop_label] [ WHEN condition ];
```

For example, this procedure doesn't print during its first two iterations because the CONTINUE statement executes and moves on to the next iteration of the loop before control reaches the RAISE NOTICE statement:

```
=> DO $$
BEGIN
   FOR i IN RANGE 1..5 LOOP
      IF i < 3 THEN
         CONTINUE;
      END IF;
      RAISE NOTICE 'i = %', i;
   END LOOP;
END;
$$;

NOTICE 2005:  i = 3
NOTICE 2005:  i = 4
NOTICE 2005:  i = 5
```

Errors and diagnostics

ASSERT

ASSERT is a debugging feature that checks whether a condition is true . If the condition is false , ASSERT raises an ASSERT_FAILURE exception with an optional error message.

To escape a ' (single quote) character, use '' . Similarly, to escape a " (double quote) character, use "" .

```
ASSERT condition [ , message ];
```

For example, this procedure checks the number of rows in the products table and uses ASSERT to check that the table is populated. If the table is empty, Vertica raises an error:

```
=> CREATE TABLE products(id UUID, name VARCHARE, price MONEY);
CREATE TABLE

=> SELECT * FROM products;
 id | name | price
----+------+-------
(0 rows)

DO $$
DECLARE
   prod_count INT;
BEGIN
   prod_count := SELECT count(*) FROM products;
   ASSERT prod_count > 0, 'products table is empty';
END;
$$;

ERROR 2005:  products table is empty
```

To stop Vertica from checking ASSERT statements, you can set the boolean session-level parameter PLpgSQLCheckAsserts .

RAISE

RAISE can throw errors or print a user-specified error message, one of the following:

```
RAISE [ level ] 'format' [, arg_expression [, ... ]] [ USING option = expression [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sql-state' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
```

level	VARCHAR, one of the following: • LOG: Sends the *format* to vertica.log • INFO: Prints an INFO message in VSQL • NOTICE: Prints a NOTICE in VSQL • WARNING: Prints a WARNING in VSQL • EXCEPTION: Throws catchable exception **Default:** EXCEPTION
format	VARCHAR, a string literal error message where the percent character % is substituted with the * arg_expression *s. %% escapes the substitution and results in a single % in plaintext. If the number of % characters doesn't equal the number of arguments, Vertica throws an error. To escape a ' (single quote) character, use '' . Similarly, to escape a " (double quote) character, use "" .
arg_expression	An expression that substitutes for the percent character (%) in the *format* string.
option = *expression*	*option* must be one of the following and paired with an *expression* that elaborates on the *option* :

option	***expression* content**
MESSAGE	An error message. **Default:** the ERRCODE associated with the exception
DETAIL	Details about the error.
HINT	A hint message.
ERRCODE	The error code to report, one of the following: • A condition name specified in the description column of the SQL state list (with optional ERRCODE_ prefix) • A code that satisfies the SQLSTATE formatting: 5-character sequence of numbers and capital letters (not necessarily on the SQL State List) **Default:** ERRCODE_RAISE_EXCEPTION (V0002)
COLUMN	A column name relevant to the error
CONSTRAINT	A constraint relevant to the error
DATATYPE	A data type relevant to the error
TABLE	A table name relevant to the error
SCHEMA	A schema name relevant to the error

This procedure demonstrates various RAISE levels:

```
=> DO $$
DECLARE
   logfile varchar := 'vertica.log';
BEGIN
   RAISE LOG 'this message was sent to %', logfile;
   RAISE INFO 'info';
   RAISE NOTICE 'notice';
   RAISE WARNING 'warning';
   RAISE EXCEPTION 'exception';

   RAISE NOTICE 'exception changes control flow; this is not printed';
END;
$$;

INFO 2005:  info
NOTICE 2005:  notice
WARNING 2005:  warning
ERROR 2005:  exception

$ grep 'this message was sent to vertica.log' v_vmart_node0001_catalog/vertica.log
<LOG> @v_vmart_node0001: V0002/2005: this message is sent to vertica.log
```

Exceptions

EXCEPTION blocks let you catch and handle exceptions that might get thrown from *statements* :

```
[ <<label>> ]
[ DECLARE
   declarations ]
BEGIN
   statements
EXCEPTION
   WHEN exception_condition [ OR exception_condition ... ] THEN
      handler_statements
   [ WHEN exception_condition [ OR exception_condition ... ] THEN
      handler_statements
     ... ]
END [ label ];
```

exception_condition has one of the following forms:

```
WHEN errcode_division_by_zero THEN ...
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
WHEN OTHERS THEN ...
```

OTHERS is a special condition that catches all exceptions except QUERY_CANCELLED , ASSERT_FAILURE , and FEATURE_NOT_SUPPORTED .

When an exception is thrown, Vertica checks the list of exceptions for a matching *exception_condition* from top to bottom. If it finds a match, it executes the *handler_statements* and then leaves the exception block's scope.

If Vertica can't find a match, it propagates the exception up to the next enclosing block. You can do this manually within an exception handler with RAISE:

```
RAISE;
```

For example, the following procedure divides 3 by 0 in the inner_block , which is an illegal operation that throws the exception division_by_zero with SQL state 22012. Vertica checks the inner EXCEPTION block for a matching condition:

1. The first condition checks for SQL state 42501, so Vertica, so Vertica moves to the next condition.
2. WHEN OTHERS THEN catches all exceptions, so it executes that block.
3. The bare RAISE then propagates the exception to the outer_block .
4. The outer EXCEPTION block successfully catches the exception and prints a message.

```
=> DO $$
<<outer_block>>
BEGIN
   <<inner_block>>
   DECLARE
      x int;
   BEGIN
      x := 3 / 0; -- throws exception division_by_zero, SQLSTATE 22012
   EXCEPTION -- this block is checked first for matching exceptions
      WHEN SQLSTATE '42501' THEN
         RAISE NOTICE 'caught insufficient_privilege exception';
      WHEN OTHERS THEN -- catches all exceptions
         RAISE; -- manually propagate the exception to the next enclosing block
   END inner_block;
EXCEPTION -- exception is propagated to this block
   WHEN division_by_zero THEN
      RAISE NOTICE 'caught division_by_zero exception';
END outer_block;
$$;

NOTICE 2005:  caught division_by_zero exception
```

SQLSTATE and SQLERRM variables

When handling an exception, you can use the following variables to retrieve error information:

- SQLSTATE contains the SQL state
- SQLERRM contains the error message

For details, see SQL state list .

This procedure catches the exception thrown by attempting to assign NULL to a NOT NULL variable and prints the SQL state and error message:

```
DO $$
DECLARE
   i int NOT NULL := 1;
BEGIN
   i := NULL; -- illegal, i was declared with NOT NULL
EXCEPTION
   WHEN OTHERS THEN
      RAISE WARNING 'SQL State: %', SQLSTATE;
      RAISE WARNING 'Error message: %', SQLERRM;
END;
$$;

WARNING 2005:  SQLSTATE: 42809
WARNING 2005:  SQLERRM: Cannot assign null into NOT NULL variable
```

Retrieving exception information

You can retrieve information about exceptions inside exception handlers with GET STACKED DIAGNOSTICS:

```
GET STACKED DIAGNOSTICS variable_name { = | := } item [, ... ];
```

Where *item* can be any of the following:

item	**Description**
RETRUNED_SQLSTATE	SQLSTATE error code of the exception
COLUMN_NAME	Name of the column related to exception
CONSTRAINT_NAME	Name of the constraint related to exception

DATATYPE_NAME	Name of the data type related to exception
MESSAGE_TEXT	Text of the exception's primary message
TABLE_NAME	Name of the table related to exception
SCHEMA_NAME	Name of the schema related to exception
DETAIL_TEXT	Text of the exception's detail message, if any
HINT_TEXT	Text of the exception's hint message, if any
EXCEPTION_CONTEXT	Description of the call stack at the time of the exception

For example, this procedure has an EXCEPTION block that catches the division_by_zero error and prints SQL state, error message, and the exception context:

```
=> DO $$
DECLARE
   message_1 varchar;
   message_2 varchar;
   message_3 varchar;
   x int;
BEGIN
   x := 5 / 0;
EXCEPTION
   WHEN OTHERS THEN -- OTHERS catches all exceptions
   GET STACKED DIAGNOSTICS message_1 = RETURNED_SQLSTATE,
                           message_2 = MESSAGE_TEXT,
                           message_3 = EXCEPTION_CONTEXT;

   RAISE INFO 'SQLSTATE: %', message_1;
   RAISE INFO 'MESSAGE: %', message_2;
   RAISE INFO 'EXCEPTION_CONTEXT: %', message_3;
END;
$$;

INFO 2005:  SQLSTATE: 22012
INFO 2005:  MESSAGE: Division by zero
INFO 2005:  EXCEPTION_CONTEXT: PL/vSQL procedure inline_code_block line 8 at static SQL
```

Cursors

A cursor is a reference to the result set of a query and allows you to view the results one row at a time. Cursors remember their positions in result sets, which can be one of the following:

- a result row
- before the first row
- after the last row

You can also iterate over unopened, bound cursors with a FOR loop. See Control Flow for more information.

Declaring cursors

Bound cursors

To bind a cursor to a *statement* on declaration, use the FOR keyword:

```
cursor_name CURSOR [ ( arg_name arg_type [, ...] ) ] FOR statement;
```

The arguments to a cursor give you more control over which rows to process. For example, suppose you have the following table:

```
=> SELECT * FROM coordinates_xy;
 x | y
---+----
 1 |  2
 9 |  5
 7 | 13
...
(100000 rows)
```

If you're only interested in the rows where y is 6, you might declare the following cursor and then provide the argument 6 when you OPEN the cursor:

```
c CURSOR (key int) FOR SELECT * FROM coordinates_xy WHERE y=key;
```

Unbound cursors

To declare a cursor without binding it to a particular query, use the refcursor type:

```
cursor_name refcursor;
```

You can bind an unbound cursor at any time with OPEN .

For example, to declare the cursor my_unbound_cursor :

```
my_unbound_cursor refcursor;
```

Opening and closing cursors

OPEN

Opening a cursor executes the query with the given arguments, and puts the cursor before the first row of the result set. The ordering of query results (and therefore, the start of the result set) is non-deterministic, unless you specify an ORDER BY clause .

OPEN a bound cursor

To open a cursor that was bound during declaration:

```
OPEN bound_cursor [ ( [ arg_name := ] arg_value [, ...] ) ];
```

For example, given the following declaration:

```
c CURSOR (key int) FOR SELECT * FROM t1 WHERE y=key;
```

You can open the cursor with one of the following:

```
OPEN c(5);
OPEN c(key := 5);
```

CLOSE

Open cursors are automatically closed when the cursor leaves scope, but you can close the cursor preemptively with CLOSE. Closed cursors can be reopened later, which re-executes the query and prepares a new result set.

```
CLOSE cursor;
```

OPEN an unbound cursor

To bind an unbound cursor and then open it:

```
OPEN unbound_cursor FOR statement;
```

You can also use EXECUTE because it's a statement:

```
OPEN unbound_cursor FOR EXECUTE statement_string [ USING expression [, ... ] ];
```

For example, to bind the cursor c to a query to a table product_data :

```
OPEN c for SELECT * FROM product_data;
```

FETCH rows

FETCH statements:

1. Retrieve the row that the specified cursor currently points to and stores it in some variable.
2. Advance the cursor to the next position.

```
variable [, ...] := FETCH opened_cursor;
```

The retrieved value is stored in *variable* . Rows typically have more than one value, so you can use one variable for each.

If FETCH successfully retrieves a value, the special variable FOUND is set to true . Otherwise, if you call FETCH when the cursor is past the final row of the result set, it returns NULL and the special variable FOUND is set to false .

The following procedure creates a cursor c , binding it to a SELECT query on the coordinates table. The procedures passes the argument 1 to the cursor, so the cursor only retrieves rows where the y-coordinate is 1, storing the coordinates in the variables x_ , y_ , and z_ .

Only two rows have a y-coordinate of 1, so after using FETCH twice, the third FETCH starts to return NULL values and FOUND is set to false :

```
=> SELECT * FROM coordinates;
 x  | y | z
----+---+----
 14 | 6 | 19
  1 | 6 |  2
 10 | 6 | 39
 10 | 2 |  1
  7 | 1 | 10
 67 | 1 | 77
(6 rows)

DO $$
DECLARE
   c CURSOR (key int) FOR SELECT * FROM coordinates WHERE y=key;
   x_ int;
   y_ int;
   z_ int;
BEGIN
   OPEN c(1); -- only retrieve rows where y=1
   x_,y_,z_ := FETCH c;
   RAISE NOTICE 'cursor returned %, %, %, FOUND=%',x_, y_, z_, FOUND;
   x_,y_,z_ := FETCH c; -- fetches the last set of results and moves to the end of the result set
   RAISE NOTICE 'cursor returned %, %, %, FOUND=%',x_, y_, z_, FOUND;
   x_,y_,z_ := FETCH c; -- cursor has advanced past the final row
   RAISE NOTICE 'cursor returned %, %, %, FOUND=%',x_, y_, z_, FOUND;
END;
$$;

NOTICE 2005:  cursor returned 7, 1, 10, FOUND=t
NOTICE 2005:  cursor returned 67, 1, 77, FOUND=t
NOTICE 2005:  cursor returned <NULL>, <NULL>, <NULL>, FOUND=f
```

MOVE cursors

MOVE advances an open cursor to the next position without retrieving the row. The special FOUND variable is set to true if the cursor's position (before MOVE) was not past the final row—that is, if calling FETCH instead of MOVE would have retrieved the row.

```
MOVE bound_cursor;
```

For example, this cursor only retrieves rows where the y-coordinate is 2. The result set is only one row, so using MOVE twice advances past the first (and last) row, setting FOUND to false:

```
 => SELECT * FROM coordinates WHERE y=2;
 x  | y | z
----+---+---
 10 | 2 | 1
(1 row)

DO $$
DECLARE
   c CURSOR (key int) FOR SELECT * FROM coordinates WHERE y=key;
BEGIN
   OPEN c(2); -- only retrieve rows where y=2, cursor starts before the first row
   MOVE c; -- cursor advances to the first (and last) row
   RAISE NOTICE 'FOUND=%', FOUND; -- FOUND is true because the cursor points to a row in the result set
   MOVE c; -- cursor advances past the final row
   RAISE NOTICE 'FOUND=%', FOUND; -- FOUND is false because the cursor is past the final row
END;
$$;

NOTICE 2005:  FOUND=t
NOTICE 2005:  FOUND=f
```

PL/pgSQL to PL/vSQL migration guide

While Vertica PL/vSQL is largely compatible with PostgreSQL PL/pgSQL, there are some easily-resolved semantic and SQL-level differences when migrating from PostgreSQL PL/pgSQL.

Language-level differences

The following is a list of notable differences between Vertica PL/vSQL and PostgreSQL PL/pgSQL. In Vertica PL/vSQL:

- You must use the PERFORM statement for SQL statements that return no value.
- UPDATE/DELETE WHERE CURRENT OF is not supported.
- FOR loops have additional keywords:
  - FOR (RANGE) loops: RANGE keyword
  - FOR (QUERY) loops: QUERY keyword
  - FOR (CURSOR) loops: CURSOR keyword
- By default, NULL cannot be coerced to FALSE.

Workaround: coercing NULL to FALSE

Unlike PostgreSQL PL/pgSQL, in Vertica PL/vSQL NULLs are not coercible to false, and expressions that expect a boolean value throw an exception when given a NULL:

```
=> DO $$
BEGIN
   IF NULL THEN -- boolean value expected for IF
   END IF;
END;
$$;

ERROR 10268:  Query returned null where a value was expected
```

To enable NULL-to-false coercion, enable the configuration parameter PLvSQLCoerceNull:

```
=> ALTER DATABASE DEFAULT SET PLvSQLCoerceNull = 1;
```

Planned features

Support for the following features is planned for a future release:

- Full transaction and session semantics. Currently, stored procedures commit the transaction before executing, and each embedded SQL statement executes in its own autocommitted transaction. This has the following implications: * You cannot ROLLBACK. * Session-level changes like creating directed queries or setting session-level parameters will succeed, but have no effect.
- OUT/INOUT parameter modes .
- FOREACH (ARRAY) loops .
- Using the following types as arguments: * DECIMAL * NUMERIC * NUMBER * MONEY * UUID

- Non-forward moving cursors .
- CONTEXT/EXCEPTION_CONTEXT for diagnostics .
- The special variable ROW_COUNT.

> **Note**
> As a workaround, you can rely on INSERT, UPDATE, and DELETE to return the number of rows affected:

```
=> CREATE TABLE t1(i int);
CREATE TABLE
=> DO $$
DECLARE
x int;
BEGIN
x := INSERT INTO t1 VALUES (200);
RAISE INFO 'rows inserted: %', x;
END;
$$;


INFO 2005:  rows inserted: 1
```

SQL-level differences

The following list notes significant architectural, SQL-level differences between Vertica and PostgreSQL. In Vertica:

- Some data types are different sizes—for example, the standard INTEGER type in Vertica is 8 bytes, but 4 bytes in PostgreSQL.
- INSERT, UPDATE, and DELETE to return the number of rows affected.
- Certain SQLSTATE codes are different, which affects exception handling .

Parameter modes

Stored procedures support IN parameters. OUT and INOUT parameters are currently not supported.

If unspecified, a parameter's mode defaults to IN.

IN

IN parameters specify the name and type of an argument. These parameters determine a procedure's signature. When an overloaded procedure is called , Vertica runs the procedure whose signature matches the types of the arguments passed in the invocation.

For example, the caller of this procedure must pass in an INT and a VARCHAR value. Both x and y are IN parameters:

```
=> CREATE PROCEDURE raiseXY(IN x INT, y VARCHAR) LANGUAGE PLvSQL AS $$
BEGIN
   RAISE NOTICE 'x = %', x;
   RAISE NOTICE 'y = %', y;
   -- some processing statements
END
$$;

CALL raiseXY(3, 'some string');
NOTICE 2005:  x = 3
NOTICE 2005:  y = some string
```

For more information on RAISE NOTICE, see Errors and diagnostics .

Executing stored procedures

If you have EXECUTE privileges on a stored procedure, you can execute it with a CALL statement that specifies the procedure and its IN arguments.

Syntax

```
CALL stored_procedure_name();
```

For example, the stored procedure raiseXY() is defined as:

```
=> CREATE PROCEDURE raiseXY(IN x INT, y VARCHAR) LANGUAGE PLvSQL AS $$
BEGIN
   RAISE NOTICE 'x = %', x;
   RAISE NOTICE 'y = %', y;
   -- some processing statements
END
$$;

CALL raiseXY(3, 'some string');
NOTICE 2005:  x = 3
NOTICE 2005:  y = some string
```

For more information on RAISE NOTICE, see Errors and diagnostics .

You can execute an anonymous (unnamed) procedure with DO . This requires no privileges:

```
=> DO $$
BEGIN
   RAISE NOTICE '% ran an anonymous procedure', current_user();
END;
$$;

NOTICE 2005:  Bob ran an anonymous procedure
```

Limiting runtime

You can set the maximum runtime of a procedure with session parameter RUNTIMECAP.

This example sets the runtime of all stored procedures to one second for duration of session and runs an anonymous procedure with an infinite loop . Vertica terminates the procedure after it runs for more than one second:

```
=> SET SESSION RUNTIMECAP '1 SECOND';

=> DO $$
BEGIN
   LOOP
   END LOOP;
END;
$$;

ERROR 0:  Query exceeded maximum runtime
HINT:  Change the maximum runtime using SET SESSION RUNTIMECAP
```

Execution security and privileges

By default, stored procedures execute with the privileges of the caller (invoker), so callers must have the necessary privileges on the catalog objects accessed by the stored procedure. You can allow callers to execute the procedure with the privileges, default roles , user parameters , and user attributes (RESOURCE_POOL, MEMORY_CAP_KB, TEMP_SPACE_CAP_KB, RUNTIMECAP) of the definer by specifying DEFINER for the SECURITY option .

For example, the following procedure inserts a value into table s1.t1 . If the DEFINER has the required privileges (USAGE on the schema and INSERT on table), this requirement is waived for callers.

```
=> CREATE PROCEDURE insert_into_s1_t1(IN x int, IN y int)
LANGUAGE PLvSQL
SECURITY DEFINER AS $$
BEGIN
   PERFORM INSERT INTO s1.t1 VALUES(x,y);
END;
$$;
```

A procedure with SECURITY DEFINER effectively executes the procedure as that user, so changes to the database appear to be performed by the procedure's definer rather than its caller.

Caution
Improper use of SECURITY DEFINER can lead to the confused deputy problem and introduce vulnerabilities into your system like SQL injection.

Examples

In this example, this table:

```
records(i INT, updated_date TIMESTAMP DEFAULT sysdate, updated_by VARCHAR(128) DEFAULT current_user())
```

Contains the following content:

```
=> SELECT * FROM records;
 i |        updated_date        | updated_by
---+----------------------------+------------
 1 | 2021-08-27 15:54:05.709044 | Bob
 2 | 2021-08-27 15:54:07.051154 | Bob
 3 | 2021-08-27 15:54:08.301704 | Bob
(3 rows)
```

Bob creates a procedure to update the table and uses the SECURITY DEFINER option and grants EXECUTE on the procedure to Alice. Alice can now use the procedure to update the table without any additional privileges:

```
=> GRANT EXECUTE ON PROCEDURE update_records(int,int) to Alice;
GRANT PRIVILEGE

=> \c - Alice
You are now connected as user "Alice".

=> CALL update_records(99,1);
 update_records
----------------
              0
(1 row)
```

Because calls to update_records() effectively run the procedure as Bob, Bob is listed as the updater of the table rather than Alice:

```
=> SELECT * FROM records;
 i |        updated_date        | updated_by
----+----------------------------+------------
 99 | 2021-08-27 15:55:42.936404 | Bob
  2 | 2021-08-27 15:54:07.051154 | Bob
  3 | 2021-08-27 15:54:08.301704 | Bob
(3 rows)
```

In this section

- Triggers

Triggers

You can automate the execution of stored procedures with triggers. A trigger listens to database events and executes its associated stored procedure when the events occur. You can use triggers with CREATE SCHEDULE to implement Scheduled execution .

Individual triggers can be enabled and disabled with ENABLE_TRIGGER , and can be manually executed with EXECUTE_TRIGGER .

In this section

- Scheduled execution

Scheduled execution

Stored procedures can be scheduled to execute automatically with the privileges of the trigger definer. You can use this to automate various tasks, like logging database activity, revoking privileges, or creating roles.

Enabling and disabling scheduling

Scheduling can be toggled at the database level with the EnableStoredProcedureScheduler configuration parameter:

```
-- Enable scheduler
=> SELECT SET_CONFIG_PARAMETER('EnableStoredProcedureScheduler', 1);

-- Disable scheduler
=> SELECT SET_CONFIG_PARAMETER('EnableStoredProcedureScheduler', 0);
```

You can toggle an individual schedule ENABLE_TRIGGER, or disable it by dropping the schedule's associated trigger.

Scheduling a stored procedure

The general workflow for implementing scheduled execution for a single stored procedure is as follows:

1. Create a stored procedure.
2. Create a schedule. A schedule can either use a list of timestamps for one-off triggers or a cron expression for recurring events.
3. Create a trigger, associating it with the stored procedure and trigger.
4. (Optional) Manually execute the trigger to test it.

One-off triggers

One-off triggers run a finite number of times.

The following example creates a trigger that revokes privileges on the customer_dimension table from the user Bob after 24 hours:

1. Create a stored procedure to revoke privileges from Bob:
   ```
   => CREATE OR REPLACE PROCEDURE revoke_all_on_table(table_name VARCHAR, user_name VARCHAR)
   LANGUAGE PLvSQL
   AS $$
   BEGIN
      EXECUTE 'REVOKE ALL ON ' || QUOTE_IDENT(table_name) || ' FROM ' || QUOTE_IDENT(user_name);
   END;
   $$;
   ```
2. Create a schedule with a timestamp for 24 hours later:
   ```
   => CREATE SCHEDULE 24_hours_later USING DATETIMES('2022-12-16 12:00:00');
   ```
3. Create a trigger with the stored procedure and schedule:
   ```
   => CREATE TRIGGER revoke_trigger ON SCHEDULE 24_hours_later EXECUTE PROCEDURE revoke_all_on_table('customer_dimension', 'Bob') AS
   DEFINER;
   ```

Recurring triggers

Recurring triggers run at a recurring date or time.

The following example creates a weekly trigger that logs to the USER_COUNT table the number of users in the database:

```
=> SELECT * FROM USER_COUNT;
 total |           timestamp
-------+-------------------------------
   293 | 2022-12-04 00:00:00.346664-00
   302 | 2022-12-11 00:00:00.782242-00
   301 | 2022-12-18 00:00:00.144633-00
   301 | 2022-12-25 00:00:00.548832-00
(4 rows)
```

1. Create the table to log the user counts:
   ```
   => CREATE TABLE USER_COUNT(total INT, timestamp TIMESTAMPTZ)
   ```
2. Create the stored procedure to log to the table:

```
=> CREATE OR REPLACE PROCEDURE log_user_count()
LANGUAGE PLvSQL
AS $$
DECLARE
   num_users int := SELECT count (user_id) FROM users;
   timestamp datetime := SELECT NOW();
BEGIN
   PERFORM INSERT INTO USER_COUNT VALUES(num_users, timestamp);
END;
$$;
```

3. Create the schedule for 12:00 AM on Sunday:
```
=> CREATE SCHEDULE weekly_sunday USING CRON '0 0 * * 0';
```

4. Create the trigger with the stored procedure and schedule:
```
=> CREATE TRIGGER user_log_trigger ON SCHEDULE weekly_sunday EXECUTE PROCEDURE log_user_count() AS DEFINER;
```

Viewing upcoming schedules

Schedules are managed and coordinated by the Active Scheduler Node (ASN). If the ASN goes down, a different node is automatically designated as the new ASN. To view scheduled tasks, query SCHEDULER_TIME_TABLE on the ASN.

1. Determine the ASN with ACTIVE_SCHEDULER_NODE :
```
=> SELECT active_scheduler_node();
active_scheduler_node
----------------------
 initiator
(1 row)
```

2. On the ASN, query SCHEDULER_TIME_TABLE :
```
=> SELECT * FROM scheduler_time_table;

 schedule_name  | attached_trigger | scheduled_execution_time
----------------+------------------+--------------------------
daily_1am_gmt   | log_user_actions | 2022-12-15 01:00:00-00
24_hours_later  | revoke_trigger   | 2022-12-16 12:00:00-00
```

Altering stored procedures

You can alter a stored procedure and retain its grants with ALTER PROCEDURE .

Examples

The examples below use the following procedure:

```
=> CREATE PROCEDURE echo_integer(IN x int) LANGUAGE PLvSQL AS $$
BEGIN
   RAISE INFO 'x is %', x;
END;
$$;
```

By default, stored procedures execute with the privileges of the caller (invoker), so callers must have the necessary privileges on the catalog objects accessed by the stored procedure. You can allow callers to execute the procedure with the privileges, default roles , user parameters , and user attributes (RESOURCE_POOL, MEMORY_CAP_KB, TEMP_SPACE_CAP_KB, RUNTIMECAP) of the definer by specifying DEFINER for the SECURITY option .

To execute the procedure with privileges of the...

- Definer (owner):
```
=> ALTER PROCEDURE echo_integer(int) SECURITY DEFINER;
```

- Invoker:
```
=> ALTER PROCEDURE echo_integer(int) SECURITY INVOKER;
```

To change a procedure's source code:

```
=> ALTER PROCEDURE echo_integer(int) SOURCE TO $$
   BEGIN
      RAISE INFO 'the integer is: %', x;
   END;
$$;
```

To change a procedure's owner (definer):

```
=> ALTER PROCEDURE echo_integer(int) OWNER TO u1;
```

To change a procedure's schema:

```
=> ALTER PROCEDURE echo_integer(int) SET SCHEMA s1;
```

To rename a procedure:

```
=> ALTER PROCEDURE echo_integer(int) RENAME TO echo_int;
```

Stored procedures: use cases and examples

Stored procedures in Vertica are best suited for complex, analytical workflows rather than small, transaction-heavy ones. Some recommended use cases include information lifecycle management (ILM) activities like extract, transform, and load (ETL), and data preparation for more complex analytical tasks like machine learning. For example:

- Swapping partitions according to age
- Exporting data at end-of-life and dropping the partitions
- Saving inputs, outputs, and metadata from a machine learning model (e.g. who ran the model, the version of the model, how many times the model was run, who received the results, etc.) for auditing purposes

Searching for a value

The find_my_value() procedure searches for a user-specified value in any table column in a given schema and stores the locations of instances of the value in a user-specified table:

```
=> CREATE PROCEDURE find_my_value(p_table_schema VARCHAR(128), p_search_value VARCHAR(1000), p_results_schema VARCHAR(128),
p_results_table VARCHAR(128)) AS $$
DECLARE
   sql_cmd VARCHAR(65000);
   sql_cmd_result VARCHAR(65000);
   results VARCHAR(65000);
BEGIN
   IF p_table_schema IS NULL OR p_table_schema = '' OR
      p_search_value IS NULL OR p_search_value = '' OR
      p_results_schema IS NULL OR p_results_schema = '' OR
      p_results_table IS NULL OR p_results_table = '' THEN
      RAISE EXCEPTION 'Please provide a schema to search, a search value, a results table schema, and a results table name.';
      RETURN;
   END IF;

   sql_cmd := 'CREATE TABLE IF NOT EXISTS ' || QUOTE_IDENT(p_results_schema) || '.' || QUOTE_IDENT(p_results_table) ||
      '(found_timestamp TIMESTAMP, found_value VARCHAR(1000), table_name VARCHAR(128), column_name VARCHAR(128));';

   sql_cmd_result := EXECUTE 'SELECT LISTAGG(c USING PARAMETERS max_length=1000000, separator="") 
      FROM (SELECT ''
      (SELECT ''''' || NOW() || '''''::TIMESTAMP , ''''' || QUOTE_IDENT(p_search_value) || ''''','''''' || table_name || '''''', '''''' || column_name || ''''''
         FROM '' || table_schema || ''.'' || table_name || ''
         WHERE '' || column_name || ''::'' ||
         CASE
            WHEN data_type_id IN (17, 115, 116, 117) THEN data_type
            ELSE ''VARCHAR('' || LENGTH('''' || QUOTE_IDENT(p_search_value)|| '''') || '')'' END || '' = '''''' || QUOTE_IDENT(p_search_value) || '''''''' ||
DECODE(LEAD(column_name) OVER(ORDER BY table_schema, table_name, ordinal_position), NULL, '' LIMIT 1);'', '' LIMIT 1)

      UNION ALL '') c
         FROM (SELECT table_schema, table_name, column_name, ordinal_position, data_type_id, data_type
         FROM columns WHERE NOT is_system_table AND table_schema ILIKE ''' || QUOTE_IDENT(p_table_schema) || ''' AND data_type_id < 1000
         ORDER BY table_schema, table_name, ordinal_position) foo) foo;';

   results := EXECUTE 'INSERT INTO ' || QUOTE_IDENT(p_results_schema) || '.' || QUOTE_IDENT(p_results_table) || ' ' || sql_cmd_result;

   RAISE INFO 'Matches Found: %', results;
END;
$$;
```

For example, to search the public schema for instances of the string ' dog ' and then store the results in public.table_list :

```
=> CALL find_my_value('public', 'dog', 'public', 'table_list');
 find_my_value
---------------
             0
(1 row)

=> SELECT * FROM public.table_list;
     found_timestamp       | found_value |  table_name   | column_name
---------------------------+-------------+---------------+-------------
 2021-08-25 22:13:20.147889 | dog         | another_table | b
 2021-08-25 22:13:20.147889 | dog         | some_table    | c
(2 rows)
```

Optimizing tables

You can automate loading data from Parquet files and optimizing your queries with the create_optimized_table() procedure. This procedure:

1. Creates an external table whose structure is built from Parquet files using the Vertica INFER_TABLE_DDL function.
2. Creates a native Vertica table, like the external table, resizing all VARCHAR columns to the MAX length of the data to be loaded.
3. Creates a super projection using the optional segmentation/order by columns passed in as a parameter.
4. Adds an optional primary key to the native table passed in as a parameter.
5. Loads a sample data set (1 million rows) from the external table into the native table.

6. Drops the external table.
7. Runs the ANALYZE_STATISTICS function on the native table.
8. Runs the DESIGNER_DESIGN_PROJECTION_ENCODINGS function to get a properly encoded super projection for the native table.
9. Truncates the now-optimized native table (we will load the entire data set in a separate script/stored procedure).

```
=> CREATE OR REPLACE PROCEDURE create_optimized_table(p_file_path VARCHAR(1000), p_table_schema VARCHAR(128), p_table_name
VARCHAR(128), p_seg_columns VARCHAR(1000), p_pk_columns VARCHAR(1000)) LANGUAGE PLvSQL AS $$
DECLARE
   command_sql VARCHAR(1000);
   seg_columns VARCHAR(1000);
   BEGIN

-- First 3 parms are required.
-- Segmented and PK columns names, if present, must be Unquoted Identifiers
   IF p_file_path IS NULL OR p_file_path = '' THEN
      RAISE EXCEPTION 'Please provide a file path.';
   ELSEIF p_table_schema IS NULL OR p_table_schema = '' THEN
      RAISE EXCEPTION 'Please provide a table schema.';
   ELSEIF p_table_name IS NULL OR p_table_name = '' THEN
      RAISE EXCEPTION 'Please provide a table name.';
   END IF;

-- Pass optional segmented columns parameter as null or empty string if not used
   IF p_seg_columns IS NULL OR p_seg_columns = '' THEN
      seg_columns := '';
   ELSE
      seg_columns := 'ORDER BY ' || p_seg_columns || ' SEGMENTED BY HASH(' || p_seg_columns || ') ALL NODES';
   END IF;

-- Add '_external' to end of p_table_name for the external table and drop it if it already exists
   EXECUTE 'DROP TABLE IF EXISTS ' || QUOTE_IDENT(p_table_schema) || '.' || QUOTE_IDENT(p_table_name) || '_external CASCADE;';

-- Execute INFER_TABLE_DDL to generate CREATE EXTERNAL TABLE from the Parquet files
   command_sql := EXECUTE 'SELECT infer_table_ddl(' || QUOTE_LITERAL(p_file_path) || ' USING PARAMETERS format = "parquet", table_schema = "' ||
QUOTE_IDENT(p_table_schema) || '", table_name = "' || QUOTE_IDENT(p_table_name) || '_external", table_type = "external");';

-- Run the CREATE EXTERNAL TABLE DDL
   EXECUTE command_sql;

-- Generate the Internal/ROS Table DDL and generate column lengths based on maximum column lengths found in external table
   command_sql := EXECUTE 'SELECT LISTAGG(y USING PARAMETERS separator="") 
      FROM ((SELECT 0 x, "SELECT ""CREATE TABLE ' || QUOTE_IDENT(p_table_schema) || '.' || QUOTE_IDENT(p_table_name) || '(" y
         UNION ALL SELECT ordinal_position, column_name || " " ||
         CASE WHEN data_type LIKE "varchar%"
            THEN "varchar(" || (SELECT MAX(LENGTH(" || column_name || "))
               FROM " || table_schema || "." || table_name || ") || ")" ELSE data_type END || NVL2(LEAD(" || column_name || ", 1) OVER (ORDER BY
ordinal_position), ",", ")")
               FROM columns WHERE table_schema = "' || QUOTE_IDENT(p_table_schema) || '" AND table_name = "' || QUOTE_IDENT(p_table_name) ||
'_external"
               UNION ALL SELECT 10000, "' || seg_columns || '" UNION ALL SELECT 10001, ";""") ORDER BY x) foo WHERE y <> """;';
   command_sql := EXECUTE command_sql;
   EXECUTE command_sql;

-- Alter the Internal/ROS Table if primary key columns were passed as a parameter
   IF p_pk_columns IS NOT NULL AND p_pk_columns <> '' THEN
      EXECUTE 'ALTER TABLE ' || QUOTE_IDENT(p_table_schema) || '.' || QUOTE_IDENT(p_table_name) || ' ADD CONSTRAINT ' ||
QUOTE_IDENT(p_table_name) || '_pk PRIMARY KEY (' || p_pk_columns || ') ENABLED;';
   END IF;

-- Insert 1M rows into the Internal/ROS Table, analyze stats, and generate encodings
   EXECUTE 'INSERT INTO ' || QUOTE_IDENT(p_table_schema) || '.' || QUOTE_IDENT(p_table_name) || ' SELECT * FROM ' ||
```

```
   EXECUTE 'INSERT INTO ' || QUOTE_IDENT(p_table_schema) || '.' || QUOTE_IDENT(p_table_name) || ' SELECT * FROM ' ||
QUOTE_IDENT(p_table_schema) || '.' || QUOTE_IDENT(p_table_name) || '_external LIMIT 1000000;';

   EXECUTE 'SELECT analyze_statistics('' || QUOTE_IDENT(p_table_schema) || '.' || QUOTE_IDENT(p_table_name) || '');';

   EXECUTE 'SELECT designer_design_projection_encodings('' || QUOTE_IDENT(p_table_schema) || '.' || QUOTE_IDENT(p_table_name) || '',
''/tmp/toss.sql'', TRUE, TRUE);';


-- Truncate the Internal/ROS Table and you are now ready to load all rows
-- Drop the external table

   EXECUTE 'TRUNCATE TABLE ' || QUOTE_IDENT(p_table_schema) || '.' || QUOTE_IDENT(p_table_name) || ';';

   EXECUTE 'DROP TABLE IF EXISTS ' || QUOTE_IDENT(p_table_schema) || '.' || QUOTE_IDENT(p_table_name) || '_external CASCADE;';

 END;

 $$;
```

```
=> call create_optimized_table('/home/dbadmin/parquet_example/*','public','parquet_table','c1,c2','c1');

create_optimized_table
------------------------
                     0
(1 row)

=> select export_objects('', 'public.parquet_table');
      export_objects
-------------------------------------------
CREATE TABLE public.parquet_table
(
   c1 int NOT NULL,
   c2 varchar(36),
   c3 date,
   CONSTRAINT parquet_table_pk PRIMARY KEY (c1) ENABLED
);


CREATE PROJECTION public.parquet_table_super /*+createtype(D)*/
(
c1 ENCODING COMMONDELTA_COMP,
c2 ENCODING ZSTD_FAST_COMP,
c3 ENCODING COMMONDELTA_COMP
)
AS
SELECT parquet_table.c1,
       parquet_table.c2,
       parquet_table.c3
FROM public.parquet_table
ORDER BY parquet_table.c1,
         parquet_table.c2
SEGMENTED BY hash(parquet_table.c1, parquet_table.c2) ALL NODES OFFSET 0;

SELECT MARK_DESIGN_KSAFE(0);

(1 row)
```

Pivoting tables dynamically

The stored procedure unpivot() takes as input a source table and target table. It unpivots the source table and outputs it into a target table.

This example uses the following table:

```
=> SELECT * FROM make_the_columns_into_rows;
c1  | c2  |                  c3                  |            c4             |    c5    | c6
-----+-----+--------------------------------------+---------------------------+----------+----
123 | ABC | cf470c5b-50e3-492a-8483-b9e4f20d195a | 2021-08-24 18:49:40.835802 |  1.72964 | t
567 | EFG | 25ea7636-d924-4b4f-81b5-1e1c884b06e3 | 2021-08-04 18:49:40.835802 | 41.46100 | f
890 | XYZ | f588935a-35a4-4275-9e7f-ebb3986390e3 | 2021-08-29 19:53:39.465778 |  8.58207 | t
(3 rows)
```

This table contains the following columns:

```
=> \d make_the_columns_into_rows
                                  List of Fields by Tables
Schema |          Table           | Column |     Type      | Size | Default | Not Null | Primary Key | Foreign Key
--------+--------------------------+--------+---------------+------+---------+----------+-------------+-------------
public | make_the_columns_into_rows | c1     | int           |    8 |         | f        | f           |
public | make_the_columns_into_rows | c2     | varchar(80)   |   80 |         | f        | f           |
public | make_the_columns_into_rows | c3     | uuid          |   16 |         | f        | f           |
public | make_the_columns_into_rows | c4     | timestamp     |    8 |         | f        | f           |
public | make_the_columns_into_rows | c5     | numeric(10,5) |    8 |         | f        | f           |
public | make_the_columns_into_rows | c6     | boolean       |    1 |         | f        | f           |
(6 rows)
```

The target table has columns from the source table pivoted into rows as key/value pairs. It also has a ROWID column to tie the key/value pairs back to their original row from the source table:

```
=> CREATE PROCEDURE unpivot(p_source_table_schema VARCHAR(128), p_source_table_name VARCHAR(128), p_target_table_schema
VARCHAR(128), p_target_table_name VARCHAR(128)) AS $$
DECLARE
   explode_command VARCHAR(10000);
BEGIN
   explode_command := EXECUTE 'SELECT "explode(string_to_array(""[" || " || LISTAGG("NVL(" || column_name || "::VARCHAR, """")" USING
PARAMETERS separator=" || ""","" || ") || " || "]""")) OVER (PARTITION BY rn)" explode_command FROM (SELECT table_schema, table_name,
column_name, ordinal_position FROM columns ORDER BY table_schema, table_name, ordinal_position LIMIT 10000000) foo WHERE table_schema = ''' ||
QUOTE_IDENT(p_source_table_schema) || ''' AND table_name = ''' || QUOTE_IDENT(p_source_table_name) || ''';';

   EXECUTE 'CREATE TABLE ' || QUOTE_IDENT(p_target_table_schema) || '.' || QUOTE_IDENT(p_target_table_name) || '
      AS SELECT rn rowid, column_name key, value FROM (SELECT (ordinal_position - 1) op, column_name
         FROM columns WHERE table_schema = ''' || QUOTE_IDENT(p_source_table_schema) || '''
         AND table_name = ''' || QUOTE_IDENT(p_source_table_name) || ''') a
         JOIN (SELECT rn, ' || explode_command || '
         FROM (SELECT ROW_NUMBER() OVER() rn, *
         FROM ' || QUOTE_IDENT(p_source_table_schema) || '.' || QUOTE_IDENT(p_source_table_name) || ') foo) b ON b.position = a.op';
END;
$$;
```

Call the procedure:

```
=> CALL unpivot('public', 'make_the_columns_into_rows', 'public', 'columns_into_rows');
 unpivot
---------
       0
(1 row)

=> SELECT * FROM columns_into_rows ORDER BY rowid, key;
rowid | key |                value
-------+-----+--------------------------------------
     1 | c1  | 123
     1 | c2  | ABC
     1 | c3  | cf470c5b-50e3-492a-8483-b9e4f20d195a
     1 | c4  | 2021-08-24 18:49:40.835802
     1 | c5  | 1.72964
     1 | c6  | t
     2 | c1  | 890
     2 | c2  | XYZ
     2 | c3  | f588935a-35a4-4275-9e7f-ebb3986390e3
     2 | c4  | 2021-08-29 19:53:39.465778
     2 | c5  | 8.58207
     2 | c6  | t
     3 | c1  | 567
     3 | c2  | EFG
     3 | c3  | 25ea7636-d924-4b4f-81b5-1e1c884b06e3
     3 | c4  | 2021-08-04 18:49:40.835802
     3 | c5  | 41.46100
     3 | c6  | f
(18 rows)
```

The unpivot() procedure can handle new columns in the source table as well.

Add a new column z to the source table, and then unpivot the table with the same procedure:

```
=> ALTER TABLE make_the_columns_into_rows ADD COLUMN z VARCHAR;
ALTER TABLE

=> UPDATE make_the_columns_into_rows SET z = 'ZZZ' WHERE c1 IN (123, 890);
OUTPUT
--------
      2
(1 row)

=> CALL unpivot('public', 'make_the_columns_into_rows', 'public', 'columns_into_rows');
unpivot
---------
       0
(1 row)

=> SELECT * FROM columns_into_rows;
rowid | key |                value
-------+-----+--------------------------------------
     1 | c1  | 567
     1 | c2  | EFG
     1 | c3  | 25ea7636-d924-4b4f-81b5-1e1c884b06e3
     1 | c4  | 2021-08-04 18:49:40.835802
     1 | c5  | 41.46100
     1 | c6  | f
     1 | z   |   -- new column
     2 | c1  | 123
     2 | c2  | ABC
     2 | c3  | cf470c5b-50e3-492a-8483-b9e4f20d195a
     2 | c4  | 2021-08-24 18:49:40.835802
     2 | c5  | 1.72964
     2 | c6  | t
     2 | z   | ZZZ   -- new column
     3 | c1  | 890
     3 | c2  | XYZ
     3 | c3  | f588935a-35a4-4275-9e7f-ebb3986390e3
     3 | c4  | 2021-08-29 19:53:39.465778
     3 | c5  | 8.58207
     3 | c6  | t
     3 | z   | ZZZ   -- new column
(21 rows)
```

Machine learning: optimizing AUC estimation

The ROC function can approximate the AUC (area under the curve), the accuracy of which depends on the num_bins parameter; greater values of num_bins give you more precise approximations, but may impact performance.

You can use the stored procedure accurate_auc() to approximate the AUC, which automatically determines the optimal num_bins value for a given epsilon (error term):

```
=> CREATE PROCEDURE accurate_auc(relation VARCHAR, observation_col VARCHAR, probability_col VARCHAR, epsilon FLOAT) AS $$
DECLARE
    auc_value FLOAT;
    previous_auc FLOAT;
    nbins INT;
BEGIN
    IF epsilon > 0.25 THEN
        RAISE EXCEPTION 'epsilon must not be bigger than 0.25';
    END IF;
    IF epsilon < 1e-12 THEN
        RAISE EXCEPTION 'epsilon must be bigger than 1e-12';
    END IF;
    auc_value := 0.5;
    previous_auc := 0; -- epsilon and auc should be always less than 1
    nbins := 100;
    WHILE abs(auc_value - previous_auc) > epsilon and nbins < 1000000 LOOP
        RAISE INFO 'auc_value: %', auc_value;
        RAISE INFO 'previous_auc: %', previous_auc;
        RAISE INFO 'nbins: %', nbins;
        previous_auc := auc_value;
        auc_value := EXECUTE 'SELECT auc FROM (select roc(' || QUOTE_IDENT(observation_col) || ',' || QUOTE_IDENT(probability_col) || ' USING
parameters num_bins=$1, auc=true) over() FROM ' || QUOTE_IDENT(relation) || ') subq WHERE auc IS NOT NULL' USING nbins;
        nbins := nbins * 2;
    END LOOP;
    RAISE INFO 'Result_auc_value: %', auc_value;
END;
$$;
```

For example, given the following data in test_data.csv :

```
1,0,0.186
1,1,0.993
1,1,0.9
1,1,0.839
1,0,0.367
1,0,0.362
0,1,0.6
1,1,0.726
...
```

(see test_data.csv for the complete set of data)

You can load the data into table categorical_test_data as follows:

```
=> \set datafile '\'/data/test_data.csv\''
=> CREATE TABLE categorical_test_data(obs INT, pred INT, prob FLOAT);
CREATE TABLE

=> COPY categorical_test_data FROM :datafile DELIMITER ',';
```

Call accurate_auc() . For this example, the approximated AUC will be within the an epsilon of 0.01:

```
=> CALL accurate_auc('categorical_test_data', 'obs', 'prob', 0.01);
INFO 2005:  auc_value: 0.5
INFO 2005:  previous_auc: 0
INFO 2005:  nbins: 100
INFO 2005:  auc_value: 0.749597423510467
INFO 2005:  previous_auc: 0.5
INFO 2005:  nbins: 200
INFO 2005:  Result_auc_value: 0.750402576489533
```

test_data.csv

```
1,0,0.186
```

1,1,0.993
1,1,0.9
1,1,0.839
1,0,0.367
1,0,0.362
0,1,0.6
1,1,0.726
0,0,0.087
0,0,0.004
0,1,0.562
1,0,0.477
0,0,0.258
1,0,0.143
0,0,0.403
1,1,0.978
1,1,0.58
1,1,0.51
0,0,0.424
0,1,0.546
0,1,0.639
0,1,0.676
0,1,0.639
1,1,0.757
1,1,0.883
1,0,0.301
1,1,0.846
1,0,0.129
1,1,0.76
1,0,0.351
1,1,0.803
1,1,0.527
1,1,0.836
1,0,0.417
1,1,0.656
1,1,0.977
1,1,0.815
1,1,0.869
0,0,0.474
0,0,0.346
1,0,0.188
0,1,0.805
1,1,0.872
1,0,0.466
1,1,0.72
0,0,0.163
0,0,0.085
0,0,0.124
1,1,0.876
0,0,0.451
0,0,0.185
1,1,0.937
1,1,0.615
0,0,0.312
1,1,0.924
1,1,0.638
1,1,0.891
0,1,0.621
1,0,0.421
0,0,0.254
0,0,0.225
1,1,0.577

```
0,1,0.579
0,1,0.628
0,1,0.855
1,1,0.955
0,0,0.331
1,0,0.298
0,0,0.047
0,0,0.173
1,1,0.96
0,0,0.481
0,0,0.39
0,0,0.088
1,0,0.417
0,0,0.12
1,1,0.871
0,1,0.522
0,0,0.312
1,1,0.695
0,0,0.155
0,0,0.352
1,1,0.561
0,0,0.076
0,1,0.923
1,0,0.169
0,0,0.032
1,1,0.63
0,0,0.126
0,0,0.15
1,0,0.348
0,0,0.188
0,1,0.755
1,1,0.813
0,0,0.418
1,0,0.161
1,0,0.316
0,1,0.558
1,1,0.641
1,0,0.305
```

User-defined extensions

A user-defined extension (UDx) is a component that expands Vertica functionality—for example, new types of data analysis and the ability to parse and load new types of data.

This section provides an overview of how to install and use a UDx. If you are using a UDx developed by a third party, consult its documentation for detailed installation and usage instructions.

In this section

- Loading UDxs
- Installing Java on Vertica hosts
- UDx restrictions
- Fenced and unfenced modes
- Updating UDx libraries
- Listing the UDxs contained in a library
- Using wildcards in your UDx

Loading UDxs

User-defined extensions (UDxs) are contained in libraries. A library can contain multiple UDxs. To add UDxs to Vertica, you must:

1. Deploy the library (once per library).
2. Create each UDx (once per UDx).

If you are using UDxs written in Java, you must also set up a Java runtime environment. See Installing Java on Vertica hosts .

Deploying libraries

To deploy a library to your Vertica database:

1. Copy the UDx shared library file (.so), Python file, Java JAR file, or R functions file that contains your function to a node on your Vertica cluster. You do not need to copy it to every node.
2. Connect to the node where you copied the library (for example, using vsql).
3. Add your library to the database catalog using the CREATE LIBRARY statement.

```
=> CREATE LIBRARY libname AS '/path_to_lib/filename'
  LANGUAGE 'language';
```

libname is the name you want to use to reference the library. *path_to_lib/filename* is the fully-qualified path to the library or JAR file you copied to the host. *language* is the implementation language.

For example, if you created a JAR file named TokenizeStringLib.jar and copied it to the dbadmin account's home directory, you would use this command to load the library:

```
=> CREATE LIBRARY tokenizelib AS '/home/dbadmin/TokenizeStringLib.jar'
  LANGUAGE 'Java';
```

You can load any number of libraries into Vertica.

Privileges

Superusers can create, modify, and drop any library. Users with the UDXDEVELOPER role or explicit grants can also act on libraries, as shown in the following table:

Operation	**Requires**
CREATE LIBRARY	UDXDEVELOPER
Replace a library (CREATE OR REPLACE LIBRARY)	UDXDEVELOPER and one of: • owner of library being replaced • DROP privilege on the target library
DROP LIBRARY	UDXDEVELOPER and one of: • owner of library being dropped • DROP privilege on the target library
ALTER LIBRARY	UDXDEVELOPER and owner

Creating UDx functions

After the library is loaded, define individual UDxs using SQL statements such as CREATE FUNCTION and CREATE SOURCE . These statements assign SQL function names to the extension classes in the library. They add the UDx to the database catalog and remain available after a database restart.

The statement you use depends on the type of UDx you are declaring, as shown in the following table:

UDx Type	**SQL Statement**
Aggregate Function (UDAF)	CREATE AGGREGATE FUNCTION
Analytic Function (UDAnF)	CREATE ANALYTIC FUNCTION
Scalar Function (UDSF)	CREATE FUNCTION (scalar)
Transform Function (UDTF)	CREATE TRANSFORM FUNCTION
Load (UDL): Source	CREATE SOURCE
Load (UDL): Filter	CREATE FILTER

Load (UDL): Parser	CREATE PARSER

If a UDx of the given name already exists, you can replace it or instruct Vertica to not replace it. To replace it, use the OR REPLACE syntax, as in the following example:

```
=> CREATE OR REPLACE TRANSFORM FUNCTION tokenize
   AS LANGUAGE 'C++' NAME 'TokenFactory' LIBRARY TransformFunctions;
CREATE TRANSFORM FUNCTION
```

You might want to replace an existing function to change between fenced and unfenced modes.

Alternatively, you can use IF NOT EXISTS to prevent the function from being created again if it already exists. You might want to use this in upgrade or test scripts that require, and therefore load, UDxs. By using IF NOT EXISTS, you preserve the original definition including fenced status. The following example shows this syntax:

```
--- original creation:
=> CREATE TRANSFORM FUNCTION tokenize
   AS LANGUAGE 'C++' NAME 'TokenFactory' LIBRARY TransformFunctions NOT FENCED;
CREATE TRANSFORM FUNCTION

--- function is not replaced (and is still unfenced):
=> CREATE TRANSFORM FUNCTION IF NOT EXISTS tokenize
   AS LANGUAGE 'C++' NAME 'TokenFactory' LIBRARY TransformFunctions FENCED;
CREATE TRANSFORM FUNCTION
```

After you add the UDx to the database, you can use your extension within SQL statements. The database superuser can grant access privileges to the UDx for users. See GRANT (user defined extension) for details.

When you call a UDx, Vertica creates an instance of the UDx class on each node in the cluster and provides it with the data it needs to process.

Installing Java on Vertica hosts

If you are using UDxs written in Java, follow the instructions in this section.

You must install a Java Virtual Machine (JVM) on every host in your cluster in order for Vertica to be able to execute your Java UDxs.

Installing Java on your Vertica cluster is a two-step process:

1. Install a Java runtime on all of the hosts in your cluster.
2. Set the JavaBinaryForUDx configuration parameter to tell Vertica the location of the Java executable.

Installing a Java runtime

For Java-based features, Vertica requires a 64-bit Java 6 (Java version 1.6) or later Java runtime. Vertica supports runtimes from either Oracle or OpenJDK. You can choose to install either the Java Runtime Environment (JRE) or Java Development Kit (JDK), since the JDK also includes the JRE.

Many Linux distributions include a package for the OpenJDK runtime. See your Linux distribution's documentation for information about installing and configuring OpenJDK.

To install the Oracle Java runtime, see the Java Standard Edition (SE) Download Page. You usually run the installation package as root in order to install it. See the download page for instructions.

Once you have installed a JVM on each host, ensure that the java command is in the search path and calls the correct JVM by running the command:

```
$ java -version
```

This command should print something similar to:

```
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)
```

> **Note**
> Any previously installed Java VM on your hosts may interfere with a newly installed Java runtime. See your Linux distribution's documentation for instructions on configuring which JVM is the default. Unless absolutely required, you should uninstall any incompatible version of Java before

installing the Java 6 or Java 7 runtime.

Setting the JavaBinaryForUDx configuration parameter

The JavaBinaryForUDx configuration parameter tells Vertica where to look for the JRE to execute Java UDxs. After you have installed the JRE on all of the nodes in your cluster, set this parameter to the absolute path of the Java executable. You can use the symbolic link that some Java installers create (for example /usr/bin/java). If the Java executable is in your shell search path, you can get the path of the Java executable by running the following command from the Linux command line shell:

```
$ which java
/usr/bin/java
```

If the java command is not in the shell search path, use the path to the Java executable in the directory where you installed the JRE. Suppose you installed the JRE in /usr/java/default (which is where the installation package supplied by Oracle installs the Java 1.6 JRE). In this case the Java executable is /usr/java/default/bin/java .

You set the configuration parameter by executing the following statement as a database superuser :

```
=> ALTER DATABASE DEFAULT SET PARAMETER JavaBinaryForUDx = '/usr/bin/java';
```

See ALTER DATABASE for more information on setting configuration parameters.

To view the current setting of the configuration parameter, query the CONFIGURATION_PARAMETERS system table:

```
=> \x
Expanded display is on.
=> SELECT * FROM CONFIGURATION_PARAMETERS WHERE parameter_name = 'JavaBinaryForUDx';
-[ RECORD 1 ]----------------+-----------------------------------------------------------
node_name                    | ALL
parameter_name               | JavaBinaryForUDx
current_value                | /usr/bin/java
default_value                |
change_under_support_guidance | f
change_requires_restart      | f
description                  | Path to the java binary for executing UDx written in Java
```

Once you have set the configuration parameter, Vertica can find the Java executable on each node in your cluster.

Note
Since the location of the Java executable is set by a single configuration parameter for the entire cluster, you must ensure that the Java executable is installed in the same path on all of the hosts in the cluster.

UDx restrictions

Some UDx types have special considerations or restrictions.

UDxs written in Java and R do not support complex types.

Aggregate functions

You cannot use the DISTINCT clause in queries with more than one aggregate function or provide inputs or return values containing complex types.

Analytic functions

UDAnFs do not support framing windows using ROWS .

Only UDAnFs written in C++ can use complex types.

As with Vertica's built-in analytic functions, UDAnFs cannot be used with MATCH clause functions .

Scalar functions

If the result of applying a UDSF is an invalid record, COPY aborts the load even if CopyFaultTolerantExpressions is set to true.

A ROW returned from a UDSF cannot be used as an argument to COUNT.

Transform functions

A query that includes a UDTF cannot:

- Include statements other than the SELECT statement that calls the UDTF and a PARTITION BY expression unless the UDTF is marked as a one-to-many UDTF
- Call an analytic function
- Call another UDTF
- Include one of the following clauses:
  - TIMESERIES
  - Pattern matching
  - Gap filling and interpolation

Load functions

Installing an untrusted UDL function can compromise the security of the server. UDxs can contain arbitrary code. In particular, user-defined source functions can read data from any arbitrary location. It is up to the developer of the function to enforce proper security limitations. Superusers must not grant access to UDxs to untrusted users.

You cannot ALTER UDL functions.

UDFilter and UDSource functions do not support complex types.

Fenced and unfenced modes

User-defined extensions (UDxs) written in the C++ programming language have the option of running in fenced or unfenced mode. Fenced mode runs the UDx code outside of the main Vertica process in a separate zygote process. UDxs that use unfenced mode run directly within the Vertica process.

Fenced mode

You can run most C++ UDxs in fenced mode. Fenced mode uses a separate zygote process, so fenced UDx crashes do not impact the core Vertica process. There is a small performance impact when running UDx code in fenced mode. On average, using fenced mode adds about 10% more time to execution compared to unfenced mode.

Fenced mode is currently available for all C++ UDxs with the exception of user-defined aggregates. All UDxs developed in the Python, R, and Java programming languages must run in fenced mode, since the Python, R, and Java runtimes cannot run directly within the Vertica process.

Using fenced mode does not affect the development of your UDx. Fenced mode is enabled by default for UDxs that support fenced mode. Optionally, you can issue the CREATE FUNCTION command with the NOT FENCED modifier to disable fenced mode for the function. Additionally, you can enable or disable fenced mode on any fenced mode-supported C++ UDx by using the ALTER FUNCTION command.

Unfenced mode

Unfenced UDxs run within Vertica, so they have little overhead, and can perform almost as fast as Vertica's own built-in functions. However, because they run within Vertica directly, any bugs in their code (memory leaks, for example) can destabilize the main Vertica process and bring one or more database nodes down.

Important
Always carefully review code downloaded from third-party sources and test UDxs in a test or staging environment before deciding to run them in unfenced mode in production.

About the zygote process

The Vertica zygote process starts when Vertica starts. Each node has a single zygote process. Side processes are created "on demand". The zygote listens for requests and spawns a UDx side session that runs the UDx in fenced mode when a UDx is called by the user.

About fenced mode logging:

UDx code that runs in fenced mode is logged in the UDxZygote.log and is stored in the UDxLogs directory in the catalog directory of Vertica. Log entries for the side process are denoted by the UDx language (for example, C++), node, zygote process ID, and the UDxSideProcess ID.

For example, for the following query returns the current fenced processes:

```
=> SELECT * FROM UDX_FENCED_PROCESSES;
   node_name     |   process_type   |            session_id            |  pid  | port  | status
-----------------+------------------+----------------------------------+-------+-------+--------
 v_vmart_node0001 | UDxZygoteProcess |                                  | 27468 | 51900 | UP
 v_vmart_node0001 | UDxSideProcess   | localhost.localdoma-27465:0x800b |  5677 | 44123 | UP
```

Below is the corresponding log file for the fenced processes returned in the previous query:

```
2016-05-16 11:24:43.990 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 UDx side process started
 11:24:43.996 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 Finished setting up signal handlers.
 11:24:43.996 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 My port: 44123
 11:24:43.996 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 My address: 0.0.0.0
 11:24:43.996 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 Vertica port: 51900
 11:24:43.996 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 Vertica address: 127.0.0.1
 11:25:19.749 [C++-localhost.localdoma-27465:0x800b-5677]  0x41837940 Setting memory resource limit to -1
 11:30:11.523 [C++-localhost.localdoma-27465:0x800b-5677]  0x41837940 Exiting UDx side process
```

The last line indicates that the side process was killed. In this case it was killed when the user session (vsql) closed.

About fenced mode configuration parameters

Fenced mode supports the following configuration parameters :

- FencedUDxMemoryLimitMB : The maximum memory size, in MB, to use for fenced mode processes. The default is -1 (no limit). The side process is killed if this limit is exceeded.
- ForceUDxFencedMode : When set to 1 , force all UDx's that support fenced mode to run in fenced mode even if their definition specified NOT FENCED. The default is 0 (disabled).
- UDxFencedBlockTimeout : The maximum time, in seconds, that the Vertica server waits for a UDx to return before aborting with ERROR 3399. The default is 60 .

See also

- CREATE LIBRARY
- CREATE FUNCTION
- CREATE TRANSFORM FUNCTION
- CREATE ANALYTIC FUNCTION
- ALTER FUNCTION (scalar)
- UDX_FENCED_PROCESSES

Updating UDx libraries

There are two cases where you need to update libraries that you have already deployed:

- When you have upgraded Vertica to a new version that contains changes to the SDK API. For your libraries to work with the new server version, you need to recompile them with new version of the SDK. See UDx library compatibility with new server versions for more information.
- When you have made changes to your UDxs and you want to deploy these changes. Before updating your UDx library, you need to determine if you have changed the signature of any of the functions contained in the library. If you have, you need to drop the functions from the Vertica catalog before you update the library.

In this section

- UDx library compatibility with new server versions
- Determining if a UDx signature has changed
- Deploying a new version of your UDx library

UDx library compatibility with new server versions

The Vertica SDK defines an application programming interface (API) that UDxs use to interact with the database. When developers compile their UDx code, it is linked to the SDK code to form a library. This library is only compatible with Vertica servers that support the version of the SDK API used to compile the code. The library and servers that share the same API version are compatible on a binary level (referred to as "binary compatible").

The Vertica server returns an error message if you attempt to load a library that is not binary compatible with it. Similarly, if you upgrade your Vertica server to a version that supports a new SDK API, any existing UDx that relies on newly-incompatible libraries returns an error messages when you call it:

```
ERROR 2858:  Could not find function definition
HINT:
This usually happens due to missing or corrupt libraries, libraries built
with the wrong SDK version, or due to a concurrent session dropping the library
or function. Try recreating the library and function
```

To resolve this issue, you must install UDx libraries that have been recompiled with the correct version of the SDK.

New versions of the Vertica server do not always change the SDK API version. The SDK API version changes whenever OpenText changes the components that make up the SDK. If the SDK API does not change in a new version of the server, then the old libraries remain compatible with the new server.

The SDK API almost always changes in Vertica releases (major, minor, service pack) as OpenText expands the SDK's features. Vertica will never change the API in a hotfix patch.

These policies mean that you must update UDx libraries when you upgrade between major versions. For example, if you upgrade from version 10.0 to 10.1, you must update your UDx libraries.

> **Note**
> A UDx written in a scripting language has no compiled binary, and so does not need to maintain binary compatibility from one version to another. UDxs written in scripting languages only become incompatible if the APIs used in the SDK actually change. For example, if the number of arguments to an API call changes, a UDx has to be changed to use the new number of arguments.

Pre-upgrade steps

Before upgrading your Vertica server, consider whether you have any UDx libraries that may be incompatible with the new version. Consult the release notes of the new server version to determine whether the SDK API has changed between the version of Vertica server you currently have installed and the new version. As mentioned previously, only upgrades from a previous major version or from the initial release of a major version to a service pack release can cause your currently-loaded UDx libraries to become incompatible with the server.

Any UDx libraries that are incompatible with the new version of the Vertica server must be recompiled. If you got the UDx library from a third party, you need to see if a new version has been released. If so, deploy the new version after you have upgraded the server (see Deploying a new version of your UDx library).

If you developed the UDx yourself (or if you have the source code) you must:

1. Recompile your UDx library using the new version of the Vertica SDK. See Compiling your C++ library or Compiling and packaging a Java library for more information.
2. Deploy the new version of your library. See Deploying a new version of your UDx library .

Determining if a UDx signature has changed

You need to be careful when making changes to UDx libraries that contain functions you have already deployed in your Vertica database. When you deploy a new version of your UDx library, Vertica does not ensure that the signatures of the functions that are defined in the library match the signature of the function that is already defined in the Vertica catalog. If you have changed the signature of a UDx in the library then update the library in the Vertica database, calls to the altered UDx will produce errors.

Making any of the following changes to a UDx alters its signature:

- Changing the number of arguments accepted or the data type of any argument accepted by your function (not including polymorphic functions).
- Changing the number or data types of any return values or output columns.
- Changing the name of the factory class that Vertica uses to create an instance of your function code.
- Changing the null handling or volatility behavior of your function.
- Removed the function's factory class from the library completely.

The following changes do not alter the signature of your function, and do not require you to drop the function before updating the library:

- Changing the number or type of arguments handled by a polymorphic function. Vertica does not process the arguments the user passes to a polymorphic function.
- Changing the name, data type, or number of parameters accepted by your function. The parameters your function accepts are not determined by the function signature. Instead, Vertica passes all of the parameters the user included in the function call, and your function processes them at runtime. See UDx parameters for more information about parameters.
- Changing any of the internal processing performed by your function.
- Adding new UDxs to the library.

After you drop any functions whose signatures have changed, you load the new library file, then re-create your altered functions. If you have not made any changes to the signature of your UDxs, you can just update the library file in your Vertica database without having to drop or alter your function definitions. As long as the UDx definitions in the Vertica catalog match the signatures of the functions in your library, function calls will work transparently after you have updated the library. See Deploying a new version of your UDx library .

Deploying a new version of your UDx library

You need to deploy a new version of your UDx library if:

- You have made changes to the library that you now want to roll out to your Vertica database.
- You have upgraded Vertica to a new version whose SDK is incompatible with the previous version.

The process of deploying a new version of your library is similar to deploying it initially.

1. If you are deploying a UDx library developed in C++ or Java, you must compile it with the current version of the Vertica SDK.
2. Copy your UDx's library file (a .so file for libraries developed in C++, a .py file for libraries developed in Python, or a .jar file for libraries developed in Java) or R source file to a host in your Vertica database.
3. Connect to the host using vsql .
4. If you have changed the signature of any of the UDxs in the shared library, you must drop them using DROP statements such as DROP FUNCTION or DROP SOURCE . If you are unsure whether any of the signatures of your functions have changed, see Determining if a UDx signature has changed .

   > **Note**
   > If all of the UDx signatures in your library have changed, you may find it more convenient to drop the library using the DROP LIBRARY statement with the CASCADE option to drop the library and all of the functions and loaders that reference it. Droping the library can save you the time it would take to drop each UDx individually. You can then reload the library and recreate all of the extensions using the same process you used to deploy the library in the first place. See CREATE LIBRARY .

5. Use the ALTER LIBRARY statement to update the UDx library definition with the file you copied in step 1. For example, if you want to update the library named ScalarFunctions with a file named ScalarFunctions-2.0.so in the dbadmin user's home directory, you could use the command:

   ```
   => ALTER LIBRARY ScalarFunctions AS '/home/dbadmin/ScalarFunctions-2.0.so';
   ```

   After you have updated the UDx library definition to use the new version of your shared library, the UDxs that are defined using classes in your UDx library begin using the new shared library file without any further changes.
6. If you had to drop any functions in step 4, recreate them using the new signature defined by the factory classes in your library. See CREATE FUNCTION statements .

Listing the UDxs contained in a library

Once a library has been loaded using the CREATE LIBRARY statement, you can find the UDxs and UDLs it contains by querying the USER_LIBRARY_MANIFEST system table:

```
=> CREATE LIBRARY ScalarFunctions AS '/home/dbadmin/ScalarFunctions.so';
CREATE LIBRARY
=> \x
Expanded display is on.
=> SELECT * FROM USER_LIBRARY_MANIFEST WHERE lib_name = 'ScalarFunctions';
-[ RECORD 1 ]-------------------
schema_name | public
lib_name    | ScalarFunctions
lib_oid     | 45035996273792402
obj_name    | RemoveSpaceFactory
obj_type    | Scalar Function
arg_types   | Varchar
return_type | Varchar
-[ RECORD 2 ]-------------------
schema_name | public
lib_name    | ScalarFunctions
lib_oid     | 45035996273792402
obj_name    | Div2intsInfo
obj_type    | Scalar Function
arg_types   | Integer, Integer
return_type | Integer
-[ RECORD 3 ]-------------------
schema_name | public
lib_name    | ScalarFunctions
lib_oid     | 45035996273792402
obj_name    | Add2intsInfo
obj_type    | Scalar Function
arg_types   | Integer, Integer
return_type | Integer
```

The obj_name column lists the factory classes contained in the library. These are the names you use to define UDxs and UDLs in the database catalog using statements such as CREATE FUNCTION and CREATE SOURCE .

Using wildcards in your UDx

Vertica supports wildcard * characters in the place of column names in user-defined functions.

You can use wildcards when:

- Your query contains a table in the FROM clause
- You are using a Vertica-supported development language
- Your UDx is running in fenced or unfenced mode

Supported SQL statements

The following SQL statements can accept wildcards:

- DELETE
- INSERT
- SELECT
- UPDATE

Unsupported configurations

The following situations do not support wildcards:

- You cannot pass a wildcard in the OVER clause of a query
- You cannot us a wildcard with a DROP statement
- You cannot use wildcards with any other arguments

Examples

These examples show wildcards and user-defined functions in a range of data manipulation operations.

DELETE statements:

```
=> DELETE FROM tablename WHERE udf(tablename.*) = 5;
```

INSERT statements:

```
=> INSERT INTO table1 SELECT udf(*) FROM table2;
```

SELECT statements:

```
=> SELECT udf(*) FROM tablename;
=> SELECT udf(tablename.*) FROM tablename;
=> SELECT udf(f.*) FROM table f;
=> SELECT udf(*) FROM table1,table2;
=> SELECT udf1( udf2(*) ) FROM table1,table2;
=> SELECT udf( db.schema.table.*) FROM tablename;
=> SELECT udf(sub.*) FROM (select col1, col2 FROM table) sub;
=> SELECT x FROM tablename WHERE udf(*) = y;
=> WITH sub as (SELECT * FROM tablename) select x, udf(*) FROM sub;
=> SELECT udf( * using parameters x=1) FROM tablename;
=> SELECT udf(table1.*, table2.col2) FROM table1,table2;
```

UPDATE statements:

```
=> UPDATE tablename set col1 = 4 FROM tablename WHERE udf(*) = 3;
```

Developing user-defined extensions (UDxs)

User-defined extensions (UDxs) are functions contained in external libraries that are developed in C++, Python, Java, or R using the Vertica SDK. The external libraries are defined in the Vertica catalog using the CREATE LIBRARY statement. They are best suited for analytic operations that are difficult to perform in SQL, or that need to be performed frequently enough that their speed is a major concern.

The primary strengths of UDxs are:

- They can be used anywhere an internal function can be used.
- They take full advantage of Vertica's distributed computing features. The extensions usually execute in parallel on each node in the cluster.
- They are distributed to all nodes by Vertica. You only need to copy the library to the initiator node.
- All of the complicated aspects of developing a distributed piece of analytic code are handled for you by Vertica. Your main programming task is to read in data, process it, and then write it out using the Vertica SDK APIs.

There are a few things to keep in mind about developing UDxs:

- UDxs can be developed in the programming languages C++, Python, Java, and R. (Not all UDx types support all languages.)
- UDxs written in Java always run in fenced mode, because the Java Virtual Machine that executes Java programs cannot run directly within the Vertica process.
- UDxs written in Python and R always run in fenced mode.
- UDxs developed in C++ have the option of running in unfenced mode, which means they load and run directly in the Vertica database process. This option provides the lowest overhead and highest speed. However, any bugs in the UDx's code can cause database instability. You must thoroughly test any UDxs you intend to run in unfenced mode before deploying them in a live environment. Consider whether the performance boost of running a C++ UDx unfenced is worth the potential database instability that a buggy UDx can cause.
- Because a UDx runs on the Vertica cluster, it can take processor time and memory away from the database processes. A UDx that consumes large amounts of computing resources can negatively impact database performance.

Types of UDxs

Vertica supports five types of user-defined extensions:

- User-defined scalar functions (UDSFs) take in a single row of data and return a single value. These functions can be used anywhere a native function can be used, except CREATE TABLE BY PARTITION and SEGMENTED BY expressions. UDSFs can be developed in C++, Python, Java, and R.
- User-defined aggregate functions (UDAF) allow you to create custom Aggregate functions specific to your needs. They read one column of data, and return one output column. UDAFs can be developed in C++.
- User-defined analytic functions (UDAnF) are similar to UDSFs, in that they read a row of data and return a single row. However, the function can read input rows independently of outputting rows, so that the output values can be calculated over several input rows. The function can be used with the query's OVER() clause to partition rows. UDAnFs can be developed in C++ and Java.
- User-defined transform functions (UDTFs) operate on table partitions (as specified by the query's OVER() clause) and return zero or more rows of data. The data they return can be an entirely new table, unrelated to the schema of the input table, with its own ordering and segmentation expressions. They can only be used in the SELECT list of a query. UDTFs can be developed in C++, Python, Java, and R.
  To optimize query performance, you can use live aggregate projections to pre-aggregate the data that a UDTF returns. For more information, see Pre-aggregating UDTF results.

- User-defined load allows you to create custom sources , filters , and parsers to load data. These extensions can be used in COPY statements. UDLs can be developed C++, Java and Python.

While each UDx type has a unique base class, developing them is similar in many ways. Different UDx types can also share the same library.

Structure

Each UDx type consists of two primary classes. The main class does the actual work (a transformation, an aggregation, and so on). The class usually has at least three methods: one to set up, one to tear down (release reserved resources), and one to do the work. Sometimes additional methods are defined.

The main processing method receives an instance of the ServerInterface class as an argument. This object is used by the underlying Vertica SDK code to make calls back into the Vertica process, for example to allocate memory. You can use this class to write to the server log during UDx execution.

The second class is a singleton factory. It defines one method that produces instances of the first class, and might define other methods to manage parameters.

When implementing a UDx you must subclass both classes.

Conventions

The C++, Python, and Java APIs are nearly identical. Where possible, this documentation describes these interfaces without respect to language. Documentation specific to C++, Python, or Java is covered in language-specific sections.

Because some documentation is language-independent, it is not always possible to use ideal, language-based terminology. This documentation uses the term "method" to refer to a Java method or a C++ member function.

See also

Loading UDxs

In this section

- Developing with the Vertica SDK
- Arguments and return values
- UDx parameters
- Errors, warnings, and logging
- Handling cancel requests
- Aggregate functions (UDAFs)
- Analytic functions (UDAnFs)
- Scalar functions (UDSFs)
- Transform functions (UDTFs)
- User-defined load (UDL)

Developing with the Vertica SDK

Before you can write a user-defined extension you must set up a development environment. After you do so, a good test is to download, build, and run the published examples.

In addition to covering how to set up your environment, this section covers general information about working with the Vertica SDK, including language-specific considerations.

In this section

- Setting up a development environment
- Downloading and running UDx example code
- C++ SDK
- Java SDK
- Python SDK
- R SDK
- Debugging tips

Setting up a development environment

Before you start developing your UDx, you need to configure your development and test environments. Development and test environments must use the same operating system and Vertica version as the production environment.

For additional language-specific requirements, see the following topics:

- Setting up the C++ SDK

- Setting up the Java SDK
- Python SDK
- Installing/upgrading the R language pack for Vertica

Development environment options

The language that you use to develop your UDx determines the setup options and requirements for your development environment. C++ developers can use the C++ UDx container , and all developers can use a non-production Vertica environment .

C++ UDx container

C++ developers can develop with the C++ UDx container. The UDx-container GitHub repository provides the tools to build a container that packages the binaries, libraries, and compilers required to develop C++ Vertica extensions. The C++ UDx container has the following build options:

- CentOS or Ubuntu base image
- Vertica 10.x and 11.x versions

For requirement, build, and test details, see the repository README .

Non-production Vertica environments

You can use a node in a non-production Vertica database or another machine that runs the same operating system and Vertica version as your production environment. For specific requirements and dependencies, refer to Operating System Requirements and Language Requirements .

Test environment options

To test your UDx, you need access to a non-production Vertica database. You have the following options:

- Install a single-node Vertica database on your development machine.
- Download and build a containerized test environment.

Containerized test environments

Vertica provides the following containerized options to simplify your test environment setup:

- Vertica Community Edition (CE) container image from the Vertica DockerHub registry . See the Overview for base image and Vertica version details.
  For details about Vertica CE, see Vertica community edition (CE) .
- Custom CE image with the one-node-ce GitHub repository . You can build CentOS- and Debian-based containers that are compatible with Vertica 10.x and 11.x versions.

Operating system requirements

Develop your UDx code on the same Linux platform that you use for your production Vertica database cluster. Centos- and Debian-based operating systems each require that you download additional packages.

CentOS-based operating systems

Installations on the following CentOS-based operating systems require the devtoolset-7 package:

- CentOS
- Red Hat Enterprise Linux
- Oracle Enterprise Linux

Consult the documentation for your operating system for the specific installation command.

Debian-based operating systems

Installations on the following Debian-based operating systems require the GCC package version 7 or later:

> **Note**
> Vertica has not tested UDx builds that use GCC versions later than GCC 8 .

- Debian
- Ubuntu
- SUSE
- OpenSUSE
- Amazon Linux (The GCC package is pre-installed on Amazon Linux)

Consult the documentation for your operating system for the specific installation command.

Downloading and running UDx example code

You can download all of the examples shown in this documentation, and many more, from the Vertica GitHub repository . This repository includes examples of all types of UDxs.

You can download the examples in either of two ways:

- Download the ZIP file. Extract the contents of the file into a directory.
- Clone the repository. Using a terminal window, run the following command:

```
$ git clone https://github.com/vertica/UDx-Examples.git
```

The repository includes a makefile that you can use to compile the C++ and Java examples. It also includes .sql files that load and use the examples. See the README file for instructions on compiling and running the examples. To compile the examples you will need g++ or a JDK and make. See Setting up a development environment for related information.

Running the examples not only helps you understand how a UDx works, but also helps you ensure your development environment is properly set up to compile UDx libraries.

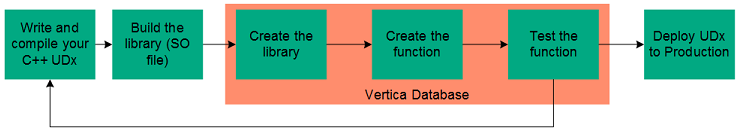
See also

- Aggregate functions (UDAFs)
- Analytic functions (UDAnFs)
- Scalar functions (UDSFs)
- Transform functions (UDTFs)
- User-defined load (UDL)

C++ SDK

The Vertica SDK supports writing both fenced and unfenced UDxs in C++ 11. You can download, compile, and run the examples; see Downloading and running UDx example code . Running the examples is a good way to verify that your development environment has all needed libraries.

If you do not have access to a Vertica test environment, you can install Vertica on your development machine and run a single node. Each time you rebuild your UDx library, you need to re-install it into Vertica. The following diagram illustrates the typical development cycle.



This section covers C++-specific topics that apply to all UDx types. For information that applies to all languages, see Arguments and return values , UDx parameters , Errors, warnings, and logging , Handling cancel requests and the sections for specific UDx types. For full API documentation, see the C++ SDK Documentation .

In this section

- Setting up the C++ SDK
- Compiling your C++ library
- Adding metadata to C++ libraries
- C++ SDK data types
- Resource use for C++ UDxs

Setting up the C++ SDK

The Vertica C++ Software Development Kit (SDK) is distributed as part of the server installation. It contains the source and header files you need to create your UDx library. For examples that you can compile and run, see Downloading and running UDx example code .

Requirements

At a minimum, install the following on your development machine:

- devtoolset-7 package (CentOS) or GCC package (Debian), including GCC version 7 or later and an up-to-date libstdc++ package.

> **Note**
> Vertica has not tested UDx builds that use GCC versions later than GCC 8 .

- g++ and its associated toolchain, such as ld . Some Linux distributions package g++ separately from GCC .
- A copy of the Vertica SDK.

> **Note**
> The Vertica binaries are compiled using the default version of g++ installed on the supported Linux platforms.

You must compile with a std flag value of c++11 or later.

The following optional software packages can simplify development:

- make , or some other build-management tool.
- gdb , or some other debugger.
- Valgrind, or similar tools that detect memory leaks.

If you want to use any third-party libraries, such as statistical analysis libraries, you need to install them on your development machine. If you do not statically link these libraries into your UDx library, you must install them on every node in the cluster. See Compiling your C++ library for details.

SDK files

The SDK files are located in the sdk subdirectory under the root Vertica server directory (usually, /opt/vertica/sdk). This directory contains a subdirectory, include , which contains the headers and source files needed to compile UDx libraries.

There are two files in the include directory you need when compiling your UDx:

- Vertica.h is the main header file for the SDK. Your UDx code needs to include this file in order to find the SDK's definitions.
- Vertica.cpp contains support code that needs to be compiled into the UDx library.

Much of the Vertica SDK API is defined in the VerticaUDx.h header file (which is included by the Vertica.h file). If you're curious, you might want to review the contents of this file in addition to reading the API documentation.

Finding the current SDK version

You must develop your UDx using the same SDK version as the database in which you plan to use it. To display the SDK version currently installed on your system, run the following command in vsql:

```
=> SELECT sdk_version();
```

Running the examples

You can download the examples from the GitHub repository (see Downloading and running UDx example code). Compiling and running the examples helps you to ensure that your development environment is properly set up.

To compile all of the examples, including the Java examples, issue the following command in the Java-and-C++ directory under the examples directory:

```
$ make
```

> **Note**
> To compile the examples, you must have a g++ development environment installed. To install a g++ development environment on Red Hat systems, run yum install gcc gcc-c++ make.

Compiling your C++ library

GNU g++ is the only supported compiler for compiling UDx libraries. Always compile your UDx code on the same version of Linux that you use on your Vertica cluster.

When compiling your library, you must always:

- Compile with a -std flag value of c++11 or later.
- Pass the -shared and -fPIC flags to the linker. The simplest method is to just pass these flags to g++ when you compile and link your library.
- Use the -Wno-unused-value flag to suppress warnings when macro arguments are not used. If you do not use this flag, you may get "left-hand operand of comma has no effect" warnings.

- Compile sdk/include/Vertica.cpp and link it into your library. This file contains support routines that help your UDx communicate with Vertica. The easiest way to do this is to include it in the g++ command to compile your library. Vertica supplies this file as C++ source rather than a library to limit library compatibility issues.
- Add the Vertica SDK include directory in the include search path using the g++ -I flag.

The SDK examples include a working makefile. See Downloading and running UDx example code .

Example of compiling a UDx

The following command compiles a UDx contained in a single source file named MyUDx.cpp into a shared library named MyUDx.so :

```
g++ -I /opt/vertica/sdk/include -Wall -shared -Wno-unused-value \
    -fPIC -o MyUDx.so MyUDx.cpp /opt/vertica/sdk/include/Vertica.cpp
```

Important
Vertica only supports UDx development on 64-bit architectures.

After you debug your UDx, you are ready to deploy it. Recompile your UDx using the -O3 flag to enable compiler optimization.

You can add additional source files to your library by adding them to the command line. You can also compile them separately and then link them together.

> **Tip**
> The examples subdirectory in the Vertica SDK directory contains a make file that you can use as starting point for your own UDx project.

Handling external libraries

You must link your UDx library to any supporting libraries that your UDx code relies on.These libraries might be either ones you developed or others provided by third parties. You have two options for linking:

- Statically link the support libraries into your UDx. The benefit of this method is that your UDx library does not rely on external files. Having a single UDx library file simplifies deployment because you just transfer a single file to your Vertica cluster. This method's main drawback is that it increases the size of your UDx library file.
- Dynamically link the library to your UDx. You must sometimes use dynamic linking if a third-party library does not allow static linking. In this case, you must copy the libraries to your Vertica cluster in addition to your UDx library file.

Adding metadata to C++ libraries

You can add metadata, such as author name, the version of the library, a description of your library, and so on to your library. This metadata lets you track the version of your function that is deployed on a Vertica Analytic Database cluster and lets third-party users of your function know who created the function. Your library's metadata appears in the USER_LIBRARIES system table after your library has been loaded into the Vertica Analytic Database catalog.

You declare the metadata for your library by calling the RegisterLibrary() function in one of the source files for your UDx. If there is more than one function call in the source files for your UDx, whichever gets interpreted last as Vertica Analytic Database loads the library is used to determine the library's metadata.

The RegisterLibrary() function takes eight string parameters:

```
RegisterLibrary(author,
                library_build_tag,
                library_version,
                library_sdk_version,
                source_url,
                description,
                licenses_required,
                signature);
```

- author contains whatever name you want associated with the creation of the library (your own name or your company's name for example).
- library_build_tag is a string you want to use to represent the specific build of the library (for example, the SVN revision number or a timestamp of when the library was compiled). This is useful for tracking instances of your library as you are developing them.
- library_version is the version of your library. You can use whatever numbering or naming scheme you want.
- library_sdk_version is the version of the Vertica Analytic Database SDK Library for which you've compiled the library.

> **Note**
> This field isn't used to determine whether a library is compatible with a version of the Vertica Analytic Database server. The version of the Vertica Analytic Database SDK you use to compile your library is embedded in the library when you compile it. It is this information that Vertica Analytic Database server uses to determine if your library is compatible with it.

- source_url is a URL where users of your function can find more information about it. This can be your company's website, the GitHub page hosting your library's source code, or whatever site you like.
- description is a concise description of your library.
- licenses_required is a placeholder for licensing information. You must pass an empty string for this value.
- signature is a placeholder for a signature that will authenticate your library. You must pass an empty string for this value.

For example, the following code demonstrates adding metadata to the Add2Ints example (see C++ example: Add2Ints).

```
// Register the factory with Vertica
RegisterFactory(Add2IntsFactory);

// Register the library's metadata.
RegisterLibrary("Whizzo Analytics Ltd.",
                "1234",
                "2.0",
                "7.0.0",
                "http://www.example.com/add2ints",
                "Add 2 Integer Library",
                "",
                "");
```

Loading the library and querying the USER_LIBRARIES system table shows the metadata supplied in the call to RegisterLibrary() :

```
=> CREATE LIBRARY add2intslib AS '/home/dbadmin/add2ints.so';
CREATE LIBRARY
=> \x
Expanded display is on.
=> SELECT * FROM USER_LIBRARIES WHERE lib_name = 'add2intslib';
-[ RECORD 1 ]-----+----------------------------------------
schema_name       | public
lib_name          | add2intslib
lib_oid           | 45035996273869808
author            | Whizzo Analytics Ltd.
owner_id          | 45035996273704962
lib_file_name     | public_add2intslib_45035996273869808.so
md5_sum           | 732c9e145d447c8ac6e7304313d3b8a0
sdk_version       | v7.0.0-20131105
revision          | 125200
lib_build_tag     | 1234
lib_version       | 2.0
lib_sdk_version   | 7.0.0
source_url        | http://www.example.com/add2ints
description       | Add 2 Integer Library
licenses_required |
signature         |
```

C++ SDK data types

The Vertica SDK has typedefs and classes for representing Vertica data types within your UDx code. Using these typedefs ensures data type compatibility between the data your UDx processes and generates and the Vertica database. The following table describes some of the typedefs available. Consult the C++ SDK Documentation for a complete list, as well as lists of helper functions to convert and manipulate these data types.

For information about SDK support for complex data types, see Complex Types as Arguments and Return Values .

Type Definition	Description
Interval	A Vertica interval
IntervalYM	A Vertica year-to-month interval.
Timestamp	A Vertica timestamp
vint	A standard Vertica 64-bit integer
vbool	A Boolean value in Vertica
vbool_null	A null value for a Boolean data types
vfloat	A Vertica floating point value
VString	String data types (such as varchar and char) **Note:** Do not use a VString object to hold an intermediate result. Use a std::string or char[] instead.
VNumeric	Fixed-point data types from Vertica
VUuid	A Vertica universally unique identifier

Notes

- When making some Vertica SDK API calls (such as VerticaType::getNumericLength()) on objects, make sure they have the correct data type. To minimize overhead and improve performance, most of the APIs do not check the data types of the objects on which they are called. Calling a function on an incorrect data type can result in an error.
- You cannot create instances of VString or VNumeric yourself. You can manipulate the values of existing objects of these classes that Vertica passes to your UDx, and extract values from them. However, only Vertica can instantiate these classes.

Resource use for C++ UDxs

Your UDxs consume at least a small amount of memory by instantiating classes and creating local variables. This basic memory usage by UDxs is small enough that you do not need to be concerned about it.

If your UDx needs to allocate more than one or two megabytes of memory for data structures, or requires access to additional resources such as files, you must inform Vertica about its resource use. Vertica can then ensure that the resources your UDx requires are available before running a query that uses it. Even moderate memory use (10MB per invocation of a UDx, for example) can become an issue if there are many simultaneous queries that call it.

> **Note**
> If your UDx allocates its own memory, you must make **absolutely sure** it properly frees it. Failing to free even a single byte of allocated memory can have significant consequences at scale. Instead of having your code allocate its own memory, you should use the C++ vt_alloc macro, which uses Vertica's own memory manager to allocate and track memory. This memory is guaranteed to be properly disposed of when your UDx completes execution. See Allocating resources for UDxs for more information.

In this section

- Allocating resources for UDxs
- Allocating resources with the SDK macros
- Informing Vertica of resource requirements
- Setting memory limits for fenced-mode UDxs
- How resource limits are enforced

Allocating resources for UDxs

You have two options for allocating memory and file handles for your user-defined extensions (UDxs):

- Use Vertica SDK macros to allocate resources. This is the best method, since it uses Vertica's own resource manager, and guarantees that

resources used by your UDx are reclaimed. See Allocating resources with the SDK macros .

- While not the recommended option, you can allocate resources in your UDxs yourself using standard C++ methods (instantiating objects using new , allocating memory blocks using malloc() , etc.). You must manually free these resources before your UDx exits.

> **Note**
> You must be extremely careful if you choose to allocate your own resources in your UDx. Failing to free resources properly will have significant negative impact, especially if your UDx is running in unfenced mode.

Whichever method you choose, you usually allocate resources in a function named setup() in your UDx class. This function is called after your UDx function object is instantiated, but before Vertica calls it to process data.

If you allocate memory on your own in the setup() function, you must free it in a corresponding function named destroy() . This function is called after your UDx has performed all of its processing. This function is also called if your UDx returns an error (see Handling errors).

> **Note**
> Always use the setup() and destroy() functions to allocate and free resources instead of your own constructors and destructors. The memory for your UDx object is allocated from one of Vertica's own memory pools. Vertica always calls your UDx's destroy() function before it deallocates the object's memory. There is no guarantee that your UDx's destructor is will be called before the object is deallocated. Using the destroy() function ensures that your UDx has a chance to free its allocated resources before it is destroyed.

The following code fragment demonstrates allocating and freeing memory using a setup() and destroy() function.

```
class MemoryAllocationExample : public ScalarFunction
{
public:
   uint64* myarray;
   // Called before running the UDF to allocate memory used throughout
   // the entire UDF processing.
   virtual void setup(ServerInterface &srvInterface, const SizedColumnTypes
                      &argTypes)
   {
      try
      {
         // Allocate an array. This memory is directly allocated, rather than
         // letting Vertica do it. Remember to properly calculate the amount
         // of memory you need based on the data type you are allocating.
         // This example divides 500MB by 8, since that's the number of
         // bytes in a 64-bit unsigned integer.
         myarray = new uint64[1024 * 1024 * 500 / 8];
      }
      catch (std::bad_alloc &ba)
      {
         // Always check for exceptions caused by failed memory
         // allocations.
         vt_report_error(1, "Couldn't allocate memory :[%s]", ba.what());
      }

   }

   // Called after the UDF has processed all of its information. Use to free
   // any allocated resources.
   virtual void destroy(ServerInterface &srvInterface, const SizedColumnTypes
                        &argTypes)
   {
      // srvInterface.log("RowNumber processed %d records", *count_ptr);
      try
      {
         // Properly dispose of the allocated memory.
         delete[] myarray;
      }
      catch (std::bad_alloc &ba)
      {
         // Always check for exceptions caused by failed memory
         // allocations.
         vt_report_error(1, "Couldn't free memory :[%s]", ba.what());
      }

   }
```

Allocating resources with the SDK macros

The Vertica SDK provides three macros to allocate memory:

- vt_alloc allocates a block of memory to fit a specific data type (vint, struct, etc.).
- vt_allocArray allocates a block of memory to hold an array of a specific data type.
- vt_allocSize allocates an arbitrarily-sized block of memory.

All of these macros allocate their memory from memory pools managed by Vertica. The main benefit of allowing Vertica to manage your UDx's memory is that the memory is automatically reclaimed after your UDx has finished. This ensures there is no memory leaks in your UDx.

Because Vertica frees this memory automatically, do not attempt to free any of the memory you allocate through any of these macros. Attempting to free this memory results in run-time errors.

Informing Vertica of resource requirements

When you run your UDx in fenced mode, Vertica monitors its use of memory and file handles. If your UDx uses more than a few megabytes of memory or any file handles, it should tell Vertica about its resource requirements. Knowing the resource requirements of your UDx allows Vertica to determine whether it can run the UDx immediately or needs to queue the request until enough resources become available to run it.

Determining how much memory your UDx requires can be difficult in some cases. For example, if your UDx extracts unique data elements from a data set, there is potentially no bound on the number of data items. In this case, a useful technique is to run your UDx in a test environment and monitor its memory use on a node as it handles several differently-sized queries, then extrapolate its memory use based on the worst-case scenario it may face in your production environment. In all cases, it's usually a good idea to add a safety margin to the amount of memory you tell Vertica your UDx uses.

> **Note**
> The information on your UDx's resource needs that you pass to Vertica is used when planning the query execution. There is no way to change the amount of resources your UDx requests from Vertica while the UDx is actually running.

Your UDx informs Vertica of its resource needs by implementing the getPerInstanceResources() function in its factory class (see Vertica::UDXFactory::getPerInstanceResources() in the SDK documentation). If your UDx's factory class implements this function, Vertica calls it to determine the resources your UDx requires.

The getPerInstanceResources() function receives an instance of the Vertica::VResources struct. This struct contains fields that set the amount of memory and the number of file handles your UDx needs. By default, the Vertica server allocates zero bytes of memory and 100 file handles for each instance of your UDx.

Your implementation of the getPerInstanceResources() function sets the fields in the VResources struct based on the maximum resources your UDx may consume for each instance of the UDx function. So, if your UDx's processBlock() function creates a data structure that uses at most 100MB of memory, your UDx must set the VResources.scratchMemory field to at least 104857600 (the number of bytes in 100MB). Leave yourself a safety margin by increasing the number beyond what your UDx should normally consume. In this example, allocating 115000000 bytes (just under 110MB) is a good idea.

The following ScalarFunctionFactory class demonstrates calling getPerInstanceResources() to inform Vertica about the memory requirements of the MemoryAllocationExample class shown in Allocating resources for UDxs . It tells Vertica that the UDSF requires 510MB of memory (which is a bit more than the UDSF actually allocates, to be on the safe size).

```
class MemoryAllocationExampleFactory : public ScalarFunctionFactory
{
  virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface
                                                         &srvInterface)
  {
    return vt_createFuncObj(srvInterface.allocator, MemoryAllocationExample);
  }
  virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                            Vertica::ColumnTypes &argTypes,
                            Vertica::ColumnTypes &returnType)
  {
    argTypes.addInt();
    argTypes.addInt();
    returnType.addInt();
  }
  // Tells Vertica the amount of resources that this UDF uses.
  virtual void getPerInstanceResources(ServerInterface &srvInterface,
                                       VResources &res)
  {
    res.scratchMemory += 1024LL * 1024 * 510; // request 510MB of memory
  }
};
```

Setting memory limits for fenced-mode UDxs

Vertica calls a fenced-mode UDx's implementation of Vertica::UDXFactory::getPerInstanceResources() to determine if there are enough free resources to run the query containing the UDx (see Informing Vertica of resource requirements). Since these reports are not generated by actual memory use, they can be inaccurate. Once started by Vertica, a UDx could allocate far more memory or file handles than it reported it needs.

The FencedUDxMemoryLimitMB configuration parameter lets you create an absolute memory limit for UDxs. Any attempt by a UDx to allocate more memory than this limit results in a bad_alloc exception. For an example of setting FencedUDxMemoryLimitMB, see How resource limits are enforced .

How resource limits are enforced

Before running a query, Vertica determines how much memory it requires to run. If the query contains a fenced-mode UDx which implements the getPerInstanceResources() function in its factory class, Vertica calls it to determine the amount of memory the UDx needs and adds this to the total required for the query. Based on these requirements, Vertica decides how to handle the query:

- If the total amount of memory required (including the amount that the UDxs report that they need) is larger than the session's MEMORYCAP or resource pool's MAXMEMORYSIZE setting, Vertica rejects the query. For more information about resource pools, see Resource pool architecture .
- If the amount of memory is below the limit set by the session and resource pool limits, but there is currently not enough free memory to run the query, Vertica queues it until enough resources become available.
- If there are enough free resources to run the query, Vertica executes it.

> **Note**
> Vertica has no other way to determine the amount of resources a UDx requires other than the values it reports using the getPerInstanceResources() function. A UDx could use more resources than it claims, which could cause performance issues for other queries that are denied resources. You can set an absolute limit on the amount of memory UDxs can allocate. See Setting memory limits for fenced-mode UDxs for more information.

If the process executing your UDx attempts to allocate more memory than the limit set by the FencedUDxMemoryLimitMB configuration parameter, it receives a bad_alloc exception. For more information about FencedUDxMemoryLimitMB, see Setting memory limits for fenced-mode UDxs .

Below is the output of loading a UDSF that consumes 500MB of memory, then changing the memory settings to cause out-of-memory errors. The MemoryAllocationExample UDSF in the following example is just the Add2Ints UDSF example altered as shown in Allocating resources for UDxs and Informing Vertica of resource requirements to allocate 500MB of RAM.

```
=> CREATE LIBRARY mylib AS '/home/dbadmin/MemoryAllocationExample.so';
CREATE LIBRARY
=> CREATE FUNCTION usemem AS NAME 'MemoryAllocationExampleFactory' LIBRARY mylib
-> FENCED;
CREATE FUNCTION
=> SELECT usemem(1,2);
 usemem
--------
      3
(1 row)
```

The following statements demonstrate setting the session's MEMORYCAP to lower than the amount of memory that the UDSF reports it uses. This causes Vertica to return an error before it executes the UDSF.

```
=> SET SESSION MEMORYCAP '100M';
SET
=> SELECT usemem(1,2);
ERROR 3596:  Insufficient resources to execute plan on pool sysquery
[Request exceeds session memory cap: 520328KB > 102400KB]
=> SET SESSION MEMORYCAP = default;
SET
```

The resource pool can also prevent a UDx from running if it requires more memory than is available in the pool. The following statements demonstrate the effect of creating and using a resource pool that has too little memory for the UDSF to run. Similar to the session's MAXMEMORYCAP limit, the pool's MAXMEMORYSIZE setting prevents Vertica from executing the query containing the UDSF.

```
=> CREATE RESOURCE POOL small MEMORYSIZE '100M' MAXMEMORYSIZE '100M';
CREATE RESOURCE POOL
=> SET SESSION RESOURCE POOL small;
SET
=> CREATE TABLE ExampleTable(a int, b int);
CREATE TABLE
=> INSERT /*+direct*/ INTO ExampleTable VALUES (1,2);
 OUTPUT
--------
      1
(1 row)
=> SELECT usemem(a, b) FROM ExampleTable;
ERROR 3596:  Insufficient resources to execute plan on pool small
[Request Too Large:Memory(KB) Exceeded: Requested = 523136, Free = 102400 (Limit = 102400, Used = 0)]
=> DROP RESOURCE POOL small; --Dropping the pool resets the session's pool
DROP RESOURCE POOL
```

Finally, setting the FencedUDxMemoryLimitMB configuration parameter to lower than the UDx actually allocates results in the UDx throwing an exception. This is a different case than either of the previous two examples, since the query actually executes. The UDx's code needs to catch and handle the exception. In this example, it uses the vt_report_error macro to report the error back to Vertica and exit.

```
=> ALTER DATABASE DEFAULT SET FencedUDxMemoryLimitMB = 300;

=> SELECT usemem(1,2);
    ERROR 3412:  Failure in UDx RPC call InvokeSetup(): Error calling setup() in
    User Defined Object [usemem] at [MemoryAllocationExample.cpp:32], error code:
     1, message: Couldn't allocate memory :[std::bad_alloc]

=> ALTER DATABASE DEFAULT SET FencedUDxMemoryLimitMB = -1;

=> SELECT usemem(1,2);
 usemem
--------
      3
(1 row)
```

See also

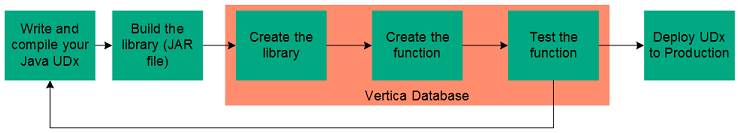
- SET SESSION RESOURCE_POOL
- SET SESSION MEMORYCAP

Java SDK

The Vertica SDK supports writing Java UDxs of all types except aggregate functions. All Java UDxs are fenced.

You can download, compile, and run the examples; see Downloading and running UDx example code . Running the examples is a good way to verify that your development environment has all needed libraries.

If you do not have access to a Vertica test environment, you can install Vertica on your development machine and run a single node. Each time you rebuild your UDx library, you need to re-install it into Vertica. The following diagram illustrates the typical development cycle.



This section covers Java-specific topics that apply to all UDx types. For information that applies to all languages, see Arguments and return values , UDx parameters , Errors, warnings, and logging , Handling cancel requests and the sections for specific UDx types. For full API documentation, see the Java SDK Documentation .

In this section

- Setting up the Java SDK
- Compiling and packaging a Java library
- Handling Java UDx dependencies

- Java and Vertica data types
- Handling NULL values
- Adding metadata to Java UDx libraries
- Java UDx resource management

Setting up the Java SDK

The Vertica Java Software Development Kit (SDK) is distributed as part of the server installation. It contains the source and JAR files you need to create your UDx library. For examples that you can compile and run, see Downloading and running UDx example code .

Requirements

At a minimum, install the following on your development machine:

- The Java Development Kit (JDK) version that matches the Java version you have installed on your database hosts (see Installing Java on Vertica Hosts).
- A copy of the Vertica SDK.

Optionally, you can simplify development with a build-management tool, such as make .

SDK files

To use the SDK you need two files from the Java support package:

- /opt/vertica/bin/VerticaSDK.jar contains the Vertica Java SDK and other supporting files.
- /opt/vertica/sdk/BuildInfo.java contains version information about the SDK. You must compile this file and include it within your Java UDx JAR files.

If you are not doing your development on a database node, you can copy these two files from one of the database nodes to your development system.

The BuildInfo.java and VerticaSDK.jar files that you use to compile your UDx must be from the same SDK version. Both files must also match the version of the SDK files on your Vertica hosts. Versioning is only an issue if you are not compiling your UDxs on a Vertica host. If you are compiling on a separate development system, always refresh your copies of these two files and recompile your UDxs just before deploying them.

Finding the current SDK version

You must develop your UDx using the same SDK version as the database in which you plan to use it. To display the SDK version currently installed on your system, run the following command in vsql:

```
=> SELECT sdk_version();
```

Compiling BuildInfo.java

You need to compile the BuildInfo.java file into a class file, so you can include it in your Java UDx JAR library. If you are using a Vertica node as a development system, you can either:

- Copy the BuildInfo.java file to another location on your host.
- If you have root privileges, compile the BuildInfo.java file in place. (Only the root user has privileges to write files to the /opt/vertica/sdk directory.)

Compile the file using the following command. Replace *path* with the path to the file and *output-directory* with the directory where you will compile your UDxs.

```
$ javac -classpath /opt/vertica/bin/VerticaSDK.jar \
    /path/BuildInfo.java -d output-directory
```

If you use an IDE such as Eclipse, you can include the BuildInfo.java file in your project instead of compiling it separately. You must also add the VerticaSDK.jar file to the project's build path. See your IDE's documentation for details on how to include files and libraries in your projects.

Running the examples

You can download the examples from the GitHub repository (see Downloading and running UDx example code). Compiling and running the examples helps you to ensure that your development environment is properly set up.

If you have not already done so, set the JAVA_HOME environment variable to your JDK (not JRE) directory.

To compile all of the examples, including the Java examples, issue the following command in the Java-and-C++ directory under the examples directory:

```
$ make
```

To compile only the Java examples, issue the following command in the Java-and-C++ directory under the examples directory:

```
$ make JavaFunctions
```

Compiling and packaging a Java library

Before you can use your Java UDx, you need to compile it and package it into a JAR file.

The SDK examples include a working makefile. See Downloading and running UDx example code .

Compile your Java UDx

You must include the SDK JAR file in the CLASSPATH when you compile your Java UDx source files so the Java compiler can resolve the Vertica API calls. If you are using the command-line Java compiler on a host in your database cluster, enter this command:

```
$ javac -classpath /opt/vertica/bin/VerticaSDK.jar factorySource.java \
    [functionSource.java...] -d output-directory
```

If all of your source files are in the same directory, you can use *.java on the command line instead of listing the files individually.

If you are using an IDE, verify that a copy of the VerticaSDK.jar file is in the build path.

UDx class file organization

After you compile your UDx, you must package its class files and the BuildInfo.class file into a JAR file.

> **Note**
> You can package as many UDxs as you want into the same JAR file. Bundling your UDxs together saves you from having to load multiple libraries.

To use the jar command packaged as part of the JDK, you must organize your UDx class files into a directory structure matching your class package structure. For example, suppose your UDx's factory class has a fully-qualified name of com.mycompany.udfs.Add2ints . In this case, your class files must be in the directory hierarchy com/mycompany/udfs relative to your project's base directory. In addition, you must have a copy of the BuildInfo.class file in the path com/vertica/sdk so that it can be included in the JAR file. This class must appear in your JAR file to indicate the SDK version that was used to compile your Java UDx.

The JAR file for the Add2ints UDSF example has the following directory structure after compilation:

```
com/vertica/sdk/BuildInfo.class
com/mycompany/example/Add2intsFactory.class
com/mycompany/example/Add2intsFactory$Add2ints.class
```

Package your UDx into a JAR file

To create a JAR file from the command line:

1. Change to the root directory of your project.
2. Use the jar command to package the BuildInfo.class file and all of the classes in your UDx:

   ```
   # jar -cvf libname.jar com/vertica/sdk/BuildInfo.class \
       packagePath/*.class
   ```

   When you type this command, *libname* is the filename you have chosen for your JAR file (choose whatever name you like), and *packagePath* is the path to the directory containing your UDx's class files.
   - For example, to package the files from the Add2ints example, you use the command:

     ```
     # jar -cvf Add2intsLib.jar com/vertica/sdk/BuildInfo.class \
     com/mycompany/example/*.class
     ```

   - More simply, if you compiled BuildInfo.class and your class files into the same root directory, you can use the following command:

     ```
     # jar -cvf Add2intsLib.jar .
     ```

   You must include all of the class files that make up your UDx in your JAR file. Your UDx always consists of at least two classes (the factory class and the function class). Even if you defined your function class as an inner class of your factory class, Java generates a separate class file for the inner class.

After you package your UDx into a JAR file, you are ready to deploy it to your Vertica database.

Handling Java UDx dependencies

If your Java UDx relies on one or more external libraries, you can handle the dependencies in one of three ways:

- Bundle the JAR files into your UDx JAR file using a tool such as One-JAR or Eclipse Runnable JAR Export Wizard.
- Unpack the JAR file and then repack its contents in your UDx's JAR file.
- Copy the libraries to your Vertica cluster in addition to your UDx library.Then, use the DEPENDS keyword of the CREATE LIBRARY statement to tell Vertica that the UDx library depends on the external libraries. This keyword acts as a library-specific CLASSPATH setting. Vertica distributes the support libraries to all of the nodes in the cluster and sets the class path for the UDx so it can find them.
  If your UDx depends on native libraries (SO files), use the DEPENDS keyword to specify their path. When you call System.loadLibrary in your UDx (which you must do before using a native library), this function uses the DEPENDS path to find them. You do not need to also set the LD_LIBRARY_PATH environment variable.

External library example

The following example demonstrates using an external library with a Java UDx.

The following sample code defines a simple class, named VowelRemover . It contains a single method, named removevowels , that removes all of the vowels (the letters *a* , *e* , *i* , *o u* , and *y*) from a string.

```
package com.mycompany.libs;

public class VowelRemover {
   public String removevowels(String input) {
      return input.replaceAll("(?i)[aeiouy]", "");
   }
}
```

You can compile this class and package it into a JAR file with the following commands:

```
$ javac -g com/mycompany/libs/VowelRemover.java
$ jar cf mycompanylibs.jar com/mycompany/libs/VowelRemover.class
```

The following code defines a Java UDSF, named DeleteVowels , that uses the library defined in the preceding example code. DeleteVowels accepts a single VARCHAR as input, and returns a VARCHAR.

```
package com.mycompany.udx;
// Import the support class created earlier
import com.mycompany.libs.VowelRemover;
// Import the Vertica SDK
import com.vertica.sdk.*;

public class DeleteVowelsFactory extends ScalarFunctionFactory {

    @Override
    public ScalarFunction createScalarFunction(ServerInterface arg0) {
        return new DeleteVowels();
    }

    @Override
    public void getPrototype(ServerInterface arg0, ColumnTypes argTypes,
            ColumnTypes returnTypes) {
        // Accept a single string and return a single string.
        argTypes.addVarchar();
        returnTypes.addVarchar();
    }

    @Override
    public void getReturnType(ServerInterface srvInterface,
            SizedColumnTypes argTypes,
            SizedColumnTypes returnType){
        returnType.addVarchar(
        // Output will be no larger than the input.
        argTypes.getColumnType(0).getStringLength(), "RemovedVowels");
    }

    public class DeleteVowels extends ScalarFunction
    {
        @Override
        public void processBlock(ServerInterface arg0, BlockReader argReader,
                BlockWriter resWriter) throws UdfException, DestroyInvocation {

            // Create an instance of the  VowelRemover object defined in
            // the library.
            VowelRemover remover = new VowelRemover();

            do {
                String instr = argReader.getString(0);
                // Call the removevowels method defined in the library.
                resWriter.setString(remover.removevowels(instr));
                resWriter.next();
            } while (argReader.next());
        }
    }
```

Use the following commands to build the example UDSF and package it into a JAR:

- The first javac command compiles the SDK's BuildInfo class. Vertica requires all UDx libraries to contain this class. The javac command's -d option outputs the class file in the directory structure of your UDSF's source.
- The second javac command compiles the UDSF class. It adds the previously-created mycompanylibs.jar file to the class path so compiler can find the the VowelRemover class.
- The jar command packages the BuildInfo and the classes for the UDx library together.

```
$ javac -g -cp /opt/vertica/bin/VerticaSDK.jar\
  /opt/vertica/sdk/com/vertica/sdk/BuildInfo.java -d .
$ javac -g -cp mycompanylibs.jar:/opt/vertica/bin/VerticaSDK.jar\
 com/mycompany/udx/DeleteVowelsFactory.java
$ jar cf DeleteVowelsLib.jar com/mycompany/udx/*.class \
  com/vertica/sdk/*.class
```

To install the UDx library, you must copy both of the JAR files to a node in the Vertica cluster. Then, connect to the node to execute the CREATE LIBRARY statement.

The following example demonstrates how to load the UDx library after you copy the JAR files to the home directory of the dbadmin user. The DEPENDS keyword tells Vertica that the UDx library depends on the mycompanylibs.jar file.

```
=> CREATE LIBRARY DeleteVowelsLib AS
  '/home/dbadmin/DeleteVowelsLib.jar' DEPENDS '/home/dbadmin/mycompanylibs.jar'
  LANGUAGE 'JAVA';
CREATE LIBRARY
=> CREATE FUNCTION deleteVowels AS language 'java' NAME
 'com.mycompany.udx.DeleteVowelsFactory' LIBRARY DeleteVowelsLib;
CREATE FUNCTION
=> SELECT deleteVowels('I hate vowels!');
 deleteVowels
--------------
 ht vwls!
(1 row)
```

Java and Vertica data types

The Vertica Java SDK converts Vertica's native data types into the appropriate Java data type. The following table lists the Vertica data types and their corresponding Java data types.

Vertica Data Type	Java Data Type
INTEGER	long
FLOAT	double
NUMERIC	com.vertica.sdk.VNumeric
DATE	java.sql.Date
CHAR, VARCHAR, LONG VARCHAR	com.vertica.sdk.VString
BINARY, VARBINARY, LONG VARBINARY	com.vertica.sdk.VString
TIMESTAMP	java.sql.Timestamp

> **Note**
> Some Vertica data types are not supported.

Setting BINARY, VARBINARY, and LONG VARBINARY values

The Vertica BINARY, VARBINARY, and LONG VARBINARY data types are converted as the Java UDx SDK 's VString class. You can also set the value of a column with one of these data types with a ByteBuffer object (or a byte array wrapped in a ByteBuffer) using the PartitionWriter.setStringBytes() method. See the Java API UDx entry for PartitionWriter.setStringBytes() for more information.

Timestamps and time zones

When the SDK converts a Vertica timestamp into a Java timestamp, it uses the time zone of the JVM. If the JVM is running in a different time zone than the one used by Vertica, the results can be confusing.

Vertica stores timestamps in the database in UTC. (If a database time zone is set, the conversion is done at query time.) To prevent errors from the JVM time zone, add the following code to the processing method of your UDx:

```
TimeZone.setDefault(TimeZone.getTimeZone("UTC"));
```

Strings

The Java SDK contains a class named StringUtils that assists you when manipulating string data. One of its more useful features is its getStringBytes() method. This method extracts bytes from a String in a way that prevents the creation of invalid strings. If you attempt to extract a substring that would split part of a multi-byte UTF-8 character, getStringBytes() truncates it to the nearest whole character.

Handling NULL values

Your UDxs must be prepared to handle NULL values. These values usually must be handled separately from regular values.

Reading NULL values

Your UDx reads data from instances of the the BlockReader or PartitionReader classes. If the value of a column is NULL, the methods you use to get data (such as getLong) return a Java null reference. If you attempt to use the value without checking for NULL, the Java runtime will throw a null pointer exception.

You can test for null values before reading columns by using the data-type-specific methods (such as isLongNull , isDoubleNull , and isBooleanNull). For example, to test whether the INTEGER first column of your UDx's input is a NULL, you would use the statement:

```
// See if the Long value in column 0 is a NULL
if (inputReader.isLongNull(0)) {
  // value is null
  . . .
```

Writing NULL values

You output NULL values using type-specific methods on the BlockWriter and PartitionWriter classes (such as setLongNull and setStringNull). These methods take the column number to receive the NULL value. In addition, the PartitionWriter class has data-type specific set value methods (such as setLongValue and setStringValue). If you pass these methods a value, they set the output column to that value. If you pass them a Java null reference, they set the output column to NULL.

Adding metadata to Java UDx libraries

You can add metadata, such as author name, the version of the library, a description of your library, and so on to your library. This metadata lets you track the version of your function that is deployed on a Vertica Analytic Database cluster and lets third-party users of your function know who created the function. Your library's metadata appears in the USER_LIBRARIES system table after your library has been loaded into the Vertica Analytic Database catalog.

To add metadata to your Java UDx library, you create a subclass of the UDXLibrary class that contains your library's metadata. You then include this class within your JAR file. When you load your class into the Vertica Analytic Database catalog using the CREATE LIBRARY statement, looks for a subclass of UDXLibrary for the library's metadata.

In your subclass of UDXLibrary , you need to implement eight getters that return String values containing the library's metadata. The getters in this class are:

- getAuthor() returns the name you want associated with the creation of the library (your own name or your company's name for example).
- getLibraryBuildTag() returns whatever String you want to use to represent the specific build of the library (for example, the SVN revision number or a timestamp of when the library was compiled). This is useful for tracking instances of your library as you are developing them.
- getLibraryVersion() returns the version of your library. You can use whatever numbering or naming scheme you want.
- getLibrarySDKVersion() returns the version of the Vertica Analytic Database SDK Library for which you've compiled the library.

> **Note**
> This field isn't used to determine whether a library is compatible with a version of the Vertica Analytic Database server. The version of the Vertica Analytic Database SDK you use to compile your library is embedded in the library when you compile it. It is this information that Vertica Analytic Database server uses to determine if your library is compatible with it.

- getSourceUrl() returns a URL where users of your function can find more information about it. This can be your company's website, the GitHub page hosting your library's source code, or whatever site you like.

- getDescription() returns a concise description of your library.
- getLicensesRequired() returns a placeholder for licensing information. You must pass an empty string for this value.
- getSignature() returns a placeholder for a signature that will authenticate your library. You must pass an empty string for this value.

For example, the following code demonstrates creating a UDXLibrary subclass to be included in the Add2Ints UDSF example JAR file (see /opt/vertica/sdk/examples/JavaUDx/ScalarFunctions on any Vertica node).

```
// Import the UDXLibrary class to hold the metadata
import com.vertica.sdk.UDXLibrary;

public class Add2IntsLibrary extends UDXLibrary
{
    // Return values for the metadata about this library.

    @Override public String getAuthor() {return "Whizzo Analytics Ltd.";}
    @Override public String getLibraryBuildTag() {return "1234";}
    @Override public String getLibraryVersion() {return "1.0";}
    @Override public String getLibrarySDKVersion() {return "7.0.0";}
    @Override public String getSourceUrl() {
        return "http://example.com/add2ints";
    }
    @Override public String getDescription() {
        return "My Awesome Add 2 Ints Library";
    }
    @Override public String getLicensesRequired() {return "";}
    @Override public String getSignature() {return "";}
}
```

When the library containing the Add2IntsLibrary class loaded, the metadata appears in the USER_LIBRARIES system table:

```
=> CREATE LIBRARY JavaAdd2IntsLib AS :libfile LANGUAGE 'JAVA';
CREATE LIBRARY
=> CREATE FUNCTION JavaAdd2Ints as LANGUAGE 'JAVA'  name 'com.mycompany.example.Add2IntsFactory' library JavaAdd2IntsLib;
CREATE FUNCTION
=> \x
Expanded display is on.
=> SELECT * FROM USER_LIBRARIES WHERE lib_name = 'JavaAdd2IntsLib';
-[ RECORD 1 ]-----+-------------------------------------------
schema_name       | public
lib_name          | JavaAdd2IntsLib
lib_oid           | 45035996273869844
author            | Whizzo Analytics Ltd.
owner_id          | 45035996273704962
lib_file_name     | public_JavaAdd2IntsLib_45035996273869844.jar
md5_sum           | f3bfc76791daee95e4e2c0f8a8d2737f
sdk_version       | v7.0.0-20131105
revision          | 125200
lib_build_tag     | 1234
lib_version       | 1.0
lib_sdk_version   | 7.0.0
source_url        | http://example.com/add2ints
description       | My Awesome Add 2 Ints Library
licenses_required |
signature         |
```

Java UDx resource management

Java Virtual Machines (JVMs) allocate a set amount of memory when they start. This set memory allocation complicates memory management for Java UDxs, because memory cannot be dynamically allocated and freed by the UDx as it is processing data. This is differs from C++ UDxs which can dynamically allocate resources.

To control the amount of memory consumed by Java UDxs, Vertica has a memory pool named jvm that it uses to allocate memory for JVMs. If this memory pool is exhausted, queries that call Java UDxs block until enough memory in the pool becomes free to start a new JVM.

By default, the jvm pool has:

- no memory of its own assigned to it, so it borrows memory from the GENERAL pool.
- its MAXMEMORYSIZE set to either 10% of system memory or 2GB, whichever is smaller.
- its PLANNEDCONCURRENCY set to AUTO, so that it inherits the GENERAL pool's PLANNEDCONCURRENCY setting.

You can view the current settings for the jvm pool by querying the RESOURCE_POOLS table:

```
=> SELECT MAXMEMORYSIZE,PLANNEDCONCURRENCY FROM V_CATALOG.RESOURCE_POOLS WHERE NAME = 'jvm';
 MAXMEMORYSIZE | PLANNEDCONCURRENCY
---------------+--------------------
10%            | AUTO
```

When a SQL statement calls a Java UDx, Vertica checks if the jvm memory pool has enough memory to start a new JVM instance to execute the function call. Vertica starts each new JVM with its heap memory size set to approximately the jvm pool's MAXMEMORYSIZE parameter divided by its PLANNEDCONCURRENCY parameter. If the memory pool does not contain enough memory, the query blocks until another JVM exits and return their memory to the pool.

If your Java UDx attempts to consume more memory than has been allocated to the JVM's heap size, it exits with a memory error. You can attempt to resolve this issue by:

- increasing the jvm pool's MAXMEMORYSIZE parameter.
- decreasing the jvm pool's PLANNEDCONCURRENCY parameter.
- changing your Java UDx's code to consume less memory.

Adjusting the jvm pool

When adjusting the jvm pool to your needs, you must consider two factors:

- the amount of RAM your Java UDx requires to run
- how many concurrent Java UDx functions you expect your database to run

You can learn the amount of memory your Java UDx needs using several methods. For example, your code can use Java's Runtime class to get an estimate of the total memory it has allocated and then log the value using ServerInterface.log() . (An instance of this class is passed to your UDx.) If you have multiple Java UDxs in your database, set the jvm pool memory size based on the UDx that uses the most memory.

The number of concurrent sessions that need to run Java UDxs may not be the same as the global PLANNEDCONCURRENCY setting. For example, you may have just a single user who runs a Java UDx, which means you can lower the jvm pool's PLANNEDCONCURRENCY setting to 1.

When you have an estimate for the amount of RAM and the number of concurrent user sessions that need to run Java UDXs, you can adjust the jvm pool to an appropriate size. Set the pool's MAXMEMORYSIZE to the maximum amount of RAM needed by the most demanding Java UDx multiplied by the number of concurrent user sessions that need to run Java UDxs. Set the pool's PLANNEDCONCURENCY to the numebr of simultaneous user sessions that need to run Java UDxs.

For example, suppose your Java UDx requires up to 4GB of memory to run and you expect up to two user sessions use Java UDx's. You would use the following command to adjust the jvm pool:

```
=> ALTER RESOURCE POOL jvm MAXMEMORYSIZE '8G' PLANNEDCONCURRENCY 2;
```

The MEMORYSIZE is set to 8GB, which is the 4GB maximum memory use by the Java UDx multiplied by the 2 concurrent user sessions.

> **Note**
> The PLANNEDCONCURRENCY value is **not** the number of calls to Java UDx that you expect to happen simultaneously. Instead, it is the number of concurrently open user sessions that call Java UDxs at any time during the session. See below for more information.

See Managing workloads for more information on tuning the jvm and other resource pools.

Freeing JVM memory

The first time users call a Java UDx during their session, Vertica allocates memory from the jvm pool and starts a new JVM. This JVM remains running for as long as the user's session is open so it can process other Java UDx calls. Keeping the JVM running lowers the overhead of executing multiple Java UDxs by the same session. If the JVM did not remain open, each call to a Java UDx would require additional time for Vertica to allocate resources and

start a new JVM. However, having the JVM remain open means that the JVM's memory remains allocated for the life of the session whether or not it will be used again.

If the jvm memory pool is depleted, queries containing Java UDxs either block until memory becomes available or eventually fail due a lack of resources. If you find queries blocking or failing for this reason, you can allocate more memory to the jvm pool and increase its PLANNEDCONCURRENCY. Another option is to ask users to call the RELEASE_JVM_MEMORY function when they no longer need to run Java UDxs. This function closes any JVM belonging to the user's session and returns its allocated memory to the jvm memory pool.

The following example demonstrates querying V_MONITOR.SESSIONS to find the memory allocated to JVMs by all sessions. It also demonstrates how the memory is allocated by a call to a Java UDx, and then freed by calling RELEASE_JVM_MEMORY.

```
=> SELECT USER_NAME,EXTERNAL_MEMORY_KB FROM V_MONITOR.SESSIONS;
 user_name | external_memory_kb
-----------+--------------------
 dbadmin   |                  0
(1 row)

=> -- Call a Java UDx
=> SELECT add2ints(123,456);
 add2ints
----------
      579
(1 row)
=> -- JVM is now running and memory is allocated to it.
=> SELECT USER_NAME,EXTERNAL_MEMORY_KB FROM V_MONITOR.SESSIONS;
 USER_NAME | EXTERNAL_MEMORY_KB
-----------+--------------------
 dbadmin   |              79705
(1 row)

=> -- Shut down the JVM and deallocate memory
=> SELECT RELEASE_JVM_MEMORY();
           RELEASE_JVM_MEMORY
-----------------------------------------
 Java process killed and memory released
(1 row)

=> SELECT USER_NAME,EXTERNAL_MEMORY_KB FROM V_MONITOR.SESSIONS;
 USER_NAME | EXTERNAL_MEMORY_KB
-----------+--------------------
 dbadmin   |                  0
(1 row)
```

In rare cases, you may need to close all JVMs. For example, you may need to free memory for an important query, or several instances of a Java UDx may be taking too long to complete. You can use the RELEASE_ALL_JVM_MEMORY to close all of the JVMs in all user sessions:

```
=> SELECT USER_NAME,EXTERNAL_MEMORY_KB FROM V_MONITOR.SESSIONS;
 USER_NAME  | EXTERNAL_MEMORY_KB
------------+--------------------
ExampleUser |              79705
dbadmin     |              79705
(2 rows)

=> SELECT RELEASE_ALL_JVM_MEMORY();
                        RELEASE_ALL_JVM_MEMORY
-------------------------------------------------------------------------------
 Close all JVM sessions command sent. Check v_monitor.sessions for progress.
(1 row)

=> SELECT USER_NAME,EXTERNAL_MEMORY_KB FROM V_MONITOR.SESSIONS;
 USER_NAME | EXTERNAL_MEMORY_KB
-----------+--------------------
 dbadmin   |                  0
(1 row)
```

Caution

This function terminates all JVMs, including ones that are currently executing Java UDXs. This will cause any query that is currently executing a Java UDx to return an error.

Notes

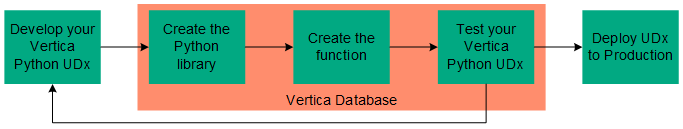
- The jvm resource pool is used only to allocate memory for the Java UDx function calls in a statement. The rest of the resources required by the SQL statement come from other memory pools.
- The first time a Java UDx is called, Vertica starts a JVM to execute some Java methods to get metadata about the UDx during the query planning phase. The memory for this JVM is also taken from the jvm memory pool.

Python SDK

The Vertica SDK supports writing UDxs of some types in Python 3.

The Python SDK does not require any additional system configuration or header files. This low overhead allows you to develop and deploy new capabilities to your Vertica cluster in a short amount of time.

The following workflow is typical for the Python SDK:



Because Python has an interpreter, you do not have to compile your program before loading the UDx in Vertica. However, you should expect to do some debugging of your code after you create your function and begin testing it in Vertica.

When Vertica calls your UDx, it starts a side process that manages the interaction between the server and the Python interpreter.

This section covers Python-specific topics that apply to all UDx types. For information that applies to all languages, see Arguments and return values , UDx parameters , Errors, warnings, and logging , Handling cancel requests and the sections for specific UDx types. For full API documentation, see the Python SDK .

Important

Your UDx must be able to run with the version of Python bundled with Vertica. You can find this with `/opt/vertica/sbin/python3 --version` . You cannot change the version used by the Vertica Python interpreter.

In this section

- Setting up a Python development environment
- Python and Vertica data types

Setting up a Python development environment

To avoid problems when loading and executing your UDxs, develop your UDxs using the same version of Python that Vertica uses. To do this without changing your environment for projects that might require other Python versions, you can use a Python virtual environment (venv) . You can install libraries that your UDx depends on into your venv and use that path when you create your UDx library with CREATE LIBRARY .

Setting up venv

Set up venv using the Python version bundled with Vertica. If you have direct access to a database node, you can use that Python binary directly to create your venv :

```
$ /opt/vertica/sbin/python3 -m venv /path/to/new/environment
```

The result is a directory with a default environment, including a site-packages directory:

```
$ ls venv/lib/
python3.9
$ ls venv/lib/python3.9/
site-packages
```

If your UDx depends on libraries that are not packaged with Vertica, install them into this directory:

```
$ source venv/bin/activate
(venv) $ pip install numpy
...
```

The lib/python3.9/site-packages directory now contains the installed library. The change affects only your virtual environment.

UDx imports

Your UDx code must import, in addition to any libraries you add, the vertica_sdk library:

```
# always required:
import vertica_sdk
# other libs:
import numpy as np
# ...
```

The vertica_sdk library is included as a part of the Vertica server. You do not need to add it to site-packages or declare it as a dependency.

Deployment

For libraries you add, you must declare dependencies when using CREATE LIBRARY . This declaration allows Vertica to find the libraries and distribute them to all database nodes. You can supply a path instead of enumerating the libraries:

```
=> CREATE OR REPLACE LIBRARY pylib AS
  '/path/to/udx/add2ints.py'
  DEPENDS '/path/to/new/environment/lib/python3.9/site-packages/*'
  LANGUAGE 'Python';

=> CREATE OR REPLACE FUNCTION add2ints AS LANGUAGE 'Python'
  NAME 'add2ints_factory' LIBRARY pylib;
```

CREATE LIBRARY copies the UDx and the contents of the DEPENDS path and stores them with the database. Vertica then distributes copies to all database nodes.

Python and Vertica data types

The Vertica Python SDK converts native Vertica data types into the appropriate Python data types. The following table describes some of the data type conversions. Consult the Python SDK for a complete list, as well as lists of helper functions to convert and manipulate these data types.

For information about SDK support for complex data types, see Complex Types as Arguments and Return Values .

Vertica Data Type	**Python Data Type**
INTEGER	int

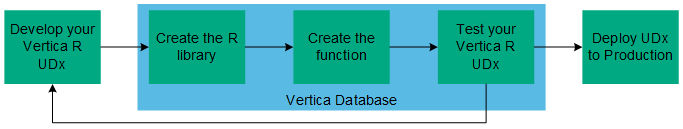
FLOAT	float
NUMERIC	decimal.Decimal
DATE	datetime.date
CHAR, VARCHAR, LONG VARCHAR	string (UTF-8 encoded)
BINARY, VARBINARY, LONG VARBINARY	binary
TIMESTAMP	datetime.datetime
TIME	datetime.time
ARRAY	list **Note:** Nested ARRAY types are also converted into lists.
ROW	collections.OrderedDict **Note:** Nested ROW types are also converted into collections.OrderedDicts.

Note
Some Vertica data types are not supported in Python. For a list of all Vertica data types, see Data types .

R SDK

The Vertica R SDK extends the capabilities of the Vertica Analytic Database so you can leverage additional R libraries. Before you can begin developing User Defined Extensions (UDxs) in R, you must install the R Language Pack for Vertica on each of the nodes in your cluster. The R SDK supports scalar and transform functions in fenced mode. Other UDx types are not supported.

The following workflow is typical for the R SDK:



You can find detailed documentation of all of the classes in the Vertica R SDK .

In this section

- Installing/upgrading the R language pack for Vertica
- R packages
- R and Vertica data types
- Adding metadata to R libraries
- Setting null input and volatility behavior for R functions

Installing/upgrading the R language pack for Vertica

To create R UDxs in Vertica, install the R Language Pack package that matches your server version. The R Language Pack includes the R runtime and associated libraries for interfacing with Vertica. You must use this version of the R runtime; you cannot upgrade it.

You must install the R Language Pack on each node in the cluster. The Vertica R Language Pack must be the only R Language Pack installed on the node.

Vertica R language pack prerequisites

The R Language Pack package requires a number of packages for installation and execution. The names of these dependencies vary among Linux distributions. For Vertica-supported Linux platforms the packages are:

- RHEL/CentOS: libfortran , xz-libs , libgomp

- SUSE Linux Enterprise Server: libfortran3 , liblzma5 , libgomp1
- Debian/Ubuntu: libfortran3 , liblzma5 , libgomp1
- Amazon Linux 2.0: compat-gcc-48-libgfortran , xz-libs , libgomp

Vertica requires a version of the libgfortran4 library later than 7.1 to create R extensions. The libgfortran library is included by default with the devtool and gcc packages.

Installing the Vertica R language pack

If you use your operating systems package manager, rather than the rpm or dpkg command, for installation, you do not need to manually install the R Language Pack. The native package managers for each supported Linux version are:

- RHEL/CentOS: yum
- SUSE Linux Enterprise Server: zypper
- Debian/Ubuntu: apt-get
- Amazon Linux 2.0: yum

1. Download the R language package by browsing to the Vertica website .
2. On the **Support** tab, select **Customer Downloads** .
3. When prompted, log in using your Micro Focus credentials.
4. Located and select the vertica-R-lang_ *version* .rpm or vertica-R-lang_ *version* .deb file for your server version. The R language package version must match your server version to three decimal points.
5. Install the package as root or using sudo:
   - RHEL/CentOS
     ```
     $ yum install vertica-R-lang-<version>.rpm
     ```
   - SUSE Linux Enterprise Server
     ```
     $ zypper install vertica-R-lang-<version>.rpm
     ```
   - Debian
     ```
     $ apt-get install ./vertica-R-lang_<version>.deb
     ```
   - Amazon Linux 2.0
     ```
     $ yum install vertica-R-lang-<version>.AMZN.rpm
     ```

The installer puts the R binary in /opt/vertica/R .

Upgrading the Vertica R language pack

When upgrading, some R packages you have manually installed may not work and may have to be reinstalled. If you do not update your package(s), then R returns an error if the package cannot be used. Instructions for upgrading these packages are below.

> **Note**
> The R packages provided in the R Language Pack are automatically upgraded and do not need to be reinstalled.

1. You must uninstall the R Language package before upgrading Vertica. Any additional R packages you manually installed remain in /opt/vertica/R and are not removed when you uninstall the package.
2. Upgrade your server package as detailed in Upgrading Vertica to a New Version .
3. After the server package has been updated, install the new R Language package on each host.

If you have installed additional R packages, on each node:

1. As root run /opt/vertica/R/bin/R and issue the command:
   ```
   > update.packages(checkBuilt=TRUE)
   ```
2. Select a CRAN mirror from the list displayed.
3. You are prompted to update each package that has an update available for it. You must update any packages that you manually installed and are not compatible with the current version of R in the R Language Pack.
   Do **NOT** update:
   - Rcpp
   - Rinside

   The packages you selected to be updated are installed. Quit R with the command:
   ```
   > quit()
   ```

Vertica UDx functions written in R do not need to be compiled and you do not need to reload your Vertica-R libraries and functions after an upgrade.

R packages

The Vertica R Language Pack includes the following R packages in addition to the default packages bundled with R:

- Rcpp
- RInside
- lpSolve
- lpSolveAPI

You can install additional R packages not included in the Vertica R Language Pack by using one of two methods. You must install the same packages on all nodes.

Installing R packages

You can install additional R packages by using one of the two following methods.

Using the install.packages() R command:

```
$ sudo /opt/vertica/R/bin/R
> install.packages("Zelig");
```

Using CMD INSTALL:

```
/opt/vertica/R/bin/R CMD INSTALL <path-to-package-tgz>
```

The installed packages are located in: /opt/vertica/R/library .

R and Vertica data types

The following data types are supported when passing data to/from an R UDx:

Vertica Data Type	**R Data Type**
BOOLEAN	logical
DATE, DATETIME, SMALLDATETIME, TIME, TIMESTAMP, TIMESTAMPTZ, TIMETZ	numeric
DOUBLE PRECISION, FLOAT, REAL	numeric
BIGINT, DECIMAL, INT, NUMERIC, NUMBER, MONEY	numeric
BINARY, VARBINARY	character
CHAR, VARCHAR	character

NULL values in Vertica are translated to R NA values when sent to the R function. R NA values are translated into Vertica null values when returned from the R function to Vertica.

Important

When specifying LONG VARCHAR or LONG VARBINARY data types, include the space between the two words. For example, datatype = c("long varchar")

.

Adding metadata to R libraries

You can add metadata, such as author name, the version of the library, a description of your library, and so on to your library. This metadata lets you track the version of your function that is deployed on a Vertica Analytic Database cluster and lets third-party users of your function know who created the function. Your library's metadata appears in the USER_LIBRARIES system table after your library has been loaded into the Vertica Analytic Database catalog.

You declare the metadata for your library by calling the RegisterLibrary() function in one of the source files for your UDx. If there is more than one function call in the source files for your UDx, whichever gets interpreted last as Vertica Analytic Database loads the library is used to determine the library's metadata.

The RegisterLibrary() function takes eight string parameters:

```
RegisterLibrary(author,
                library_build_tag,
                library_version,
                library_sdk_version,
                source_url,
                description,
                licenses_required,
                signature);
```

- author contains whatever name you want associated with the creation of the library (your own name or your company's name for example).
- library_build_tag is a string you want to use to represent the specific build of the library (for example, the SVN revision number or a timestamp of when the library was compiled). This is useful for tracking instances of your library as you are developing them.
- library_version is the version of your library. You can use whatever numbering or naming scheme you want.
- library_sdk_version is the version of the Vertica Analytic Database SDK Library for which you've compiled the library.

> **Note**
> This field isn't used to determine whether a library is compatible with a version of the Vertica Analytic Database server. The version of the Vertica Analytic Database SDK you use to compile your library is embedded in the library when you compile it. It is this information that Vertica Analytic Database server uses to determine if your library is compatible with it.

- source_url is a URL where users of your function can find more information about it. This can be your company's website, the GitHub page hosting your library's source code, or whatever site you like.
- description is a concise description of your library.
- licenses_required is a placeholder for licensing information. You must pass an empty string for this value.
- signature is a placeholder for a signature that will authenticate your library. You must pass an empty string for this value.

The following example shows how to add metadata to an R UDx.

```
RegisterLibrary("Speedy Analytics Ltd.",
                "1234",
                "1.0",
                "8.1.0",
                "http://www.example.com/sales_tax_calculator.R",
                "Sales Tax R Library",
                "",
                "")
```

Loading the library and querying the USER_LIBRARIES system table shows the metadata supplied in the call to RegisterLibrary :

```
=> CREATE LIBRARY rLib AS '/home/dbadmin/sales_tax_calculator.R' LANGUAGE 'R';
CREATE LIBRARY
=> SELECT * FROM USER_LIBRARIES WHERE lib_name = 'rLib';
-[ RECORD 1 ]-----+-------------------------------------------------------
schema_name       | public
lib_name          | rLib
lib_oid           | 45035996273708350
author            | Speedy Analytics Ltd.
owner_id          | 45035996273704962
lib_file_name     | rLib_02552872a35d9352b4907d3fcd03cf9700a0000000000d3e.R
md5_sum           | 30da555537c4d93c352775e4f31332d2
sdk_version       |
revision          |
lib_build_tag     | 1234
lib_version       | 1.0
lib_sdk_version   | 8.1.0
source_url        | http://www.example.com/sales_tax_calculator.R
description       | Sales Tax R Library
licenses_required |
signature         |
dependencies      |
is_valid          | t
sal_storage_id    | 02552872a35d9352b4907d3fcd03cf9700a0000000000d3e
```

Setting null input and volatility behavior for R functions

Vertica supports defining volatility and null-input settings for UDxs written in R. Both settings aid in the performance of your R function.

Volatility settings

Volatility settings describe the behavior of the function to the Vertica optimizer. For example, if you have identical rows of input data and you know the UDx is immutable, then you can define the UDx as IMMUTABLE. This tells the Vertica optimizer that it can return a cached value for subsequent identical rows on which the function is called rather than having the function run on each identical row.

To indicate your UDx's volatility, set the volatility parameter of your R factory function to one of the following values:

Value	Description
VOLATILE	Repeated calls to the function with the same arguments always result in different values. Vertica always calls volatile functions for each invocation.
IMMUTABLE	Calls to the function with the same arguments always results in the same return value.
STABLE	Repeated calls to the function with the same arguments *within the same statement* returns the same output. For example, a function that returns the current user name is stable because the user cannot change within a statement. The user name could change between statements.
DEFAULT_VOLATILITY	The default volatility. This is the same as VOLATILE.

If you do not define a volatility, then the function is considered to be VOLATILE.

The following example sets the volatility to STABLE in the multiplyTwoIntsFactory function:

```
multiplyTwoIntsFactory <- function() {
list(name                = multiplyTwoInts,
     udxtype             = c("scalar"),
     intype              = c("float","float"),
     outtype             = c("float"),
     volatility          = c("stable"),
     parametertypecallback = multiplyTwoIntsParameters)
}
```

Null input behavior

Null input setting determine how to respond to rows that have null input. For example, you can choose to return null if any inputs are null rather than calling the function and having the function deal with a NULL input.

To indicate how your UDx reacts to NULL input, set the strictness parameter of your R factory function to one of the following values:

Value	Description
CALLED_ON_NULL_INPUT	The function must be called, even if one or more arguments are NULL.
RETURN_NULL_ON_NULL_INPUT	The function always returns a NULL value if any of its arguments are NULL.
STRICT	A synonym for RETURN_NULL_ON_NULL_INPUT
DEFAULT_STRICTNESS	The default strictness setting. This is the same as CALLED_ON_NULL_INPUT.

If you do not define a null input behavior, then the function is called on every row of data regardless of the presence of NULL values.

The following example sets the NULL input behavior to STRICT in the multiplyTwoIntsFactory function:

```
multiplyTwoIntsFactory <- function() {
list(name                = multiplyTwoInts,
     udxtype             = c("scalar"),
     intype              = c("float","float"),
     outtype             = c("float"),
     strictness          = c("strict"),
     parametertypecallback = multiplyTwoIntsParameters)
}
```

Debugging tips

The following tips can help you debug your UDx before deploying it in a production environment.

Use a single node for initial debugging

You can attach to the Vertica process using a debugger such as gdb to debug your UDx code. Doing this in a multi-node environment, however, is very difficult. Therefore, consider setting up a single-node Vertica test environment to initially debug your UDx.

Use logging

Each UDx has an associated ServerInterface instance. The ServerInterface provides functions to write to the Vertica log and, in the C++ API only, a system table. See Logging for more information.

Arguments and return values

For all UDx types except load (UDL), the factory class declares the arguments and return type of the associated function. Factories have two methods for this purpose:

- getPrototype() (required): declares input and output types
- getReturnType() (sometimes required): declares the return types, including length and precision, when applicable

The getPrototype() method receives two ColumnTypes parameters, one for input and one for output. The factory in C++ example: string tokenizer takes a single input string and returns a string:

```
virtual void getPrototype(ServerInterface &srvInterface,
                          ColumnTypes &argTypes, ColumnTypes &returnType)
{
    argTypes.addVarchar();
    returnType.addVarchar();
}
```

The ColumnTypes class provides "add" methods for each supported type, like addVarchar() . This class supports complex types with the addArrayType() and addRowType() methods; see Complex Types as Arguments . If your function is polymorphic, you can instead call addAny() . You are then responsible for validating your inputs and outputs. For more information about implementing polymorphic UDxs, see Creating a polymorphic UDx .

The getReturnType() method computes a maximum length for the returned value. If your UDx returns a sized column (a return data type whose length can vary, such as a VARCHAR), a value that requires precision, or more than one value, implement this factory method. (Some UDx types require you to implement it.)

The input is a SizedColumnTypes containing the input argument types along with their lengths. Depending on the input types, add one of the following to the output types:

- CHAR, (LONG) VARCHAR, BINARY, and (LONG) VARBINARY: return the maximum length.
- NUMERIC types: specify the precision and scale.
- TIME and TIMESTAMP values (with or without timezone): specify precision.
- INTERVAL YEAR TO MONTH: specify range.
- INTERVAL DAY TO SECOND: specify precision and range.
- ARRAY: specify the maximum number of array elements.

In the case of the string tokenizer, the output is a VARCHAR and the function determines its maximum length:

```
// Tell Vertica what our return string length will be, given the input
// string length
virtual void getReturnType(ServerInterface &srvInterface,
                           const SizedColumnTypes &inputTypes,
                           SizedColumnTypes &outputTypes)
{
    // Error out if we're called with anything but 1 argument
    if (inputTypes.getColumnCount() != 1)
        vt_report_error(0, "Function only accepts 1 argument, but %zu provided", inputTypes.getColumnCount());

    int input_len = inputTypes.getColumnType(0).getStringLength();

    // Our output size will never be more than the input size
    outputTypes.addVarchar(input_len, "words");
}
```

Complex types as arguments and return values

The ColumnTypes class supports ARRAY and ROW types. Arrays have elements and rows have fields, both of which have types that you need to describe. To work with complex types, you build ColumnTypes objects for the array or row and then add them to the ColumnTypes objects representing the function inputs and outputs.

In the following example, the input to a transform function is an array of orders, which are rows, and the output is the individual rows with their positions in the array. An order consists of a shipping address (VARCHAR) and an array of product IDs (INT).

C++

Python

The factory's getPrototype() method first creates ColumnTypes for the array and row elements and then calls addArrayType() and addRowType() using them:

```
void getPrototype(ServerInterface &srv,
        ColumnTypes &argTypes,
        ColumnTypes &retTypes)
  {
    // item ID (int), to be used in an array
    ColumnTypes itemIdProto;
    itemIdProto.addInt();

    // row: order = address (varchar) + array of previously-created item IDs
    ColumnTypes orderProto;
    orderProto.addVarchar();                /* address */
    orderProto.addArrayType(itemIdProto);   /* array of item ID */

    /* argument (input) is array of orders */
    argTypes.addArrayType(orderProto);

    /* return values: index in the array, order */
    retTypes.addInt();                      /* index of element */
    retTypes.addRowType(orderProto);        /* element return type */
  }
```

The arguments include a sized type (the VARCHAR). The getReturnType() method uses a similar approach, using the Fields class to build the two fields in the order.

```
void getReturnType(ServerInterface &srv,
        const SizedColumnTypes &argTypes,
        SizedColumnTypes &retTypes)
  {
    Fields itemIdElementFields;
    itemIdElementFields.addInt("item_id");

    Fields orderFields;
    orderFields.addVarchar(32, "address");
    orderFields.addArrayType(itemIdElementFields[0], "item_id");
    // optional third arg: max length, default unbounded

    /* declare return type */
    retTypes.addInt("index");
    static_cast<Fields &>(retTypes).addRowType(orderFields, "element");

    /* NOTE: presumably we have verified that the arguments match the prototype, so really we could just do this: */
    retTypes.addInt("index");
    retTypes.addArg(argTypes.getColumnType(0).getElementType(), "element");
  }
```

To access complex types in the UDx processing method, use the ArrayReader , ArrayWriter , StructReader , and StructWriter classes.

See C++ example: using complex types for a polymorphic function that uses arrays.

Handling different numbers and types of arguments

You can create UDxs that handle multiple signatures, or even accept all arguments supplied to them by the user, using either overloading or polymorphism.

You can overload your UDx by assigning the same SQL function name to multiple factory classes, each of which defines a unique function signature. When a user uses the function name in a query, Vertica tries to match the signature of the function call to the signatures declared by the factory's getPrototype() method. This is the best technique to use if your UDx needs to accept a few different signatures (for example, accepting two required and one optional argument).

Alternatively, you can write a polymorphic function, writing one factory method instead of several and declaring that it accepts any number and type of arguments. When a user uses the function name in a query, Vertica calls your function regardless of the signature. In exchange for this flexibility, your UDx's main "process" method has to determine whether it can accept the arguments and emit errors if not.

All UDx types can use polymorphic inputs. Transform functions and analytic functions can also use polymorphic outputs. This means that getPrototype() can declare a return type of "any" and set the actual return type at runtime. For example, a function that returns the largest value in an input would return the same type as the input type.

In this section

- Overloading your UDx
- Creating a polymorphic UDx

Overloading your UDx

You may want your UDx to accept several different signatures (sets of arguments). For example, you might want your UDx to accept:

- One or more optional arguments.
- One or more arguments that can be one of several data types.
- Completely distinct signatures (either all INTEGER or all VARCHAR, for example).

You can create a function with this behavior by creating several factory classes, each of which accepts a different signature (the number and data types of arguments). You can then associate a single SQL function name with all of them. You can use the same SQL function name to refer to multiple factory classes as long as the signature defined by each factory is unique. When a user calls your UDx, Vertica matches the number and types of arguments supplied by the user to the arguments accepted by each of your function's factory classes. If one matches, Vertica uses it to instantiate a function class to process the data.

Multiple factory classes can instantiate the same function class, so you can re-use one function class that is able to process multiple sets of arguments and then create factory classes for each of the function signatures. You can also create multiple function classes if you want.

See the C++ example: overloading your UDx and Java example: overloading your UDx examples.

In this section

- C++ example: overloading your UDx
- Java example: overloading your UDx

C++ example: overloading your UDx

The following example code demonstrates creating a user-defined scalar function (UDSF) that adds two or three integers together. The Add2or3ints class is prepared to handle two or three arguments. The processBlock() function checks the number of arguments that have been passed to it, and adds all two or three of them together. It also exits with an error message if it has been called with less than 2 or more than 3 arguments. In theory, this should never happen, since Vertica only calls the UDSF if the user's function call matches a signature on one of the factory classes you create for your function. In practice, it is a good idea to perform this sanity checking, in case your (or someone else's) factory class inaccurately reports a set of arguments your function class cannot handle.

```
#include "Vertica.h"
using namespace Vertica;
using namespace std;
// a ScalarFunction that accepts two or three
// integers and adds them together.
class Add2or3ints : public Vertica::ScalarFunction

public:
    virtual void processBlock(Vertica::ServerInterface &srvInterface,
                              Vertica::BlockReader &arg_reader,
                              Vertica::BlockWriter &res_writer)
    {
        const size_t numCols = arg_reader.getNumCols();

        // Ensure that only two or three parameters are passed in
        if ( numCols < 2 || numCols > 3)
            vt_report_error(0, "Function only accept 2 or 3 arguments, "
                            "but %zu provided", arg_reader.getNumCols());
        // Add two integers together
        do {
```

```
        const vint a = arg_reader.getIntRef(0);
        const vint b = arg_reader.getIntRef(1);
        vint c = 0;
      // Check for third argument, add it in if it exists.
        if (numCols == 3)
            c = arg_reader.getIntRef(2);
        res_writer.setInt(a+b+c);
        res_writer.next();
      } while (arg_reader.next());
    }
};
// This factory accepts function calls with two integer arguments.
class Add2intsFactory : public Vertica::ScalarFunctionFactory
{
    virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface
            &srvInterface)
    { return vt_createFuncObj(srvInterface.allocator, Add2or3ints); }
    virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                              Vertica::ColumnTypes &argTypes,
                              Vertica::ColumnTypes &returnType)
    {   // Accept 2 integer values
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }
};
RegisterFactory(Add2intsFactory);
// This factory defines a function that accepts 3 ints.
class Add3intsFactory : public Vertica::ScalarFunctionFactory
{
    virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface
            &srvInterface)
    { return vt_createFuncObj(srvInterface.allocator, Add2or3ints); }
    virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                              Vertica::ColumnTypes &argTypes,
                              Vertica::ColumnTypes &returnType)
    {   // accept 3 integer values
        argTypes.addInt();
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }
};
RegisterFactory(Add3intsFactory);
```

The example has two ScalarFunctionFactory classes, one for each signature that the function accepts (two integers and three integers). There is nothing unusual about these factory classes, except that their implementation of ScalarFunctionFactory::createScalarFunction() both create Add2or3ints objects.

The final step is to bind the same SQL function name to both factory classes. You can assign multiple factories to the same SQL function, as long as the signatures defined by each factory's getPrototype() implementation are different.

```
=> CREATE LIBRARY add2or3IntsLib AS '/home/dbadmin/Add2or3Ints.so';
CREATE LIBRARY
=> CREATE FUNCTION add2or3Ints as NAME 'Add2intsFactory' LIBRARY add2or3IntsLib FENCED;
CREATE FUNCTION
=> CREATE FUNCTION add2or3Ints as NAME 'Add3intsFactory' LIBRARY add2or3IntsLib FENCED;
CREATE FUNCTION
=> SELECT add2or3Ints(1,2);
 add2or3Ints
-------------
           3
(1 row)
=> SELECT add2or3Ints(1,2,4);
 add2or3Ints
-------------
           7
(1 row)
=> SELECT add2or3Ints(1,2,3,4); -- Will generate an error
ERROR 3467:  Function add2or3Ints(int, int, int, int) does not exist, or
permission is denied for add2or3Ints(int, int, int, int)
HINT:  No function matches the given name and argument types. You may
need to add explicit type casts
```

The error message in response to the final call to the add2or3Ints function was generated by Vertica, since it could not find a factory class associated with add2or3Ints that accepted four integer arguments. To expand add2or3Ints further, you could create another factory class that accepted this signature, and either change the Add2or3ints ScalarFunction class or create a totally different class to handle adding more integers together. However, adding more classes to accept each variation in the arguments quickly becomes overwhelming. In that case, you should consider creating a polymorphic UDx.

Java example: overloading your UDx

The following example code demonstrates creating a user-defined scalar function (UDSF) that adds two or three integers together. The Add2or3ints class is prepared to handle two or three arguments. It checks the number of arguments that have been passed to it, and adds all two or three of them together. The processBlock() method checks whether it has been called with less than 2 or more than 3 arguments. In theory, this should never happen, since Vertica only calls the UDSF if the user's function call matches a signature on one of the factory classes you create for your function. In practice, it is a good idea to perform this sanity checking, in case your (or someone else's) factory class reports that your function class accepts a set of arguments that it actually does not.

```
// You need to specify the full package when creating functions based on
// the classes in your library.
package com.mycompany.multiparamexample;
// Import the entire Vertica SDK
import com.vertica.sdk.*;
// This ScalarFunction accepts two or three integer arguments. It tests
// the number of input columns to determine whether to read two or three
// arguments as input.
public class Add2or3ints extends ScalarFunction
{
  @Override
  public void processBlock(ServerInterface srvInterface,
                           BlockReader argReader,
                           BlockWriter resWriter)
          throws UdfException, DestroyInvocation
  {
    // See how many arguments were passed in
    int numCols = argReader.getNumCols();

    // Return an error if less than two or more than 3 aerguments
    // were given. This error only occurs if a Factory class that
    // accepts the wrong number of arguments instantiates this
    // class.
    if (numCols < 2 || numCols > 3) {
      throw new UdfException(0,
          "Must supply 2 or 3 integer arguments");
    }

    // Process all of the rows of input.
    do {
      // Get the first two integer arguments from the BlockReader
      long a = argReader.getLong(0);
      long b = argReader.getLong(1);

      // Assume no third argument.
      long c = 0;

      // Get third argument value if it exists
      if (numCols == 3) {
        c = argReader.getLong(2);
      }

      // Process the arguments and come up with a result. For this
      // example, just add the three arguments together.
      long result = a+b+c;

      // Write the integer output value.
      resWriter.setLong(result);

      // Advance the output BlockWriter to the next row.
      resWriter.next();

      // Continue processing input rows until there are no more.
    } while (argReader.next());
  }
```

The main difference between the Add2ints class and the Add2or3ints class is the inclusion of a section that gets the number of arguments by calling BlockReader.getNumCols() . This class also tests the number of columns it received from Vertica to ensure it is in the range it is prepared to handle. This test will only fail if you create a ScalarFunctionFactory whose getPrototype() method defines a signature that accepts less than two or more than

three arguments. This is not really necessary in this simple example, but for a more complicated class it is a good idea to test the number of columns and data types that Vertica passed your function class.

Within the do loop, Add2or3ints uses a default value of zero if Vertica sent it two input columns. Otherwise, it retrieves the third value and adds that to the other two. Your own class needs to use default values for missing input columns or alter its processing in some other way to handle the variable columns.

You must define your function class in its own source file, rather than as an inner class of one of your factory classes since Java does not allow the instantiation of an inner class from outside its containing class. You factory class has to be available for instantiation by multiple factory classes.

Once you have created a function class or classes, you create a factory class for each signature you want your function class to handle. These factory classes can call individual function classes, or they can all call the same class that is prepared to accept multiple sets of arguments.

The following example's createScalarFunction() method instantiates a member of the Add2or3ints class.

```
// You will need to specify the full package when creating functions based on
// the classes in your library.
package com.mycompany.multiparamexample;
// Import the entire Vertica SDK
import com.vertica.sdk.*;
public class Add2intsFactory extends ScalarFunctionFactory
{

  @Override
  public void getPrototype(ServerInterface srvInterface,
                           ColumnTypes argTypes,
                           ColumnTypes returnType)
  {
    // Accept two integers as input
    argTypes.addInt();
    argTypes.addInt();
    // writes one integer as output
    returnType.addInt();
  }
  @Override
  public ScalarFunction createScalarFunction(ServerInterface srvInterface)
  {
    // Instantiate the class that can handle either 2 or 3 integers.
    return new Add2or3ints();
  }
```

The following ScalarFunctionFactory subclass accepts three integers as input. It, too, instantiates a member of the Add2or3ints class to process the function call:

```
// You will need to specify the full package when creating functions based on
// the classes in your library.
package com.mycompany.multiparamexample;
// Import the entire Vertica SDK
import com.vertica.sdk.*;
public class Add3intsFactory extends ScalarFunctionFactory
{
   @Override
   public void getPrototype(ServerInterface srvInterface,
                            ColumnTypes argTypes,
                            ColumnTypes returnType)
   {
      // Accepts three integers as input
      argTypes.addInt();
      argTypes.addInt();
      argTypes.addInt();
      // Returns a single integer
      returnType.addInt();
   }
   @Override
   public ScalarFunction createScalarFunction(ServerInterface srvInterface)
   {
      // Instantiates the Add2or3ints ScalarFunction class, which is able to
      // handle eitehr 2 or 3 integers as arguments.
      return new Add2or3ints();
   }
}
```

The factory classes and the function class or classes they call must be packaged into the same JAR file (see Compiling and packaging a Java library for details). If a host in the database cluster has the JDK installed on it, you could use the following commands to compile and package the example:

```
$ cd pathToJavaProject$ javac -classpath /opt/vertica/bin/VerticaSDK.jar \
> com/mycompany/multiparamexample/*.java
$ jar -cvf Add2or3intslib.jar com/vertica/sdk/BuildInfo.class \
> com/mycompany/multiparamexample/*.class
added manifest
adding: com/vertica/sdk/BuildInfo.class(in = 1202) (out= 689)(deflated 42%)
adding: com/mycompany/multiparamexample/Add2intsFactory.class(in = 677) (out= 366)(deflated 45%)
adding: com/mycompany/multiparamexample/Add2or3ints.class(in = 919) (out= 601)(deflated 34%)
adding: com/mycompany/multiparamexample/Add3intsFactory.class(in = 685) (out= 369)(deflated 46%)
```

Once you have packaged your overloaded UDx, you deploy it the same way as you do a regular UDx, except you use multiple CREATE FUNCTION statements to define the function, once for each factory class.

```
=> CREATE LIBRARY add2or3intslib as '/home/dbadmin/Add2or3intslib.jar'
-> language 'Java';
CREATE LIBRARY
=> CREATE FUNCTION add2or3ints as LANGUAGE 'Java' NAME 'com.mycompany.multiparamexample.Add2intsFactory' LIBRARY add2or3intslib;
CREATE FUNCTION
=> CREATE FUNCTION add2or3ints as LANGUAGE 'Java' NAME 'com.mycompany.multiparamexample.Add3intsFactory' LIBRARY add2or3intslib;
CREATE FUNCTION
```

You call the overloaded function the same way you call any other function.

```
=> SELECT add2or3ints(2,3);
 add2or3ints
-------------
           5
(1 row)
=> SELECT add2or3ints(2,3,4);
 add2or3ints
-------------
           9
(1 row)
=> SELECT add2or3ints(2,3,4,5);
ERROR 3457:  Function add2or3ints(int, int, int, int) does not exist, or permission is denied for add2or3ints(int, int, int, int)
HINT:  No function matches the given name and argument types. You may need to add explicit type casts
```

The last error was generated by Vertica, not the UDx code. It returns an error if it cannot find a factory class whose signature matches the function call's signature.

Creating an overloaded UDx is useful if you want your function to accept a limited set of potential arguments. If you want to create a more flexible function, you can create a polymorphic function.

Creating a polymorphic UDx

Polymorphic UDxs accept any number and type of argument that the user supplies. Transform functions (UDTFs), analytic functions (UDAnFs), and aggregate functions (UDAFs) can define their output return types at runtime, usually based on the input arguments. For example, a UDTF that adds two numbers could return an integer or a float, depending on the input types.

Vertica does not check the number or types of argument that the user passes to the UDx—it just passes the UDx all of the arguments supplied by the user. It is up to your polymorphic UDx's main processing function (for example, processBlock() in user-defined scalar functions) to examine the number and types of arguments it received and determine if it can handle them. UDxs support up to 9800 arguments.

Polymorphic UDxs are more flexible than using multiple factory classes for your function (see Overloading your UDx). They also allow you to write more concise code, instead of writing versions for each data type. The tradeoff is that your polymorphic function needs to perform more work to determine whether it can process its arguments.

Your polymorphic UDx declares that it accepts any number of arguments in its factory's getPrototype() function by calling the addAny() function on the ColumnTypes object that defines its arguments, as follows:

```
// C++ example
void getPrototype(ServerInterface &srvInterface,
                  ColumnTypes &argTypes,
                  ColumnTypes &returnType)
{
    argTypes.addAny(); // Must be only argument type.
    returnType.addInt(); // or whatever the function returns
}
```

This "any parameter" argument type is the only one that your function can declare. You cannot define required arguments and then call addAny() to declare the rest of the signature as optional. If your function has requirements for the arguments it accepts, your process() function must enforce them.

The getPrototype() example shown previously accepts any type and declares that it returns an integer. The following example shows a version of the method that defers resolving the return type until runtime. You can only use the "any" return type for transform and analytic functions.

```
void getPrototype(ServerInterface &srvInterface,
                  ColumnTypes &argTypes,
                  ColumnTypes &returnType)
{
    argTypes.addAny();
    returnType.addAny(); // type determined at runtime
}
```

If you use polymorphic return types, you must also define getReturnType() in your factory. This function is called at runtime to determine the actual return type. See C++ example: PolyNthValue for an example.

Polymorphic UDxs and schema search paths

If a user does not supply a schema name as part of a UDx call, Vertica searches each schema in the schema search path for a function whose name and signature match the function call. See Setting search paths for more information about schema search paths.

Because polymorphic UDxs do not have specific signatures associated with them, Vertica initially skips them when searching for a function to handle the function call. If none of the schemas in the search path contain a UDx whose name and signature match the function call, Vertica searches the schema search path again for a polymorphic UDx whose name matches the function name in the function call.

This behavior gives precedence to a UDx whose signature exactly matches the function call. It allows you to create a "catch-all" polymorphic UDx that Vertica calls only when none of the non-polymorphic UDxs with the same name have matching signatures.

This behavior may cause confusion if your users expect the first polymorphic function in the schema search path to handle a function call. To avoid confusion, you should:

- Avoid using the same name for different UDxs. You should always uniquely name UDxs unless you intend to create an overloaded UDx with multiple signatures.
- When you cannot avoid having UDxs with the same name in different schemas, always supply the schema name as part of the function call. Using the schema name prevents ambiguity and ensures that Vertica uses the correct UDx to process your function calls.

In this section

- C++ example: PolyNthValue
- Java example: AddAnyInts
- R example: kmeansPoly

C++ example: PolyNthValue

The PolyNthValue example is an analytic function that returns the value in the Nth row in each partition in its input. This function is a generalization of FIRST_VALUE [analytic] and LAST_VALUE [analytic] .

The values can be of any primitive data type.

For the complete source code, see PolymorphicNthValue.cpp in the examples (in /opt/vertica/sdk/examples/AnalyticFunctions/).

Loading and using the example

Load the library and create the function as follows:

```
=> CREATE LIBRARY AnalyticFunctions AS '/home/dbadmin/AnalyticFns.so';
CREATE LIBRARY

=> CREATE ANALYTIC FUNCTION poly_nth_value AS LANGUAGE 'C++'
   NAME 'PolyNthValueFactory' LIBRARY AnalyticFunctions;
CREATE ANALYTIC FUNCTION
```

Consider a table of scores for different test groups:

```
=> SELECT cohort, score FROM trials;
 cohort | score
--------+-------
   1    | 9
   1    | 8
   1    | 7
   3    | 3
   3    | 2
   3    | 1
   2    | 4
   2    | 5
   2    | 6
(9 rows)
```

Call the function in a query that uses an OVER clause to partition the data. This example returns the second-highest score in each cohort:

```
=> SELECT cohort, score, poly_nth_value(score USING PARAMETERS n=2) OVER (PARTITION BY cohort) AS nth_value
FROM trials;
 cohort | score | nth_value
--------+-------+-----------
  1   | 9   |         8
  1   | 8   |         8
  1   | 7   |         8
  3   | 3   |         2
  3   | 2   |         2
  3   | 1   |         2
  2   | 4   |         5
  2   | 5   |         5
  2   | 6   |         5
(9 rows)
```

Factory implementation

The factory declares that the class is polymorphic, and then sets the return type based on the input type. Two factory methods specify the argument and return types.

Use the getPrototype() method to declare that the analytic function takes and returns any type:

```
void getPrototype(ServerInterface &srvInterface, ColumnTypes &argTypes, ColumnTypes &returnType)
{
    // This function supports any argument data type
    argTypes.addAny();

    // Output data type will be the same as the argument data type
    // We will specify that in getReturnType()
    returnType.addAny();
}
```

The getReturnType() method is called at runtime. This is where you set the return type based on the input type:

```
void getReturnType(ServerInterface &srvInterface, const SizedColumnTypes &inputTypes,
                   SizedColumnTypes &outputTypes)
{
    // This function accepts only one argument
    // Complain if we find a different number
    std::vector<size_t> argCols;
    inputTypes.getArgumentColumns(argCols); // get argument column indices

    if (argCols.size() != 1)
    {
        vt_report_error(0, "Only one argument is expected but %s provided",
                        argCols.size()? std::to_string(argCols.size()).c_str() : "none");
    }

    // Define output type the same as argument type
    outputTypes.addArg(inputTypes.getColumnType(argCols[0]), inputTypes.getColumnName(argCols[0]));
}
```

Function implementation

The analytic function itself is type-agnostic:

```
void processPartition(ServerInterface &srvInterface, AnalyticPartitionReader &inputReader,
                      AnalyticPartitionWriter &outputWriter)
{
    try {
        const SizedColumnTypes &inTypes = inputReader.getTypeMetaData();
        std::vector<size_t> argCols; // Argument column indexes.
        inTypes.getArgumentColumns(argCols);

        vint currentRow = 1;
        bool nthRowExists = false;

        // Find the value of the n-th row
        do {
            if (currentRow == this->n) {
                nthRowExists = true;
                break;
            } else {
                currentRow++;
            }
        } while (inputReader.next());

        if (nthRowExists) {
            do {
                // Return n-th value
                outputWriter.copyFromInput(0 /*dest column*/, inputReader,
                                           argCols[0] /*source column*/);
            } while (outputWriter.next());
        } else {
            // The partition has less than n rows
            // Return NULL value
            do {
                outputWriter.setNull(0);
            } while (outputWriter.next());
        }
    } catch(std::exception& e) {
        // Standard exception. Quit.
        vt_report_error(0, "Exception while processing partition: [%s]", e.what());
    }
}
};
```

Java example: AddAnyInts

The following example shows an implementation of a Java ScalarFunction that adds together two or more integers.

For the complete source code, see AddAnyIntsInfo.java in the examples (in /opt/vertica/sdk/examples/JavaUDx/ScalarFunctions).

Loading and using the example

Load the library and create the function as follows:

```
=> CREATE LIBRARY JavaScalarFunctions AS '/home/dbadmin/JavaScalarLib.jar' LANGUAGE 'JAVA';
CREATE LIBRARY

=> CREATE FUNCTION addAnyInts AS LANGUAGE 'Java' NAME 'com.vertica.JavaLibs.AddAnyIntsInfo'
   LIBRARY JavaScalarFunctions;
CREATE FUNCTION
```

Call the function with two or more integer arguments:

```
=> SELECT addAnyInts(1,2);
 addAnyInts
------------
          3
(1 row)

=> SELECT addAnyInts(1,2,3,40,50,60,70,80,900);
 addAnyInts
------------
       1206
(1 row)
```

Calling the function with too few arguments, or with non-integer arguments, produces errors that are generated from the processBlock() method. It is up to your UDx to ensure that the user supplies the correct number and types of arguments to your function and exit with an error if it cannot process them.

Function implementation

Most of the work in the example is done by the processBlock() method. It performs two checks on the arguments that have been passed in through the BlockReader object:

- There are at least two arguments.
- The data types of all arguments are integers.

It is up to your polymorphic UDx to determine that all of the input passed to it is valid.

Once the processBlock() method validates its arguments, it loops over them, adding them together.

```
    @Override
    public void processBlock(ServerInterface srvInterface,
                             BlockReader arg_reader,
                             BlockWriter res_writer)
                throws UdfException, DestroyInvocation
    {
    SizedColumnTypes inTypes = arg_reader.getTypeMetaData();
    ArrayList<Integer> argCols = new ArrayList<Integer>(); // Argument column indexes.
    inTypes.getArgumentColumns(argCols);
    // While we have inputs to process
        do {
    long sum = 0;
    for (int i = 0; i < argCols.size(); ++i){
        long a = arg_reader.getLong(i);
        sum += a;
    }
            res_writer.setLong(sum);
            res_writer.next();
        } while (arg_reader.next());
    }
}
```

Factory implementation

The factory declares the number and type of arguments in the getPrototype() function.

```
  @Override
  public void getPrototype(ServerInterface srvInterface,
                           ColumnTypes argTypes,
                           ColumnTypes returnType)
  {
  argTypes.addAny();
    returnType.addInt();
  }
```

R example: kmeansPoly

The following example shows an implementation of a Transform Function (UDTF) that performs kmeans clustering on one or more input columns.

```
kmeansPoly <- function(v.data.frame,v.param.list) {
  # Computes clusters using the kmeans algorithm.
  #
  # Input: A dataframe and a list of parameters.
  # Output: A dataframe with one column that tells the cluster to which each data
  #         point belongs.
  # Args:
  #  v.data.frame: The data from Vertica cast as an R data frame.
  #  v.param.list: List of function parameters.
  #
  # Returns:
  #  The cluster associated with each data point.
  # Ensure k is not null.
  if(!is.null(v.param.list['k'])) {
    number_of_clusters <- as.numeric(v.param.list['k'])
  } else {
    stop("k cannot be NULL! Please use a valid value.")
  }
  # Run the kmeans algorithm.
  kmeans_clusters <- kmeans(v.data.frame, number_of_clusters)
  final.output <- data.frame(kmeans_clusters$cluster)
  return(final.output)
}

kmeansFactoryPoly <- function() {
  # This function tells Vertica the name of the R function,
  # and the polymorphic parameters.
  list(name=kmeansPoly, udxtype=c("transform"), intype=c("any"),
       outtype=c("int"), parametertypecallback=kmeansParameters)
}

kmeansParameters <- function() {
  # Callback function for the parameter types.
  function.parameters <- data.frame(datatype=rep(NA, 1), length=rep(NA,1),
                                    scale=rep(NA,1), name=rep(NA,1))
  function.parameters[1,1] = "int"
  function.parameters[1,4] = "k"
  return(function.parameters)
}
```

The polymorphic R function declares it accepts any number of arguments in its factory function by specifying "any" as the argument to the intype parameter and optionally the outtype parameter. If you define "any" argument for intype or outtype , then it is the only type that your function can declare for the respective parameter. You cannot define required arguments and then call "any" to declare the rest of the signature as optional. If your function has requirements for the arguments it accepts, your process function must enforce them.

The outtypecallback method is used to indicate the argument types and sizes it has been called with, and is expected to indicate the types and sizes that the function returns. The outtypecallback method can also be used to check for unsupported types and/or number of arguments. For example, the function may require only integers, with no more than 10 of them.

You assign a SQL name to your polymorphic UDx using the same statement you use to assign one to a non-polymorphic UDx. The following statements show how you load and call the polymorphic function from the example.

```
=> CREATE LIBRARY rlib2 AS '/home/dbadmin/R_UDx/poly_kmeans.R' LANGUAGE 'R';
CREATE LIBRARY
=> CREATE TRANSFORM FUNCTION kmeansPoly AS LANGUAGE 'R' name 'kmeansFactoryPoly' LIBRARY rlib2;
CREATE FUNCTION
=> SELECT spec, kmeansPoly(sl,sw,pl,pw USING PARAMETERS k = 3)
   OVER(PARTITION BY spec) AS Clusters
    FROM iris;
     spec       | Clusters
-----------------+----------
 Iris-setosa     |        1
 Iris-setosa     |        1
 Iris-setosa     |        1
 Iris-setosa     |        1
.
.
.
(150 rows)
```

UDx parameters

Parameters let you define arguments for your UDxs that remain constant across all of the rows processed by the SQL statement that calls your UDx. Typically, your UDxs accept arguments that come from columns in a SQL statement. For example, in the following SQL statement, the arguments a and b to the add2ints UDSF change value for each row processed by the SELECT statement:

```
=> SELECT a, b, add2ints(a,b) AS 'sum' FROM example;
a | b  | sum
---+----+-----
1 |  2 |   3
3 |  4 |   7
5 |  6 |  11
7 |  8 |  15
9 | 10 |  19
(5 rows)
```

Parameters remain constant for all the rows your UDx processes. You can also make parameters optional so that if the user does not supply it, your UDx uses a default value. For example, the following example demonstrates calling a UDSF named add2intsWithConstant that has a single parameter value named constant whose value is added to each the arguments supplied in each row of input:

```
=> SELECT a, b, add2intsWithConstant(a, b USING PARAMETERS constant=42)
   AS 'a+b+42' from example;
a | b  | a+b+42
---+----+--------
1 |  2 |     45
3 |  4 |     49
5 |  6 |     53
7 |  8 |     57
9 | 10 |     61
(5 rows)
```

> **Note**
> When calling a UDx with parameters, there is no comma between the last argument and the USING PARAMETERS clause.

The topics in this section explain how to develop UDxs that accept parameters.

In this section

- Defining UDx parameters
- Getting parameter values in UDxs
- Calling UDxs with parameters
- Specifying the behavior of passing unregistered parameters
- User-defined session parameters

- C++ example: defining parameters
- C++ example: using session parameters
- Java example: defining parameters
- Java example: using session parameters

Defining UDx parameters

You define the parameters that your UDx accepts in its factory class (ScalarFunctionFactory , AggregateFunctionFactory , and so on) by implementing getParameterType() . This method is similar to getReturnType() : you call data-type-specific methods on a SizedColumnTypes object that is passed in as a parameter. Each function call sets the name, data type, and width or precision (if the data type requires it) of the parameter.

> **Note**
> Parameter names in the __ *param-name* __ format are reserved for internal use.

Setting parameter properties (C++ only)

When you add parameters to the getParameterType() function using the C++ API, you can also set properties for each parameter. For example, you can define a parameter as being required by the UDx. Doing so lets the Vertica server know that every UDx invocation must provide the specified parameter, or the query fails.

By passing an object to the SizedColumnTypes::Properties class, you can define the following four parameter properties:

Parameter	Type	Description
visible	BOOLEAN	If set to TRUE, the parameter appears in the USER_FUNCTION_PARAMETERS table. You may want to set this to FALSE to declare a parameter for internal use only.
required	BOOLEAN	If set to TRUE: • The parameter is required when invoking the UDx. • Invoking the UDx without supplying the parameter results in an error, and the UDx does not run.
canBeNull	BOOLEAN	If set to TRUE, the parameter can have a NULL value. If set to FALSE, make sure that the supplied parameter does not contain a NULL value when invoking the UDx. Otherwise, an error results, and the UDx does not run.
comment	VARCHAR(128)	A comment to describe the parameter. If you exceed the 128 character limit, Vertica generates an error when you run the CREATE_FUNCTION command. Additionally, if you replace the existing function definition in the comment parameter, make sure that the new definition does not exceed 128 characters. Otherwise, you delete all existing entries in the USER_FUNCTION_PARAMETERS table related to the UDx.

Setting parameter properties (R only)

When using parameters in your R UDx, you must specify a field in the factory function called parametertypecallback . This field points to the callback function that defines the parameters expected by the function. The callback function defines a four-column data frame with the following properties:

Parameter	Type	Description
datatype	VARCHAR(128)	The data type of the parameter.
length	INTEGER	The dimension of the parameter.
scale	INTEGER	The proportional dimensions of the parameter.
name	VARCHAR(128)	The name of the parameter.

If any of the columns are left blank (or the parametertypecallback function is omitted), then Vertica uses default values.

For more information, see Parametertypecallback function .

Redacting UDx parameters

If a parameter name meets any of the following criteria, its value is automatically redacted from logs and system tables like QUERY_REQUESTS :

- Named "secret" or "password"
- Ends with "_secret" or "_password"

Getting parameter values in UDxs

Your UDx uses the parameter values it declared in its factory class (see Defining UDx parameters) in its function class's processing method (for example, processBlock() or processPartition()). It gets its parameter values from a ParamReader object, which is available from the ServerInterface object that is passed to your processing method. Reading parameters from this object is similar to reading argument values from BlockReader or PartitionReader objects: you call a data-type-specific function with the name of the parameter to retrieve its value. For example, in C++:

```
// Get the parameter reader from the ServerInterface to see if there are supplied parameters.
ParamReader paramReader = srvInterface.getParamReader();
// Get the value of an int parameter named constant.
const vint constant = paramReader.getIntRef("constant");
```

> **Note**
> String data values do not have any of their escape characters processed before they are passed to your function. Therefore, your function may need to process the escape sequences itself if it needs to operate on unescaped character values.

Using parameters in the factory class

In addition to using parameters in your UDx function class, you can also access the parameters in the factory class. You may want to access the parameters to let the user control the input or output values of your function in some way. For example, your UDx can have a parameter that lets the user choose to have your UDx return a single- or double-precision value. The process of accessing parameters in the factory class is the same as accessing it in the function class: get a ParamReader object from the ServerInterface 's getParamReader() method, them read the parameter values.

Testing whether the user supplied parameter values

Unlike its handling of arguments, Vertica does not immediately return an error if a user's function call does not include a value for a parameter defined by your UDx's factory class. This means that your function can attempt to read a parameter value that the user did not supply. If it does so, by default Vertica returns a non-existent parameter warning to the user, and the query containing the function call continues.

If you want your parameter to be optional, you can test whether the user supplied a value for the parameter before attempting to access its value. Your function determines if a value exists for a particular parameter by calling the ParamReader 's containsParameter() method with the parameter's name. If this call returns true, your function can safely retrieve the value. If this call returns false, your UDx can use a default value or change its processing in some other way to compensate for not having the parameter value. As long as your UDx does not try to access the non-existent parameter value, Vertica does not generate an error or warning about missing parameters.

> **Note**
> If the user passes your UDx a parameter that it has not defined, by default Vertica issues a warning that the parameter is not used. It still executes the SQL statement, ignoring the parameter. You can change this behavior by altering the `StrictUDxParameterChecking` configuration parameter.

See C++ example: defining parameters for an example.

Calling UDxs with parameters

You pass parameters to a UDx by adding a USING PARAMETERS clause in the function call after the last argument.

- Do *not* insert a comma between the last argument and the USING PARAMETERS clause.
- After the USING PARAMETERS clause, add one or more parameter definitions, in the following form:
  ```
  <parameter name> = <parameter value>
  ```
- Separate parameter definitions by commas.

Parameter values can be a constant expression (for example 1234 + SQRT(5678)). You cannot use volatile functions (such as RANDOM) in the expression, because they do not return a constant value. If you do supply a volatile expression as a parameter value, by default, Vertica returns an incorrect parameter type warning. Vertica then tries to run the UDx without the parameter value. If the UDx requires the parameter, it returns its own error, which cancels the query.

Calling a UDx with a single parameter

The following example demonstrates how you can call the Add2intsWithConstant UDSF example shown in C++ example: defining parameters :

```
=> SELECT a, b, Add2intsWithConstant(a, b USING PARAMETERS constant=42) AS 'a+b+42' from example;
 a | b  | a+b+42
---+----+--------
 1 |  2 |     45
 3 |  4 |     49
 5 |  6 |     53
 7 |  8 |     57
 9 | 10 |     61
(5 rows)
```

To remove the first instance of the number 3, you can call the RemoveSymbol UDSF example:

```
=> SELECT '3re3mo3ve3sy3mb3ol' original_string, RemoveSymbol('3re3mo3ve3sy3mb3ol' USING PARAMETERS symbol='3');
  original_string   |   RemoveSymbol
--------------------+-------------------
 3re3mo3ve3sy3mb3ol | re3mo3ve3sy3mb3ol
(1 row)
```

Calling a UDx with multiple parameters

The following example shows how you can call a version of the tokenize UDTF. This UDTF includes parameters to limit the shortest allowed word and force the words to be output in uppercase. Separate multiple parameters with commas.

```
=> SELECT url, tokenize(description USING PARAMETERS minLength=4, uppercase=true) OVER (partition by url) FROM T;
       url       |   words
-----------------+-----------
 www.amazon.com  | ONLINE
 www.amazon.com  | RETAIL
 www.amazon.com  | MERCHANT
 www.amazon.com  | PROVIDER
 www.amazon.com  | CLOUD
 www.amazon.com  | SERVICES
 www.dell.com    | LEADING
 www.dell.com    | PROVIDER
 www.dell.com    | COMPUTER
 www.dell.com    | HARDWARE
 www.vertica.com | WORLD'S
 www.vertica.com | FASTEST
 www.vertica.com | ANALYTIC
 www.vertica.com | DATABASE
(16 rows)
```

The following example calls the RemoveSymbol UDSF. By changing the value of the optional parameter, n , you can remove all instances of the number 3:

```
=> SELECT '3re3mo3ve3sy3mb3ol' original_string, RemoveSymbol('3re3mo3ve3sy3mb3ol' USING PARAMETERS symbol='3', n=6);
  original_string   | RemoveSymbol
--------------------+--------------
 3re3mo3ve3sy3mb3ol | removesymbol
(1 row)
```

Calling a UDx with optional or incorrect parameters

You can optionally add the Add2intsWithConstant UDSF's constant parameter. Calling this constraint without the parameter does not return an error or warning:

```
=> SELECT a,b,Add2intsWithConstant(a, b) AS 'sum' FROM example;
 a | b  | sum
---+----+-----
 1 |  2 |   3
 3 |  4 |   7
 5 |  6 |  11
 7 |  8 |  15
 9 | 10 |  19
(5 rows)
```

Although calling a UDx with incorrect parameters generates a warning, by default, the query still runs. For further information on setting the behavior of your UDx when you supply incorrect parameters, see Specifying the behavior of passing unregistered parameters .

```
=> SELECT a, b,  add2intsWithConstant(a, b USING PARAMETERS wrongparam=42) AS 'result' from example;
WARNING 4332:  Parameter wrongparam was not registered by the function and cannot
be coerced to a definite data type
 a | b  | result
---+----+--------
 1 |  2 |      3
 3 |  4 |      7
 5 |  6 |     11
 7 |  8 |     15
 9 | 10 |     19
(5 rows)
```

Specifying the behavior of passing unregistered parameters

By default, Vertica issues a warning message when you pass a UDx an unregistered parameter. An *unregistered parameter* is one that you did not declare in the getParameterType() method.

You can control the behavior of your UDx when you pass it an unregistered parameter by altering the StrictUDxParameterChecking configuration parameter.

Unregistered parameter behavior settings

You can specify the behavior of your UDx in response to one or more unregistered parameters. To do so, set the StrictUDxParameterChecking configuration parameter to one of the following values:

- 0: Allows unregistered parameters to be accessible to the UDx. The ParamReader class's getType() method determines the data type of the unregistered parameter. Vertica does not display any warning or error message.
- 1 (default): Ignores the unregistered parameter and allows the function to run. Vertica displays a warning message.
- 2: Returns an error and does not allow the function to run.

Examples

The following examples demonstrate the behavior you can specify using different values with the StrictUDxParameterChecking parameter.

View the current value of StrictUDxParameterChecking

To view the current value of the StrictUDxParameterChecking configuration parameter, run the following query:

```
=> \x
Expanded display is on.
=> SELECT * FROM configuration_parameters WHERE parameter_name = 'StrictUDxParameterChecking';
-[ RECORD 1 ]----------------+--------------------------------------------------------------------
node_name                    | ALL
parameter_name               | StrictUDxParameterChecking
current_value                | 1
restart_value                | 1
database_value               | 1
default_value                | 1
current_level                | DATABASE
restart_level                | DATABASE
is_mismatch                  | f
groups                       |
allowed_levels               | DATABASE
superuser_only               | f
change_under_support_guidance | f
change_requires_restart      | f
description                  | Sets the behavior to deal with undeclared UDx function parameters
```

Change the value of StrictUDxParameterChecking

You can change the value of the StrictUDxParameterChecking configuration parameter at the database, node, or session level. For example, you can change the value to '0' to specify that unregistered parameters can pass to the UDx without displaying a warning or error message:

```
=> ALTER DATABASE DEFAULT SET StrictUDxParameterChecking = 0;
ALTER DATABASE
```

Invalid parameter behavior with RemoveSymbol

The following example demonstrates how to call the RemoveSymbol UDSF example. The RemoveSymbol UDSF has a required parameter, symbol , and an optional parameter, n . In this case, you do not use the optional parameter.

If you pass both symbol and an additional parameter called wrongParam , which is not declared in the UDx, the behavior of the UDx changes corresponding to the value of StrictUDxParameterChecking .

When you set StrictUDxParameterChecking to '0', the UDx runs normally without a warning. Additionally, wrongParam becomes accessible to the UDx through the ParamReader object of the ServerInterface object:

```
=> ALTER DATABASE DEFAULT SET StrictUDxParameterChecking = 0;
ALTER DATABASE

=> SELECT '3re3mo3ve3sy3mb3ol' original_string, RemoveSymbol('3re3mo3ve3sy3mb3ol' USING PARAMETERS symbol='3', wrongParam='x');
  original_string   |   RemoveSymbol
--------------------+-------------------
 3re3mo3ve3sy3mb3ol | re3mo3ve3sy3mb3ol
(1 row)
```

When you set StrictUDxParameterChecking to '1', the UDx ignores wrongParam and runs normally. However, it also issues a warning message:

```
=> ALTER DATABASE DEFAULT SET StrictUDxParameterChecking = 1;
ALTER DATABASE

=> SELECT '3re3mo3ve3sy3mb3ol' original_string, RemoveSymbol('3re3mo3ve3sy3mb3ol' USING PARAMETERS symbol='3', wrongParam='x');
WARNING 4320:  Parameter wrongParam was not registered by the function and cannot be coerced to a definite data type
  original_string   |   RemoveSymbol
--------------------+-------------------
 3re3mo3ve3sy3mb3ol | re3mo3ve3sy3mb3ol
(1 row)
```

When you set StrictUDxParameterChecking to '2', the UDx encounters an error when it tries to call wrongParam and does not run. Instead, it generates an error message:

```
=> ALTER DATABASE DEFAULT SET StrictUDxParameterChecking = 2;
ALTER DATABASE

=> SELECT '3re3mo3ve3sy3mb3ol' original_string, RemoveSymbol('3re3mo3ve3sy3mb3ol' USING PARAMETERS symbol='3', wrongParam='x');
ERROR 0:  Parameter wrongParam was not registered by the function
```

User-defined session parameters

User-defined session parameters allow you to write more generalized parameters than what Vertica provides. You can configure user-defined session parameters in these ways:

- From the client—for example, with ALTER SESSION
- Through the UDx itself

A user-defined session parameter can be passed into any type of UDx supported by Vertica. You can also set parameters for your UDx at the session level. By specifying a user-defined session parameter, you can have the state of a parameter saved continuously. Vertica saves the state of the parameter even when the UDx is invoked multiple times during a single session.

The RowCount example uses a user-defined session parameter. This parameter counts the total number of rows processed by the UDx each time it runs. RowCount then displays the aggregate number of rows processed for all executions. See C++ example: using session parameters and Java example: using session parameters for implementations.

Viewing the user-defined session parameter

Enter the following command to see the value of all session parameters:

```
=> SHOW SESSION UDPARAMETER all;
schema | library | key | value
--------+---------+-----+-------
(0 rows)
```

No value has been set, so the table is empty. Now, execute the UDx:

```
=> SELECT RowCount(5,5);
RowCount
----------
10
(1 row)
```

Again, enter the command to see the value of the session parameter:

```
=> SHOW SESSION UDPARAMETER all;
schema |  library  |   key    | value
--------+-----------+----------+-------
public | UDSession | rowcount | 1
(1 row)
```

The library column shows the name of the library containing the UDx. This is the name set with CREATE LIBRARY . Because the UDx has processed one row, the value of the rowcount session parameter is now 1. Running the UDx two more times should increment the value twice.

```
=> SELECT RowCount(10,10);
RowCount
----------
20
(1 row)
=> SELECT RowCount(15,15);
RowCount
----------
30
(1 row)
```

You have now executed the UDx three times, obtaining the sum of 5 + 5, 10 + 10, and 15 + 15. Now, check the value of rowcount.

```
=> SHOW SESSION UDPARAMETER all;
schema |  library  |   key    | value
--------+-----------+----------+-------
public | UDSession | rowcount | 3
(1 row)
```

Altering the user-defined session parameter

You can also manually alter the value of rowcount. To do so, enter the following command:

```
=> ALTER SESSION SET UDPARAMETER FOR UDSession rowcount = 25;
ALTER SESSION
```

Check the value of RowCount:

```
=> SHOW SESSION UDPARAMETER all;
schema |  library  |   key    | value
--------+-----------+----------+-------
public | UDSession | rowcount | 25
(1 row)
```

Clearing the user-defined session parameter

From the client :

To clear the current value of rowcount, enter the following command:

```
=> ALTER SESSION CLEAR UDPARAMETER FOR UDSession rowcount;
ALTER SESSION
```

Verify that rowcount has been cleared:

```
=> SHOW SESSION UDPARAMETER all;
schema | library | key | value
--------+---------+-----+-------
(0 rows)
```

Through the UDx in C++ :

You can set the session parameter to clear through the UDx itself. For example, to clear rowcount when its value reaches 10 or greater, do the following:

1. Remove the following line from the destroy() method in the RowCount class:
   ```
   udParams.getUDSessionParamWriter("library").getStringRef("rowCount").copy(i_as_string);
   ```
2. Replace the removed line from the destroy() method with the following code:
   ```
   if (rowCount < 10)
   {
   udParams.getUDSessionParamWriter("library").getStringRef("rowCount").copy(i_as_string);
   }
   else
   {
   udParams.getUDSessionParamWriter("library").clearParameter("rowCount");
   }
   ```
3. To see the UDx clear the session parameter, set rowcount to a value of 9:
   ```
   => ALTER SESSION SET UDPARAMETER FOR UDSession rowcount = 9;
   ALTER SESSION
   ```
4. Check the value of rowcount:
   ```
   => SHOW SESSION UDPARAMETER all;
    schema |  library  |   key    | value
   --------+-----------+----------+-------
    public | UDSession | rowcount | 9
    (1 row)
   ```

5. Invoke RowCount so that its value becomes 10:

```
=> SELECT RowCount(15,15);
RowCount
----------
    30
(1 row)
```

6. Check the value of rowcount again. Because the value has reached 10, the threshold specified in the UDx, expect that rowcount is cleared:

```
=> SHOW SESSION UDPARAMETER all;
 schema | library | key | value
--------+---------+-----+-------
(0 rows)
```

As expected, RowCount is cleared.

Through the UDx in Java :

1. Remove the following lines from the destroy() method in the RowCount class:

```
udParams.getUDSessionParamWriter("library").setString("rowCount", Integer.toString(rowCount));
srvInterface.log("RowNumber processed %d records", count);
```

2. Replace the removed lines from the destroy() method with the following code:

```
if (rowCount < 10)
{
udParams.getUDSessionParamWriter("library").setString("rowCount", Integer.toString(rowCount));
srvInterface.log("RowNumber processed %d records", count);
}
else
{
udParams.getUDSessionParamWriter("library").clearParameter("rowCount");
}
```

3. To see the UDx clear the session parameter, set rowcount to a value of 9:

```
=> ALTER SESSION SET UDPARAMETER FOR UDSession rowcount = 9;
ALTER SESSION
```

4. Check the value of rowcount:

```
=> SHOW SESSION UDPARAMETER all;
 schema |  library  |   key    | value
--------+-----------+----------+-------
 public | UDSession | rowcount | 9
(1 row)
```

5. Invoke RowCount so that its value becomes 10:

```
=> SELECT RowCount(15,15);
RowCount
----------
      30
(1 row)
```

6. Check the value of rowcount. Since the value has reached 10, the threshold specified in the UDx, expect that rowcount is cleared:

```
=> SHOW SESSION UDPARAMETER all;
 schema | library | key | value
--------+---------+-----+-------
(0 rows)
```

As expected, rowcount is cleared.

Read-only and hidden session parameters

If you don't want a parameter to be set anywhere except in the UDx, you can make it read-only. If, additionally, you don't want a parameter to be visible in the client, you can make it hidden.

To make a parameter read-only, meaning that it cannot be set in the client, but can be viewed, add a single underscore before the parameter's name. For example, to make rowCount read-only, change all instances in the UDx of "rowCount" to "_rowCount".

To make a parameter hidden, meaning that it cannot be viewed in the client nor set, add two underscores before the parameter's name. For example, to make rowCount hidden, change all instances in the UDx of "rowCount" to "__rowCount".

Redacted parameters

If a parameter name meets any of the following criteria, its value is automatically redacted from logs and system tables like QUERY_REQUESTS :

- Named "secret" or "password"
- Ends with "_secret" or "_password"

See also

Kafka user-defined session parameters

C++ example: defining parameters

The following code fragment demonstrates adding a single parameter to the C++ add2ints UDSF example. The getParameterType() function defines a single integer parameter that is named constant .

```
class Add2intsWithConstantFactory : public ScalarFunctionFactory
{
    // Return an instance of Add2ints to perform the actual addition.
    virtual ScalarFunction *createScalarFunction(ServerInterface &interface)
    {
        // Calls the vt_createFuncObj to create the new Add2ints class instance.
        return vt_createFuncObj(interface.allocator, Add2intsWithConstant);
    }
    // Report the argument and return types to Vertica.
    virtual void getPrototype(ServerInterface &interface,
                              ColumnTypes &argTypes,
                              ColumnTypes &returnType)
    {
        // Takes two ints as inputs, so add ints to the argTypes object.
        argTypes.addInt();
        argTypes.addInt();
        // Returns a single int.
        returnType.addInt();
    }
    // Defines the parameters for this UDSF. Works similarly to defining arguments and return types.
    virtual void getParameterType(ServerInterface &srvInterface,
                                  SizedColumnTypes &parameterTypes)
    {
        // One int parameter named constant.
        parameterTypes.addInt("constant");
    }
};
RegisterFactory(Add2intsWithConstantFactory);
```

See the Vertica SDK entry for SizedColumnTypes for a full list of the data-type-specific functions you can call to define parameters.

The following code fragment demonstrates using the parameter value. The Add2intsWithConstant class defines a function that adds two integer values. If the user supplies it, the function also adds the value of the optional integer parameter named constant.

```
/**
 * A UDSF that adds two numbers together with a constant value.
 *
 */
class Add2intsWithConstant : public ScalarFunction
{
public:
    // Processes a block of data sent by Vertica.
    virtual void processBlock(ServerInterface &srvInterface,
                              BlockReader &arg_reader,
                              BlockWriter &res_writer)
    {
        try
            {
                // The default value for the constant parameter is 0.
                vint constant = 0;

                // Get the parameter reader from the ServerInterface to see if there are supplied parameters.
                ParamReader paramReader = srvInterface.getParamReader();
                // See if the user supplied the constant parameter.
                if (paramReader.containsParameter("constant"))
                    // There is a parameter, so get its value.
                    constant = paramReader.getIntRef("constant");
                // While we have input to process:
                do
                    {
                        // Read the two integer input parameters by calling the BlockReader.getIntRef class function.
                        const vint a = arg_reader.getIntRef(0);
                        const vint b = arg_reader.getIntRef(1);
                        // Add arguments plus constant.
                        res_writer.setInt(a+b+constant);
                        // Finish writing the row, and advance to the next output row.
                        res_writer.next();
                        // Continue looping until there are no more input rows.
                    }
                while (arg_reader.next());
            }
        catch (exception& e)
            {
                // Standard exception. Quit.
                vt_report_error(0, "Exception while processing partition: %s",
                    e.what());
            }
    }
};
```

C++ example: using session parameters

The RowCount example uses a user-defined session parameter, also called RowCount. This parameter counts the total number of rows processed by the UDx each time it runs. RowCount then displays the aggregate number of rows processed for all executions.

```
#include <string>
#include <sstream>
#include <iostream>
#include "Vertica.h"
#include "VerticaUDx.h"

using namespace Vertica;

class RowCount : public Vertica::ScalarFunction
{
private:
```

```
private:
  int rowCount;
  int count;

public:

  virtual void setup(Vertica::ServerInterface &srvInterface, const Vertica::SizedColumnTypes &argTypes) {
    ParamReader pSessionParams = srvInterface.getUDSessionParamReader("library");
    std::string rCount = pSessionParams.containsParameter("rowCount")?
      pSessionParams.getStringRef("rowCount").str(): "0";
    rowCount=atoi(rCount.c_str());

  }
  virtual void processBlock(Vertica::ServerInterface &srvInterface, Vertica::BlockReader &arg_reader, Vertica::BlockWriter &res_writer) {

    count = 0;
    if(arg_reader.getNumCols() != 2)
      vt_report_error(0, "Function only accepts two arguments, but %zu provided", arg_reader.getNumCols());

    do {
      const Vertica::vint a = arg_reader.getIntRef(0);
      const Vertica::vint b = arg_reader.getIntRef(1);
      res_writer.setInt(a+b);
      count++;
      res_writer.next();
    } while (arg_reader.next());

    srvInterface.log("count %d", count);

    }

    virtual void destroy(ServerInterface &srvInterface, const SizedColumnTypes &argTypes, SessionParamWriterMap &udParams) {
      rowCount = rowCount + count;

      std:ostringstream s;
      s << rowCount;
      const std::string i_as_string(s.str());

      udParams.getUDSessionParamWriter("library").getStringRef("rowCount").copy(i_as_string);

    }
};

class RowCountsInfo : public Vertica::ScalarFunctionFactory {
  virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface &srvInterface)
  { return Vertica::vt_createFuncObject<RowCount>(srvInterface.allocator);
  }

  virtual void getPrototype(Vertica::ServerInterface &srvInterface, Vertica::ColumnTypes &argTypes, Vertica::ColumnTypes &returnType)
  {
    argTypes.addInt();
    argTypes.addInt();
    returnType.addInt();
  }
};

RegisterFactory(RowCountsInfo);
```

Java example: defining parameters

The following code fragment demonstrates adding a single parameter to the Java add2ints UDSF example. The getParameterType() method defines a single integer parameter that is named constant.

```
package com.mycompany.example;
import com.vertica.sdk.*;
public class Add2intsWithConstantFactory extends ScalarFunctionFactory
{

  @Override
  public void getPrototype(ServerInterface srvInterface,
                           ColumnTypes argTypes,
                           ColumnTypes returnType)
  {
    argTypes.addInt();
    argTypes.addInt();
    returnType.addInt();
  }

  @Override
  public void getReturnType(ServerInterface srvInterface,
                            SizedColumnTypes argTypes,
                            SizedColumnTypes returnType)
  {
    returnType.addInt("sum");
  }

  // Defines the parameters for this UDSF. Works similarly to defining
  // arguments and return types.
  public void getParameterType(ServerInterface srvInterface,
                               SizedColumnTypes parameterTypes)
  {
    // One INTEGER parameter named constant
    parameterTypes.addInt("constant");
  }

  @Override
  public ScalarFunction createScalarFunction(ServerInterface srvInterface)
  {
    return new Add2intsWithConstant();
  }
```

See the Vertica Java SDK entry for SizedColumnTypes for a full list of the data-type-specific methods you can call to define parameters.

Java example: using session parameters

The RowCount example uses a user-defined session parameter, also called RowCount. This parameter counts the total number of rows processed by the UDx each time it runs. RowCount then displays the aggregate number of rows processed for all executions.

```
package com.mycompany.example;

import com.vertica.sdk.*;

public class RowCountFactory extends ScalarFunctionFactory {

   @Override
   public void getPrototype(ServerInterface srvInterface, ColumnTypes argTypes, ColumnTypes returnType)
   {
      argTypes.addInt();
      argTypes.addInt();
    returnType.addInt();
   }

public class RowCount extends ScalarFunction {

   private Integer count;
   private Integer rowCount;

   // In the setup method, you look for the rowCount parameter. If it doesn't exist, it is created.
   // Look in the default namespace which is "library," but it could be anything else, most likely "public" if not "library".
   @Override
      public void setup(ServerInterface srvInterface, SizedColumnTypes argTypes) {
   count = new Integer(0);
   ParamReader pSessionParams = srvInterface.getUDSessionParamReader("library");
   String rCount = pSessionParams.containsParameter("rowCount")?
   pSessionParams.getString("rowCount"): "0";
   rowCount = Integer.parseInt(rCount);

   }

   @Override
   public void processBlock(ServerInterface srvInterface, BlockReader arg_reader, BlockWriter res_writer)
      throws UdfException, DestroyInvocation {
      do {
      ++count;
      long a = arg_reader.getLong(0);
      long b = arg_reader.getLong(1);

      res_writer.setLong(a+b);
      res_writer.next();
      } while (arg_reader.next());
   }

   @Override
   public void destroy(ServerInterface srvInterface, SizedColumnTypes argTypes, SessionParamWriterMap udParams){
      rowCount = rowCount+count;
      udParams.getUDSessionParamWriter("library").setString("rowCount", Integer.toString(rowCount));
      srvInterface.log("RowNumber processed %d records", count);
      }
   }

   @Override
   public ScalarFunction createScalarFunction(ServerInterface srvInterface){
      return new RowCount();
   }
```

Errors, warnings, and logging

The SDK provides several ways for a UDx to report errors, warnings, and other messages. For a UDx written in C++ or Python, use the messaging APIs described in Sending messages . UDxs in all languages can halt execution with an error, as explained in Handling errors .

UDxs can also write messages to the Vertica log , and UDxs written in C++ can write messages to a system table.

In this section

- Sending messages
- Handling errors
- Logging

Sending messages

A UDx can handle a problem by reporting an error and terminating execution, but in some cases you might want to send a warning and proceed. For example, a UDx might ignore or use a default for an unexpected input and report that it did so. The C++ and Python messaging APIs support reporting messages at different severity levels.

A UDx has access to a ServerInterface instance. This class has the following methods for reporting messages, in order of severity:

- reportError (also terminates execution)
- reportWarning
- reportNotice
- reportInfo

Each method produces messages with the following components:

- ID code: an identification code, any integer. This code does not interact with Vertica error codes.
- Message string: a succinct description of the issue.
- Optional details string: provides more contextual information.
- Optional hint string: provides other guidance.

Duplicate messages are condensed into a single report if they have the same code and message string, even if the details and hint strings differ.

Constructing messages

The UDx should report errors immediately, usually during a process call. For all other message types, record information during processing and call the reporting methods from the UDx's destroy method. The other reporting methods do not produce output if called during processing.

The process of constructing messages is language-specific.

C++

Each ServerInterface reporting method takes a ClientMessage argument. The ClientMessage class has the following methods to set the code and message, detail, and hint:

- makeMessage: sets the ID code and message string.
- setDetail: sets the optional detail string.
- setHint: sets the optional hint string.

These method calls can be chained to simplify creating and passing the message.

All strings support printf -style arguments and formatting.

In the following example, a function records issues in processBlock and reports them in destroy :

```
class PositiveIdentity : public Vertica::ScalarFunction
{
public:
    using ScalarFunction::destroy;
    bool hitNotice = false;

    virtual void processBlock(Vertica::ServerInterface &srvInterface,
                              Vertica::BlockReader &arg_reader,
                              Vertica::BlockWriter &res_writer)
    {
        do {
            const Vertica::vint a = arg_reader.getIntRef(0);
            if (a < 0 && a != vint_null) {
                hitNotice = true;
                res_writer.setInt(null);
            } else {
                res_writer.setInt(a);
            }
            res_writer.next();
        } while (arg_reader.next());
    }

    virtual void destroy(ServerInterface &srvInterface,
                         const SizedColumnTypes &argTypes) override
    {
        if (hitNotice) {
            ClientMessage msg = ClientMessage::makeMessage(100, "Passed negative argument")
                                .setDetail("Value set to null");
            srvInterface.reportNotice(msg);
        }
    }
```

Python

Each ServerInterface reporting method has the following positional and keyword arguments:

- *idCode* : integer ID code, positional argument.
- *message* : message text, positional argument.
- *hint* : optional hint text, keyword argument.
- *detail* : optional detail text, keyword argument.

All arguments support str.format() and f-string formatting.

In the following example, a function records issues in processBlock and reports them in destroy :

```
class PositiveIdentity(vertica_sdk.ScalarFunction):
  def __init__(self):
    self.hitNotice = False

  def processBlock(self, server_interface, arg_reader, res_writer):
    while True:
      arg = arg_reader.getInt(0)
      if arg < 0 and arg is not None:
        self.hitNotice = True
        res_writer.setNull()
      else:
        res_writer.setInt(arg)
      res_writer.next()
      if not arg_reader.next():
        break

  def destroy(self, srv, argType):
    if self.hitNotice:
      srv.reportNotice(100, "Passed negative arguement", detail="Value set to null")
    return
```

API

C++

Python

Before calling a ServerInterface reporting method, construct and populate a message with the ClientMessage class.

The ServerInterface API provides the following methods for reporting messages:

```
// ClientMessage methods
template<typename... Argtypes>
static ClientMessage makeMessage(int errorcode, const char *fmt, Argtypes&&... args);

template <typename... Argtypes>
ClientMessage & setDetail(const char *fmt, Argtypes&&... args);

template <typename... Argtypes>
ClientMessage & setHint(const char *fmt, Argtypes&&... args);

// ServerInterface reporting methods
virtual void reportError(ClientMessage msg);

virtual void reportInfo(ClientMessage msg);

virtual void reportNotice(ClientMessage msg);

virtual void reportWarning(ClientMessage msg);
```

Handling errors

If your UDx encounters an unrecoverable error, it should report the error and terminate. How you do this depends on the language:

- C++: Consider using the API described in Sending messages , which is more expressive than the error-handling described in this topic. Alternatively, you can use the vt_report_error macro to report an error and exit. The macro takes two parameters: an error number and an error message string. Both the error number and message appear in the error that Vertica reports to the user. The error number is not defined by Vertica. You can use whatever value that you wish.

- Java: Instantiate and throw a UdfException , which takes a numeric code and a message string to report to the user.
- Python: Consider using the API described in Sending messages , which is more expressive than the error-handling described in this topic. Alternatively, raise an exception built into the Python language; the SDK does not include a UDx-specific exception.
- R: Use stop to halt execution with a message.

An exception or halt causes the transaction containing the function call to be rolled back.

The following examples demonstrate error-handling:

C++

Java

Python

R

The following function divides two integers. To prevent division by zero, it tests the second parameter and fails if it is zero:

```
class Div2ints : public ScalarFunction
{
public:
  virtual void processBlock(ServerInterface &srvInterface,
                            BlockReader &arg_reader,
                            BlockWriter &res_writer)
  {
    // While we have inputs to process
    do
      {
        const vint a = arg_reader.getIntRef(0);
        const vint b = arg_reader.getIntRef(1);
        if (b == 0)
          {
            vt_report_error(1,"Attempted divide by zero");
          }
        res_writer.setInt(a/b);
        res_writer.next();
      }
    while (arg_reader.next());
  }
};
```

Loading and invoking the function demonstrates how the error appears to the user. Fenced and unfenced modes use different error numbers.

```
=> CREATE LIBRARY Div2IntsLib AS '/home/dbadmin/Div2ints.so';
CREATE LIBRARY
=> CREATE FUNCTION div2ints AS LANGUAGE 'C++' NAME 'Div2intsInfo' LIBRARY Div2IntsLib;
CREATE FUNCTION
=> SELECT div2ints(25, 5);
 div2ints
----------
        5
(1 row)
=> SELECT * FROM MyTable;
 a  | b
----+---
 12 | 6
  7 | 0
 12 | 2
 18 | 9
(4 rows)
=> SELECT * FROM MyTable WHERE div2ints(a, b) > 2;
ERROR 3399:  Error in calling processBlock() for User Defined Scalar Function
div2ints at Div2ints.cpp:21, error code: 1, message: Attempted divide by zero
```

To report additional diagnostic information about the error, you can write messages to a log file before throwing the exception (see Logging).

Your UDx must not consume exceptions that it did not throw. Intercepting server exceptions can lead to database instability.

Logging

Each UDx written in C++, Java, or Python has an associated instance of ServerInterface . The ServerInterface class provides a function to write to the Vertica log, and the C++ implementation also provides a function to log events in a system table.

Writing messages to the Vertica log

You can write to log files using the ServerInterface.log() function. The function acts similarly to printf() , taking a formatted string and an optional set of values and writing the string to the log file. Where the message is written depends on whether your function runs in fenced mode or unfenced mode:

- Functions running in unfenced mode write their messages into the vertica.log file in the catalog directory.
- Functions running in fenced mode write their messages into a log file named UDxLogs/UDxFencedProcesses.log in the catalog directory.

To help identify your function's output, Vertica adds the SQL function name bound to your UDx to the log message.

The following example logs a UDx's input values:

C++

Java

Python

```
virtual void processBlock(ServerInterface &srvInterface,
                          BlockReader &argReader,
                          BlockWriter &resWriter)
{
    try {
        // While we have inputs to process
        do {
            if (argReader.isNull(0) || argReader.isNull(1)) {
                resWriter.setNull();
            } else {
                const vint a = argReader.getIntRef(0);
                const vint b = argReader.getIntRef(1);
                srvInterface.log("got a: %d and b: %d", (int) a, (int) b);
                resWriter.setInt(a+b);
            }
            resWriter.next();
        } while (argReader.next());
    } catch(std::exception& e) {
        // Standard exception. Quit.
        vt_report_error(0, "Exception while processing block: [%s]", e.what());
    }
}
```

The log() function generates entries in the log file like the following:

```
$ tail /home/dbadmin/py_db/v_py_db_node0001_catalog/UDxLogs/UDxFencedProcesses.log
 07:52:12.862 [Python-v_py_db_node0001-7524:0x206c-40575]  0x7f70eee2f780 PythonExecContext::processBlock
07:52:12.862 [Python-v_py_db_node0001-7524:0x206c-40575]  0x7f70eee2f780 [UserMessage] add2ints - Python UDx - Adding 2 ints!
07:52:12.862 [Python-v_py_db_node0001-7524:0x206c-40575]  0x7f70eee2f780 [UserMessage] add2ints - Values: first_int is 100 second_int is 100
```

For details on viewing the Vertica log files, see Monitoring log files .

Writing messages to the UDX_EVENTS table (C++ only)

In the C++ API, you can write messages to the UDX_EVENTS system table instead of or in addition to writing to the log. Writing to a system table allows you to collect events from all nodes in one place.

You can write to this table using the ServerInterface.logEvent() function. The function takes one argument, a map. The map is written into the __RAW__ column of the table as a Flex VMap. The following example shows how the Parquet exporter creates and logs this map.

```
// Log exported parquet file details to v_monitor.udx_events
std::map<std::string, std::string> details;
details["file"] = escapedPath;
details["created"] = create_timestamp_;
details["closed"] = close_timestamp_;
details["rows"] = std::to_string(num_rows_in_file);
details["row_groups"] = std::to_string(num_row_groups_in_file);
details["size_mb"] = std::to_string((double)outputStream->Tell()/(1024*1024));
srvInterface.logEvent(details);
```

You can select individual fields from the VMap as in the following example.

```
=> SELECT __RAW__['file'] FROM UDX_EVENTS;
                                __RAW__
------------------------------------------------------------------------------
 /tmp/export_tmpzLkrKq3a/450c4213-v_vmart_node0001-139770732459776-0.parquet
 /tmp/export_tmpzLkrKq3a/9df1c797-v_vmart_node0001-139770860660480-0.parquet
(2 rows)
```

Alternatively, you can define a view to make it easier to query fields directly, as columns. See Monitoring exports for an example.

Handling cancel requests

Users of your UDx might cancel the operation while it is running. How Vertica handles the cancellation of the query and your UDx depends on whether your UDx is running in fenced or unfenced mode:

- If your UDx is running in unfenced mode, Vertica either stops the function when it requests a new block of input or output, or waits until your function completes running and discards the results.
- If your UDx is running in Fenced and unfenced modes , Vertica kills the zygote process that is running your function if it continues processing past a timeout.

In addition, you can implement the cancel() method in any UDx to perform any necessary additional work. Vertica calls your function when a query is canceled. This cancellation can occur at any time during your UDx's lifetime, from setup() through destroy() .

You can check for cancellation before starting an expensive operation by calling isCanceled() .

In this section

- Implementing the cancel callback
- Checking for cancellation during execution
- C++ example: cancelable UDSource

Implementing the cancel callback

Your UDx can implement a cancel() callback function. Vertica calls this function if the query that invoked the UDx has been canceled.

You usually implement this function to perform an orderly shutdown of any additional processing that your UDx spawned. For example, you can have your cancel() function shut down threads that your UDx has spawned or signal a third-party library that it needs to stop processing and exit. Your cancel() function should leave your UDx's function class ready to be destroyed, because Vertica calls the UDx's destroy() function after the cancel() function has exited.

A UDx's default cancel() behavior is to do nothing.

The contract for cancel() is:

- Vertica will call cancel() at most once per UDx instance.
- Vertica can call cancel() concurrently with any other method of the UDx object except the constructor and destructor.
- Vertica can call cancel() from another thread, so implementations should be thread-safe.
- Vertica will call cancel() for either an explicit user cancellation or an error in the query.
- Vertica does not guarantee that cancel() will run to completion. Long-running cancellations might be aborted.

The call to cancel() is not synchronized in any way with your UDx's other functions. If you need your processing function to exit before your cancel() function performs some action (killing threads, for example), you must have the two function synchronize their actions.

Vertica always calls destroy() if it called setup() . Cancellation does not prevent destruction.

See C++ example: cancelable UDSource for an example that implements cancel() .

Checking for cancellation during execution

You can call the isCanceled() method to check for user cancellation. Typically you check for cancellation from the method that does the main processing in your UDx before beginning expensive operations. If isCanceled() returns true, the query has been canceled and your method should exit immediately to prevent it from wasting CPU time. If your UDx is not running fenced mode, Vertica cannot halt your function and has to wait for it to finish. If it is running in fenced mode, Vertica eventually kills the side process running it.

See C++ example: cancelable UDSource for an example that uses isCanceled() .

C++ example: cancelable UDSource

The FifoSource example, found in filelib.cpp in the SDK examples, demonstrates use of cancel() and isCanceled() . This source reads from a named pipe. Unlike reads from files, reads from pipes can block. Therefore, we need to be able to cancel a load from this source.

To manage cancellation, the UDx uses a pipe , a data channel used for inter-process communication. A process can write data to the write end of the pipe, and it remains available until another process reads it from the read end of the pipe. This example doesn't pass data through this pipe; rather, it uses the pipe to manage cancellation, as explained further below. In addition to the pipe's two file descriptors (one for each end), the UDx creates a file descriptor for the file to read from. The setup() function creates the pipe and then opens the file.

```
virtual void setup(ServerInterface &srvInterface) {
 // cancelPipe is a pipe used only for checking cancellation
 if (pipe(cancelPipe)) {
  vt_report_error(0, "Error opening control structure");
 }

 // handle to the named pipe from which we read data
 namedPipeFd = open(filename.c_str(), O_RDONLY | O_NONBLOCK);
 if (namedPipeFd < 0) {
  vt_report_error(0, "Error opening fifo [%s]", filename.c_str());
 }
}
```

We now have three file descriptors: `namedPipeFd` , `cancelPipe[PIPE_READ]` , and `cancelPipe[PIPE_WRITE]` . Each of these must eventually be closed.

This UDx uses the `poll()` system call to wait either for data to arrive from the named pipe (`namedPipeFd`) or for a cancellation (`cancelPipe[PIPE_READ]`). The `process()` function polls, checks for results, checks for cancellation, writes output if needed, and returns.

```
virtual StreamState process(ServerInterface &srvInterface, DataBuffer &output) {
 struct pollfd pollfds[2] = {
  { namedPipeFd,          POLLIN, 0 },
  { cancelPipe[PIPE_READ], POLLIN, 0 }
 };

 if (poll(pollfds, 2, -1) < 0) {
  vt_report_error(1, "Error reading [%s]", filename.c_str());
 }

 if (pollfds[1].revents & (POLLIN | POLLHUP)) {
  /* This can only happen after cancel() has been called */
  VIAssert(isCanceled());
  return DONE;
 }

 VIAssert(pollfds[PIPE_READ].revents & (POLLIN | POLLHUP));

 const ssize_t amount = read(namedPipeFd, output.buf + output.offset, output.size - output.offset);
 if (amount < 0) {
  vt_report_error(1, "Error reading from fifo [%s]", filename.c_str());
 }

 if (amount == 0 || isCanceled()) {
  return DONE;
 } else {
  output.offset += amount;
  return OUTPUT_NEEDED;
 }
}
```

If the query is canceled, the `cancel()` function closes the write end of the pipe. The next time `process()` polls for input, it finds no input on the read end of the pipe and exits. Otherwise, it continues. The function also calls `isCanceled()` to check for cancellation before returning `OUTPUT_NEEDED` , the signal that it has filled its buffer and is waiting for it to be processed downstream.

The `cancel()` function does only the work needed to interrupt a call to `process()` . Cleanup that is always needed, not just for cancellation, is instead done in `destroy()` or the destructor. The `cancel()` function closes the write end of the pipe. (The helper function will be shown later.)

```
virtual void cancel(ServerInterface &srvInterface) {
 closeIfNeeded(cancelPipe[PIPE_WRITE]);
```

It is not safe to close the named pipe in cancel() , because closing it could create a race condition if another process (like another query) were to reuse the file descriptor number for a new descriptor before the UDx finishes. Instead we close it, and the read end of the pipe, in destroy() .

```
virtual void destroy(ServerInterface &srvInterface) {
 closeIfNeeded(namedPipeFd);
 closeIfNeeded(cancelPipe[PIPE_READ]);
```

It is not safe to close the write end of the pipe in destroy() , because cancel() closes it and can be called concurrently with destroy() . Therefore, we close it in the destructor.

```
~FifoSource() {
 closeIfNeeded(cancelPipe[PIPE_WRITE]);
```

The UDx uses a helper function, closeIfNeeded() , to make sure each file descriptor is closed exactly once.

```
void closeIfNeeded(int &fd) {
 if (fd >= 0) {
  close(fd);
  fd = -1;
 }
```

Aggregate functions (UDAFs)

Aggregate functions perform an operation on a set of values and return one value. Vertica provides standard built-in aggregate functions such as AVG , MAX , and MIN . User-defined aggregate functions (UDAFs) provide similar functionality:

- Support a single input column (or set) of values and provide a single output column.
- Support RLE decompression. RLE input is decompressed before it is sent to a UDAF.
- Support use with GROUP BY and HAVING clauses. Only columns appearing in the GROUP BY clause can be selected.

Restrictions

The following restrictions apply to UDAFs:

- Available for C++ only.
- Cannot be run in fenced mode.
- Cannot be used with correlated subqueries.

In this section

- AggregateFunction class
- AggregateFunctionFactory class
- UDAF performance in statements containing a GROUP BY clause
- C++ example: average

AggregateFunction class

The AggregateFunction class performs the aggregation. It computes values on each database node where relevant data is stored and then combines the results from the nodes. You must implement the following methods:

- initAggregate() - Initializes the class, defines variables, and sets the starting value for the variables. This function must be idempotent.
- aggregate() - The main aggregation operation, executed on each node.
- combine() - If multiple invocations of aggregate() are needed, Vertica calls combine() to combine all the sub-aggregations into a final aggregation. Although this method might not be called, you must define it.
- terminate() - Terminates the function and returns the result as a column.

Important

The aggregate() function might not operate on the complete input set all at once. For this reason, initAggregate() must be idempotent.

The AggregateFunction class also provides optional methods that you can implement to allocate and free resources: setup() and destroy() . You should use these methods to allocate and deallocate resources that you do not allocate through the UDAF API (see Allocating resources for UDxs for details).

API

Aggregate functions are supported for C++ only.

The AggregateFunction API provides the following methods for extension by subclasses:

```
virtual void setup(ServerInterface &srvInterface,
    const SizedColumnTypes &argTypes);

virtual void initAggregate(ServerInterface &srvInterface, IntermediateAggs &aggs)=0;

void aggregate(ServerInterface &srvInterface, BlockReader &arg_reader,
    IntermediateAggs &aggs);

virtual void combine(ServerInterface &srvInterface, IntermediateAggs &aggs_output,
    MultipleIntermediateAggs &aggs_other)=0;

virtual void terminate(ServerInterface &srvInterface, BlockWriter &res_writer,
    IntermediateAggs &aggs);

virtual void cancel(ServerInterface &srvInterface);

virtual void destroy(ServerInterface &srvInterface, const SizedColumnTypes &argTypes);
```

AggregateFunctionFactory class

The AggregateFunctionFactory class specifies metadata information such as the argument and return types of your aggregate function. It also instantiates your AggregateFunction subclass. Your subclass must implement the following methods:

- getPrototype() - Defines the number of parameters and data types accepted by the function. There is a single parameter for aggregate functions.
- getIntermediateTypes() - Defines the intermediate variable(s) used by the function. These variables are used when combining the results of aggregate() calls.
- getReturnType() - Defines the type of the output column.

Your function may also implement getParameterType() , which defines the names and types of parameters that this function uses.

Vertica uses this data when you call the CREATE AGGREGATE FUNCTION SQL statement to add the function to the database catalog.

API

Aggregate functions are supported for C++ only.

The AggregateFunctionFactory API provides the following methods for extension by subclasses:

```
virtual AggregateFunction *
    createAggregateFunction ServerInterface &srvInterface)=0;

virtual void getPrototype(ServerInterface &srvInterface,
    ColumnTypes &argTypes, ColumnTypes &returnType)=0;

virtual void getIntermediateTypes(ServerInterface &srvInterface,
    const SizedColumnTypes &inputTypes, SizedColumnTypes &intermediateTypeMetaData)=0;

virtual void getReturnType(ServerInterface &srvInterface,
    const SizedColumnTypes &argTypes, SizedColumnTypes &returnType)=0;

virtual void getParameterType(ServerInterface &srvInterface,
    SizedColumnTypes &parameterTypes);
```

UDAF performance in statements containing a GROUP BY clause

You may see slower-than-expected performance from your UDAF if the SQL statement calling it also contains a GROUP BY clause . For example:

```
=> SELECT a, MYUDAF(b) FROM sampletable GROUP BY a;
```

In statements like this one, Vertica does not consolidate row data together before calling your UDAF's aggregate() method. Instead, it calls aggregate() once for each row of data. Usually, the overhead of having Vertica consolidate the row data is greater than the overhead of calling aggregate() for each row of data. However, if your UDAF's aggregate() method has significant overhead, then you might notice an impact on your UDAF's performance.

For example, suppose aggregate() allocates memory. When called in a statement with a GROUP BY clause, it performs this memory allocation for each row of data. Because memory allocation is a relatively expensive process, this allocation can impact the overall performance of your UDAF and the query.

There are two ways you can address UDAF performance in a statement containing a GROUP BY clause:

- Reduce the overhead of each call to aggregate() . If possible, move any allocation or other setup operations to the UDAF's setup() function.
- Declare a special parameter that tells Vertica to group row data together when calling a UDAF. This technique is explained below.

Using the _minimizeCallCount parameter

Your UDAF can tell Vertica to always batch row data together to reduce the number of calls to its aggregate() method. To trigger this behavior, your UDAF must declare an integer parameter named _minimizeCallCount . You do not need to set a value for this parameter in your SQL statement. The fact that your UDAF declares this parameter triggers Vertica to group row data together when calling aggregate() .

You declare the _minimizeCallCount parameter the same way you declare other UDx parameters. See UDx parameters for more information.

Important

Always test the performance of your UDAF before and after implementing the _minimizeCallCount parameter to ensure that it improves performance. You might find that the overhead of having Vertica group row data for your UDAF is greater than the cost of the repeated calls to aggregate() .

C++ example: average

The Average aggregate function created in this example computes the average of values in a column.

You can find the source code used in this example on the Vertica GitHub page .

Loading the example

Use CREATE LIBRARY and CREATE AGGREGATE FUNCTION to declare the function:

```
=> CREATE LIBRARY AggregateFunctions AS
'/opt/vertica/sdk/examples/build/AggregateFunctions.so';
CREATE LIBRARY
=> CREATE aggregate function ag_avg AS LANGUAGE 'C++'
name 'AverageFactory' library AggregateFunctions;
CREATE AGGREGATE FUNCTION
```

Using the example

Use the function as part of a SELECT statement:

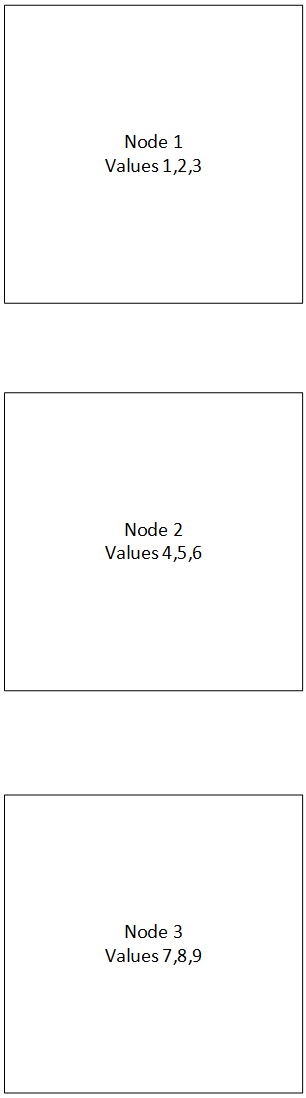
```
=> SELECT * FROM average;
id | count
----+---------
A  |     8
B  |     3
C  |     6
D  |     2
E  |     9
F  |     7
G  |     5
H  |     4
I  |     1
(9 rows)
=> SELECT ag_avg(count) FROM average;
ag_avg
--------
  5
(1 row)
```

AggregateFunction implementation

This example adds the input argument values in the aggregate() method and keeps a counter of the number of values added. The server runs aggregate() on every node and different data chunks, and combines all the individually added values and counters in the combine() method. Finally, the average value is computed in the terminate() method by dividing the total sum by the total number of values processed.

For this discussion, assume the following environment:

- A three-node Vertica cluster
- A table column that contains nine values that are evenly distributed across the nodes. Schematically, the nodes look like the following figure:



The function uses sum and count variables. *Sum* contains the sum of the values, and *count* contains the count of values.

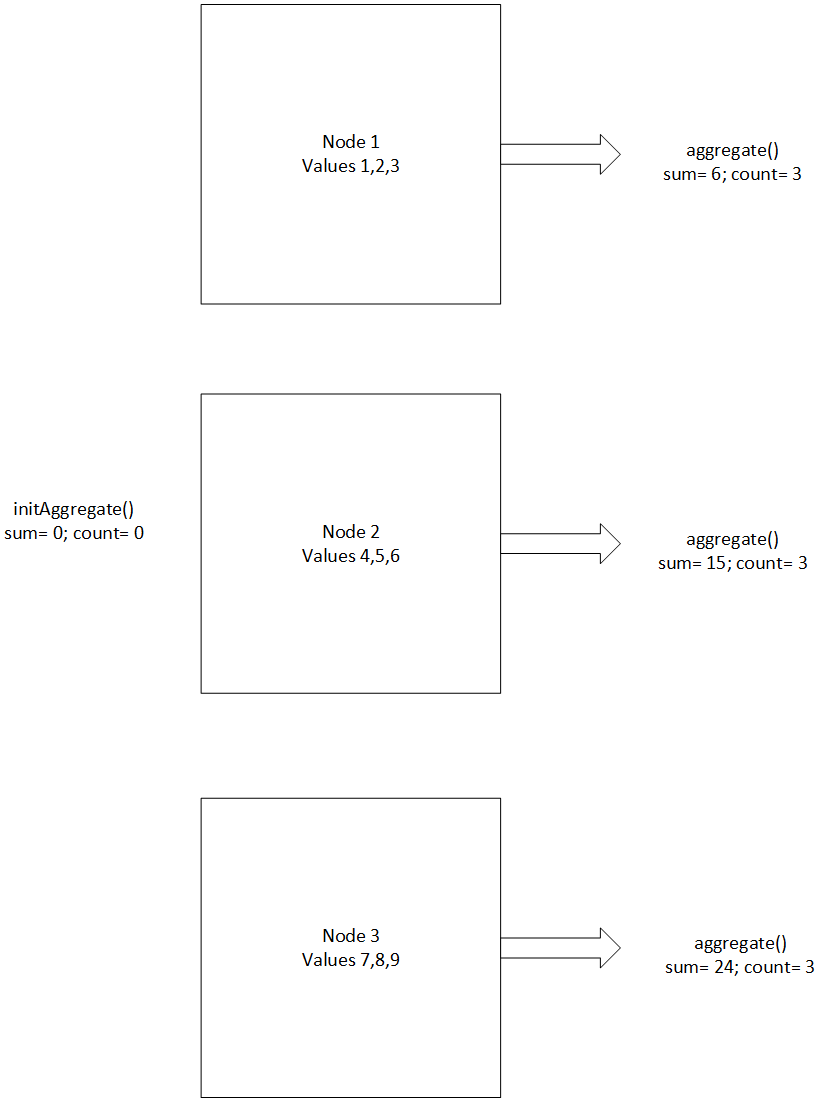
First, initAggregate() initializes the variables and sets their values to zero.

```
virtual void initAggregate(ServerInterface &srvInterface,
                           IntermediateAggs &aggs)
{
 try {
  VNumeric &sum = aggs.getNumericRef(0);
  sum.setZero();

  vint &count = aggs.getIntRef(1);
  count = 0;
 }
 catch(std::exception &e) {
  vt_ report_ error(0, "Exception while initializing intermediate aggregates: [% s]", e.what());
 }
}
```

The aggregate() function reads the block of data on each node and calculates partial aggregates.

```
void aggregate(ServerInterface &srvInterface,
               BlockReader &argReader,
               IntermediateAggs &aggs)
{
  try {
      VNumeric &sum = aggs.getNumericRef(0);
      vint     &count = aggs.getIntRef(1);
      do {
          const VNumeric &input = argReader.getNumericRef(0);
          if (!input.isNull()) {
           sum.accumulate(&input);
       count++;
          }
      } while (argReader.next());
  } catch(std::exception &e) {
    vt_report_ error(0, " Exception while processing aggregate: [%s]", e.what());
  }
```

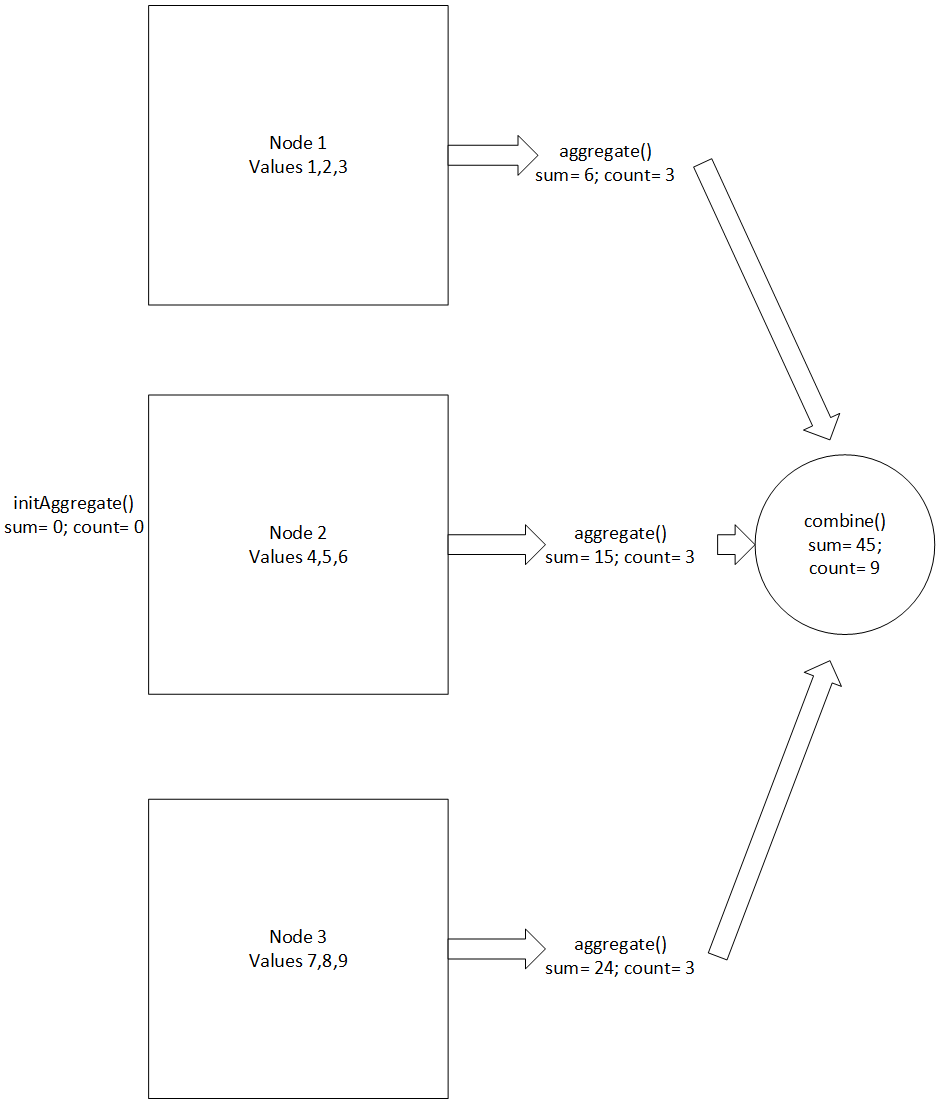
Each completed instance of the aggregate() function returns multiple partial aggregates for sum and count. The following figure illustrates this process using the aggregate() function:



The combine() function puts together the partial aggregates calculated by each instance of the average function.

```
virtual void combine(ServerInterface &srvInterface,
                     IntermediateAggs &aggs,
                     MultipleIntermediateAggs &aggsOther)
{
  try {
    VNumeric      &mySum      = aggs.getNumericRef(0);
    vint          &myCount    = aggs.getIntRef(1);

    // Combine all the other intermediate aggregates
    do {
      const VNumeric &otherSum   = aggsOther.getNumericRef(0);
      const vint     &otherCount = aggsOther.getIntRef(1);

      // Do the actual accumulation
      mySum.accumulate(&otherSum);
      myCount += otherCount;

    } while (aggsOther.next());
  } catch(std::exception &e) {
    // Standard exception. Quit.
    vt_report_error(0, "Exception while combining intermediate aggregates: [%s]", e.what());
  }
```

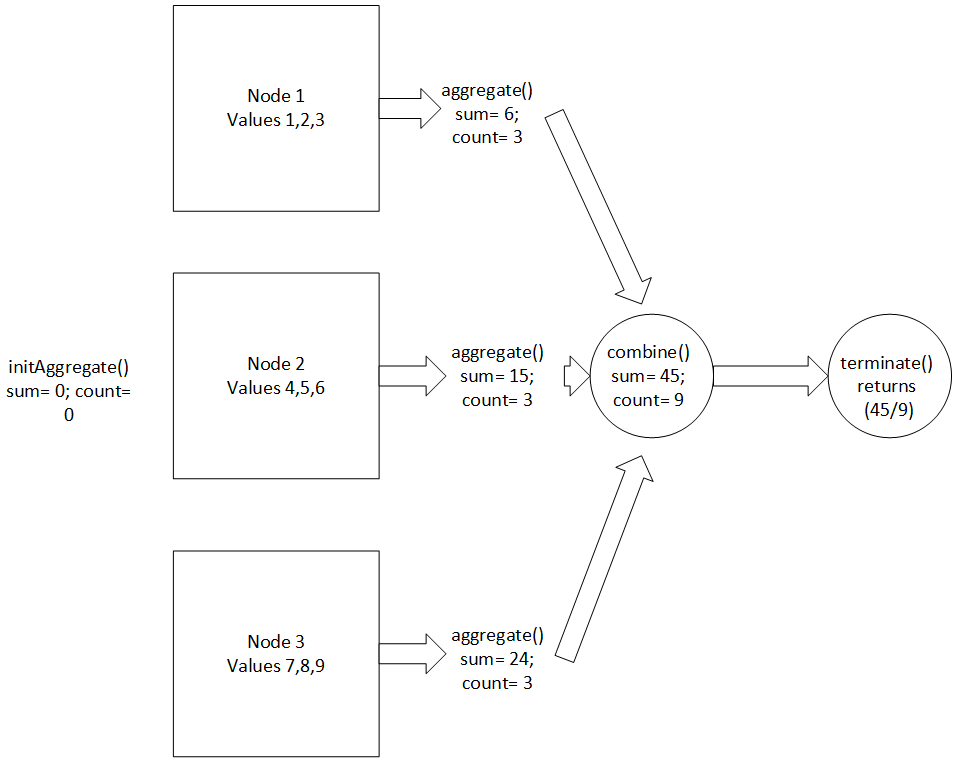
The following figure shows how each partial aggregate is combined:



After all input has been evaluated by the aggregate() function Vertica calls the terminate() function. It returns the average to the caller.

```
virtual void terminate(ServerInterface &srvInterface,
                       BlockWriter &resWriter,
                       IntermediateAggs &aggs)
{
  try {
    const int32 MAX_INT_PRECISION = 20;
    const int32 prec = Basics::getNumericWordCount(MAX_INT_PRECISION);
    uint64 words[prec];
    VNumeric count(words,prec,0/*scale*/);
    count.copy(aggs.getIntRef(1));
    VNumeric &out = resWriter.getNumericRef();
    if (count.isZero()) {
      out.setNull();
    } else
      const VNumeric &sum = aggs.getNumericRef(0);
      out.div(&sum, &count);
  }
```

The following figure shows the implementation of the terminate() function:



AggregateFunctionFactory implementation

The getPrototype() function allows you to define the variables that are sent to your aggregate function and returned to Vertica after your aggregate function runs. The following example accepts and returns a numeric value:

```
virtual void getPrototype(ServerInterface &srvInterface,
                          ColumnTypes &argTypes,
                          ColumnTypes &returnType)
{
  argTypes.addNumeric();
  returnType.addNumeric();
}
```

The getIntermediateTypes() function defines any intermediate variables that you use in your aggregate function. *Intermediate variables* are values used to pass data among multiple invocations of an aggregate function. They are used to combine results until a final result can be computed. In this example, there are two results - total (numeric) and count (int).

```
virtual void getIntermediateTypes(ServerInterface &srvInterface,
                                  const SizedColumnTypes &inputTypes,
                                  SizedColumnTypes &intermediateTypeMetaData)
{
    const VerticaType &inType = inputTypes.getColumnType(0);
    intermediateTypeMetaData.addNumeric(interPrec, inType.getNumericScale());
    intermediateTypeMetaData.addInt();
}
```

The getReturnType() function defines the output data type:

```
virtual void getReturnType(ServerInterface &srvfloaterface,
                           const SizedColumnTypes &inputTypes,
                           SizedColumnTypes &outputTypes)
{
    const VerticaType &inType = inputTypes.getColumnType(0);
    outputTypes.addNumeric(inType.getNumericPrecision(),
    inType.getNumericScale());
}
```

Analytic functions (UDAnFs)

User-defined analytic functions (UDAnFs) are used for analytics. See SQL analytics for an overview of Vertica's built-in analytics. Like user-defined scalar functions (UDSFs), UDAnFs must output a single value for each row of data read and can have no more than 9800 arguments.

Unlike UDSFs, the UDAnF's input reader and output reader can be advanced independently. This feature lets you create analytic functions where the output value is calculated over multiple rows of data. By advancing the reader and writer independently, you can create functions similar to the built-in analytic functions such as LAG , which uses data from prior rows to output a value for the current row.

In this section

- AnalyticFunction class
- AnalyticFunctionFactory class
- C++ example: rank

AnalyticFunction class

The AnalyticFunction class performs the analytic processing. Your subclass must define the processPartition() method to perform the operation. It may define methods to set up and tear down the function.

Performing the operation

The processPartition() method reads a partition of data, performs some sort of processing, and outputs a single value for each input row.

Vertica calls processPartition() once for each partition of data. It supplies the partition using an AnalyticPartitionReader object from which you read its input data. In addition, there is a unique method on this object named isNewOrderByKey() , which returns a Boolean value indicating whether your function has seen a row with the same ORDER BY key (or keys). This method is very useful for analytic functions (such as the example RANK function) which need to handle rows with identical ORDER BY keys differently than rows with different ORDER BY keys.

> **Note**
> You can specify multiple ORDER BY columns in the SQL query you use to call your UDAnF. The isNewOrderByKey method returns true if any of the ORDER BY keys are different than the previous row.

Once your method has finished processing the row of data, you advance it to the next row of input by calling next() on AnalyticPartitionReader .

Your method writes its output value using an AnalyticPartitionWriter object that Vertica supplies as a parameter to processPartition() . This object has data-type-specific methods to write the output value (such as setInt()). After setting the output value, call next() on AnalyticPartitionWriter to advance to the next row in the output.

> **Note**
> You must be sure that your function produces a row of output for each row of input in the partition. You must also not output more rows than are in the partition, otherwise the zygote size process (if running in Fenced and unfenced modes) or Vertica itself could generate an out of

bounds error.

Setting up and tearing down

The AnalyticFunction class defines two additional methods that you can optionally implement to allocate and free resources: setup() and destroy() . You should use these methods to allocate and deallocate resources that you do not allocate through the UDx API (see Allocating resources for UDxs for details).

API

C++

Java

The AnalyticFunction API provides the following methods for extension by subclasses:

```
virtual void setup(ServerInterface &srvInterface,
    const SizedColumnTypes &argTypes);

virtual void processPartition (ServerInterface &srvInterface,
    AnalyticPartitionReader &input_reader,
    AnalyticPartitionWriter &output_writer)=0;

virtual void cancel(ServerInterface &srvInterface);

virtual void destroy(ServerInterface &srvInterface, const SizedColumnTypes &argTypes);
```

AnalyticFunctionFactory class

The AnalyticFunctionFactory class tells Vertica metadata about your UDAnF: its number of parameters and their data types, as well as the data type of its return value. It also instantiates a subclass of AnalyticFunction .

Your AnalyticFunctionFactory subclass must implement the following methods:

- getPrototype() describes the input parameters and output value of your function. You set these values by calling functions on two ColumnTypes objects that are passed to your method.
- createAnalyticFunction() supplies an instance of your AnalyticFunction that Vertica can call to process a UDAnF function call.
- getReturnType() provides details about your function's output. This method is where you set the width of the output value if your function returns a variable-width value (such as VARCHAR) or the precision of the output value if it has a settable precision (such as TIMESTAMP).

API

C++

Java

The AnalyticFunctionFactory API provides the following methods for extension by subclasses:

```
virtual AnalyticFunction * createAnalyticFunction (ServerInterface &srvInterface)=0;

virtual void getPrototype(ServerInterface &srvInterface,
    ColumnTypes &argTypes, ColumnTypes &returnType)=0;

virtual void getReturnType(ServerInterface &srvInterface,
    const SizedColumnTypes &argTypes, SizedColumnTypes &returnType)=0;

virtual void getParameterType(ServerInterface &srvInterface,
    SizedColumnTypes &parameterTypes);
```

C++ example: rank

The Rank analytic function ranks rows based on how they are ordered. A Java version of this UDx is included in /opt/vertica/sdk/examples .

Loading and using the example

The following example shows how to load the function into Vertica. It assumes that the AnalyticFunctions.so library that contains the function has been copied to the dbadmin user's home directory on the initiator node.

```
=> CREATE LIBRARY AnalyticFunctions AS '/home/dbadmin/AnalyticFunctions.so';
CREATE LIBRARY
=> CREATE ANALYTIC FUNCTION an_rank AS LANGUAGE 'C++'
   NAME 'RankFactory' LIBRARY AnalyticFunctions;
CREATE ANALYTIC FUNCTION
```

An example of running this rank function, named an_rank , is:

```
=> SELECT * FROM hits;
     site       |    date    | num_hits
----------------+------------+----------
 www.example.com | 2012-01-02 |       97
 www.vertica.com | 2012-01-01 |   343435
 www.example.com | 2012-01-01 |      123
 www.example.com | 2012-01-04 |      112
 www.vertica.com | 2012-01-02 |   503695
 www.vertica.com | 2012-01-03 |   490387
 www.example.com | 2012-01-03 |      123
(7 rows)
=> SELECT site,date,num_hits,an_rank()
   OVER (PARTITION BY site ORDER BY num_hits DESC)
   AS an_rank FROM hits;
     site       |    date    | num_hits | an_rank
----------------+------------+----------+---------
 www.example.com | 2012-01-03 |      123 |       1
 www.example.com | 2012-01-01 |      123 |       1
 www.example.com | 2012-01-04 |      112 |       3
 www.example.com | 2012-01-02 |       97 |       4
 www.vertica.com | 2012-01-02 |   503695 |       1
 www.vertica.com | 2012-01-03 |   490387 |       2
 www.vertica.com | 2012-01-01 |   343435 |       3
(7 rows)
```

As with the built-in RANK analytic function, rows that have the same value for the ORDER BY column (num_hits in this example) have the same rank, but the rank continues to increase, so that the next row that has a different ORDER BY key gets a rank value based on the number of rows that preceded it.

AnalyticFunction implementation

The following code defines an AnalyticFunction subclass named Rank . It is based on example code distributed in the examples directory of the SDK.

```
/**
 * User-defined analytic function: Rank - works mostly the same as SQL-99 rank
 * with the ability to define as many order by columns as desired
 *
 */
class Rank : public AnalyticFunction
{
  virtual void processPartition(ServerInterface &srvInterface,
                                AnalyticPartitionReader &inputReader,
                                AnalyticPartitionWriter &outputWriter)
  {
    // Always use a top-level try-catch block to prevent exceptions from
    // leaking back to Vertica or the fenced-mode side process.
    try {
      rank = 1; // The rank to assign a row
      rowCount = 0; // Number of rows processed so far
      do {
        rowCount++;
        // Do we have a new order by row?
        if (inputReader.isNewOrderByKey()) {
          // Yes, so set rank to the total number of rows that have been
          // processed. Otherwise, the rank remains the same value as
          // the previous iteration.
          rank = rowCount;
        }
        // Write the rank
        outputWriter.setInt(0, rank);
        // Move to the next row of the output
        outputWriter.next();
      } while (inputReader.next()); // Loop until no more input
    } catch(exception& e) {
      // Standard exception. Quit.
      vt_report_error(0, "Exception while processing partition: %s", e.what());
    }
  }
private:
  vint rank, rowCount;
};
```

In this example, the processPartition() method does not actually read any of the data from the input row; it just advances through the rows. It does not need to read data; it just counts the rows that have been read and determine whether those rows have the same ORDER BY key as the previous row. If the current row is a new ORDER BY key, then the rank is set to the total number of rows that have been processed. If the current row has the same ORDER BY value as the previous row, then the rank remains the same.

Note that the function has a top-level try-catch block. All of your UDx functions should always have one to prevent stray exceptions from being passed back to Vertica (if you run the function unfenced) or the side process.

AnalyticFunctionFactory implementation

The following code defines the AnalyticFunctionFactory that corresponds with the Rank analytic function.

```
class RankFactory : public AnalyticFunctionFactory
{
  virtual void getPrototype(ServerInterface &srvInterface,
                            ColumnTypes &argTypes, ColumnTypes &returnType)
  {
    returnType.addInt();
  }
  virtual void getReturnType(ServerInterface &srvInterface,
                             const SizedColumnTypes &inputTypes,
                             SizedColumnTypes &outputTypes)
  {
    outputTypes.addInt();
  }
  virtual AnalyticFunction *createAnalyticFunction(ServerInterface
                                                   &srvInterface)
  { return vt_createFuncObj(srvInterface.allocator, Rank); }
};
```

The first method defined by the RankFactory subclass, getPrototype() , sets the data type of the return value. Because the Rank UDAnF does not read input, it does not define any arguments by calling methods on the ColumnTypes object passed in the argTypes parameter.

The next method is getReturnType() . If your function returns a data type that needs to define a width or precision, your implementation of the getReturnType() method calls a method on the SizedColumnType object passed in as a parameter to tell Vertica the width or precision. Rank returns a fixed-width data type (an INTEGER) so it does not need to set the precision or width of its output; it just calls addInt() to report its output data type.

Finally, RankFactory defines the createAnalyticFunction() method that returns an instance of the AnalyticFunction class that Vertica can call. This code is mostly boilerplate. All you need to do is add the name of your analytic function class in the call to vt_createFuncObj() , which takes care of allocating the object for you.

Scalar functions (UDSFs)

A user-defined scalar function (UDSF) returns a single value for each row of data it reads. You can use a UDSF anywhere you can use a built-in Vertica function. You usually develop a UDSF to perform data manipulations that are too complex or too slow to perform using SQL statements and functions. UDSFs also let you use analytic functions provided by third-party libraries within Vertica while still maintaining high performance.

A UDSF returns a single column. You can automatically return multiple values in a ROW . A ROW is a group of property-value pairs. In the following example, div_with_rem is a UDSF that performs a division operation, returning the quotient and remainder as integers:

```
=> SELECT div_with_rem(18,5);
      div_with_rem
------------------------------
 {"quotient":3,"remainder":3}
(1 row)
```

A ROW returned from a UDSF cannot be used as an argument to COUNT.

Alternatively, you can construct a complex return value yourself, as described in Complex Types as Arguments .

Your UDSF must return a value for every input row (unless it generates an error; see Handling errors for details). Failure to return a value for an input row results in incorrect results and potentially destabilizes the Vertica server if not run in Fenced and unfenced modes .

A UDSF can have up to 9800 arguments.

In this section

- ScalarFunction class
- ScalarFunctionFactory class
- Setting null input and volatility behavior
- Improving query performance (C++ only)
- C++ example: Add2Ints
- Python example: currency_convert
- Python example: validate_url
- Python example: matrix multiplication
- R example: SalesTaxCalculator

- R example: kmeans
- C++ example: using complex types
- C++ example: returning multiple values
- C++ example: calling a UDSF from a check constraint

ScalarFunction class

The ScalarFunction class is the heart of a UDSF. Your subclass must define the processBlock() method to perform the scalar operation. It may define methods to set up and tear down the function.

For scalar functions written in C++, you can provide information that can help with query optimization. See Improving query performance (C++ only) .

Performing the operation

The processBlock() method carries out all of the processing that you want your UDSF to perform. When a user calls your function in a SQL statement, Vertica bundles together the data from the function parameters and passes it to processBlock() .

The input and output of the processBlock() method are supplied by objects of the BlockReader and BlockWriter classes. They define methods that you use to read the input data and write the output data for your UDSF.

The majority of the work in developing a UDSF is writing processBlock() . This is where all of the processing in your function occurs. Your UDSF should follow this basic pattern:

- Read in a set of arguments from the BlockReader object using data-type-specific methods.
- Process the data in some manner.
- Output the resulting value using one of the BlockWriter class's data-type-specific methods.
- Advance to the next row of output and input by calling BlockWriter.next() and BlockReader.next() .

This process continues until there are no more rows of data to be read (BlockReader.next() returns false).

You must make sure that processBlock() reads all of the rows in its input and outputs a single value for each row. Failure to do so can corrupt the data structures that Vertica reads to get the output of your UDSF. The only exception to this rule is if your processBlock() function reports an error back to Vertica (see Handling errors). In that case, Vertica does not attempt to read the incomplete result set generated by the UDSF.

Setting up and tearing down

The ScalarFunction class defines two additional methods that you can optionally implement to allocate and free resources: setup() and destroy() . You should use these methods to allocate and deallocate resources that you do not allocate through the UDx API (see Allocating resources for UDxs for details).

Notes

- While the name you choose for your ScalarFunction subclass does not have to match the name of the SQL function you will later assign to it, Vertica considers making the names the same a best practice.
- Do not assume that your function will be called from the same thread that instantiated it.
- The same instance of your ScalarFunction subclass can be called on to process multiple blocks of data.
- The rows of input sent to processBlock() are not guaranteed to be any particular order.
- Writing too many output rows can cause Vertica to emit an out-of-bounds error.

API

C++

Java

Python

R

The ScalarFunction API provides the following methods for extension by subclasses:

```
virtual void setup(ServerInterface &srvInterface,
    const SizedColumnTypes &argTypes);

virtual void processBlock(ServerInterface &srvInterface,
    BlockReader &arg_reader, BlockWriter &res_writer)=0;

virtual void getOutputRange (ServerInterface &srvInterface,
    ValueRangeReader &inRange, ValueRangeWriter &outRange)

virtual void cancel(ServerInterface &srvInterface);

virtual void destroy(ServerInterface &srvInterface, const SizedColumnTypes &argTypes);
```

ScalarFunctionFactory class

The ScalarFunctionFactory class tells Vertica metadata about your UDSF: its number of parameters and their data types, as well as the data type of its return value. It also instantiates a subclass of ScalarFunction .

Methods

You must implement the following methods in your ScalarFunctionFactory subclass:

- createScalarFunction() instantiates a ScalarFunction subclass. If writing in C++, you can call the vt_createFuncObj macro with the name of the ScalarFunction subclass. This macro takes care of allocating and instantiating the class for you.
- getPrototype() tells Vertica about the parameters and return type(s) for your UDSF. In addition to a ServerInterface object, this method gets two ColumnTypes objects. All you need to do in this function is to call class functions on these two objects to build the list of parameters and the return value type(s). If you return more than one value, the results are packaged into a ROW type.

After defining your factory class, you need to call the RegisterFactory macro. This macro instantiates a member of your factory class, so Vertica can interact with it and extract the metadata it contains about your UDSF.

Declaring return values

If your function returns a sized column (a return data type whose length can vary, such as a VARCHAR), a value that requires precision, or more than one value, you must implement getReturnType() . This method is called by Vertica to find the length or precision of the data being returned in each row of the results. The return value of this method depends on the data type your processBlock() method returns:

- CHAR, (LONG) VARCHAR, BINARY, and (LONG) VARBINARY return the maximum length.
- NUMERIC types specify the precision and scale.
- TIME and TIMESTAMP values (with or without timezone) specify precision.
- INTERVAL YEAR TO MONTH specifies range.
- INTERVAL DAY TO SECOND specifies precision and range.
- ARRAY types specify the maximum number of elements.

If your UDSF does not return one of these data types and returns a single value, it does not need a getReturnType() method.

The input to the getReturnType() method is a SizedColumnTypes object that contains the input argument types along with their lengths. This object will be passed to an instance of your processBlock() function. Your implementation of getReturnType() must extract the data types and lengths from this input and determine the length or precision of the output rows. It then saves this information in another instance of the SizedColumnTypes class.

API

C++

Java

Python

R

The ScalarFunctionFactory API provides the following methods for extension by subclasses:

```
virtual ScalarFunction * createScalarFunction(ServerInterface &srvInterface)=0;

virtual void getPrototype(ServerInterface &srvInterface,
      ColumnTypes &argTypes, ColumnTypes &returnType)=0;

virtual void getReturnType(ServerInterface &srvInterface,
      const SizedColumnTypes &argTypes, SizedColumnTypes &returnType);

virtual void getParameterType(ServerInterface &srvInterface,
      SizedColumnTypes &parameterTypes);
```

Setting null input and volatility behavior

Normally, Vertica calls your UDSF for every row of data in the query. There are some cases where Vertica can avoid executing your UDSF. You can tell Vertica when it can skip calling your function and just supply a return value itself by changing your function's volatility and strictness settings.

- Your function's **volatility** indicates whether it always returns the same output value when passed the same arguments. Depending on its behavior, Vertica can cache the arguments and the return value. If the user calls the UDSF with the same set of arguments, Vertica returns the cached value instead of calling your UDSF.
- Your function's **strictness** indicates how it reacts to NULL arguments. If it always returns NULL when *any* argument is NULL, Vertica can just return NULL without having to call the function. This optimization also saves you work, because you do not need to test for and handle null arguments in your UDSF code.

You indicate the volatility and null handling of your function by setting the vol and strict fields in your ScalarFunctionFactory class's constructor.

Volatility settings

To indicate your function's volatility, set the vol field to one of the following values:

Value	Description
VOLATILE	Repeated calls to the function with the same arguments always result in different values. Vertica always calls volatile functions for each invocation.
IMMUTABLE	Calls to the function with the same arguments always results in the same return value.
STABLE	Repeated calls to the function with the same arguments *within the same statement* returns the same output. For example, a function that returns the current user name is stable because the user cannot change within a statement. The user name could change between statements.
DEFAULT_VOLATILITY	The default volatility. This is the same as VOLATILE.

Example

C++

Java

The following example shows a version of the Add2ints example factory class that makes the function immutable.

```
class Add2intsImmutableFactory : public Vertica::ScalarFunctionFactory
{
    virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface &srvInterface)
    { return vt_createFuncObj(srvInterface.allocator, Add2ints); }
    virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                              Vertica::ColumnTypes &argTypes,
                              Vertica::ColumnTypes &returnType)
    {
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }

public:
    Add2intsImmutableFactory() {vol = IMMUTABLE;}
};
RegisterFactory(Add2intsImmutableFactory);
```

Null input behavior

To indicate how your function reacts to NULL input, set the strictness field to one of the following values.

Value	Description
CALLED_ON_NULL_INPUT	The function must be called, even if one or more arguments are NULL.
RETURN_NULL_ON_NULL_INPUT	The function always returns a NULL value if any of its arguments are NULL.
STRICT	A synonym for RETURN_NULL_ON_NULL_INPUT
DEFAULT_STRICTNESS	The default strictness setting. This is the same as CALLED_ON_NULL_INPUT.

Example

The following C++ example demonstrates setting the null behavior of Add2ints so Vertica does not call the function with NULL values.

```
class Add2intsNullOnNullInputFactory : public Vertica::ScalarFunctionFactory
{
    virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface &srvInterface)
    { return vt_createFuncObj(srvInterface.allocator, Add2ints); }
    virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                              Vertica::ColumnTypes &argTypes,
                              Vertica::ColumnTypes &returnType)
    {
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }

public:
    Add2intsNullOnNullInputFactory() {strict = RETURN_NULL_ON_NULL_INPUT;}
};
RegisterFactory(Add2intsNullOnNullInputFactory);
```

Improving query performance (C++ only)

When evaluating a query, Vertica can take advantage of available information about the ranges of values. For example, if data is partitioned and a query restricts output by the partitioned value, Vertica can ignore partitions that cannot possibly contain data that would satisfy the query. Similarly, for a scalar function, Vertica can skip processing rows in the data where the value returned from the function cannot possibly affect the results.

Consider a table with millions of rows of data on customer orders and a scalar function that computes the total price paid for everything in an order. A query uses a WHERE clause to restrict results to orders above a given value. A scalar function is called on a block of data; if no rows within that block could produce the target value, skipping the processing of the block could improve query performance.

A scalar function written in C++ can implement the getOutputRange method. Before calling processBlock , Vertica calls getOutputRange to determine the minimum and maximum return values from this block given the input ranges. It then decides whether to call processBlock to perform the computations.

The Add2Ints example implements this function. The minimum output value is the sum of the smallest values of each of the two inputs, and the maximum output is the sum of the largest values of each of the inputs. This function does not consider individual rows. Consider the following inputs:

```
 a  | b
----+----
 21 | 92
500 | 19
111 | 11
```

The smallest values of the two inputs are 21 and 11, so the function reports 32 as the low end of the output range. The largest input values are 500 and 92, so it reports 592 as the high end of the output range. 592 is larger than the value returned for any of the input rows and 32 is smaller than any row's return value.

The purpose of getOutputRange is to quickly eliminate calls where outputs would definitely be out of range. For example, if the query included "WHERE Add2Ints(a,b) > 600", this block of data could be skipped. There can still be cases where, after calling getOutputRange , processBlock returns no results. If the query included "WHERE Add2Ints(a,b) > 500", getOutputRange would not eliminate this block of data.

Add2Ints implements getOutputRange as follows:

```
/*
 * This method computes the output range for this scalar function from
 *   the ranges of its inputs in a single invocation.
 *
 * The input ranges are retrieved via inRange
 * The output range is returned via outRange
 */
virtual void getOutputRange(Vertica::ServerInterface &srvInterface,
                            Vertica::ValueRangeReader &inRange,
                            Vertica::ValueRangeWriter &outRange)
{
    if (inRange.hasBounds(0) && inRange.hasBounds(1)) {
        // Input ranges have bounds defined
        if (inRange.isNull(0) || inRange.isNull(1)) {
            // At least one range has only NULL values.
            // Output range can only have NULL values.
            outRange.setNull();
            outRange.setHasBounds();
            return;
        } else {
            // Compute output range
            const vint& a1LoBound = inRange.getIntRefLo(0);
            const vint& a2LoBound = inRange.getIntRefLo(1);
            outRange.setIntLo(a1LoBound + a2LoBound);

            const vint& a1UpBound = inRange.getIntRefUp(0);
            const vint& a2UpBound = inRange.getIntRefUp(1);
            outRange.setIntUp(a1UpBound + a2UpBound);
        }
    } else {
        // Input ranges are unbounded. No output range can be defined
        return;
    }

    if (!inRange.canHaveNulls(0) && !inRange.canHaveNulls(1)) {
        // There cannot be NULL values in the output range
        outRange.setCanHaveNulls(false);
    }

    // Let Vertica know that the output range is bounded
    outRange.setHasBounds();
}
```

If getOutputRange produces an error, Vertica issues a warning and does not call the method again for the current query.

C++ example: Add2Ints

The following example shows a basic subclass of ScalarFunction called Add2ints . As the name implies, it adds two integers together, returning a single integer result.

For the complete source code, see /opt/vertica/sdk/examples/ScalarFunctions/Add2Ints.cpp . Java and Python versions of this UDx are included in /opt/vertica/sdk/examples .

Loading and using the example

Use CREATE LIBRARY to load the library containing the function, and then use CREATE FUNCTION (scalar) to declare the function as in the following example:

```
=> CREATE LIBRARY ScalarFunctions AS '/home/dbadmin/examples/ScalarFunctions.so';

=> CREATE FUNCTION add2ints AS LANGUAGE 'C++' NAME 'Add2IntsFactory' LIBRARY ScalarFunctions;
```

The following example shows how to use this function:

```
=> SELECT Add2Ints(27,15);
 Add2ints
----------
       42
(1 row)
=> SELECT * FROM MyTable;
 a  | b
-----+----
  7 |  0
 12 |  2
 12 |  6
 18 |  9
  1 |  1
 58 |  4
450 | 15
(7 rows)
=> SELECT * FROM MyTable WHERE Add2ints(a, b) > 20;
 a  | b
-----+----
 18 |  9
 58 |  4
450 | 15
(3 rows)
```

Function implementation

A scalar function does its computation in the processBlock method:

```
class Add2Ints : public ScalarFunction
{
  public:
  /*
   * This method processes a block of rows in a single invocation.
   *
   * The inputs are retrieved via argReader
   * The outputs are returned via resWriter
   */
  virtual void processBlock(ServerInterface &srvInterface,
                            BlockReader &argReader,
                            BlockWriter &resWriter)
  {
    try {
      // While we have inputs to process
      do {
        if (argReader.isNull(0) || argReader.isNull(1)) {
          resWriter.setNull();
        } else {
          const vint a = argReader.getIntRef(0);
          const vint b = argReader.getIntRef(1);
          resWriter.setInt(a+b);
        }
        resWriter.next();
      } while (argReader.next());
    } catch(std::exception& e) {
      // Standard exception. Quit.
      vt_report_error(0, "Exception while processing block: [%s]", e.what());
    }
  }

// ...
};
```

Implementing getOutputRange , which is optional, allows your function to skip rows where the result would not be within a target range. For example, if a WHERE clause restricts the query results to those in a certain range, calling the function for cases that could not possibly be in that range is unnecessary.

```
/*
 * This method computes the output range for this scalar function from
 *   the ranges of its inputs in a single invocation.
 *
 * The input ranges are retrieved via inRange
 * The output range is returned via outRange
 */
virtual void getOutputRange(Vertica::ServerInterface &srvInterface,
                            Vertica::ValueRangeReader &inRange,
                            Vertica::ValueRangeWriter &outRange)
{
    if (inRange.hasBounds(0) && inRange.hasBounds(1)) {
        // Input ranges have bounds defined
        if (inRange.isNull(0) || inRange.isNull(1)) {
            // At least one range has only NULL values.
            // Output range can only have NULL values.
            outRange.setNull();
            outRange.setHasBounds();
            return;
        } else {
            // Compute output range
            const vint& a1LoBound = inRange.getIntRefLo(0);
            const vint& a2LoBound = inRange.getIntRefLo(1);
            outRange.setIntLo(a1LoBound + a2LoBound);

            const vint& a1UpBound = inRange.getIntRefUp(0);
            const vint& a2UpBound = inRange.getIntRefUp(1);
            outRange.setIntUp(a1UpBound + a2UpBound);
        }
    } else {
        // Input ranges are unbounded. No output range can be defined
        return;
    }

    if (inRange.canHaveNulls(0) && inRange.canHaveNulls(1)) {
        // There cannot be NULL values in the output range
        outRange.setCanHaveNulls(false);
    }

    // Let Vertica know that the output range is bounded
    outRange.setHasBounds();
}
```

Factory implementation

The factory instantiates a member of the class (createScalarFunction), and also describes the function's inputs and outputs (getPrototype):

```
class Add2IntsFactory : public ScalarFunctionFactory
{
  // return an instance of Add2Ints to perform the actual addition.
  virtual ScalarFunction *createScalarFunction(ServerInterface &interface)
  { return vt_createFuncObject<Add2Ints>(interface.allocator); }

  // This function returns the description of the input and outputs of the
  // Add2Ints class's processBlock function.  It stores this information in
  // two ColumnTypes objects, one for the input parameters, and one for
  // the return value.
  virtual void getPrototype(ServerInterface &interface,
                            ColumnTypes &argTypes,
                            ColumnTypes &returnType)
  {
    argTypes.addInt();
    argTypes.addInt();

    // Note that ScalarFunctions *always* return a single value.
    returnType.addInt();
  }
};
```

The RegisterFactory macro

Use the RegisterFactory macro to register a UDx. This macro instantiates the factory class and makes the metadata it contains available for Vertica to access. To call this macro, pass it the name of your factory class:

```
RegisterFactory(Add2IntsFactory);
```

Python example: currency_convert

The currency_convert scalar function reads two values from a table, a currency and a value. It then converts the item's value to USD, returning a single float result.

You can find more UDx examples in the Vertica Github repository, https://github.com/vertica/UDx-Examples .

UDSF Python code

```
import vertica_sdk
import decimal

rates2USD = {'USD': 1.000,
             'EUR': 0.89977,
             'GBP': 0.68452,
             'INR': 67.0345,
             'AUD': 1.39187,
             'CAD': 1.30335,
             'ZAR': 15.7181,
             'XXX': -1.0000}

class currency_convert(vertica_sdk.ScalarFunction):
    """Converts a money column to another currency

    Returns a value in USD.

    """
    def __init__(self):
        pass

    def setup(self, server_interface, col_types):
        pass

    def processBlock(self, server_interface, block_reader, block_writer):
        while(True):
            currency = block_reader.getString(0)
            try:
                rate = decimal.Decimal(rates2USD[currency])

            except KeyError:
                server_interface.log("ERROR: {} not in dictionary.".format(currency))
                # Scalar functions always need a value to move forward to the
                # next input row. Therefore, we need to assign it a value to
                # move beyond the error.
                currency = 'XXX'
                rate = decimal.Decimal(rates2USD[currency])

            starting_value = block_reader.getNumeric(1)
            converted_value = decimal.Decimal(starting_value  / rate)
            block_writer.setNumeric(converted_value)
            block_writer.next()
            if not block_reader.next():
                break

    def destroy(self, server_interface, col_types):
        pass

class currency_convert_factory(vertica_sdk.ScalarFunctionFactory):

    def createScalarFunction(self, srv):
        return currency_convert()

    def getPrototype(self, srv_interface, arg_types, return_type):
        arg_types.addVarchar()
        arg_types.addNumeric()
        return_type.addNumeric()

    def getReturnType(self, srv_interface, arg_types, return_type):
        return_type.addNumeric(9,4)
```

Load the function and library

Create the library and the function.

```
=> CREATE LIBRARY pylib AS '/home/dbadmin/python_udx/currency_convert/currency_convert.py' LANGUAGE 'Python';
CREATE LIBRARY
=> CREATE FUNCTION currency_convert AS LANGUAGE 'Python' NAME 'currency_convert_factory' LIBRARY pylib fenced;
CREATE FUNCTION
```

Querying data with the function

The following query shows how you can run a query with the UDSF.

```
=> SELECT product, currency_convert(currency, value) AS cost_in_usd
    FROM items;
   product    | cost_in_usd
--------------+-------------
 Shoes        |    133.4008
 Soccer Ball  |    110.2817
 Coffee       |     13.5190
 Surfboard    |    176.2593
 Hockey Stick |     76.7177
 Car          |  17000.0000
 Software     |     10.4424
 Hamburger    |      7.5000
 Fish         |    130.4272
 Cattle       |    269.2367
(10 rows)
```

Python example: validate_url

The validate_url scalar function reads a string from a table, a URL. It then validates if the URL is responsive, returning a status code or a string indicating the attempt failed.

You can find more UDx examples in the Vertica Github repository, https://github.com/vertica/UDx-Examples .

UDSF Python code

```
import vertica_sdk
import urllib.request
import time

class validate_url(vertica_sdk.ScalarFunction):
    """Validates HTTP requests.

    Returns the status code of a webpage. Pages that cannot be accessed return
    "Failed to load page."

    """

    def __init__(self):
        pass

    def setup(self, server_interface, col_types):
        pass

    def processBlock(self, server_interface, arg_reader, res_writer):
        # Writes a string to the UDx log file.
        server_interface.log("Validating webpage accessibility - UDx")

        while(True):
            url = arg_reader.getString(0)
            try:
                status = urllib.request.urlopen(url).getcode()
                # Avoid overwhelming web servers -- be nice.
                time.sleep(2)
            except (ValueError, urllib.error.HTTPError, urllib.error.URLError):
                status = 'Failed to load page'
            res_writer.setString(str(status))
            res_writer.next()
            if not arg_reader.next():
                # Stop processing when there are no more input rows.
                break

    def destroy(self, server_interface, col_types):
        pass

class validate_url_factory(vertica_sdk.ScalarFunctionFactory):

    def createScalarFunction(self, srv):
        return validate_url()

    def getPrototype(self, srv_interface, arg_types, return_type):
        arg_types.addVarchar()
        return_type.addChar()

    def getReturnType(self, srv_interface, arg_types, return_type):
        return_type.addChar(20)
```

Load the function and library

Create the library and the function.

```
=> CREATE OR REPLACE LIBRARY pylib AS 'webpage_tester/validate_url.py' LANGUAGE 'Python';
=> CREATE OR REPLACE FUNCTION validate_url AS LANGUAGE 'Python' NAME 'validate_url_factory' LIBRARY pylib fenced;
```

Querying data with the function

The following query shows how you can run a query with the UDSF.

```
=> SELECT url, validate_url(url) AS url_status FROM webpages;
                    url                     |      url_status
------------------------------------------------+----------------------
 http://www.vertica.com/documentation/vertica/ | 200
 http://www.google.com/                         | 200
 http://www.mass.gov.com/                       | Failed to load page
 http://www.espn.com                            | 200
 http://blah.blah.blah.blah                     | Failed to load page
 http://www.vertica.com/                        | 200
(6 rows)
```

Python example: matrix multiplication

Python UDxs can accept and return complex types. The MatrixMultiply class multiplies input matrices and returns the resulting matrix product. These matrices are represented as two-dimensional arrays. In order to perform the matrix multiplication operation, the number of columns in the first input matrix must equal the number of rows in the second input matrix.

The complete source code is in /opt/vertica/sdk/examples/python/ScalarFunctions.py .

Loading and using the example

Load the library and create the function as follows:

```
=> CREATE OR REPLACE LIBRARY ScalarFunctions AS '/home/dbadmin/examples/python/ScalarFunctions.py' LANGUAGE 'Python';

=> CREATE FUNCTION MatrixMultiply AS LANGUAGE 'Python' NAME 'matrix_multiply_factory' LIBRARY ScalarFunctions;
```

You can create input matrices and then call the function, for example:

```

=> CREATE TABLE mn (id INTEGER, data ARRAY[ARRAY[INTEGER, 3], 2]);
CREATE TABLE

=> CREATE TABLE np (id INTEGER, data ARRAY[ARRAY[INTEGER, 2], 3]);
CREATE TABLE

=> COPY mn FROM STDIN PARSER fjsonparser();
{"id": 1, "data": [[1, 2, 3], [4, 5, 6]] }
{"id": 2, "data": [[7, 8, 9], [10, 11, 12]] }
\.

=> COPY np FROM STDIN PARSER fjsonparser();
{"id": 1, "data": [[0, 0], [0, 0], [0, 0]] }
{"id": 2, "data": [[1, 1], [1, 1], [1, 1]] }
{"id": 3, "data": [[2, 0], [0, 2], [2, 0]] }
\.

=> SELECT mn.id, np.id, MatrixMultiply(mn.data, np.data) FROM mn CROSS JOIN np ORDER BY 1, 2;
id | id |   MatrixMultiply
---+----+--------------------
1  |  1 | [[0,0],[0,0]]
1  |  2 | [[6,6],[15,15]]
1  |  3 | [[8,4],[20,10]]
2  |  1 | [[0,0],[0,0]]
2  |  2 | [[24,24],[33,33]]
2  |  3 | [[32,16],[44,22]]
(6 rows)
```

Setup

All Python UDxs must import the Vertica SDK library:

```
import vertica_sdk
```

Factory implementation

The getPrototype() method declares that the function arguments and return type must all be two-dimensional arrays, represented as arrays of integer arrays:

```
def getPrototype(self, srv_interface, arg_types, return_type):
    array1dtype = vertica_sdk.ColumnTypes.makeArrayType(vertica_sdk.ColumnTypes.makeInt())
    arg_types.addArrayType(array1dtype)
    arg_types.addArrayType(array1dtype)
    return_type.addArrayType(array1dtype)
```

getReturnType() validates that the product matrix has the same number of rows as the first input matrix and the same number of columns as the second input matrix:

```
def getReturnType(self, srv_interface, arg_types, return_type):
    (_, a1type) = arg_types[0]
    (_, a2type) = arg_types[1]
    m = a1type.getArrayBound()
    p = a2type.getElementType().getArrayBound()
    return_type.addArrayType(vertica_sdk.SizedColumnTypes.makeArrayType(vertica_sdk.SizedColumnTypes.makeInt(), p), m)
```

Function implementation

The processBlock() method is called with a BlockReader and a BlockWriter , named arg_reader and res_writer respectively. To access elements of the input arrays, the method uses ArrayReader instances. The arrays are nested, so an ArrayReader must be instantiated for both the outer and inner arrays. List comprehension simplifies the process of reading the input arrays into lists. The method performs the computation and then uses an ArrayWriter instance to construct the product matrix.

```
def processBlock(self, server_interface, arg_reader, res_writer):
    while True:
        lmat = [[cell.getInt(0) for cell in row.getArrayReader(0)] for row in arg_reader.getArrayReader(0)]
        rmat = [[cell.getInt(0) for cell in row.getArrayReader(0)] for row in arg_reader.getArrayReader(1)]
        omat = [[0 for c in range(len(rmat[0]))] for r in range(len(lmat))]

        for i in range(len(lmat)):
            for j in range(len(rmat[0])):
                for k in range(len(rmat)):
                    omat[i][j] += lmat[i][k] * rmat[k][j]

        res_writer.setArray(omat)
        res_writer.next()

        if not arg_reader.next():
            break
```

R example: SalesTaxCalculator

The SalesTaxCalculator scalar function reads a float and a varchar from a table, an item's price and the state abbreviation. It then uses the state abbreviation to find the sales tax rate from a list and calculates the item's price including the state's sales tax, returning the total cost of the item.

You can find more UDx examples in the Vertica Github repository, https://github.com/vertica/UDx-Examples .

Load the function and library

Create the library and the function.

```
=> CREATE OR REPLACE LIBRARY rLib AS 'sales_tax_calculator.R' LANGUAGE 'R';
CREATE LIBRARY
=> CREATE OR REPLACE FUNCTION SalesTaxCalculator AS LANGUAGE 'R' NAME 'SalesTaxCalculatorFactory' LIBRARY rLib FENCED;
CREATE FUNCTION
```

Querying data with the function

The following query shows how you can run a query with the UDSF.

```
=> SELECT item, state_abbreviation,
        price, SalesTaxCalculator(price, state_abbreviation) AS Price_With_Sales_Tax
   FROM inventory;
   item     | state_abbreviation | price | Price_With_Sales_Tax
------------+--------------------+-------+----------------------
 Scarf      | AZ                 |  6.88 |              7.53016
 Software   | MA                 | 88.31 |             96.655295
 Soccer Ball | MS                | 12.55 |             13.735975
 Beads      | LA                 |  0.99 |              1.083555
 Baseball   | TN                 | 42.42 |             46.42869
 Cheese     | WI                 | 20.77 |             22.732765
 Coffee Mug | MA                 |  8.99 |              9.839555
 Shoes      | TN                 | 23.99 |             26.257055
(8 rows)
```

UDSF R code

```
SalesTaxCalculator <- function(input.data.frame) {
  # Not a complete list of states in the USA, but enough to get the idea.
  state.sales.tax <- list(ma = 0.0625,
                          az = 0.087,
                          la = 0.0891,
                          tn = 0.0945,
                          wi = 0.0543,
                          ms = 0.0707)
  for ( state_abbreviation in input.data.frame[, 2] ) {
    # Ensure state abbreviations are lowercase.
    lower_state <- tolower(state_abbreviation)
    # Check if the state is in our state.sales.tax list.
    if (is.null(state.sales.tax[[lower_state]])) {
      stop("State is not in our small sample!")
    } else {
      sales.tax.rate <- state.sales.tax[[lower_state]]
      item.price <- input.data.frame[, 1]
      # Calculate the price including sales tax.
      price.with.sales.tax <- (item.price) + (item.price * sales.tax.rate)
    }
  }
  return(price.with.sales.tax)
}

SalesTaxCalculatorFactory <- function() {
  list(name    = SalesTaxCalculator,
       udxtype = c("scalar"),
       intype  = c("float", "varchar"),
       outtype = c("float"))
}
```

R example: kmeans

The KMeans_User scalar function reads any number of columns from a table, the observations. It then uses the observations and the two parameters when applying the kmeans clustering algorithm to the data, returning an integer value associated with the cluster of the row.

You can find more UDx examples in the Vertica Github repository, https://github.com/vertica/UDx-Examples .

Load the function and library

Create the library and the function:

```
=> CREATE OR REPLACE LIBRARY rLib AS 'kmeans.R' LANGUAGE 'R';
CREATE LIBRARY
=> CREATE OR REPLACE FUNCTION KMeans_User AS LANGUAGE 'R' NAME 'KMeans_UserFactory' LIBRARY rLib FENCED;
CREATE FUNCTION
```

Querying data with the function

The following query shows how you can run a query with the UDSF:

```
=> SELECT spec,
          KMeans_User(sl, sw, pl, pw USING PARAMETERS clusters = 3, nstart = 20)
     FROM iris;
      spec       | KMeans_User
-----------------+-------------
 Iris-setosa     |           2
 Iris-setosa     |           2
 Iris-setosa     |           2
 Iris-setosa     |           2
 Iris-setosa     |           2
 Iris-setosa     |           2
 Iris-setosa     |           2
 Iris-setosa     |           2
 Iris-setosa     |           2
 Iris-setosa     |           2
 Iris-setosa     |           2
.
.
.
(150 rows)
```

UDSF R code

```
KMeans_User <- function(input.data.frame, parameters.data.frame) {
  # Take the clusters and nstart parameters passed by the user and assign them
  # to variables in the function.
  if ( is.null(parameters.data.frame[['clusters']]) ) {
    stop("NULL value for clusters! clusters cannot be NULL.")
  } else {
    clusters.value <- parameters.data.frame[['clusters']]
  }
  if ( is.null(parameters.data.frame[['nstart']]) ) {
    stop("NULL value for nstart! nstart cannot be NULL.")
  } else {
    nstart.value <- parameters.data.frame[['nstart']]
  }
  # Apply the algorithm to the data.
  kmeans.clusters <- kmeans(input.data.frame[, 1:length(input.data.frame)],
                            clusters.value, nstart = nstart.value)
  final.output <- data.frame(kmeans.clusters$cluster)
  return(final.output)
}

KMeans_UserFactory <- function() {
  list(name    = KMeans_User,
       udxtype = c("scalar"),
       # Since this is a polymorphic function the intype must be any
       intype  = c("any"),
       outtype = c("int"),
       parametertypecallback=KMeansParameters)
}

KMeansParameters <- function() {
  parameters <- list(datatype = c("int", "int"),
                     length   = c("NA", "NA"),
                     scale    = c("NA", "NA"),
                     name     = c("clusters", "nstart"))
  return(parameters)
}
```

C++ example: using complex types

UDxs can accept and return complex types. The ArraySlice example takes an array and two indices as inputs and returns an array containing only the values in that range. Because array elements can be of any type, the function is polymorphic.

The complete source code is in /opt/vertica/sdk/examples/ScalarFunctions/ArraySlice.cpp .

Loading and using the example

Load the library and create the function as follows:

```
=> CREATE OR REPLACE LIBRARY ScalarFunctions AS '/home/dbadmin/examplesUDSF.so';

=> CREATE FUNCTION ArraySlice AS
LANGUAGE 'C++' NAME 'ArraySliceFactory' LIBRARY ScalarFunctions;
```

Create some data and call the function on it as follows:

```
=> CREATE TABLE arrays (id INTEGER, aa ARRAY[INTEGER]);
COPY arrays FROM STDIN;
1|[]
2|[1,2,3]
3|[5,4,3,2,1]
\.

=> CREATE TABLE slices (b INTEGER, e INTEGER);
COPY slices FROM STDIN;
0|2
1|3
2|4
\.

=> SELECT id, b, e, ArraySlice(aa, b, e) AS slice FROM arrays, slices;
 id | b | e | slice
----+---+---+-------
  1 | 0 | 2 | []
  1 | 1 | 3 | []
  1 | 2 | 4 | []
  2 | 0 | 2 | [1,2]
  2 | 1 | 3 | [2,3]
  2 | 2 | 4 | [3]
  3 | 0 | 2 | [5,4]
  3 | 1 | 3 | [4,3]
  3 | 2 | 4 | [3,2]
(9 rows)
```

Factory implementation

Because the function is polymorphic, getPrototype() declares that the inputs and outputs can be of any type, and type enforcement must be done elsewhere:

```
void getPrototype(ServerInterface &srvInterface,
                  ColumnTypes &argTypes,
                  ColumnTypes &returnType) override

  /*
   * This is a polymorphic function that accepts any array
   * and returns an array of the same type
   */
  argTypes.addAny();
  returnType.addAny();
```

The factory validates input types and determines the return type in getReturnType() :

```
void getReturnType(ServerInterface &srvInterface,
                   const SizedColumnTypes &argTypes,
                   SizedColumnTypes &returnType) override
{
  /*
   * Three arguments: (array, slicebegin, sliceend)
   * Validate manually since the prototype accepts any arguments.
   */
  if (argTypes.size() != 3) {
    vt_report_error(0, "Three arguments (array, slicebegin, sliceend) expected");
  } else if (!argTypes[0].getType().isArrayType()) {
    vt_report_error(1, "Argument 1 is not an array");
  } else if (!argTypes[1].getType().isInt()) {
    vt_report_error(2, "Argument 2 (slicebegin) is not an integer)");
  } else if (!argTypes[2].getType().isInt()) {
    vt_report_error(3, "Argument 3 (sliceend) is not an integer)");
  }

  /* return type is the same as the array arg type, copy it over */
  returnType.push_back(argTypes[0]);
}
```

Function implementation

The processBlock() method is called with a BlockReader and a BlockWriter . The first argument is an array. To access elements of the array, the method uses an ArrayReader . Similarly, it uses an ArrayWriter to construct the output.

```
void processBlock(ServerInterface &srvInterface,
    BlockReader &argReader,
    BlockWriter &resWriter) override
{
  do {
    if (argReader.isNull(0) || argReader.isNull(1) || argReader.isNull(2)) {
      resWriter.setNull();
    } else {
      Array::ArrayReader argArray  = argReader.getArrayRef(0);
      const vint slicebegin = argReader.getIntRef(1);
      const vint sliceend   = argReader.getIntRef(2);

      Array::ArrayWriter outArray = resWriter.getArrayRef(0);
      if (slicebegin < sliceend) {
        for (int i = 0; i < slicebegin && argArray->hasData(); i++) {
          argArray->next();
        }
        for (int i = slicebegin; i < sliceend && argArray->hasData(); i++) {
          outArray->copyFromInput(*argArray);
          outArray->next();
          argArray->next();
        }
      }
      outArray.commit();  /* finalize the written array elements */
    }
    resWriter.next();
  } while (argReader.next());
}
```

C++ example: returning multiple values

When writing a UDSF, you can specify more than one return value. If you specify multiple values, Vertica packages them into a single ROW as a return value. You can query fields in the ROW or the entire ROW.

The following example implements a function named div (division) that returns two integers, the quotient and the remainder.

This example shows one way to return a ROW from a UDSF. Returning multiple values and letting Vertica build the ROW is convenient when inputs and outputs are all of primitive types. You can also work directly with the complex types, as described in Complex Types as Arguments and illustrated in C++ example: using complex types .

Loading and using the example

Load the library and create the function as follows:

```
=> CREATE OR REPLACE LIBRARY ScalarFunctions AS '/home/dbadmin/examplesUDSF.so';

=> CREATE FUNCTION div AS
LANGUAGE 'C++' NAME 'DivFactory' LIBRARY ScalarFunctions;
```

Create some data and call the function on it as follows:

```
=> CREATE TABLE D (a INTEGER, b INTEGER);
COPY D FROM STDIN DELIMITER ',';
10,0
10,1
10,2
10,3
10,4
10,5
\.

=> SELECT a, b, Div(a, b), (Div(a, b)).quotient, (Div(a, b)).remainder FROM D;
 a  | b |                  Div                  | quotient | remainder
----+---+---------------------------------------+----------+-----------
 10 | 0 | {"quotient":null,"remainder":null} |          |
 10 | 1 | {"quotient":10,"remainder":0}      |       10 |         0
 10 | 2 | {"quotient":5,"remainder":0}       |        5 |         0
 10 | 3 | {"quotient":3,"remainder":1}       |        3 |         1
 10 | 4 | {"quotient":2,"remainder":2}       |        2 |         2
 10 | 5 | {"quotient":2,"remainder":0}       |        2 |         0
(6 rows)
```

Factory implementation

The factory declares the two return values in getPrototype() and in getReturnType() . The factory is otherwise unremarkable.

```
void getPrototype(ServerInterface &interface,
        ColumnTypes &argTypes,
        ColumnTypes &returnType) override
{
    argTypes.addInt();
    argTypes.addInt();
    returnType.addInt(); /* quotient  */
    returnType.addInt(); /* remainder */
}

void getReturnType(ServerInterface &srvInterface,
        const SizedColumnTypes &argTypes,
        SizedColumnTypes &returnType) override
{
    returnType.addInt("quotient");
    returnType.addInt("remainder");
}
```

Function implementation

The function writes two output values in processBlock() . The number of values here must match the factory declarations.

```
class Div : public ScalarFunction {
  void processBlock(Vertica::ServerInterface &srvInterface,
        Vertica::BlockReader &argReader,
        Vertica::BlockWriter &resWriter) override
  {
    do {
      if (argReader.isNull(0) || argReader.isNull(1) || (argReader.getIntRef(1) == 0)) {
        resWriter.setNull(0);
        resWriter.setNull(1);
      } else {
        const vint dividend = argReader.getIntRef(0);
        const vint divisor  = argReader.getIntRef(1);
        resWriter.setInt(0, dividend / divisor);
        resWriter.setInt(1, dividend % divisor);
      }
      resWriter.next();
    } while (argReader.next());
  }
};
```

C++ example: calling a UDSF from a check constraint

This example shows you the C++ code needed to create a UDSF that can be called by a check constraint. The name of the sample function is LargestSquareBelow . The sample function determines the largest number whose square is less than the number in the subject column. For example, if the number in the column is 1000, the largest number whose square is less than 1000 is 31 (961).

Important
A UDSF used within a check constraint must be immutable, and the constraint must handle null values properly. Otherwise, the check constraint might not work as you intended. In addition, Vertica evaluates the predicate of an enabled check constraint on every row that is loaded or updated, so consider performance in writing your function.

For information on check constraints, see Check constraints .

Loading and using the example

The following example shows how you can create and load a library named MySqLib, using CREATE LIBRARY . Adjust the library path in this example to the absolute path and file name for the location where you saved the shared object LargestSquareBelow .

Create the library:

```
=> CREATE OR REPLACE LIBRARY MySqLib AS '/home/dbadmin/LargestSquareBelow.so';
```

After you create and load the library, add the function to the catalog using the CREATE FUNCTION (scalar) statement:

```
=> CREATE OR REPLACE FUNCTION largestSqBelow AS LANGUAGE 'C++' NAME 'LargestSquareBelowInfo' LIBRARY MySqLib;
```

Next, include the UDSF in a check constraint:

```
=> CREATE TABLE squaretest(
   ceiling INTEGER UNIQUE,
   CONSTRAINT chk_sq CHECK (largestSqBelow(ceiling) < ceiling*ceiling)
);
```

Add data to the table, squaretest :

```
=> COPY squaretest FROM stdin DELIMITER ','NULL'null';
-1
null
0
1
1000
1000000
1000001
\.
```

Your output should be similar to the following sample, based upon the data you use:

```
=> SELECT ceiling, largestSqBelow(ceiling)
FROM squaretest ORDER BY ceiling;

ceiling  | largestSqBelow
---------+----------------
         |
      -1 |
       0 |
       1 |              0
    1000 |             31
 1000000 |            999
 1000001 |           1000
(7 rows)
```

ScalarFunction implementation

This ScalarFunction implementation does the processing work for a UDSF that determines the largest number whose square is less than the number input.

```
#include "Vertica.h"
/*
 * ScalarFunction implementation for a UDSF that
 * determines the largest number whose square is less than
 * the number input.
 */
class LargestSquareBelow : public Vertica::ScalarFunction
{
public:
  /*
   * This function does all of the actual processing for the UDSF.
   * The inputs are retrieved via arg_reader
   * The outputs are returned via arg_writer
   *
   */
    virtual void processBlock(Vertica::ServerInterface &srvInterface,
                              Vertica::BlockReader &arg_reader,
                              Vertica::BlockWriter &res_writer)
    {
        if (arg_reader.getNumCols() != 1)
            vt_report_error(0, "Function only accept 1 argument, but %zu provided", arg_reader.getNumCols());
// While we have input to process
        do {
            // Read the input parameter by calling the
            // BlockReader.getIntRef class function
            const Vertica::vint a = arg_reader.getIntRef(0);
            Vertica::vint res;
            //Determine the largest square below the number
            if ((a != Vertica::vint_null) && (a > 0))
            {
                res = (Vertica::vint)sqrt(a - 1);
            }
            else
                res = Vertica::vint_null;
            //Call BlockWriter.setInt to store the output value,
            //which is the largest square
            res_writer.setInt(res);
            //Write the row and advance to the next output row
            res_writer.next();
            //Continue looping until there are no more input rows
        } while (arg_reader.next());
    }
};
```

ScalarFunctionFactory implementation

This ScalarFunctionFactory implementation does the work of handling input and output, and marks the function as immutable (a requirement if you plan to use the UDSF within a check constraint).

```
class LargestSquareBelowInfo : public Vertica::ScalarFunctionFactory
{
  //return an instance of LargestSquareBelow to perform the computation.
  virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface &srvInterface)
  //Call the vt_createFuncObj to create the new LargestSquareBelow class instance.
  { return Vertica::vt_createFuncObject<LargestSquareBelow>(srvInterface.allocator); }

  /*
   * This function returns the description of the input and outputs of the
   * LargestSquareBelow class's processBlock function.  It stores this information in
   * two ColumnTypes objects, one for the input parameter, and one for
   * the return value.
   */
  virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                            Vertica::ColumnTypes &argTypes,
                            Vertica::ColumnTypes &returnType)
  {
      // Takes one int as input, so adds int to the argTypes object
      argTypes.addInt();
      // Returns a single int, so add a single int to the returnType object.
      // ScalarFunctions always return a single value.
      returnType.addInt();
  }
public:
  // the function cannot be called within a check constraint unless the UDx author
  // certifies that the function is immutable:
  LargestSquareBelowInfo() { vol = Vertica::IMMUTABLE; }
};
```

The RegisterFactory macro

Use the RegisterFactory macro to register a ScalarFunctionFactory subclass. This macro instantiates the factory class and makes the metadata it contains available for Vertica to access. To call this macro, pass it the name of your factory class.

```
RegisterFactory(LargestSquareBelowInfo);
```

Transform functions (UDTFs)

A user-defined transform function (UDTF) lets you transform a table of data into another table. It reads one or more arguments (treated as a row of data), and returns zero or more rows of data consisting of one or more columns. A UDTF can produce any number of rows as output. However, each row it outputs must be complete. Advancing to the next row without having added a value for each column produces incorrect results.

The schema of the output table does not need to correspond to the schema of the input table—they can be totally different. The UDTF can return any number of output rows for each row of input.

Unless a UDTF is marked as one-to-many in its factory function, it can only be used in a SELECT list that contains the UDTF call and a required OVER clause. A multi-phase UDTF can make use of partition columns (PARTITION BY), but other UDTFs cannot.

UDTFs are run after GROUP BY, but before the final ORDER BY, when used in conjunction with GROUP BY and ORDER BY in a statement. The ORDER BY clause may contain only columns or expressions that are in a window partition clause (see Window partitioning).

UDTFs can take up to 9800 parameters (input columns). Attempts to pass more parameters to a UDTF return an error.

In this section

- TransformFunction class
- TransformFunctionFactory class
- MultiPhaseTransformFunctionFactory class
- Improving query performance (C++ only)
- Partitioning options for UDTFs
- C++ example: string tokenizer
- Python example: string tokenizer
- R example: log tokenizer

- C++ example: multi-phase indexer
- Python example: multi-phase calculation
- Python example: count elements
- Python example: explode

TransformFunction class

The TransformFunction class is where you perform the data-processing, transforming input rows into output rows. Your subclass must define the processPartition() method. It may define methods to set up and tear down the function.

Performing the transformation

The processPartition() method carries out all of the processing that you want your UDTF to perform. When a user calls your function in a SQL statement, Vertica bundles together the data from the function parameters and passes it to processPartition() .

The input and output of the processPartition() method are supplied by objects of the PartitionReader and PartitionWriter classes. They define methods that you use to read the input data and write the output data for your UDTF.

A UDTF does not necessarily operate on a single row the way a UDSF does. A UDTF can read any number of rows and write output at any time.

Consider the following guidelines when implementing processPartition() :

- Extract the input parameters by calling data-type-specific functions in the PartitionReader object to extract each input parameter. Each of these functions takes a single parameter: the column number in the input row that you want to read. Your function might need to handle NULL values.
- When writing output, your UDTF must supply values for all of the output columns you defined in your factory. Similarly to reading input columns, the PartitionWriter object has functions for writing each type of data to the output row.
- Use PartitionReader.next() to determine if there is more input to process, and exit when the input is exhausted.
- In some cases, you might want to determine the number and types of parameters using PartitionReader 's getNumCols() and getTypeMetaData() functions, instead of just hard-coding the data types of the columns in the input row. This is useful if you want your TransformFunction to be able to process input tables with different schemas. You can then use different TransformFunctionFactory classes to define multiple function signatures that call the same TransformFunction class. See Overloading your UDx for more information.

Setting up and tearing down

The TransformFunction class defines two additional methods that you can optionally implement to allocate and free resources: setup() and destroy() . You should use these methods to allocate and deallocate resources that you do not allocate through the UDx API (see Allocating resources for UDxs for details).

API

C++

Java

Python

R

The TransformFunction API provides the following methods for extension by subclasses:

```
virtual void setup(ServerInterface &srvInterface,
        const SizedColumnTypes &argTypes);

virtual void processPartition(ServerInterface &srvInterface,
        PartitionReader &input_reader, PartitionWriter &output_writer)=0;

virtual void cancel(ServerInterface &srvInterface);

virtual void destroy(ServerInterface &srvInterface,
        const SizedColumnTypes &argTypes);
```

The PartitionReader and PartitionWriter classes provide getters and setters for column values, along with next() to iterate through partitions. See the API reference documentation for details.

TransformFunctionFactory class

The TransformFunctionFactory class tells Vertica metadata about your UDTF: its number of parameters and their data types, as well as function properties and the data type of the return value. It also instantiates a subclass of TransformFunction .

You must implement the following methods in your TransformFunctionFactory :

- getPrototype() returns two ColumnTypes objects that describe the columns your UDTF takes as input and returns as output.
- getReturnType() tells Vertica details about the output values: the width of variable-sized data types (such as VARCHAR) and the precision of data types that have settable precision (such as TIMESTAMP). You can also set the names of the output columns using this function. While this method is optional for UDxs that return single values, you must implement it for UDTFs.
- createTransformFunction() instantiates your TransformFunction subclass.

For UDTFs written in C++ and Python, you can implement the getTransformFunctionProperties() method to set transform function class properties, including:

- isExploder : By default False, indicates whether a single-phase UDTF performs a transform from one input row to a result set of N rows, often called a one-to-many transform. If set to True, each partition to the UDTF must consist of exactly one input row. When a UDTF is labeled as one-to-many, Vertica is able to optimize query plans and users can write SELECT queries that include any expression and do not require an OVER clause. For more information about UDTF partitioning options and instructions on how to set this class property, see Partitioning options for UDTFs . See Python example: explode for an in-depth example detailing a one-to-many UDTF.

For transform functions written in C++, you can provide information that can help with query optimization. See Improving query performance (C++ only) .

API

C++

Java

Python

R

The TransformFunctionFactory API provides the following methods for extension by subclasses:

```
virtual TransformFunction *
    createTransformFunction (ServerInterface &srvInterface)=0;

virtual void getPrototype(ServerInterface &srvInterface,
            ColumnTypes &argTypes, ColumnTypes &returnType)=0;

virtual void getReturnType(ServerInterface &srvInterface,
            const SizedColumnTypes &argTypes,
            SizedColumnTypes &returnType)=0;

virtual void getParameterType(ServerInterface &srvInterface,
            SizedColumnTypes &parameterTypes);

virtual void getTransformFunctionProperties(ServerInterface &srvInterface,
            const SizedColumnTypes &argTypes,
            Properties &properties);
```

MultiPhaseTransformFunctionFactory class

Multi-phase UDTFs let you break your data processing into multiple steps. Using this feature, your UDTFs can perform processing in a way similar to Hadoop or other MapReduce frameworks. You can use the first phase to break down and gather data, and then use subsequent phases to process the data. For example, the first phase of your UDTF could extract specific types of user interactions from a web server log stored in the column of a table, and subsequent phases could perform analysis on those interactions.

Multi-phase UDTFs also let you decide where processing should occur: locally on each node, or throughout the cluster. If your multi-phase UDTF is like a MapReduce process, you want the first phase of your multi-phase UDTF to process data that is stored locally on the node where the instance of the UDTF is running. This prevents large segments of data from being copied around the Vertica cluster. Depending on the type of processing being

performed in later phases, you may choose to have the data segmented and distributed across the Vertica cluster.

Each phase of the UDTF is the same as a traditional (single-phase) UDTF: it receives a table as input, and generates a table as output. The schema for each phase's output does not have to match its input, and each phase can output as many or as few rows as it wants.

You create a subclass of TransformFunction to define the processing performed by each stage. If you already have a TransformFunction from a single-phase UDTF that performs the processing you want a phase of your multi-phase UDTF to perform, you can easily adapt it to work within the multi-phase UDTF.

What makes a multi-phase UDTF different from a traditional UDTF is the factory class you use. You define a multi-phase UDTF using a subclass of MultiPhaseTransformFunctionFactory , rather than the TransformFunctionFactory . This special factory class acts as a container for all of the phases in your multi-step UDTF. It provides Vertica with the input and output requirements of the entire multi-phase UDTF (through the getPrototype() function), and a list of all the phases in the UDTF.

Within your subclass of the MultiPhaseTransformFunctionFactory class, you define one or more subclasses of TransformFunctionPhase . These classes fill the same role as the TransformFunctionFactory class for each phase in your multi-phase UDTF. They define the input and output of each phase and create instances of their associated TransformFunction classes to perform the processing for each phase of the UDTF. In addition to these subclasses, your MultiPhaseTransformFunctionFactory includes fields that provide a handle to an instance of each of the TransformFunctionPhase subclasses.

> **Note**
> The MultiPhaseTransformFunctionFactory and TransformFunctionPhase classes do not support the isExploder class property.

API

C++

Java

Python

The MultiPhaseTransformFunctionFactory class extends TransformFunctionFactory The API provides the following additional methods for extension by subclasses:

```
virtual void getPhases(ServerInterface &srvInterface,
    std::vector< TransformFunctionPhase * > &phases)=0;
```

If using this factory you must also extend TransformFunctionPhase . See the SDK reference documentation.

Improving query performance (C++ only)

When evaluating a query, the Vertica optimizer might sort its input to improve performance. If a function already returns sorted data, this means the optimizer is doing extra work. A transform function written in C++ can declare how the data it returns is sorted, and the optimizer can take advantage of that information.

A transform function does the actual sorting in the function's processPartition() method. To take advantage of this optimization, sorts must be ascending. You need not sort all columns, but you must return the sorted column or columns first.

You can declare how the function sorts its output in the factory's getReturnType() method.

> **Caution**
> If the sorting declared in the factory does not match the sorting provided by the function, query results can be incorrect.

The PolyTopKPerPartition example sorts input columns and returns a given number of rows:

```
=> SELECT polykSort(14, a, b, c) OVER (ORDER BY a, b, c)
   AS (sort1,sort2,sort3) FROM observations ORDER BY 1,2,3;
 sort1 | sort2 | sort3
-------+-------+-------
     1 |     1 |     1
     1 |     1 |     2
     1 |     1 |     3
     1 |     2 |     1
     1 |     2 |     2
     1 |     3 |     1
     1 |     3 |     2
     1 |     3 |     3
     1 |     3 |     4
     2 |     1 |     1
     2 |     1 |     2
     2 |     2 |     3
     2 |     2 |    34
     2 |     3 |     5
(14 rows)
```

The factory declares this sorting in getReturnType() by setting the isSortedBy property on each column. Each SizedColumnType has an associated Properties object where this value can be set:

```
virtual void getReturnType(ServerInterface &srvInterface, const SizedColumnTypes &inputTypes, SizedColumnTypes &outputTypes)
{
    vector<size_t> argCols; // Argument column indexes.
    inputTypes.getArgumentColumns(argCols);
    size_t colIdx = 0;

    for (vector<size_t>::iterator i = argCols.begin() + 1; i < argCols.end(); i++)
    {
        SizedColumnTypes::Properties props;
        props.isSortedBy = true;
        std::stringstream cname;
        cname << "col" << colIdx++;
        outputTypes.addArg(inputTypes.getColumnType(*i), cname.str(), props);
    }
}
```

You can see the effects of this optimization by reviewing the EXPLAIN plans for queries with and without this setting. The following output shows the query plan for polyk , the unsorted version. Note the cost for sorting:

```
=> EXPLAIN SELECT polyk(14, a, b, c) OVER (ORDER BY a, b, c)
   FROM observations ORDER BY 1,2,3;

 Access Path:
 +-SORT [Cost: 2K, Rows: 10K] (PATH ID: 1)
 |  Order: col0 ASC, col1 ASC, col2 ASC
 | +---> ANALYTICAL [Cost: 2K, Rows: 10K] (PATH ID: 2)
 | |      Analytic Group
 | |        Functions: polyk()
 | |        Group Sort: observations.a ASC NULLS LAST, observations.b ASC NULLS LAST, observations.c ASC NULLS LAST
 | | +---> STORAGE ACCESS for observations [Cost: 2K, Rows: 10K]
 (PATH ID: 3)
 | | |      Projection: public.observations_super
 | | |      Materialize: observations.a, observations.b, observations.c
```

The query plan for the sorted version omits this step (and cost) and starts with the analytical step (the second step in the previous plan):

```
=> EXPLAN SELECT polykSort(14, a, b, c) OVER (ORDER BY a, b, c)
   FROM observations ORDER BY 1,2,3;

Access Path:
+-ANALYTICAL [Cost: 2K, Rows: 10K] (PATH ID: 2)
|  Analytic Group
|   Functions: polykSort()
|   Group Sort: observations.a ASC NULLS LAST, observations.b ASC NULLS LAST, observations.c ASC NULLS LAST
| +---> STORAGE ACCESS for observations [Cost: 2K, Rows: 10K] (PATH ID: 3)
| |      Projection: public.observations_super
| |      Materialize: observations.a, observations.b, observations.c
```

Partitioning options for UDTFs

Depending on the application, a UDTF might require the input data to be partitioned in a specific way. For example, a UDTF that processes a web server log file to count the number of hits referred by each partner web site needs to have its input partitioned by a referrer column. However, in other cases—such as a string tokenizer—the sort order of the data does not matter. Vertica provides partition options for both of these types of UDTFs.

Data sort required

In cases where a specific sort order is required, the window partitioning clause in the query that calls the UDTF should use a PARTITION BY clause. Each node in the cluster partitions the data it stores, sends some of these partitions off to other nodes, and then consolidates the partitions it receives from other nodes and runs an instance of the UDTF to process them.

For example, the following UDTF partitions the input data by store ID and then computes the count of each distinct array element in each partition:

```
=> SELECT * FROM orders;
 storeID |      productIDs
---------+-----------------------
       1 | [101,102,103]
       1 | [102,104]
       1 | [101,102,102,201,203]
       2 | [101,202,203,202,203]
       2 | [203]
       2 | []
(6 rows)

=> SELECT storeID, CountElements(productIDs) OVER (PARTITION BY storeID) FROM orders;
storeID |       element_count
--------+---------------------------
      1 | {"element":101,"count":2}
      1 | {"element":102,"count":4}
      1 | {"element":103,"count":1}
      1 | {"element":104,"count":1}
      1 | {"element":201,"count":1}
      1 | {"element":202,"count":1}
      2 | {"element":101,"count":1}
      2 | {"element":202,"count":2}
      2 | {"element":203,"count":3}
(9 rows)
```

No sort needed

Some UDTFs, such as Explode , do not need to partition input data in a particular way. In these cases, you can specify that each UDTF instance process only the data that is stored locally by the node on which it is running. By eliminating the overhead of partitioning data and the cost of sort and merge operations, processing can be much more efficient.

You can use the following window partition options for UDTFs that do not require a specific data partitioning:

- PARTITION ROW : For single-phase UDTFs where each partition is one input row, allows users to write SELECT queries that include any expression. The UDTF calls the processPartition() method once per input row. UDTFs of this type, often called one-to-many transforms, can be explicitly marked as such with the exploder class property in the TransformFunctionFactory class . This class property helps Vertica optimize query

plans and removes the need for an OVER clause. See One to Many UDTFs for details on how to set this class property for UDTFs written in C++ and Python.

- PARTITION BEST : For thread-safe UDTFs only, optimizes performance through multi-threaded queries across multiple nodes. The UDTF calls the processPartition() method once per thread per node.
- PARTITION NODES : Optimizes performance of single-threaded queries across multiple nodes. The UDTF calls the processPartition() method once per node.

For more information about these partition options, see Window partitioning .

One-to-many UDTFs

C++

Python

To mark a UDTF as one-to-many, you must set the isExploder class property to True within the getTransformFunctionProperties() method. Whether a UDTF is marked as one-to-many can be determined by the transform function's arguments and parameters, for example:

```
void getFunctionProperties(ServerInterface &srvInterface,
        const SizedColumnTypes &argTypes,
        Properties &properties) override
{
    if (argTypes.getColumnCount() > 1) {
        properties.isExploder = false;
    }
    else {
        properties.isExploder = true;
    }
}
```

If the exploder class property is set to True, the OVER clause is by default OVER(PARTITION ROW). This allows users to call the UDTF without specifying an OVER clause:

```
=> SELECT * FROM reviews;
 id |             sentence
----+----------------------------------------
  1 | Customer service was slow
  2 | Product is exactly what I needed
  3 | Price is a bit high
  4 | Highly recommended
(4 rows)

=> SELECT tokenize(sentence) FROM reviews;
   tokens
-------------
Customer
service
was
slow
Product
...
bit
high
Highly
recommended
(17 rows)
```

Note

If a user writes an OVER clause with PARTITION BY for a function marked as one-to-many, the function replaces the OVER clause with OVER(PARTITION ROW) and emits a notice to the user.

One-to-many UDTFs also support any expression in the SELECT clause, unlike UDTFs that use either the PARTITION BEST or the PARTITION NODES clause:

```
=> SELECT id, tokenize(sentence) FROM reviews;
 id |   tokens
----+-------------
  1 | Customer
  1 | service
  1 | was
  1 | respond
  2 | Product
...
  3 | high
  4 | Highly
  4 | recommended
(17 rows)
```

For an in-depth example detailing a one-to-many UDTF, see Python example: explode .

See also

- TransformFunctionFactory class
- EXPLODE
- Window partitioning
- Window partition clause

C++ example: string tokenizer

The following example shows a subclass of TransformFunction named StringTokenizer . It defines a UDTF that reads a table containing an INTEGER ID column and a VARCHAR column. It breaks the text in the VARCHAR column into tokens (individual words). It returns a table containing each token, the row it occurred in, and its position within the string.

Loading and using the example

The following example shows how to load the function into Vertica. It assumes that the TransformFunctions.so library that contains the function has been copied to the dbadmin user's home directory on the initiator node.

```
=> CREATE LIBRARY TransformFunctions AS
   '/home/dbadmin/TransformFunctions.so';
CREATE LIBRARY
=> CREATE TRANSFORM FUNCTION tokenize
   AS LANGUAGE 'C++' NAME 'TokenFactory' LIBRARY TransformFunctions;
CREATE TRANSFORM FUNCTION
```

You can then use it from SQL statements, for example:

```
=> CREATE TABLE T (url varchar(30), description varchar(2000));
CREATE TABLE
=> INSERT INTO T VALUES ('www.amazon.com','Online retail merchant and provider of cloud services');
 OUTPUT
--------
      1
(1 row)
=> INSERT INTO T VALUES ('www.vertica.com','World"s fastest analytic database');
 OUTPUT
--------
      1
(1 row)
=> COMMIT;
COMMIT

=> -- Invoke the UDTF
=> SELECT url, tokenize(description) OVER (partition by url) FROM T;
       url       |   words
-----------------+-----------
 www.amazon.com  | Online
 www.amazon.com  | retail
 www.amazon.com  | merchant
 www.amazon.com  | and
 www.amazon.com  | provider
 www.amazon.com  | of
 www.amazon.com  | cloud
 www.amazon.com  | services
 www.vertica.com | World's
 www.vertica.com | fastest
 www.vertica.com | analytic
 www.vertica.com | database
(12 rows)
```

Notice that the number of rows and columns in the result table are different than the input table. This is one of the strengths of a UDTF.

TransformFunction implementation

The following code shows the StringTokenizer class.

```
class StringTokenizer : public TransformFunction
{
 virtual void processPartition(ServerInterface &srvInterface,
                               PartitionReader &inputReader,
                               PartitionWriter &outputWriter)
 {
  try {
   if (inputReader.getNumCols() != 1)
    vt_report_error(0, "Function only accepts 1 argument, but %zu provided", inputReader.getNumCols());

   do {
    const VString &sentence = inputReader.getStringRef(0);

    // If input string is NULL, then output is NULL as well
    if (sentence.isNull())
     {
      VString &word = outputWriter.getStringRef(0);
      word.setNull();
      outputWriter.next();
     }
    else
     {
      // Otherwise, let's tokenize the string and output the words
      std::string tmp = sentence.str();
      std::istringstream ss(tmp);

      do
       {
        std::string buffer;
        ss >> buffer;

        // Copy to output
        if (!buffer.empty()) {
         VString &word = outputWriter.getStringRef(0);
         word.copy(buffer);
         outputWriter.next();
        }
       } while (ss);
     }
   } while (inputReader.next() && !isCanceled());
  } catch(std::exception& e) {
   // Standard exception. Quit.
   vt_report_error(0, "Exception while processing partition: [%s]", e.what());
  }
 }
};
```

The processPartition() function in this example follows a pattern that you will follow in your own UDTF: it loops over all rows in the table partition that Vertica sends it, processing each row and checking for cancellation before advancing. For UDTFs you do not have to actually process every row. You can exit your function without having read all of the input without any issues. You may choose to do this if your UDTF is performing some sort search or some other operation where it can determine that the rest of the input is unneeded.

In this example, processPartition() first extracts the VString containing the text from the PartitionReader object. The VString class represents a Vertica string value (VARCHAR or CHAR). If there is input, it then tokenizes it and adds it to the output using the PartitionWriter object.

Similarly to reading input columns, the PartitionWriter class has functions for writing each type of data to the output row. In this case, the example calls the PartitionWriter object's getStringRef() function to allocate a new VString object to hold the token to output for the first column, and then copies the token's value into the VString .

TranformFunctionFactory implementation

The following code shows the factory class.

```
class TokenFactory : public TransformFunctionFactory
{
 // Tell Vertica that we take in a row with 1 string, and return a row with 1 string
 virtual void getPrototype(ServerInterface &srvInterface, ColumnTypes &argTypes, ColumnTypes &returnType)
 {
  argTypes.addVarchar();
  returnType.addVarchar();
 }

 // Tell Vertica what our return string length will be, given the input
 // string length
 virtual void getReturnType(ServerInterface &srvInterface,
                            const SizedColumnTypes &inputTypes,
                            SizedColumnTypes &outputTypes)
 {
  // Error out if we're called with anything but 1 argument
  if (inputTypes.getColumnCount() != 1)
   vt_report_error(0, "Function only accepts 1 argument, but %zu provided", inputTypes.getColumnCount());

  int input_len = inputTypes.getColumnType(0).getStringLength();

  // Our output size will never be more than the input size
  outputTypes.addVarchar(input_len, "words");
 }

 virtual TransformFunction *createTransformFunction(ServerInterface &srvInterface)
 { return vt_createFuncObject<StringTokenizer>(srvInterface.allocator); }

};
```

In this example:

- The UDTF takes a VARCHAR column as input. To define the input column, getPrototype() calls addVarchar() on the ColumnTypes object that represents the input table.
- The UDTF returns a VARCHAR as output. The getPrototype() function calls addVarchar() to define the output table.

This example must return the maximum length of the VARCHAR output column. It sets the length to the length of the input string. This is a safe value, because the output will never be longer than the input string. It also sets the name of the VARCHAR output column to "words".

> **Note**
> You are not required to supply a name for an output column in this function. However, it is a best practice to do so. If you do not name an output column, getReturnType() sets the column name to "". The SQL statements that call your UDTF must provide aliases for any unnamed columns to access them or else they return an error. From a usability standpoint, it is easier for you to supply the column names here once. The alternative is to force all of the users of your function to supply their own column names for each call to the UDTF.

The implementation of the createTransformFunction() function in the example is boilerplate code. It just calls the vt_returnFuncObj macro with the name of the TransformFunction class associated with this factory class. This macro takes care of instantiating a copy of the TransformFunction class that Vertica can use to process data.

The RegisterFactory macro

The final step in creating your UDTF is to call the RegisterFactory macro. This macro ensures that your factory class is instantiated when Vertica loads the shared library containing your UDTF. Having your factory class instantiated is the only way that Vertica can find your UDTF and determine what its inputs and outputs are.

The RegisterFactory macro just takes the name of your factory class:

```
RegisterFactory(TokenFactory);
```

Python example: string tokenizer

The following example shows a transform function that breaks an input string into tokens (based on whitespace). It is similar to the tokenizer examples for C++ and Java.

Loading and using the example

Create the library and function:

```
=> CREATE LIBRARY pyudtf AS '/home/dbadmin/udx/tokenize.py' LANGUAGE 'Python';
CREATE LIBRARY
=> CREATE TRANSFORM FUNCTION tokenize AS NAME 'StringTokenizerFactory' LIBRARY pyudtf;
CREATE TRANSFORM FUNCTION
```

You can then use the function in SQL statements, for example:

```
=> CREATE TABLE words (w VARCHAR);
CREATE TABLE
=> COPY words FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> this is a test of the python udtf
>> \.

=> SELECT tokenize(w) OVER () FROM words;
  token
----------
 this
 is
 a
 test
 of
 the
 python
 udtf
(8 rows)
```

Setup

All Python UDxs must import the Vertica SDK.

```
import vertica_sdk
```

UDTF Python code

The following code defines the tokenizer and its factory.

```
class StringTokenizer(vertica_sdk.TransformFunction):
    """
    Transform function which tokenizes its inputs.
    For each input string, each of the whitespace-separated tokens of that
    string is produced as output.
    """
    def processPartition(self, server_interface, input, output):
        while True:
            for token in input.getString(0).split():
                output.setString(0, token)
                output.next()
            if not input.next():
                break


class StringTokenizerFactory(vertica_sdk.TransformFunctionFactory):
    def getPrototype(self, server_interface, arg_types, return_type):
        arg_types.addVarchar()
        return_type.addVarchar()
    def getReturnType(self, server_interface, arg_types, return_type):
        return_type.addColumn(arg_types.getColumnType(0), "tokens")
    def createTransformFunction(cls, server_interface):
        return StringTokenizer()
```

R example: log tokenizer

The LogTokenizer transform function reads a varchar from a table, a log message. It then tokenizes each of the log messages, returning each of the tokens.

You can find more UDx examples in the Vertica Github repository, https://github.com/vertica/UDx-Examples .

Load the function and library

Create the library and the function.

```
=> CREATE OR REPLACE LIBRARY rLib AS 'log_tokenizer.R' LANGUAGE 'R';
CREATE LIBRARY
=> CREATE OR REPLACE TRANSFORM FUNCTION LogTokenizer AS LANGUAGE 'R' NAME 'LogTokenizerFactory' LIBRARY rLib FENCED;
CREATE FUNCTION
```

Querying data with the function

The following query shows how you can run a query with the UDTF.

```
=> SELECT machine,
          LogTokenizer(error_log USING PARAMETERS spliton = ' ') OVER(PARTITION BY machine)
     FROM error_logs;
 machine |  Token
---------+---------
 node001 | ERROR
 node001 | 345
 node001 | -
 node001 | Broken
 node001 | pipe
 node001 | WARN
 node001 | -
 node001 | Nearly
 node001 | filled
 node001 | disk
 node002 | ERROR
 node002 | 111
 node002 | -
 node002 | Flooded
 node002 | roads
 node003 | ERROR
 node003 | 222
 node003 | -
 node003 | Plain
 node003 | old
 node003 | broken
(21 rows)
```

UDTF R code

```
LogTokenizer <- function(input.data.frame, parameters.data.frame) {
  # Take the spliton parameter passed by the user and assign it to a variable
  # in the function so we can use that as our tokenizer.
  if ( is.null(parameters.data.frame['spliton']) ) {
    stop("NULL value for spliton! Token cannot be NULL.")
  } else {
    split.on <- as.character(parameters.data.frame['spliton'])
  }
  # Tokenize the string.
  tokens <- vector(length=0)
  for ( string in input.data.frame[, 1] ) {
    tokenized.string <- strsplit(string, split.on)
    for ( token in tokenized.string ) {
      tokens <- append(tokens, token)
    }
  }
  final.output <- data.frame(tokens)
  return(final.output)
}

LogTokenizerFactory <- function() {
  list(name     = LogTokenizer,
       udxtype = c("transform"),
       intype  = c("varchar"),
       outtype = c("varchar"),
       outtypecallback=LogTokenizerReturn,
       parametertypecallback=LogTokenizerParameters)
}



LogTokenizerParameters <- function() {
  parameters <- list(datatype = c("varchar"),
                     length   = c("NA"),
                     scale    = c("NA"),
                     name     = c("spliton"))
  return(parameters)
}

LogTokenizerReturn <- function(arg.data.frame, parm.data.frame) {
  output.return.type <- data.frame(datatype = rep(NA,1),
                                   length   = rep(NA,1),
                                   scale    = rep(NA,1),
                                   name     = rep(NA,1))
  output.return.type$datatype <- c("varchar")
  output.return.type$name <- c("Token")
  return(output.return.type)
}
```

C++ example: multi-phase indexer

The following code fragment is from the InvertedIndex UDTF example distributed with the Vertica SDK. It demonstrates subclassing the MultiPhaseTransformFunctionFactory including two TransformFunctionPhase subclasses that define the two phases in this UDTF.

```
class InvertedIndexFactory : public MultiPhaseTransformFunctionFactory

public:
  /**
   * Extracts terms from documents.
   */
  class ForwardIndexPhase : public TransformFunctionPhase
  {
```

```
{
  virtual void getReturnType(ServerInterface &srvInterface,
                             const SizedColumnTypes &inputTypes,
                             SizedColumnTypes &outputTypes)
  {
    // Sanity checks on input we've been given.
    // Expected input: (doc_id INTEGER, text VARCHAR)
    vector<size_t> argCols;
    inputTypes.getArgumentColumns(argCols);
    if (argCols.size() < 2 ||
        !inputTypes.getColumnType(argCols.at(0)).isInt() ||
        !inputTypes.getColumnType(argCols.at(1)).isVarchar())
        vt_report_error(0, "Function only accepts two arguments"
                        "(INTEGER, VARCHAR)");
    // Output of this phase is:
    //   (term_freq INTEGER) OVER(PBY term VARCHAR OBY doc_id INTEGER)
    // Number of times term appears within a document.
    outputTypes.addInt("term_freq");
    // Add analytic clause columns: (PARTITION BY term ORDER BY doc_id).
    // The length of any term is at most the size of the entire document.
    outputTypes.addVarcharPartitionColumn(
        inputTypes.getColumnType(argCols.at(1)).getStringLength(),
        "term");
    // Add order column on the basis of the document id's data type.
    outputTypes.addOrderColumn(inputTypes.getColumnType(argCols.at(0)),
                               "doc_id");
  }
  virtual TransformFunction *createTransformFunction(ServerInterface
      &srvInterface)
  { return vt_createFuncObj(srvInterface.allocator, ForwardIndexBuilder); }
};
/**
 * Constructs terms' posting lists.
 */
class InvertedIndexPhase : public TransformFunctionPhase
{
  virtual void getReturnType(ServerInterface &srvInterface,
                             const SizedColumnTypes &inputTypes,
                             SizedColumnTypes &outputTypes)
  {
    // Sanity checks on input we've been given.
    // Expected input:
    //   (term_freq INTEGER) OVER(PBY term VARCHAR OBY doc_id INTEGER)
    vector<size_t> argCols;
    inputTypes.getArgumentColumns(argCols);
    vector<size_t> pByCols;
    inputTypes.getPartitionByColumns(pByCols);
    vector<size_t> oByCols;
    inputTypes.getOrderByColumns(oByCols);
    if (argCols.size() != 1 || pByCols.size() != 1 || oByCols.size() != 1 ||
        !inputTypes.getColumnType(argCols.at(0)).isInt() ||
        !inputTypes.getColumnType(pByCols.at(0)).isVarchar() ||
        !inputTypes.getColumnType(oByCols.at(0)).isInt())
        vt_report_error(0, "Function expects an argument (INTEGER) with "
                        "analytic clause OVER(PBY VARCHAR OBY INTEGER)");
    // Output of this phase is:
    //   (term VARCHAR, doc_id INTEGER, term_freq INTEGER, corp_freq INTEGER).
    outputTypes.addVarchar(inputTypes.getColumnType(
                           pByCols.at(0)).getStringLength(),"term");
    outputTypes.addInt("doc_id");
    // Number of times term appears within the document.
    outputTypes.addInt("term_freq");
```

```
        // Number of documents where the term appears in.
        outputTypes.addInt("corp_freq");
    }

    virtual TransformFunction *createTransformFunction(ServerInterface
            &srvInterface)
    { return vt_createFuncObj(srvInterface.allocator, InvertedIndexBuilder); }
};
ForwardIndexPhase fwardIdxPh;
InvertedIndexPhase invIdxPh;
virtual void getPhases(ServerInterface &srvInterface,
     std::vector<TransformFunctionPhase *> &phases)
{
    fwardIdxPh.setPrepass(); // Process documents wherever they're originally stored.
    phases.push_back(&fwardIdxPh);
    phases.push_back(&invIdxPh);
}
virtual void getPrototype(ServerInterface &srvInterface,
                          ColumnTypes &argTypes,
                          ColumnTypes &returnType)
{
    // Expected input: (doc_id INTEGER, text VARCHAR).
    argTypes.addInt();
    argTypes.addVarchar();
    // Output is: (term VARCHAR, doc_id INTEGER, term_freq INTEGER, corp_freq INTEGER)
    returnType.addVarchar();
    returnType.addInt();
    returnType.addInt();
    returnType.addInt();
}
};
RegisterFactory(InvertedIndexFactory);
```

Most of the code in this example is similar to the code in a TransformFunctionFactory class:

- Both TransformFunctionPhase subclasses implement the getReturnType() function, which describes the output of each stage. This is the similar to the getReturnType() function from the TransformFunctionFactory class. However, this function also lets you control how the data is partitioned and ordered between each phase of your multi-phase UDTF.
  The first phase calls SizedColumnTypes::addVarcharPartitionColumn() (rather than just addVarcharColumn()) to set the phase's output table to be partitioned by the column containing the extracted words. It also calls SizedColumnTypes::addOrderColumn() to order the output table by the document ID column. It calls this function instead of one of the data-type-specific functions (such as addIntOrderColumn()) so it can pass the data type of the original column through to the output column.

  > **Note**
  > Any order by column or partition by column set by the final phase of the UDTF in its getReturnType() function is ignored. Its output is returned to the initiator node rather than partitioned and reordered then sent to another phase.

- The MultiPhaseTransformFunctionFactory class implements the getPrototype() function, that defines the schemas for the input and output of the multi-phase UDTF. This function is the same as the TransformFunctionFactory::getPrototype() function.

The unique function implemented by the MultiPhaseTransformFunctionFactory class is getPhases() . This function defines the order in which the phases are executed. The fields that represent the phases are pushed into this vector in the order they should execute.

The MultiPhaseTransformFunctionFactory.getPhases() function is also where you flag the first phase of the UDTF as operating on data stored locally on the node (called a "pre-pass" phase) rather than on data partitioned across all nodes. Using this option increases the efficiency of your multi-phase UDTF by avoiding having to move significant amounts of data around the Vertica cluster.

> **Note**

> Only the first phase of your UDTF can be a pre-pass phase. You cannot have multiple pre-pass phases, and no later phase can be a pre-pass phase.

To mark the first phase as pre-pass, you call the TransformFunctionPhase::setPrepass() function of the first phase's TransformFunctionPhase instance from within the getPhase() function.

Notes

- You need to ensure that the output schema of each phase matches the input schema expected by the next phase. In the example code, each TransformFunctionPhase::getReturnType() implementation performs a sanity check on its input and output schemas. Your TransformFunction subclasses can also perform these checks in their processPartition() function.
- There is no built-in limit on the number of phases that your multi-phase UDTF can have. However, more phases use more resources. When running in fenced mode, Vertica may terminate UDTFs that use too much memory. See Resource use for C++ UDxs .

Python example: multi-phase calculation

The following example shows a multi-phase transform function that computes the average value on a column of numbers in an input table. It first defines two transform functions, and then defines a factory that creates the phases using them.

See AvgMultiPhaseUDT.py in the examples distribution for the complete code.

Loading and using the example

Create the library and function:

```
=> CREATE LIBRARY pylib_avg AS '/home/dbadmin/udx/AvgMultiPhaseUDT.py' LANGUAGE 'Python';
CREATE LIBRARY
=> CREATE TRANSFORM FUNCTION myAvg AS NAME 'MyAvgFactory' LIBRARY pylib_avg;
CREATE TRANSFORM FUNCTION
```

You can then use the function in SELECT statements:

```
=> CREATE TABLE IF NOT EXISTS numbers(num FLOAT);
CREATE TABLE

=> COPY numbers FROM STDIN delimiter ',';
1
2
3
4
\.

=> SELECT myAvg(num) OVER() FROM numbers;
 average | ignored_rows | total_rows
---------+--------------+------------
     2.5 |            0 |          4
(1 row)
```

Setup

All Python UDxs must import the Vertica SDK. This example also imports another library.

```
import vertica_sdk
import math
```

Component transform functions

A multi-phase transform function must define two or more TransformFunction subclasses to be used in the phases. This example uses two classes: LocalCalculation , which does calculations on local partitions, and GlobalCalculation , which aggregates the results of all LocalCalculation instances to calculate a final result.

In both functions, the calculation is done in the processPartition() function:

```
class LocalCalculation(vertica_sdk.TransformFunction):
    """
    This class is the first phase and calculates the local values for sum, ignored_rows and total_rows.
    """

    def setup(self, server_interface, col_types):
        server_interface.log("Setup: Phase0")
        self.local_sum = 0.0
        self.ignored_rows = 0
        self.total_rows = 0

    def processPartition(self, server_interface, input, output):
        server_interface.log("Process Partition: Phase0")

        while True:
            self.total_rows += 1

            if input.isNull(0) or math.isinf(input.getFloat(0)) or math.isnan(input.getFloat(0)):
                # Null, Inf, or Nan is ignored
                self.ignored_rows += 1
            else:
                self.local_sum += input.getFloat(0)

            if not input.next():
                break

        output.setFloat(0, self.local_sum)
        output.setInt(1, self.ignored_rows)
        output.setInt(2, self.total_rows)
        output.next()

class GlobalCalculation(vertica_sdk.TransformFunction):
    """
    This class is the second phase and aggregates the values for sum, ignored_rows and total_rows.
    """

    def setup(self, server_interface, col_types):
        server_interface.log("Setup: Phase1")
        self.global_sum = 0.0
        self.ignored_rows = 0
        self.total_rows = 0

    def processPartition(self, server_interface, input, output):
        server_interface.log("Process Partition: Phase1")

        while True:
            self.global_sum += input.getFloat(0)
            self.ignored_rows += input.getInt(1)
            self.total_rows += input.getInt(2)

            if not input.next():
                break

        average = self.global_sum / (self.total_rows - self.ignored_rows)

        output.setFloat(0, average)
        output.setInt(1, self.ignored_rows)
        output.setInt(2, self.total_rows)
        output.next()
```

Multi-phase factory

A MultiPhaseTransformFunctionFactory ties together the individual functions as phases. The factory defines a TransformFunctionPhase for each function. Each phase defines createTransformFunction() , which calls the constructor for the corresponding TransformFunction , and getReturnType() .

The first phase, LocalPhase , follows.

```
class MyAvgFactory(vertica_sdk.MultiPhaseTransformFunctionFactory):
    """ Factory class """

    class LocalPhase(vertica_sdk.TransformFunctionPhase):
        """ Phase 1 """
        def getReturnType(self, server_interface, input_types, output_types):
            # sanity check
            number_of_cols = input_types.getColumnCount()
            if (number_of_cols != 1 or not input_types.getColumnType(0).isFloat()):
                raise ValueError("Function only accepts one argument (FLOAT)")

            output_types.addFloat("local_sum");
            output_types.addInt("ignored_rows");
            output_types.addInt("total_rows");

        def createTransformFunction(cls, server_interface):
            return LocalCalculation()
```

The second phase, GlobalPhase , does not check its inputs because the first phase already did. As with the first phase, createTransformFunction merely constructs and returns the corresponding TransformFunction .

```
    class GlobalPhase(vertica_sdk.TransformFunctionPhase):
        """ Phase 2 """
        def getReturnType(self, server_interface, input_types, output_types):
            output_types.addFloat("average");
            output_types.addInt("ignored_rows");
            output_types.addInt("total_rows");

        def createTransformFunction(cls, server_interface):
            return GlobalCalculation()
```

After defining the TransformFunctionPhase subclasses, the factory instantiates them and chains them together in getPhases() .

```
    ph0Instance = LocalPhase()
    ph1Instance = GlobalPhase()

    def getPhases(cls, server_interface):
        cls.ph0Instance.setPrepass()
        phases = [cls.ph0Instance, cls.ph1Instance]
        return phases
```

Python example: count elements

The following example details a UDTF that takes a partition of arrays, computes the count of each distinct array element in the partition, and outputs each element and its count as a row value. You can call the function on tables that contain multiple partitions of arrays.

The complete source code is in /opt/vertica/sdk/examples/python/TransformFunctions.py .

Loading and using the example

Load the library and create the transform function as follows:

```
=> CREATE OR REPLACE LIBRARY TransformFunctions AS '/home/dbadmin/examples/python/TransformFunctions.py' LANGUAGE 'Python';

=> CREATE TRANSFORM FUNCTION CountElements AS LANGUAGE 'Python' NAME 'countElementsUDTFactory' LIBRARY TransformFunctions;
```

You can create some data and then call the function on it, for example:

```
=> CREATE TABLE orders (storeID int, productIDs array[int]);
CREATE TABLE

=> INSERT INTO orders VALUES
   (1, array[101, 102, 103]),
   (1, array[102, 104]),
   (1, array[101, 102, 102, 201, 203]),
   (2, array[101, 202, 203, 202, 203]),
   (2, array[203]),
   (2, array[]);
OUTPUT
--------
6
(1 row)

=> COMMIT;
COMMIT

=> SELECT storeID, CountElements(productIDs) OVER (PARTITION BY storeID) FROM orders;
storeID |       element_count
--------+---------------------------
      1 | {"element":101,"count":2}
      1 | {"element":102,"count":4}
      1 | {"element":103,"count":1}
      1 | {"element":104,"count":1}
      1 | {"element":201,"count":1}
      1 | {"element":202,"count":1}
      2 | {"element":101,"count":1}
      2 | {"element":202,"count":2}
      2 | {"element":203,"count":3}
(9 rows)
```

Setup

All Python UDxs must import the Vertica SDK library:

```
import vertica_sdk
```

Factory implementation

The getPrototype() method declares that the inputs and outputs can be of any type, which means that type enforcement must be done elsewhere:

```
def getPrototype(self, srv_interface, arg_types, return_type):
    arg_types.addAny()
    return_type.addAny()
```

getReturnType() validates that the only argument to the function is an array and that the return type is a row with 'element' and 'count' fields:

```
def getReturnType(self, srv_interface, arg_types, return_type):

    if arg_types.getColumnCount() != 1:
        srv_interface.reportError(1, 'countElements UDT should take exactly one argument')

    if not arg_types.getColumnType(0).isArrayType():
        srv_interface.reportError(2, 'Argument to countElements UDT should be an ARRAY')

    retRowFields = vertica_sdk.SizedColumnTypes.makeEmpty()
    retRowFields.addColumn(arg_types.getColumnType(0).getElementType(), 'element')
    retRowFields.addInt('count')
    return_type.addRowType(retRowFields, 'element_count')
```

The countElementsUDTFactory class also contains a createTransformFunction() method that instantiates and returns the transform function.

Function implementation

The processBlock() method is called with a BlockReader and a BlockWriter , named arg_reader and res_writer respectively. The function loops through all the input arrays in a partition and uses a dictionary to collect the frequency of each element. To access elements of each input array, the method instantiates an ArrayReader . After collecting the element counts, the function writes each element and its count to a row. This process is repeated for each partition.

```
def processPartition(self, srv_interface, arg_reader, res_writer):

    elemCounts = dict()
    # Collect element counts for entire partition
    while (True):
        if not arg_reader.isNull(0):
            arr = arg_reader.getArray(0)
            for elem in arr:
                elemCounts[elem] = elemCounts.setdefault(elem, 0) + 1

        if not arg_reader.next():
            break

    # Write at least one value for each partition
    if len(elemCounts) == 0:
        elemCounts[None] = 0

    # Write out element counts as (element, count) pairs
    for pair in elemCounts.items():
        res_writer.setRow(0, pair)
        res_writer.next()
```

Python example: explode

The following example details a UDTF that accepts a one-dimensional array as input and outputs each element of the array as a separate row, similar to functions commonly known as EXPLODE . Because this UDTF always accepts one array as input, you can explicitly mark it as a one-to-many UDTF in the factory function, which helps Vertica optimize query plans and allows users to write SELECT queries that include any expression and do not require an OVER clause.

The complete source code is in /opt/vertica/sdk/examples/python/TransformFunctions.py .

Loading and using the example

Load the library and create the transform function as follows:

```
=> CREATE OR REPLACE LIBRARY PyTransformFunctions AS '/opt/vertica/sdk/examples/python/TransformFunctions.py' LANGUAGE 'Python';
CREATE LIBRARY

=> CREATE TRANSFORM FUNCTION py_explode AS LANGUAGE 'Python' NAME 'ExplodeFactory' LIBRARY TransformFunctions;
CREATE TRANSFORM FUNCTION
```

You can then use the function in SQL statements, for example:

```
=> CREATE TABLE reviews (id INTEGER PRIMARY KEY, sentiment VARCHAR(16), review ARRAY[VARCHAR(16), 32]);
CREATE TABLE

=> INSERT INTO reviews VALUES(1, 'Very Negative', string_to_array('This was the worst restaurant I have ever had the misfortune of eating at' USING
PARAMETERS collection_delimiter = ' ')),
    (2, 'Neutral', string_to_array('This restaurant is pretty decent' USING PARAMETERS collection_delimiter = ' ')),
    (3, 'Very Positive', string_to_array('Best restaurant in the Western Hemisphere' USING PARAMETERS collection_delimiter = ' ')),
    (4, 'Positive', string_to_array('Prices low for the area' USING PARAMETERS collection_delimiter = ' '));
OUTPUT
--------
4
(1 row)

=> COMMIT;
COMMIT

=> SELECT id, sentiment, py_explode(review) FROM reviews; --no OVER clause because "is_exploder = True", see below
 id |   sentiment   |  element
----+---------------+------------
  1 | Very Negative | This
  1 | Very Negative | was
  1 | Very Negative | the
  1 | Very Negative | worst
  1 | Very Negative | restaurant
  1 | Very Negative | I
  1 | Very Negative | have
...
  3 | Very Positive | Western
  3 | Very Positive | Hemisphere
  4 | Positive      | Prices
  4 | Positive      | low
  4 | Positive      | for
  4 | Positive      | the
  4 | Positive      | area
(30 rows)
```

Setup

All Python UDxs must import the Vertica SDK library:

```
import vertica_sdk
```

Factory implementation

The following code shows the ExplodeFactory class.

```
class ExplodeFactory(vertica_sdk.TransformFunctionFactory):
    def getPrototype(self, srv_interface, arg_types, return_type):
        arg_types.addAny()
        return_type.addAny()

    def getTransformFunctionProperties(cls, server_interface, arg_types):
        props = vertica_sdk.TransformFunctionFactory.Properties()
        props.is_exploder = True
        return props

    def getReturnType(self, srv_interface, arg_types, return_type):
        if arg_types.getColumnCount() != 1:
            srv_interface.reportError(1, 'explode UDT should take exactly one argument')
        if not arg_types.getColumnType(0).isArrayType():
            srv_interface.reportError(2, 'Argument to explode UDT should be an ARRAY')

        return_type.addColumn(arg_types.getColumnType(0).getElementType(), 'element')

    def createTransformFunction(cls, server_interface):
        return Explode()
```

In this example:

- The getTransformFunctionProperties method sets the is_exploder class property to True, explicitly marking the UDTF as one-to-many. This indicates that the function uses an OVER(PARTITION ROW) clause by default and thereby removes the need to specify an OVER clause when calling the UDTF. With is_exploder set to True, users can write SELECT queries that include any expression, unlike queries that use PARTITION BEST or PARTITION NODES.
- The getReturnType method verifies that the input contains only one argument and is of type ARRAY. The method also sets the return type to that of the elements in the input array.

Function implementation

The following code shows the Explode class:

```
class Explode(vertica_sdk.TransformFunction):
"""
Transform function that turns an array into one row
for each array element.
"""
    def processPartition(self, srv_interface, arg_reader, res_writer):
        while True:
            arr = arg_reader.getArrayReader(0)
            for elt in arr:
                res_writer.copyRow(elt)
                res_writer.next()
            if not arg_reader.next():
                break;
```

The processPartition() method accepts a single row of the input data, processes each element of the array, and then breaks the loop. The method accesses the elements of the array with an ArrayReader object and then uses an ArrayWriter object to write each element of the array to a separate output row. The UDTF calls processPartition() for each row of the input data.

See also

- TransformFunctionFactory class
- EXPLODE
- Window partitioning
- Partitioning options for UDTFs

User-defined load (UDL)

COPY offers extensive options and settings to control how to load data. However, you may find that these options do not suit the type of data load that you want to perform. The user-defined load (UDL) feature lets you develop one or more functions that change how the COPY statement operates. You can create custom libraries using the Vertica SDK to handle various steps in the loading process. .

You use three types of UDL functions during development, one for each stage of the data-load process:

- User-defined source (UDSource): Controls how COPY obtains the data it loads into the database. For example, COPY might obtain data by fetching it through HTTP or cURL. Up to one UDSource reads data from a file or input stream. Your UDSource can read from more than one source, but COPY invokes only one UDSource.
  API support: C++, Java.
- User-defined filter (UDFilter): Preprocesses the data. For example, a filter might unzip a file or convert UTF-16 to UTF-8. You can chain multiple user-defined filters together, for example unzipping and then converting.
  API support: C++, Java, Python.
- User-defined parser (UDParser): Up to one parser parses the data into tuples that are ready to be inserted into a table. For example, a parser could extract data from an XML-like format. You can optionally define a user-defined chunker (UDChunker, C++ only), to have the parser perform parallel parsing.
  API support: C++, Java, Python.

After the final step, COPY inserts the data into a table, or rejects it if the format is incorrect.

In this section

- User-defined source
- User-defined filter
- User-defined parser
- Load parallelism
- Continuous load
- Buffer classes

User-defined source

A user-defined source allows you to process a source of data using a method that is not built into Vertica. For example, you can write a user-defined source to access the data from an HTTP source using cURL. While a given COPY statement can use specify only one user-defined source statement, the source function itself can pull data from multiple sources.

The UDSource class acquires data from an external source. It reads data from an input stream and produces an output stream to be filtered and parsed. If you implement a UDSource , you must also implement a corresponding SourceFactory .

In this section

- UDSource class
- SourceFactory class
- C++ example: CurlSource
- C++ example: concurrent load
- Java example: FileSource

UDSource class

You can subclass the UDSource class when you need to load data from a source type that COPY does not already support.

Each instance of your UDSource subclass reads from a single data source. Examples of a single data source are a single file or the results of a single function call to a RESTful web application.

UDSource methods

Your UDSource subclass must override process() or processWithMetadata() :

> **Note**
> processWithMetadata() is available only for user-defined extensions (UDxs) written in the C++ programming language.

- process() reads the raw input stream as one large file. If there are any errors or failures, the entire load fails.
- processWithMetadata() is useful when the data source has metadata about record boundaries available in some structured format that's separate from the data payload. With this interface, the source emits a record length for each record in addition to the data.
  By implementing processWithMetadata() instead of process() in each phase, you can retain this record length metadata throughout the load stack, which enables a more efficient parse that can recover from errors on a per-message basis, rather than a per-file or per-source basis. KafkaSource and the Kafka parsers (KafkaAvroParser , KafkaJSONParser , and KafkaParser) use this mechanism to support per-Kafka-message rejections when individual Kafka messages are cannot be parsed.

> **Note**
> To implement processWithMetadata() , you must override useSideChannel() to return true .

Additionally, you can override the other UDSource class methods.

Source execution

The following sections detail the execution sequence each time a user-defined source is called. The following example overrides the process() method.

Setting Up

COPY calls setup() before the first time it calls process() . Use setup() to perform any necessary setup steps to access the data source. This method establishes network connections, opens files, and similar tasks that need to be performed before the UDSource can read data from the data source. Your object might be destroyed and re-created during use, so make sure that your object is restartable.

Processing a Source

COPY calls process() repeatedly during query execution to read data and write it to the DataBuffer passed as a parameter. This buffer is then passed to the first filter.

If the source runs out of input, or fills the output buffer, it must return the value StreamState.OUTPUT_NEEDED . When Vertica gets this return value, it will call the method again. This second call occurs after the output buffer has been processed by the next stage in the data-load process. Returning StreamState.DONE indicates that all of the data from the source has been read.

The user can cancel the load operation, which aborts reading.

Tearing Down

COPY calls destroy() after the last time that process() is called. This method frees any resources reserved by the setup() or process() methods, such as file handles or network connections that the setup() method allocated.

Accessors

A source can define two accessors, getSize() and getUri() .

COPY might call getSize() to estimate the number of bytes of data to be read before calling process() . This value is an estimate only and is used to indicate the file size in the LOAD_STREAMS table. Because Vertica can call this method before calling setup() , getSize() must not rely on any resources allocated by setup() .

This method should not leave any resources open. For example, do not save any file handles opened by getSize() for use by the process() method. Doing so can exhaust the available resources, because Vertica calls getSize() on all instances of your UDSource subclass before any data is loaded. If many data sources are being opened, these open file handles could use up the system's supply of file handles. Thus, none would remain available to perform the actual data load.

Vertica calls getUri() during execution to update status information about which resources are currently being loaded. It returns the URI of the data source being read by this UDSource .

API

C++

Java

The UDSource API provides the following methods for extension by subclasses:

```
virtual void setup(ServerInterface &srvInterface);

virtual bool useSideChannel();

virtual StreamState process(ServerInterface &srvInterface, DataBuffer &output)=0;

virtual StreamState processWithMetadata(ServerInterface &srvInterface, DataBuffer &output, LengthBuffer &output_len

virtual void cancel(ServerInterface &srvInterface);

virtual void destroy(ServerInterface &srvInterface);

virtual vint getSize();

virtual std::string getUri();
```

SourceFactory class

If you write a source, you must also write a source factory. Your subclass of the SourceFactory class is responsible for:

- Performing the initial validation of the parameters passed to your UDSource .
- Setting up any data structures your UDSource instances need to perform their work. This information can include recording which nodes will read which data source.
- Creating one instance of your UDSource subclass for each data source (or portion thereof) that your function reads on each host.

The simplest source factory creates one UDSource instance per data source per executor node. You can also use multiple concurrent UDSource instances on each node. This behavior is called *concurrent load* . To support both options, SourceFactory has two versions of the method that creates the sources. You must implement exactly one of them.

Source factories are singletons. Your subclass must be stateless, with no fields containing data. The subclass also must not modify any global variables.

SourceFactory methods

The SourceFactory class defines several methods. Your class must override prepareUDSources() ; it may override the other methods.

Setting up

Vertica calls plan() once on the initiator node to perform the following tasks:

- Check the parameters the user supplied to the function call in the COPY statement and provide error messages if there are any issues. You can read the parameters by getting a ParamReader object from the instance of ServerInterface passed into the plan() method.
- Decide which hosts in the cluster will read the data source. How you divide up the work depends on the source your function is reading. Some sources can be split across many hosts, such as a source that reads data from many URLs. Others, such as an individual local file on a host's file system, can be read only by a single specified host.
  You store the list of hosts to read the data source by calling the setTargetNodes() method on the NodeSpecifyingPlanContext object. This object is passed into your plan() method.
- Store any information that the individual hosts need to process the data sources in the NodeSpecifyingPlanContext instance passed to the plan() method. For example, you could store assignments that tell each host which data sources to process. The plan() method runs only on the initiator node, and the prepareUDSources() method runs on each host reading from a data source. Therefore, this object is the only means of communication between them.
  You store data in the NodeSpecifyingPlanContext by getting a ParamWriter object from the getWriter() method. You then write parameters by calling methods on the ParamWriter such as setString() .

  > **Note**
  > ParamWriter offers the ability to store only simple data types. For complex types, you must serialize the data in some manner and store it as a string or long string.

Creating sources

Vertica calls prepareUDSources() on all hosts that the plan() method selected to load data. This call instantiates and returns a list of UDSource subclass

instances. If you are not using concurrent load, return one UDSource for each of the sources that the host is assigned to process. If you are using concurrent load, use the version of the method that takes an ExecutorPlanContext as a parameter, and return as many sources as you can use. Your factory must implement exactly one of these methods.

> **Note**
> In the C++ API, the function that supports concurrent load is named prepareUDSourcesExecutor() . In the Java API the class provides two overloads of prepareUDSources() .

For concurrent load, you can find out how many threads are available on the node to run UDSource instances by calling getLoadConcurrency() on the ExecutorPlanContext that is passed in.

Defining parameters

Implement getParameterTypes() to define the names and types of parameters that your source uses. Vertica uses this information to warn callers about unknown or missing parameters. Vertica ignores unknown parameters and uses default values for missing parameters. While you should define the types and parameters for your function, you are not required to override this method.

Requesting threads for concurrent load

When a source factory creates sources on an executor node, by default, it creates one thread per source. If your sources can use multiple threads, implement getDesiredThreads() . Vertica calls this method before it calls prepareUDSources() , so you can also use it to decide how many sources to create. Return the number of threads your factory can use for sources. The maximum number of available threads is passed in, so you can take that into account. The value your method returns is a hint, not a guarantee; each executor node determines the number of threads to allocate. The FilePortionSourceFactory example implements this method; see C++ example: concurrent load .

You can allow your source to have control over parallelism, meaning that it can divide a single input into multiple load streams, by implementing isSourceApportionable() . Returning true from this method does not guarantee that the source *will* apportion the load. However, returning false indicates that it will not try to do so. See Apportioned load for more information.

Often, a SourceFactory that implements getDesiredThreads() also uses apportioned load. However, using apportioned load is not a requirement. A source reading from Kafka streams, for example, could use multiple threads without ssapportioning.

API

C++

Java

The SourceFactory API provides the following methods for extension by subclasses:

```
virtual void plan(ServerInterface &srvInterface, NodeSpecifyingPlanContext &planCtxt);

// must implement exactly one of prepareUDSources() or prepareUDSourcesExecutor()
virtual std::vector< UDSource * > prepareUDSources(ServerInterface &srvInterface,
        NodeSpecifyingPlanContext &planCtxt);

virtual std::vector< UDSource * > prepareUDSourcesExecutor(ServerInterface &srvInterface,
        ExecutorPlanContext &planCtxt);

virtual void getParameterType(ServerInterface &srvInterface,
        SizedColumnTypes &parameterTypes);

virtual bool isSourceApportionable();

ssize_t getDesiredThreads(ServerInterface &srvInterface,
        ExecutorPlanContext &planContext);
```

After creating your SourceFactory , you must register it with the RegisterFactory macro.

C++ example: CurlSource

The CurlSource example allows you to use cURL to open and read in a file over HTTP. The example provided is part of:

/opt/vertica/sdk/examples/SourceFunctions/cURL.cpp .

Source implementation

This example uses the helper library available in /opt/vertica/sdk/examples/HelperLibraries/ .

CurlSource loads the data in chunks. If the parser encounters an EndOfFile marker, then the process() method returns DONE. Otherwise, the method returns OUTPUT_NEEDED and processes another chunk of data. The functions included in the helper library (such as url_fread() and url_fopen()) are based on examples that come with the libcurl library. For an example, see http://curl.haxx.se/libcurl/c/fopen.html .

The setup() function opens a file handle and the destroy() function closes it. Both use functions from the helper library.

```
class CurlSource : public UDSource {private:
    URL_FILE *handle;
    std::string url;
    virtual StreamState process(ServerInterface &srvInterface, DataBuffer &output) {
        output.offset = url_fread(output.buf, 1, output.size, handle);
        return url_feof(handle) ? DONE : OUTPUT_NEEDED;
    }
public:
    CurlSource(std::string url) : url(url) {}
    void setup(ServerInterface &srvInterface) {
        handle = url_fopen(url.c_str(),"r");
    }
    void destroy(ServerInterface &srvInterface) {
        url_fclose(handle);
    }
};
```

Factory implementation

CurlSourceFactory produces CurlSource instances.

```
class CurlSourceFactory : public SourceFactory {public:
    virtual void plan(ServerInterface &srvInterface,
            NodeSpecifyingPlanContext &planCtxt) {
        std::vector<std::string> args = srvInterface.getParamReader().getParamNames();
        /* Check parameters */
        if (args.size() != 1 || find(args.begin(), args.end(), "url") == args.end()) {
            vt_report_error(0, "You must provide a single URL.");
        }
        /* Populate planData */
        planCtxt.getWriter().getStringRef("url").copy(
                                srvInterface.getParamReader().getStringRef("url"));

        /* Assign Nodes */
        std::vector<std::string> executionNodes = planCtxt.getClusterNodes();
        while (executionNodes.size() > 1) executionNodes.pop_back();
        // Only run on the first node in the list.
        planCtxt.setTargetNodes(executionNodes);
    }
    virtual std::vector<UDSource*> prepareUDSources(ServerInterface &srvInterface,
            NodeSpecifyingPlanContext &planCtxt) {
        std::vector<UDSource*> retVal;
        retVal.push_back(vt_createFuncObj(srvInterface.allocator, CurlSource,
                planCtxt.getReader().getStringRef("url").str()));
        return retVal;
    }
    virtual void getParameterType(ServerInterface &srvInterface,
                                  SizedColumnTypes &parameterTypes) {
        parameterTypes.addVarchar(65000, "url");
    }
};
RegisterFactory(CurlSourceFactory);
```

C++ example: concurrent load

The FilePortionSource example demonstrates the use of concurrent load. This example is a refinement of the FileSource example. Each input file is divided into portions and distributed to FilePortionSource instances. The source accepts a list of offsets at which to break the input into portions; if offsets are not provided, the source divides the input dynamically.

Concurrent load is handled in the factory, so this discussion focuses on FilePortionSourceFactory . The full code for the example is located in /opt/vertica/sdk/examples/ApportionLoadFunctions . The distribution also includes a Java version of this example.

Loading and using the example

Load and use the FilePortionSource example as follows.

```
=> CREATE LIBRARY FilePortionLib AS '/home/dbadmin/FP.so';

=> CREATE SOURCE FilePortionSource AS LANGUAGE 'C++'
-> NAME 'FilePortionSourceFactory' LIBRARY FilePortionLib;

=> COPY t WITH SOURCE FilePortionSource(file='g1/*.dat', nodes='initiator,e0,e1', offsets = '0,380000,820000');

=> COPY t WITH SOURCE FilePortionSource(file='g2/*.dat', nodes='e0,e1,e2', local_min_portion_size = 2097152);
```

Implementation

Concurrent load affects the source factory in two places, getDesiredThreads() and prepareUDSourcesExecutor() .

getDesiredThreads()

The getDesiredThreads() member function determines the number of threads to request. Vertica calls this member function on each executor node before calling prepareUDSourcesExecutor() .

The function begins by breaking an input file path, which might be a glob, into individual paths. This discussion omits those details. If apportioned load is not being used, then the function allocates one source per file.

```
virtual ssize_t getDesiredThreads(ServerInterface &srvInterface,
    ExecutorPlanContext &planCtxt) {
  const std::string filename = srvInterface.getParamReader().getStringRef("file").str();

  std::vector<std::string> paths;
  // expand the glob - at least one thread per source.
  ...

  // figure out how to assign files to sources
  const std::string nodeName = srvInterface.getCurrentNodeName();
  const size_t nodeId = planCtxt.getWriter().getIntRef(nodeName);
  const size_t numNodes = planCtxt.getTargetNodes().size();

  if (!planCtxt.canApportionSource()) {
    /* no apportioning, so the number of files is the final number of sources */
    std::vector<std::string> *expanded =
        vt_createFuncObject<std::vector<std::string> >(srvInterface.allocator, paths);
    /* save expanded paths so we don't have to compute expansion again */
    planCtxt.getWriter().setPointer("expanded", expanded);
    return expanded->size();
  }

  // ...
```

If the source can be apportioned, then getDesiredThreads() uses the offsets that were passed as arguments to the factory to divide the file into portions. It then allocates portions to available nodes. This function does not actually assign sources directly; this work is done to determine how many threads to request.

```
else if (srvInterface.getParamReader().containsParameter("offsets")) {

  // if the offsets are specified, then we will have a fixed number of portions per file.
  // Round-robin assign offsets to nodes.
  // ...

  /* Construct the portions that this node will actually handle.
   * This isn't changing (since the offset assignments are fixed),
   * so we'll build the Portion objects now and make them available
   * to prepareUDSourcesExecutor() by putting them in the ExecutorContext.
   *
   * We don't know the size of the last portion, since it depends on the file
   * size.  Rather than figure it out here we will indicate it with -1 and
   * defer that to prepareUDSourcesExecutor().
   */
  std::vector<Portion> *portions =
    vt_createFuncObject<std::vector<Portion>>(srvInterface.allocator);

  for (std::vector<size_t>::const_iterator offset = offsets.begin();
       offset != offsets.end(); ++offset) {
    Portion p(*offset);
    p.is_first_portion = (offset == offsets.begin());
    p.size = (offset + 1 == offsets.end() ? -1 : (*(offset + 1) - *offset));

    if ((offset - offsets.begin()) % numNodes == nodeId) {
      portions->push_back(p);
      srvInterface.log("FilePortionSource: assigning portion %ld: [offset = %lld, size = %lld]",
                       offset - offsets.begin(), p.offset, p.size);
    }
  }
```

The function now has all the portions and thus the number of portions:

```
  planCtxt.getWriter().setPointer("portions", portions);

  /* total number of threads we want is the number of portions per file, which is fixed */
  return portions->size() * expanded->size();
} // end of "offsets" parameter
```

If offsets were not provided, the function divides the file into portions dynamically, one portion per thread. This discussion omits the details of this computation. There is no point in requesting more threads than are available, so the function calls getMaxAllowedThreads() on the PlanContext (an argument to the function) to set an upper bound:

```
if (portions->size() >= planCtxt.getMaxAllowedThreads()) {
  return paths.size();
}
```

See the full example for the details of how this function divides the file into portions.

This function uses the vt_createFuncObject template to create objects. Vertica calls the destructors of returned objects created using this macro, but it does not call destructors for other objects like vectors. You must call these destructors yourself to avoid memory leaks. In this example, these calls are made in prepareUDSourcesExecutor() .

prepareUDSourcesExecutor()

The prepareUDSourcesExecutor() member function, like getDesiredThreads() , has separate blocks of code depending on whether offsets are provided. In both cases, the function breaks input into portions and creates UDSource instances for them.

If the function is called with offsets, prepareUDSourcesExecutor() calls prepareCustomizedPortions() . This function follows.

```
/* prepare portions as determined via the "offsets" parameter */
void prepareCustomizedPortions(ServerInterface &srvInterface,
                               ExecutorPlanContext &planCtxt,
                               std::vector<UDSource *> &sources,
                               const std::vector<std::string> &expandedPaths,
                               std::vector<Portion> &portions) {
  for (std::vector<std::string>::const_iterator filename = expandedPaths.begin();
       filename != expandedPaths.end(); ++filename) {
    /*
     * the "portions" vector contains the portions which were generated in
     * "getDesiredThreads"
     */
    const size_t fileSize = getFileSize(*filename);
    for (std::vector<Portion>::const_iterator portion = portions.begin();
         portion != portions.end(); ++portion) {
      Portion fportion(*portion);
      if (fportion.size == -1) {
        /* as described above, this means from the offset to the end */
        fportion.size = fileSize - portion->offset;
        sources.push_back(vt_createFuncObject<FilePortionSource>(srvInterface.allocator,
                          *filename, fportion));
      } else if (fportion.size > 0) {
        sources.push_back(vt_createFuncObject<FilePortionSource>(srvInterface.allocator,
                          *filename, fportion));
      }
    }
  }
```

If prepareUDSourcesExecutor() is called without offsets, then it must decide how many portions to create.

The base case is to use one portion per source. However, if extra threads are available, the function divides the input into more portions so that a source can process them concurrently. Then prepareUDSourcesExecutor() calls prepareGeneratedPortions() to create the portions. This function begins by calling getLoadConcurrency() on the plan context to find out how many threads are available.

```
void prepareGeneratedPortions(ServerInterface &srvInterface,
                              ExecutorPlanContext &planCtxt,
                              std::vector<UDSource *> &sources,
                              std::map<std::string, Portion> initialPortions) {

  if ((ssize_t) initialPortions.size() >= planCtxt.getLoadConcurrency()) {
  /* all threads will be used, don't bother splitting into portions */

    for (std::map<std::string, Portion>::const_iterator file = initialPortions.begin();
         file != initialPortions.end(); ++file) {
      sources.push_back(vt_createFuncObject<FilePortionSource>(srvInterface.allocator,
                        file->first, file->second));
    } // for
    return;
  } // if

  // Now we can split files to take advantage of potentially-unused threads.
  // First sort by size (descending), then we will split the largest first.

  // details elided...
```

For more information

See the source code for the full implementation of this example.

Java example: FileSource

The example shown in this section is a simple UDL Source function named FileSource , This function loads data from files stored on the host's file system (similar to the standard COPY statement). To call FileSource , you must supply a parameter named file that contains the absolute path to one or more files on the host file system. You can specify multiple files as a comma-separated list.

The FileSource function also accepts an optional parameter, named nodes , that indicates which nodes should load the files. If you do not supply this parameter, the function defaults to loading data on the initiator node only. Because this example is simple, the nodes load only the files from their own file systems. Any files in the file parameter must exist on all of the hosts in the nodes parameter. The FileSource UDSource attempts to load all of the files in the file parameter on all of the hosts in the nodes parameter.

Generating files

You can use the following Python script to generate files and distribute them to hosts in your Vertica cluster. With these files, you can experiment with the example UDSource function. Running the function requires passwordless-SSH logins to copy the files to the other hosts. Therefore, you must run the script using the database administrator account on one of your database hosts.

```
#!/usr/bin/python
# Save this file as UDLDataGen.py
import string
import random
import sys
import os

# Read in the dictionary file to provide random words. Assumes the words
# file is located in /usr/share/dict/words
wordFile = open("/usr/share/dict/words")
wordDict = []
for line in wordFile:
    if len(line) > 6:
        wordDict.append(line.strip())

MAXSTR = 4 # Maximum number of words to concatentate
NUMROWS = 1000 # Number of rows of data to generate
#FILEPATH = '/tmp/UDLdata.txt' # Final filename to use for UDL source
TMPFILE = '/tmp/UDLtemp.txt'  # Temporary filename.

# Generate a random string by concatenating several words together. Max
# number of words set by MAXSTR
def randomWords():
    words = [random.choice(wordDict) for n in xrange(random.randint(1, MAXSTR))]
    sentence = " ".join(words)
    return sentence

# Create a temporary data file that will be moved to a node. Number of
# rows for the file is set by NUMROWS. Adds the name of the node which will
# get the file, to show which node loaded the data.
def generateFile(node):
    outFile = open(TMPFILE, 'w')
    for line in xrange(NUMROWS):
        outFile.write('{0}|{1}|{2}\n'.format(line,randomWords(),node))
    outFile.close()

# Copy the temporary file to a node. Only works if passwordless SSH login
# is enabled, which it is for the database administrator account on
# Vertica hosts.
def copyFile(fileName,node):
    os.system('scp "%s" "%s:%s"' % (TMPFILE, node, fileName) )

# Loop through the comma-separated list of nodes given in the first
# parameter, creating and copying data files whose full comma-separated
# paths are passed in the second parameter
for node in [x.strip() for x in sys.argv[1].split(',')]:
    for fileName in [y.strip() for y in sys.argv[2].split(',')]:
        print "generating file", fileName, "for", node
        generateFile(node)
        print "Copying file to",node
        copyFile(fileName,node)
```

You call this script by giving it a comma-separated list of hosts to receive the files and a comma-separated list of absolute paths of files to generate. For example:

```
$ python UDLDataGen.py v_vmart_node0001,v_vmart_node0002,v_vmart_node0003 /tmp/UDLdata01.txt,/tmp/UDLdata02.txt,UDLdata03.txt
```

This script generates files that contain a thousand rows of columns delimited with the pipe character (|). These columns contain an index value, a set of random words, and the node for which the file was generated, as shown in the following output sample:

```
0|megabits embanks|v_vmart_node0001
1|unneatly|v_vmart_node0001
2|self-precipitation|v_vmart_node0001
3|antihistamine scalados Vatter|v_vmart_node0001
```

Loading and using the example

Load and use the FileSource UDSource as follows:

```
=> --Load library and create the source function
=> CREATE LIBRARY JavaLib AS '/home/dbadmin/JavaUDlLib.jar'
-> LANGUAGE 'JAVA';
CREATE LIBRARY
=> CREATE SOURCE File as LANGUAGE 'JAVA' NAME
-> 'com.mycompany.UDL.FileSourceFactory' LIBRARY JavaLib;
CREATE SOURCE FUNCTION
=> --Create a table to hold the data loaded from files
=> CREATE TABLE t (i integer, text VARCHAR, node VARCHAR);
CREATE TABLE
=> -- Copy a single file from the currently host using the FileSource
=> COPY t SOURCE File(file='/tmp/UDLdata01.txt');
 Rows Loaded
-------------
        1000
(1 row)

=> --See some of what got loaded.
=> SELECT * FROM t WHERE i < 5 ORDER BY i;
 i |             text              |  node
---+-------------------------------+-----------------
 0 | megabits embanks              | v_vmart_node0001
 1 | unneatly                      | v_vmart_node0001
 2 | self-precipitation            | v_vmart_node0001
 3 | antihistamine scalados Vatter | v_vmart_node0001
 4 | fate-menaced toilworn         | v_vmart_node0001
(5 rows)




=> TRUNCATE TABLE t;
TRUNCATE TABLE
=> -- Now load a file from three hosts. All of these hosts must have a file
=> -- named /tmp/UDLdata01.txt, each with different data
=> COPY t SOURCE File(file='/tmp/UDLdata01.txt',
-> nodes='v_vmart_node0001,v_vmart_node0002,v_vmart_node0003');
 Rows Loaded
-------------
        3000
(1 row)

=> --Now see what has been loaded
=> SELECT * FROM t WHERE i < 5 ORDER BY i,node ;
 i |                    text                    |  node
---+-----------------------------------------------+--------
 0 | megabits embanks                           | v_vmart_node0001
 0 | nimble-eyed undupability frowsier          | v_vmart_node0002
 0 | Circean nonrepellence nonnasality          | v_vmart_node0003
 1 | unneatly                                   | v_vmart_node0001
 1 | floatmaker trabacolos hit-in               | v_vmart_node0002
 1 | revelrous treatableness Halleck            | v_vmart_node0003
 2 | self-precipitation                         | v_vmart_node0001
 2 | whipcords archipelagic protodonatan copycutter  | v_vmart_node0002
 2 | Paganalian geochemistry short-shucks       | v_vmart_node0003
```

```
2 | Paganalian geochemistry short-shucks          | v_vmart_node0003
3 | antihistamine scalados Vatter                 | v_vmart_node0001
3 | swordweed touristical subcommanders desalinized | v_vmart_node0002
3 | batboys                                       | v_vmart_node0003
4 | fate-menaced toilworn                         | v_vmart_node0001
4 | twice-wanted cirrocumulous                    | v_vmart_node0002
4 | doon-head-clock                               | v_vmart_node0003
(15 rows)

=> TRUNCATE TABLE t;
TRUNCATE TABLE
=> --Now copy from several files on several hosts
=> COPY t SOURCE File(file='/tmp/UDLdata01.txt,/tmp/UDLdata02.txt,/tmp/UDLdata03.txt'
-> ,nodes='v_vmart_node0001,v_vmart_node0002,v_vmart_node0003');
 Rows Loaded
-------------
        9000
(1 row)

=> SELECT * FROM t WHERE i = 0 ORDER BY node ;
 i |                  text                   |  node
---+-----------------------------------------+--------
 0 | Awolowo Mirabilis D'Amboise              | v_vmart_node0001
 0 | sortieing Divisionism selfhypnotization  | v_vmart_node0001
 0 | megabits embanks                         | v_vmart_node0001
 0 | nimble-eyed undupability frowsier        | v_vmart_node0002
 0 | thiaminase hieroglypher derogated soilborne | v_vmart_node0002
 0 | aurigraphy crocket stenocranial          | v_vmart_node0002
 0 | Khulna pelmets                           | v_vmart_node0003
 0 | Circean nonrepellence nonnasality        | v_vmart_node0003
 0 | matterate protarsal                      | v_vmart_node0003
(9 rows)
```

Parser implementation

The following code shows the source of the FileSource class that reads a file from the host file system. The constructor, which is called by FileSourceFactory.prepareUDSources() , gets the absolute path for the file containing the data to be read. The setup() method opens the file and the destroy() method closes it. The process() method reads from the file into a buffer provided by the instance of the DataBuffer class passed to it as a parameter. If the read operation filled the output buffer, it returns OUTPUT_NEEDED . This value tells Vertica to call the method again after the next stage of the load has processed the output buffer. If the read did not fill the output buffer, then process() returns DONE to indicate it has finished processing the data source.

```
package com.mycompany.UDL;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.RandomAccessFile;

import com.vertica.sdk.DataBuffer;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.State.StreamState;
import com.vertica.sdk.UDSource;
import com.vertica.sdk.UdfException;

public class FileSource extends UDSource {

    private String filename;  // The file for this UDSource to read
    private RandomAccessFile reader;   // handle to read from file



    // The constructor just stores the absolute filename of the file it will
```

```
// The constructor just stores the absolute filename of the file it will
// read.
public FileSource(String filename) {
    super();
    this.filename = filename;
}

// Called before Vertica starts requesting data from the data source.
// In this case, setup needs to open the file and save to the reader
// property.
@Override
public void setup(ServerInterface srvInterface ) throws UdfException{
    try {
        reader = new RandomAccessFile(new File(filename), "r");
    } catch (FileNotFoundException e) {
        // In case of any error, throw a UDfException. This will terminate
        // the data load.
         String msg = e.getMessage();
         throw new UdfException(0, msg);
    }
}

// Called after data has been loaded. In this case, close the file handle.
@Override
public void destroy(ServerInterface srvInterface ) throws UdfException {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException e) {
            String msg = e.getMessage();
             throw new UdfException(0, msg);
        }
    }
}

@Override
public StreamState process(ServerInterface srvInterface, DataBuffer output)
                        throws UdfException {

    // Read up to the size of the buffer provided in the DataBuffer.buf
    // property. Here we read directly from the file handle into the
    // buffer.
    long offset;
    try {
        offset = reader.read(output.buf,output.offset,
                         output.buf.length-output.offset);
    } catch (IOException e) {
        // Throw an exception in case of any errors.
        String msg = e.getMessage();
        throw new UdfException(0, msg);
    }

    // Update the number of bytes processed so far by the data buffer.
    output.offset +=offset;

    // See end of data source has been reached, or less data was read
    // than can fit in the buffer
    if(offset == -1 || offset < output.buf.length) {
        // No more data to read.
        return StreamState.DONE;
    }else{
        // Tell Vertica to call again when buffer has been emptied
```

```
        return StreamState.OUTPUT_NEEDED;
    }
}
```

Factory implementation

The following code is a modified version of the example Java UDsource function provided in the Java UDx support package. You can find the full example in /opt/vertica/sdk/examples/JavaUDx/UDLFuctions/com/vertica/JavaLibs/FileSourceFactory.java . Its override of the plan() method verifies that the user supplied the required file parameter. If the user also supplied the optional nodes parameter, this method verifies that the nodes exist in the Vertica cluster. If there is a problem with either parameter, the method throws an exception to return an error to the user. If there are no issues with the parameters, the plan() method stores their values in the plan context object.

```
package com.mycompany.UDL;

import java.util.ArrayList;
import java.util.Vector;
import com.vertica.sdk.NodeSpecifyingPlanContext;
import com.vertica.sdk.ParamReader;
import com.vertica.sdk.ParamWriter;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.SizedColumnTypes;
import com.vertica.sdk.SourceFactory;
import com.vertica.sdk.UDSource;
import com.vertica.sdk.UdfException;

public class FileSourceFactory extends SourceFactory {

    // Called once on the initiator host to do initial setup. Checks
    // parameters and chooses which nodes will do the work.
    @Override
    public void plan(ServerInterface srvInterface,
            NodeSpecifyingPlanContext planCtxt) throws UdfException {

        String nodes; // stores the list of nodes that will load data

        // Get  copy of the parameters the user supplied to the UDSource
        // function call.
        ParamReader args =  srvInterface.getParamReader();

        // A list of nodes that will perform work. This gets saved as part
        // of the plan context.
        ArrayList<String> executionNodes = new ArrayList<String>();

        // First, ensure the user supplied the file parameter
        if (!args.containsParameter("file")) {
            // Withut a file parameter, we cannot continue. Throw an
            // exception that will be caught by the Java UDx framework.
            throw new UdfException(0, "You must supply a file parameter");
        }

        // If the user specified nodes to read the file, parse the
        // comma-separated list and save. Otherwise, assume just the
        // initiator node has the file to read.
        if (args.containsParameter("nodes")) {
            nodes = args.getString("nodes");

            // Get list of nodes in cluster, to ensure that the node the
            // user specified actually exists. The list of nodes is available
            // from the planCTxt (plan context) object.
            ArrayList<String> clusterNodes = planCtxt.getClusterNodes();
```

```
        // Parse the string parameter "nodes" which
        // is a comma-separated list of node names.
        String[] nodeNames = nodes.split(",");

        for (int i = 0; i < nodeNames.length; i++){
            // See if the node the user gave us actually exists
            if(clusterNodes.contains(nodeNames[i]))
                // Node exists. Add it to list of nodes.
                executionNodes.add(nodeNames[i]);
            else{
                // User supplied node that doesn't exist. Throw an
                // exception so the user is notified.
                String msg = String.format("Specified node '%s' but no" +
                    " node by that name is available.  Available nodes " +
                    "are \"%s\".",
                    nodeNames[i], clusterNodes.toString());
                throw new UdfException(0, msg);
            }
        }
    } else {
        // User did not supply a list of node names. Assume the initiator
        // is the only host that will read the file. The srvInterface
        // instance passed to this method has a getter for the current
        // node.
        executionNodes.add(srvInterface.getCurrentNodeName());
    }

    // Set the target node(s) in the plan context
    planCtxt.setTargetNodes(executionNodes);

    // Set parameters for each node reading data that tells it which
    // files it will read. In this simple example, just tell it to
    // read all of the files the user passed in the file parameter
    String files = args.getString("file");

    // Get object to write parameters into the plan context object.
    ParamWriter nodeParams = planCtxt.getWriter();

    // Loop through list of execution nodes, and add a parameter to plan
    // context named for each node performing the work, which tells it the
    // list of files it will process. Each node will look for a
    // parameter named something like "filesFor_vmart_node0002" in its
    // prepareUDSources() method.
    for (int i = 0; i < executionNodes.size(); i++) {
        nodeParams.setString("filesFor" + executionNodes.get(i), files);
    }
}

// Called on each host that is reading data from a source. This method
// returns an array of UDSource objects that process each source.
@Override
public ArrayList<UDSource> prepareUDSources(ServerInterface srvInterface,
        NodeSpecifyingPlanContext planCtxt) throws UdfException {

    // An array to hold the UDSource subclasses that we instaniate
    ArrayList<UDSource> retVal = new ArrayList<UDSource>();

    // Get the list of files this node is supposed to process. This was
    // saved by the plan() method in the plancontext
    String myName = srvInterface.getCurrentNodeName();
    ParamReader params = planCtxt.getReader();
```

```
    String fileNames = params.getString("filesFor" + myName);

    // Note that you can also be lazy and directly grab the parameters
    // the user passed to the UDSource functon in the COPY statement directly
    // by getting parameters from the ServerInterface object. I.e.:

    //String fileNames = srvInterface.getParamReader().getString("file");

    // Split comma-separated list into a single list.
    String[] fileList = fileNames.split(",");
    for (int i = 0; i < fileList.length; i++){
        // Instantiate a FileSource object (which is a subclass of UDSource)
        // to read each file. The constructor for FileSource takes the
        // file name of the
        retVal.add(new FileSource(fileList[i]));
    }

    // Return the collection of FileSource objects. They will be called,
    // in turn, to read each of the files.
    return retVal;
}

// Declares which parameters that this factory accepts.
@Override
public void getParameterType(ServerInterface srvInterface,
                             SizedColumnTypes parameterTypes) {
    parameterTypes.addVarchar(65000, "file");
    parameterTypes.addVarchar(65000, "nodes");
}
```

User-defined filter

User-defined filter functions allow you to manipulate data obtained from a source in various ways. For example, a filter can:

- Process a compressed file in a compression format not natively supported by Vertica.
- Take UTF-16-encoded data and transcode it to UTF-8 encoding.
- Perform search-and-replace operations on data before it is loaded into Vertica.

You can also process data through multiple filters before it is loaded into Vertica. For instance, you could unzip a file compressed with GZip, convert the content from UTF-16 to UTF-8, and finally search and replace certain text strings.

If you implement a UDFilter , you must also implement a corresponding FilterFactory .

See UDFilter class and FilterFactory class for API details.

In this section

- UDFilter class
- FilterFactory class
- Java example: ReplaceCharFilter
- C++ example: converting encoding

UDFilter class

The UDFilter class is responsible for reading raw input data from a source and preparing it to be loaded into Vertica or processed by a parser. This preparation may involve decompression, re-encoding, or any other sort of binary manipulation.

A UDFilter is instantiated by a corresponding FilterFactory on each host in the Vertica cluster that is performing filtering for the data source.

UDFilter methods

Your UDFilter subclass must override process() or processWithMetadata() :

> **Note**
> processWithMetadata() is available only for user-defined extensions (UDxs) written in the C++ programming language.

- process() reads the raw input stream as one large file. If there are any errors or failures, the entire load fails.
  You can implement process() when the upstream source implements processWithMetadata() , but it might result in parsing errors.
- processWithMetadata() is useful when the data source has metadata about record boundaries available in some structured format that's separate from the data payload. With this interface, the source emits a record length for each record in addition to the data.
  By implementing processWithMetadata() instead of process() in each phase, you can retain this record length metadata throughout the load stack, which enables a more efficient parse that can recover from errors on a per-message basis, rather than a per-file or per-source basis. KafkaSource and the Kafka parsers (KafkaAvroParser , KafkaJSONParser , and KafkaParser) use this mechanism to support per-Kafka-message rejections when individual Kafka messages are corrupted.
  Using processWithMetadata() with your UDFilter subclass enables you to write an internal filter that integrates the record length metadata from the source into the data stream, producing a single byte stream with boundary information to help parsers extract and process individual messages. KafkaInsertDelimeters and KafkaInsertLengths use this mechanism to insert message boundary information into Kafka data streams.

  > **Note**
  > To implement processWithMetadata() , you must override useSideChannel() to return true .

Optionally, you can override other UDFilter class methods.

Filter execution

The following sections detail the execution sequence each time a user-defined filter is called. The following example overrides the process() method.

Setting Up

COPY calls setup() before the first time it calls process() . Use setup() to perform any necessary setup steps that your filter needs to operate, such as initializing data structures to be used during filtering. Your object might be destroyed and re-created during use, so make sure that your object is restartable.

Filtering Data

COPY calls process() repeatedly during query execution to filter data. The method receives two instances of the DataBuffer class among its parameters, an input and an output buffer. Your implementation should read from the input buffer, manipulate it in some manner (such as decompressing it), and write the result to the output. A one-to-one correlation between the number of bytes your implementation reads and the number it writes might not exist. The process() method should process data until it either runs out of data to read or runs out of space in the output buffer. When one of these conditions occurs, your method should return one of the following values defined by StreamState :

- OUTPUT_NEEDED if the filter needs more room in its output buffer.
- INPUT_NEEDED if the filter has run out of input data (but the data source has not yet been fully processed).
- DONE if the filter has processed all of the data in the data source.
- KEEP_GOING if the filter cannot proceed for an extended period of time. The method will be called again, so do not block indefinitely. If you do, then you prevent your user from canceling the query.

Before returning, your process() method must set the offset property in each DataBuffer . In the input buffer, set it to the number of bytes that the method successfully read. In the output buffer, set it to the number of bytes the method wrote. Setting these properties allows the next call to process() to resume reading and writing data at the correct points in the buffers.

Your process() method also needs to check the InputState object passed to it to determine if there is more data in the data source. When this object is equal to END_OF_FILE, then the data remaining in the input data is the last data in the data source. Once it has processed all of the remaining data, process() must return DONE.

Tearing Down

COPY calls destroy() after the last time it calls process() . This method frees any resources reserved by the setup() or process() methods. Vertica calls this method after the process() method indicates it has finished filtering all of the data in the data stream.

If there are still data sources that have not yet been processed, Vertica may later call setup() on the object again. On subsequent calls Vertica directs the method to filter the data in a new data stream. Therefore, your destroy() method should leave an object of your UDFilter subclass in a state where the setup() method can prepare it to be reused.

API

C++

Java

Python

The UDFilter API provides the following methods for extension by subclasses:

```
virtual void setup(ServerInterface &srvInterface);

virtual bool useSideChannel();

virtual StreamState process(ServerInterface &srvInterface, DataBuffer &input, InputState input_state, DataBuffer &outpu

virtual StreamState processWithMetadata(ServerInterface &srvInterface, DataBuffer &input,
    LengthBuffer &input_lengths, InputState input_state, DataBuffer &output, LengthBuffer &output_lengths)=0;

virtual void cancel(ServerInterface &srvInterface);

virtual void destroy(ServerInterface &srvInterface);
```

FilterFactory class

If you write a filter, you must also write a filter factory to produce filter instances. To do so, subclass the FilterFactory class.

Your subclass performs the initial validation and planning of the function execution and instantiates UDFilter objects on each host that will be filtering data.

Filter factories are singletons. Your subclass must be stateless, with no fields containing data. The subclass also must not modify any global variables.

FilterFactory methods

The FilterFactory class defines the following methods. Your subclass must override the prepare() method. It may override the other methods.

Setting up

Vertica calls plan() once on the initiator node, to perform the following tasks:

- Check any parameters that have been passed from the function call in the COPY statement and error messages if there are any issues. You read the parameters by getting a ParamReader object from the instance of ServerInterface passed into your plan() method.
- Store any information that the individual hosts need in order to filter data in the PlanContext instance passed as a parameter. For example, you could store details of the input format that the filter will read and output the format that the filter should produce. The plan() method runs only on the initiator node, and the prepare() method runs on each host reading from a data source. Therefore, this object is the only means of communication between them.
  You store data in the PlanContext by getting a ParamWriter object from the getWriter() method. You then write parameters by calling methods on the ParamWriter such as setString .

  > **Note**
  > ParamWriter offers only the ability to store simple data types. For complex types, you need to serialize the data in some manner and store it as a string or long string.

Creating filters

Vertica calls prepare() to create and initialize your filter. It calls this method once on each node that will perform filtering. Vertica automatically selects the best nodes to complete the work based on available resources. You cannot specify the nodes on which the work is done.

Defining parameters

Implement getParameterTypes() to define the names and types of parameters that your filter uses. Vertica uses this information to warn callers about unknown or missing parameters. Vertica ignores unknown parameters and uses default values for missing parameters. While you should define the types and parameters for your function, you are not required to override this method.

API

C++

Java

Python

The FilterFactory API provides the following methods for extension by subclasses:

```
virtual void plan(ServerInterface &srvInterface, PlanContext &planCtxt);

virtual UDFilter * prepare(ServerInterface &srvInterface, PlanContext &planCtxt)=0;

virtual void getParameterType(ServerInterface &srvInterface, SizedColumnTypes &parameterTypes);
```

After creating your FilterFactory , you must register it with the RegisterFactory macro.

Java example: ReplaceCharFilter

The example in this section demonstrates creating a UDFilter that replaces any occurrences of a character in the input stream with another character in the output stream. This example is highly simplified and assumes the input stream is ASCII data.

Always remember that the input and output streams in a UDFilter are actually binary data. If you are performing character transformations using a UDFilter , convert the data stream from a string of bytes into a properly encoded string. For example, your input stream might consist of UTF-8 encoded text. If so, be sure to transform the raw binary being read from the buffer into a UTF string before manipulating it.

Loading and using the example

The example UDFilter has two required parameters. The from_char parameter specifies the character to be replaced, and the to_char parameter specifies the replacement character. Load and use the ReplaceCharFilter UDFilter as follows:

```
=> CREATE LIBRARY JavaLib AS '/home/dbadmin/JavaUDlLib.jar'
->LANGUAGE 'JAVA';
CREATE LIBRARY
=> CREATE FILTER ReplaceCharFilter as LANGUAGE 'JAVA'
->name 'com.mycompany.UDL.ReplaceCharFilterFactory' library JavaLib;
CREATE FILTER FUNCTION
=> CREATE TABLE t (text VARCHAR);
CREATE TABLE
=> COPY t FROM STDIN WITH FILTER ReplaceCharFilter(from_char='a', to_char='z');
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> Mary had a little lamb
>> a man, a plan, a canal, Panama
>> \.

=> SELECT * FROM t;
             text
--------------------------------
 Mzry hzd z little lzmb
 z mzn, z plzn, z cznzl, Pznzmz
(2 rows)

=> --Calling the filter with incorrect parameters returns errors
=> COPY t from stdin with filter ReplaceCharFilter();
ERROR 3399:  Failure in UDx RPC call InvokePlanUDL(): Error in User Defined Object [
ReplaceCharFilter], error code: 0
com.vertica.sdk.UdfException: You must supply two parameters to ReplaceChar: 'from_char' and 'to_char'
        at com.vertica.JavaLibs.ReplaceCharFilterFactory.plan(ReplaceCharFilterFactory.java:22)
        at com.vertica.udxfence.UDxExecContext.planUDFilter(UDxExecContext.java:889)
        at com.vertica.udxfence.UDxExecContext.planCurrentUDLType(UDxExecContext.java:865)
        at com.vertica.udxfence.UDxExecContext.planUDL(UDxExecContext.java:821)
        at com.vertica.udxfence.UDxExecContext.run(UDxExecContext.java:242)
        at java.lang.Thread.run(Thread.java:662)
```

Parser implementation

The ReplaceCharFilter class reads the data stream, replacing each occurrence of a user-specified character with another character.

```
package com.vertica.JavaLibs;

import com.vertica.sdk.DataBuffer;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.State.InputState;
import com.vertica.sdk.State.StreamState;
import com.vertica.sdk.UDFilter;

public class ReplaceCharFilter extends UDFilter {

    private byte[] fromChar;
    private byte[] toChar;

    public ReplaceCharFilter(String fromChar, String toChar){
        // Stores the from char and to char as byte arrays. This is
        // not a robust method of doing this, but works for this simple
        // example.
        this.fromChar= fromChar.getBytes();
        this.toChar=toChar.getBytes();
    }
    @Override
    public StreamState process(ServerInterface srvInterface, DataBuffer input,
            InputState input_state, DataBuffer output) {

        // Check if there is no more input and the input buffer has been completely
        // processed. If so, filtering is done.
        if (input_state == InputState.END_OF_FILE && input.buf.length == 0) {
            return StreamState.DONE;
        }

        // Get current position in the input buffer
        int offset = output.offset;

        // Determine how many bytes to process. This is either until input
        // buffer is exhausted or output buffer is filled
        int limit = Math.min((input.buf.length - input.offset),
                (output.buf.length - output.offset));

        for (int i = input.offset; i < limit; i++) {
            // This example just replaces each instance of from_char
            // with to_char. It does not consider things such as multi-byte
            // UTF-8 characters.
            if (input.buf[i] == fromChar[0]) {
                output.buf[i+offset] = toChar[0];
            } else {
                // Did not find from_char, so copy input to the output
                output.buf[i+offset]=input.buf[i];
            }
        }

        input.offset += limit;
        output.offset += input.offset;

        if (input.buf.length - input.offset < output.buf.length - output.offset) {
            return StreamState.INPUT_NEEDED;
        } else {
            return StreamState.OUTPUT_NEEDED;
        }
    }
```

Factory implementation

ReplaceCharFilterFactory requires two parameters (from_char and to_char). The plan() method verifies that these parameters exist and are single-character strings. The method then stores them in the plan context. The prepare() method gets the parameter values and passes them to the ReplaceCharFilter objects, which it instantiates, to perform the filtering.

```
package com.vertica.JavaLibs;

import java.util.ArrayList;
import java.util.Vector;

import com.vertica.sdk.FilterFactory;
import com.vertica.sdk.PlanContext;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.SizedColumnTypes;
import com.vertica.sdk.UDFilter;
import com.vertica.sdk.UdfException;

public class ReplaceCharFilterFactory extends FilterFactory {

    // Run on the initiator node to perform varification and basic setup.
    @Override
    public void plan(ServerInterface srvInterface,PlanContext planCtxt)
            throws UdfException {
        ArrayList<String> args =
                srvInterface.getParamReader().getParamNames();

        // Ensure user supplied two arguments
        if (!(args.contains("from_char") && args.contains("to_char"))) {
            throw new UdfException(0, "You must supply two parameters" +
                    " to ReplaceChar: 'from_char' and 'to_char'");
        }

        // Verify that the from_char is a single character.
        String fromChar = srvInterface.getParamReader().getString("from_char");
        if (fromChar.length() != 1) {
            String message =  String.format("Replacechar expects a single " +
                "character in the 'from_char' parameter. Got length %d",
                fromChar.length());
            throw new UdfException(0, message);
        }

        // Save the from character in the plan context, to be read by
        // prepare() method.
        planCtxt.getWriter().setString("fromChar",fromChar);

        // Ensure to character parameter is a single characater
        String toChar = srvInterface.getParamReader().getString("to_char");
        if (toChar.length() != 1) {
            String message =  String.format("Replacechar expects a single "
                + "character in the 'to_char' parameter. Got length %d",
                toChar.length());
            throw new UdfException(0, message);
        }
        // Save the to character in the plan data
        planCtxt.getWriter().setString("toChar",toChar);
    }

    // Called on every host that will filter data. Must instantiate the
    // UDFilter subclass.
    @Override
    public UDFilter prepare(ServerInterface srvInterface, PlanContext planCtxt)
```

```
                    throws UdfException {
    // Get data stored in the context by the plan() method.
    String fromChar = planCtxt.getWriter().getString("fromChar");
    String toChar = planCtxt.getWriter().getString("toChar");

    // Instantiate a filter object to perform filtering.
    return new ReplaceCharFilter(fromChar, toChar);
  }

  // Describe the parameters accepted by this filter.
  @Override
  public void getParameterType(ServerInterface srvInterface,
          SizedColumnTypes parameterTypes) {
    parameterTypes.addVarchar(1, "from_char");
    parameterTypes.addVarchar(1, "to_char");
  }
```

C++ example: converting encoding

The following example shows how you can convert encoding for a file from one type to another by converting UTF-16 encoded data to UTF-8. You can find this example in the SDK at /opt/vertica/sdk/examples/FilterFunctions/IConverter.cpp .

Filter implementation

```
class Iconverter : public UDFilter{
private:
  std::string fromEncoding, toEncoding;
  iconv_t cd; // the conversion descriptor opened
  uint converted; // how many characters have been converted
protected:
  virtual StreamState process(ServerInterface &srvInterface, DataBuffer &input,
                              InputState input_state, DataBuffer &output)
  {
    char *input_buf = (char *)input.buf + input.offset;
    char *output_buf = (char *)output.buf + output.offset;
    size_t inBytesLeft = input.size - input.offset, outBytesLeft = output.size - output.offset;
    // end of input
    if (input_state == END_OF_FILE && inBytesLeft == 0)
    {
      // Gnu libc iconv doc says, it is good practice to finalize the
      // outbuffer for stateful encodings (by calling with null inbuffer).
      //
      // http://www.gnu.org/software/libc/manual/html_node/Generic-Conversion-Interface.html
      iconv(cd, NULL, NULL, &output_buf, &outBytesLeft);
      // output buffer can be updated by this operation
      output.offset = output.size - outBytesLeft;
      return DONE;
    }
    size_t ret = iconv(cd, &input_buf, &inBytesLeft, &output_buf, &outBytesLeft);
    // if conversion is successful, we ask for more input, as input has not reached EOF.
    StreamState retStatus = INPUT_NEEDED;
    if (ret == (size_t)(-1))
    {
      // seen an error
      switch (errno)
      {
      case E2BIG:
        // input size too big, not a problem, ask for more output.
        retStatus = OUTPUT_NEEDED;
        break;
      case EINVAL:
```

```
            // input stops in the middle of a byte sequence, not a problem, ask for more input
            retStatus = input_state == END_OF_FILE ? DONE : INPUT_NEEDED;
            break;
        case EILSEQ:
            // invalid sequence seen, throw
            // TODO: reporting the wrong byte position
            vt_report_error(1, "Invalid byte sequence when doing %u-th conversion", converted);
        case EBADF:
            // something wrong with descriptor, throw
            vt_report_error(0, "Invalid descriptor");
        default:
            vt_report_error(0, "Uncommon Error");
            break;
        }
    }
    else converted += ret;
    // move position pointer
    input.offset = input.size - inBytesLeft;
    output.offset = output.size - outBytesLeft;
    return retStatus;
  }
public:
  Iconverter(const std::string &from, const std::string &to)
  : fromEncoding(from), toEncoding(to), converted(0)
  {
    // note "to encoding" is first argument to iconv...
    cd = iconv_open(to.c_str(), from.c_str());
    if (cd == (iconv_t)(-1))
    {
        // error when creating converters.
        vt_report_error(0, "Error initializing iconv: %m");
    }
  }
  ~Iconverter()
  {
    // free iconv resources;
    iconv_close(cd);
  }
};
```

Factory implementation

```
class IconverterFactory : public FilterFactory{
public:
  virtual void plan(ServerInterface &srvInterface,
        PlanContext &planCtxt) {
    std::vector<std::string> args = srvInterface.getParamReader().getParamNames();
    /* Check parameters */
    if (!(args.size() == 0 ||
          (args.size() == 1 && find(args.begin(), args.end(), "from_encoding")
                != args.end()) || (args.size() == 2
                && find(args.begin(), args.end(), "from_encoding") != args.end()
                && find(args.begin(), args.end(), "to_encoding") != args.end()))) {
      vt_report_error(0, "Invalid arguments.  Must specify either no arguments,  or "
                      "'from_encoding' alone, or 'from_encoding' and 'to_encoding'.");
    }
    /* Populate planData */
    // By default, we do UTF16->UTF8, and x->UTF8
    VString from_encoding = planCtxt.getWriter().getStringRef("from_encoding");
    VString to_encoding = planCtxt.getWriter().getStringRef("to_encoding");
    from_encoding.copy("UTF-16");
    to_encoding.copy("UTF-8");
    if (args.size() == 2)
    {
      from_encoding.copy(srvInterface.getParamReader().getStringRef("from_encoding"));
      to_encoding.copy(srvInterface.getParamReader().getStringRef("to_encoding"));
    }
    else if (args.size() == 1)
    {
      from_encoding.copy(srvInterface.getParamReader().getStringRef("from_encoding"));
    }
    if (!from_encoding.length()) {
      vt_report_error(0, "The empty string is not a valid from_encoding value");
    }
    if (!to_encoding.length()) {
      vt_report_error(0, "The empty string is not a valid to_encoding value");
    }
  }
  virtual UDFilter* prepare(ServerInterface &srvInterface,
        PlanContext &planCtxt) {
    return vt_createFuncObj(srvInterface.allocator, Iconverter,
          planCtxt.getReader().getStringRef("from_encoding").str(),
          planCtxt.getReader().getStringRef("to_encoding").str());
  }
  virtual void getParameterType(ServerInterface &srvInterface,
                        SizedColumnTypes &parameterTypes) {
    parameterTypes.addVarchar(32, "from_encoding");
    parameterTypes.addVarchar(32, "to_encoding");
  }
};
RegisterFactory(IconverterFactory);
```

User-defined parser

A parser takes a stream of bytes and passes a corresponding sequence of tuples to the Vertica load process. You can use user-defined parser functions to parse:

- Data in formats not understood by the Vertica built-in parser.
- Data that requires more specific control than the built-in parser supplies.

For example, you can load a CSV file using a specific CSV library. See the Vertica SDK for two CSV examples.

COPY supports a single user-defined parser that you can use with a user-defined source and zero or more instances of a user-defined filter . If you implement a UDParser class , you must also implement a corresponding ParserFactory .

Sometimes, you can improve the performance of your parser by adding a *chunker* . A chunker divides up the input and uses multiple threads to parse it. Chunkers are available only in the C++ API. For details, see Cooperative parse and UDChunker class . Under special circumstances you can further improve performance by using *apportioned load* , an approach where multiple Vertica nodes parse the input.

In this section

- UDParser class
- UDChunker class
- ParserFactory class
- C++ example: BasicIntegerParser
- C++ example: ContinuousIntegerParser
- Java example: numeric text
- Java example: JSON parser
- C++ example: delimited parser and chunker
- Python example: complex types JSON parser

UDParser class

You can subclass the UDParser class when you need to parse data that is in a format that the COPY statement's native parser cannot handle.

During parser execution, Vertica always calls three methods: setup() , process() , and destroy() . It might also call getRejectedRecord() .

UDParser constructor

The UDParser class performs important initialization required by all subclasses, including initializing the StreamWriter object used by the parser. Therefore, your constructor must call super() .

UDParser methods

Your UDParser subclass must override process() or processWithMetadata() :

> **Note**
> processWithMetadata() is available only for user-defined extensions (UDxs) written in the C++ programming language.

- process() reads the raw input stream as one large file. If there are any errors or failures, the entire load fails. You can implement process() with a source or filter that implements processWithMetadata() , but it might result in parsing errors.
  You can implement process() when the upstream source or filter implements processWithMetadata() , but it might result in parsing errors.
- processWithMetadata() is useful when the data source has metadata about record boundaries available in some structured format that's separate from the data payload. With this interface, the source emits a record length for each record in addition to the data.
  By implementing processWithMetadata() instead of process() in each phase, you can retain this record length metadata throughout the load stack, which enables a more efficient parse that can recover from errors on a per-message basis, rather than a per-file or per-source basis. KafkaSource and Kafka parsers (KafkaAvroParser , KafkaJSONParser , and KafkaParser) use this mechanism to support per-Kafka-message rejections when individual Kafka messages are corrupted.

  > **Note**
  > To implement processWithMetadata() , you must override useSideChannel() to return true .

Additionally, you must override getRejectedRecord() to return information about rejected records.

Optionally, you can override the other UDParser class methods.

Parser execution

The following sections detail the execution sequence each time a user-defined parser is called. The following example overrides the process() method.

Setting up

COPY calls setup() before the first time it calls process() . Use setup() to perform any initial setup tasks that your parser needs to parse data. This setup includes retrieving parameters from the class context structure or initializing data structures for use during filtering. Vertica calls this method before calling the process() method for the first time. Your object might be destroyed and re-created during use, so make sure that your object is restartable.

Parsing

COPY calls process() repeatedly during query execution. Vertica passes this method a buffer of data to parse into columns and rows and one of the following input states defined by InputState :

- OK : currently at the start of or in the middle of a stream
- END_OF_FILE : no further data is available.
- END_OF_CHUNK : the current data ends on a record boundary and the parser should consume all of it before returning. This input state only occurs when using a chunker.
- START_OF_PORTION : the input does not start at the beginning of a source. The parser should find the first end-of-record mark. This input state only occurs when using apportioned load.You can use the getPortion() method to access the offset and size of the portion.
- END_OF_PORTION : the source has reached the end of its portion. The parser should finish processing the last record it started and advance no further. This input state only occurs when using apportioned load.

The parser must reject any data that it cannot parse, so that Vertica can report the rejection and write the rejected data to files.

The process() method must parse as much data as it can from the input buffer. The buffer might not end on a row boundary. Therefore, it might have to stop parsing in the middle of a row of input and ask for more data. The input can contain null bytes, if the source file contains them, and is *not* automatically null-terminated.

A parser has an associated StreamWriter object, which performs the actual writing of the data. When your parser extracts a column value, it uses one of the type-specific methods on StreamWriter to write that value to the output stream. See Writing Data for more information about these methods.

A single call to process() might write several rows of data. When your parser finishes processing a row of data, it must call next() on its StreamWriter to advance the output stream to a new row. (Usually a parser finishes processing a row because it encounters an end-of-row marker.)

When your process() method reaches the end of the buffer, it tells Vertica its current state by returning one of the following values defined by StreamState :

- INPUT_NEEDED : the parser has reached the end of the buffer and needs more data to parse.
- DONE : the parser has reached the end of the input data stream.
- REJECT : the parser has rejected the last row of data it read (see Rejecting Rows).

Tearing down

COPY calls destroy() after the last time that process() is called. It frees any resources reserved by the setup() or process() method.

Vertica calls this method after the process() method indicates it has completed parsing the data source. However, sometimes data sources that have not yet been processed might remain. In such cases, Vertica might later call setup() on the object again and have it parse the data in a new data stream. Therefore, write your destroy() method so that it leaves an instance of your UDParser subclass in a state where setup() can be safely called again.

Reporting rejections

If process() rejects a row, Vertica calls getRejectedRecord() to report it. Usually, this method returns an instance of the RejectedRecord class with details of the rejected row.

Writing data

A parser has an associated StreamWriter object, which you access by calling getStreamWriter() . In your process() implementation, use the set *Type* () methods on the StreamWriter object to write values in a row to specific column indexes. Verify that the data types you write match the data types expected by the schema.

The following example shows how you can write a value of type long to the fourth column (index 3) in the current row:

```
StreamWriter writer = getStreamWriter();
...
writer.setLongValue(3, 98.6);
```

StreamWriter provides methods for all the basic types, such as setBooleanValue() , setStringValue() , and so on. See the API documentation for a complete list of StreamWriter methods, including options that take primitive types or explicitly set entries to null.

Rejecting rows

If your parser finds data it cannot parse, it should reject the row by:

1. Saving details about the rejected row data and the reason for the rejection. These pieces of information can be directly stored in a RejectedRecord object, or in fields on your UDParser subclass, until they are needed.
2. Updating the row's position in the input buffer by updating input.offset so it can resume parsing with the next row.

3. Signaling that it has rejected a row by returning with the value StreamState.REJECT .
4. Returning an instance of the RejectedRecord class with the details about the rejected row.

Breaking up large loads

Vertica provides two ways to break up large loads. Apportioned load allows you to distribute a load among several database nodes. Cooperative parse (C++ only) allows you to distribute a load among several threads on one node.

API

C++

Java

Python

The UDParser API provides the following methods for extension by subclasses:

```
virtual void setup(ServerInterface &srvInterface, SizedColumnTypes &returnType);

virtual bool useSideChannel();

virtual StreamState process(ServerInterface &srvInterface, DataBuffer &input, InputState input_state)=0;

virtual StreamState processWithMetadata(ServerInterface &srvInterface, DataBuffer &input,
                LengthBuffer &input_lengths, InputState input_state)=0;

virtual void cancel(ServerInterface &srvInterface);

virtual void destroy(ServerInterface &srvInterface, SizedColumnTypes &returnType);

virtual RejectedRecord getRejectedRecord();
```

UDChunker class

You can subclass the UDChunker class to allow your parser to support Cooperative parse . This class is available only in the C++ API.

Fundamentally, a UDChunker is a very simplistic parser. Like UDParser , it has the following three methods: setup() , process() , and destroy() . You must override process() ; you may override the others. This class has one additional method, alignPortion() , which you must implement if you want to enable Apportioned load for your UDChunker .

Setting up and tearing down

As with UDParser , you can define initialization and cleanup code for your chunker. Vertica calls setup() before the first call to process() and destroy() after the last call to process() . Your object might be reused among multiple load sources, so make sure that setup() completely initializes all fields.

Chunking

Vertica calls process() to divide an input into chunks that can be parsed independently. The method takes an input buffer and an indicator of the input state:

- OK : the input buffer begins at the start of or in the middle of a stream.
- END_OF_FILE : no further data is available.
- END_OF_PORTION : the source has reached the end of its portion. This state occurs only when using apportioned load.

If the input state is END_OF_FILE , the chunker should set the input.offset marker to input.size and return DONE . Returning INPUT_NEEDED is an error.

If the input state is OK , the chunker should read data from the input buffer and find record boundaries. If it finds the end of at least one record, it should align the input.offset marker with the byte after the end of the last record in the buffer and return CHUNK_ALIGNED . For example, if the input is "abc~def" and "~" is a record terminator, this method should set input.offset to 4, the position of "d". If process() reaches the end of the input without finding a record boundary, it should return INPUT_NEEDED .

You can divide the input into smaller chunks, but consuming all available records in the input can have better performance. For example, a chunker could scan backwards from the end of the input to find a record terminator, which might be the last of many records in the input, and return it all as one chunk without scanning through the rest of the input.

If the input state is END_OF_PORTION , the chunker should behave as it does for an input state of OK , except that it should also set a flag. When called again, it should find the first record in the next portion and align the chunk to that record.

The input data can contain null bytes, if the source file contains them. The input argument is not automatically null-terminated.

The process() method must not block indefinitely. If this method cannot proceed for an extended period of time, it should return KEEP_GOING . Failing to return KEEP_GOING has several consequences, such as preventing your user from being able to cancel the query.

See C++ example: delimited parser and chunker for an example of the process() method using chunking.

Aligning portions

If your chunker supports apportioned load, implement the alignPortion() method. Vertica calls this method one or more times, before calling process() , to align the input offset with the beginning of the first complete chunk in the portion. The method takes an input buffer and an indicator of the input state:

- START_OF_PORTION : the beginning of the buffer corresponds to the start of the portion. You can use the getPortion() method to access the offset and size of the portion.
- OK : the input buffer is in the middle of a portion.
- END_OF_PORTION : the end of the buffer corresponds to the end of the portion or beyond the end of a portion.
- END_OF_FILE : no further data is available.

The method should scan from the beginning of the buffer to the start of the first complete record. It should set input.offset to this position and return one of the following values:

- DONE , if it found a chunk. input.offset is the first byte of the chunk.
- INPUT_NEEDED , if the input buffer does not contain the start of any chunk. It is an error to return this from an input state of END_OF_FILE .
- REJECT , if the portion (not buffer) does not contain the start of any chunk.

API

The UDChunker API provides the following methods for extension by subclasses:

```
virtual void setup(ServerInterface &srvInterface,
        SizedColumnTypes &returnType);

virtual StreamState alignPortion(ServerInterface &srvInterface,
        DataBuffer &input, InputState state);

virtual StreamState process(ServerInterface &srvInterface,
        DataBuffer &input, InputState input_state)=0;

virtual void cancel(ServerInterface &srvInterface);

virtual void destroy(ServerInterface &srvInterface,
        SizedColumnTypes &returnType);
```

ParserFactory class

If you write a parser, you must also write a factory to produce parser instances. To do so, subclass the ParserFactory class.

Parser factories are singletons. Your subclass must be stateless, with no fields containing data. Your subclass also must not modify any global variables.

The ParserFactory class defines the following methods. Your subclass must override the prepare() method. It may override the other methods.

Setting up

Vertica calls plan() once on the initiator node to perform the following tasks:

- Check any parameters that have been passed from the function call in the COPY statement and error messages if there are any issues. You read the parameters by getting a ParamReader object from the instance of ServerInterface passed into your plan() method.
- Store any information that the individual hosts need in order to parse the data. For example, you could store parameters in the PlanContext

instance passed in through the planCtxt parameter. The plan() method runs only on the initiator node, and the prepareUDSources() method runs on each host reading from a data source. Therefore, this object is the only means of communication between them.
You store data in the PlanContext by getting a ParamWriter object from the getWriter() method. You then write parameters by calling methods on the ParamWriter such as setString .

> **Note**
> ParamWriter offers only the ability to store simple data types. For complex types, you need to serialize the data in some manner and store it as a string or long string.

Creating parsers

Vertica calls prepare() on each node to create and initialize your parser, using data stored by the plan() method.

Defining parameters

Implement getParameterTypes() to define the names and types of parameters that your parser uses. Vertica uses this information to warn callers about unknown or missing parameters. Vertica ignores unknown parameters and uses default values for missing parameters. While you should define the types and parameters for your function, you are not required to override this method.

Defining parser outputs

Implement getParserReturnType() to define the data types of the table columns that the parser outputs. If applicable, getParserReturnType() also defines the size, precision, or scale of the data types. Usually, this method reads data types of the output table from the argType and perColumnParamReader arguments and verifies that it can output the appropriate data types. If getParserReturnType() is prepared to output the data types, it calls methods on the SizedColumnTypes object passed in the returnType argument. In addition to the data type of the output column, your method should also specify any additional information about the column's data type:

- For binary and string data types (such as CHAR, VARCHAR, and LONG VARBINARY), specify its maximum length.
- For NUMERIC types, specify its precision and scale.
- For Time/Timestamp types (with or without time zone), specify its precision (-1 means unspecified).
- For ARRAY types, specify the maximum number of elements.
- For all other types, no length or precision specification is required.

Supporting cooperative parse

To support Cooperative parse , implement prepareChunker() and return an instance of your UDChunker subclass. If isChunkerApportionable() returns true , then it is an error for this method to return null.

Cooperative parse is currently supported only in the C++ API.

Supporting apportioned load

To support Apportioned load , your parser, chunker, or both must support apportioning. To indicate that the parser can apportion a load, implement isParserApportionable() and return true . To indicate that the chunker can apportion a load, implement isChunkerApportionable() and return true .

The isChunkerApportionable() method takes a ServerInterface as an argument, so you have access to the parameters supplied in the COPY statement. You might need this information if the user can specify a record delimiter, for example. Return true from this method if and only if the factory can create a chunker for this input.

API

C++

Java

Python

The ParserFactory API provides the following methods for extension by subclasses:

```
virtual void plan(ServerInterface &srvInterface, PerColumnParamReader &perColumnParamReader, PlanContext &plan

virtual UDParser * prepare(ServerInterface &srvInterface, PerColumnParamReader &perColumnParamReader,
        PlanContext &planCtxt, const SizedColumnTypes &returnType)=0;

virtual void getParameterType(ServerInterface &srvInterface, SizedColumnTypes &parameterTypes);

virtual void getParserReturnType(ServerInterface &srvInterface, PerColumnParamReader &perColumnParamReader,
        PlanContext &planCtxt, const SizedColumnTypes &argTypes,
        SizedColumnTypes &returnType);

virtual bool isParserApportionable();

// C++ API only:
virtual bool isChunkerApportionable(ServerInterface &srvInterface);

virtual UDChunker * prepareChunker(ServerInterface &srvInterface, PerColumnParamReader &perColumnParamRead
        PlanContext &planCtxt, const SizedColumnTypes &returnType);
```

If you are using Apportioned load to divide a single input into multiple load streams, implement isParserApportionable() and/or isChunkerApportionable() and return true . Returning true from these methods does not guarantee that Vertica *will* apportion the load. However, returning false from both indicates that it will not try to do so.

If you are using Cooperative parse , implement prepareChunker() and return an instance of your UDChunker subclass. Cooperative parse is supported only for the C++ API.

Vertica calls the prepareChunker() method *only* for unfenced functions. This method is not available when you use the function in fenced mode.

If you want your chunker to be available for apportioned load, implement isChunkerApportionable() and return true .

After creating your ParserFactory , you must register it with the RegisterFactory macro.

C++ example: BasicIntegerParser

The BasicIntegerParser example parses a string of integers separated by non-numeric characters. For a version of this parser that uses continuous load, see C++ example: ContinuousIntegerParser .

Loading and using the example

Load and use the BasicIntegerParser example as follows.

```
=> CREATE LIBRARY BasicIntegerParserLib AS '/home/dbadmin/BIP.so';

=> CREATE PARSER BasicIntegerParser AS
LANGUAGE 'C++' NAME 'BasicIntegerParserFactory' LIBRARY BasicIntegerParserLib;

=> CREATE TABLE t (i integer);

=> COPY t FROM stdin WITH PARSER BasicIntegerParser();
0
1
2
3
4
5
\.
```

Implementation

The BasicIntegerParser class implements only the process() method from the API. (It also implements a helper method for type conversion.) This method processes each line of input, looking for numbers on each line. When it advances to a new line it moves the input.offset marker and checks the input state. It then writes the output.

```
virtual StreamState process(ServerInterface &srvInterface, DataBuffer &input,
        InputState input_state) {
    // WARNING: This implementation is not trying for efficiency.
    // It is trying for simplicity, for demonstration purposes.

    size_t start = input.offset;
    const size_t end = input.size;

    do {
        bool found_newline = false;
        size_t numEnd = start;
        for (; numEnd < end; numEnd++) {
            if (input.buf[numEnd] < '0' || input.buf[numEnd] > '9') {
                found_newline = true;
                break;
            }
        }

        if (!found_newline) {
            input.offset = start;
            if (input_state == END_OF_FILE) {
                // If we're at end-of-file,
                // emit the last integer (if any) and return DONE.
                if (start != end) {
                    writer->setInt(0, strToInt(input.buf + start, input.buf + numEnd));
                    writer->next();
                }
                return DONE;
            } else {
                // Otherwise, we need more data.
                return INPUT_NEEDED;
            }
        }

        writer->setInt(0, strToInt(input.buf + start, input.buf + numEnd));
        writer->next();

        start = numEnd + 1;
    } while (true);
}
};
```

In the factory, the plan() method is a no-op; there are no parameters to check. The prepare() method instantiates the parser using the macro provided by the SDK:

```
virtual UDParser* prepare(ServerInterface &srvInterface,
        PerColumnParamReader &perColumnParamReader,
        PlanContext &planCtxt,
        const SizedColumnTypes &returnType) {

    return vt_createFuncObject<BasicIntegerParser>(srvInterface.allocator);
}
```

The getParserReturnType() method declares the single output:

```
    virtual void getParserReturnType(ServerInterface &srvInterface,
                                     PerColumnParamReader &perColumnParamReader,
                                     PlanContext &planCtxt,
                                     const SizedColumnTypes &argTypes,
                                     SizedColumnTypes &returnType) {
        // We only and always have a single integer column
        returnType.addInt(argTypes.getColumnName(0));
    }
```

As for all UDxs written in C++, the example ends by registering its factory:

```
RegisterFactory(BasicIntegerParserFactory);
```

C++ example: ContinuousIntegerParser

The ContinuousIntegerParser example is a variation of BasicIntegerParser . Both examples parse integers from input strings. ContinuousIntegerParser uses Continuous load to read data.

Loading and using the example

Load the ContinuousIntegerParser example as follows.

```
=> CREATE LIBRARY ContinuousIntegerParserLib AS '/home/dbadmin/CIP.so';

=> CREATE PARSER ContinuousIntegerParser AS
LANGUAGE 'C++' NAME 'ContinuousIntegerParserFactory'
LIBRARY ContinuousIntegerParserLib;
```

Use it in the same way that you use BasicIntegerParser . See C++ example: BasicIntegerParser .

Implementation

ContinuousIntegerParser is a subclass of ContinuousUDParser . Subclasses of ContinuousUDParser place the processing logic in the run() method.

```
    virtual void run() {

        // This parser assumes a single-column input, and
        // a stream of ASCII integers split by non-numeric characters.
        size_t pos = 0;
        size_t reserved = cr.reserve(pos+1);
        while (!cr.isEof() || reserved == pos + 1) {
            while (reserved == pos + 1 && isdigit(*ptr(pos))) {
                pos++;
                reserved = cr.reserve(pos + 1);
            }

            std::string st(ptr(), pos);
            writer->setInt(0, strToInt(st));
            writer->next();

            while (reserved == pos + 1 && !isdigit(*ptr(pos))) {
                pos++;
                reserved = cr.reserve(pos + 1);
            }
            cr.seek(pos);
            pos = 0;
            reserved = cr.reserve(pos + 1);
        }
    }
};
```

For a more complex example of a ContinuousUDParser , see ExampleDelimitedParser in the examples. (See Downloading and running UDx example code .) ExampleDelimitedParser uses a chunker; see C++ example: delimited parser and chunker .

Java example: numeric text

This NumericTextParser example parses integer values spelled out in words rather than digits (for example "one two three" for one-hundred twenty three). The parser:

- Accepts a single parameter to set the character that separates columns in a row of data. The separator defaults to the pipe (|) character.
- Ignores extra spaces and the capitalization of the words used to spell out the digits.
- Recognizes the digits using the following words: zero, one, two, three, four, five, six, seven, eight, nine.
- Assumes that the words spelling out an integer are separated by at least one space.
- Rejects any row of data that cannot be completely parsed into integers.
- Generates an error, if the output table has a non-integer column.

Loading and using the example

Load and use the parser as follows:

```
=> CREATE LIBRARY JavaLib AS '/home/dbadmin/JavaLib.jar' LANGUAGE 'JAVA';
CREATE LIBRARY

=> CREATE PARSER NumericTextParser AS LANGUAGE 'java'
->    NAME 'com.myCompany.UDParser.NumericTextParserFactory'
->    LIBRARY JavaLib;
CREATE PARSER FUNCTION
=> CREATE TABLE t (i INTEGER);
CREATE TABLE
=> COPY t FROM STDIN WITH PARSER NumericTextParser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> One
>> Two
>> One Two Three
>> \.
=> SELECT * FROM t ORDER BY i;
  i
-----
   1
   2
 123
(3 rows)

=> DROP TABLE t;
DROP TABLE
=> -- Parse multi-column input
=> CREATE TABLE t (i INTEGER, j INTEGER);
CREATE TABLE
=> COPY t FROM stdin WITH PARSER NumericTextParser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> One | Two
>> Two | Three
>> One Two Three | four Five Six
>> \.
=> SELECT * FROM t ORDER BY i;
  i  |  j
-----+-----
   1 |   2
   2 |   3
 123 | 456
(3 rows)

=> TRUNCATE TABLE t;
TRUNCATE TABLE
=> -- Use alternate separator character
=> COPY t FROM STDIN WITH PARSER NumericTextParser(separator='"');
```

```
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> Five * Six
>> seven * eight
>> nine * one zero
>> \.
=> SELECT * FROM t ORDER BY i;
 i | j
---+----
 5 |  6
 7 |  8
 9 | 10
(3 rows)

=> TRUNCATE TABLE t;
TRUNCATE TABLE

=> -- Rows containing data that does not parse into digits is rejected.
=> DROP TABLE t;
DROP TABLE
=> CREATE TABLE t (i INTEGER);
CREATE TABLE
=> COPY t FROM STDIN WITH PARSER NumericTextParser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> One Zero Zero
>> Two Zero Zero
>> Three Zed Zed
>> Four Zero Zero
>> Five Zed Zed
>> \.
SELECT * FROM t ORDER BY i;
  i
-----
 100
 200
 400
(3 rows)

=> -- Generate an error by trying to copy into a table with a non-integer column
=> DROP TABLE t;
DROP TABLE
=> CREATE TABLE t (i INTEGER, j VARCHAR);
CREATE TABLE
=> COPY t FROM STDIN WITH PARSER NumericTextParser();
vsql:UDParse.sql:94: ERROR 3399:  Failure in UDx RPC call
InvokeGetReturnTypeParser(): Error in User Defined Object [NumericTextParser],
error code: 0
com.vertica.sdk.UdfException: Column 2 of output table is not an Int
        at com.myCompany.UDParser.NumericTextParserFactory.getParserReturnType
        (NumericTextParserFactory.java:70)
        at com.vertica.udxfence.UDxExecContext.getReturnTypeParser(
        UDxExecContext.java:1464)
        at com.vertica.udxfence.UDxExecContext.getReturnTypeParser(
        UDxExecContext.java:768)
        at com.vertica.udxfence.UDxExecContext.run(UDxExecContext.java:236)
        at java.lang.Thread.run(Thread.java:662)
```

Parser implementation

The following code implements the parser.

```
package com.myCompany.UDParser;

import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;

import com.vertica.sdk.DataBuffer;
import com.vertica.sdk.DestroyInvocation;
import com.vertica.sdk.RejectedRecord;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.State.InputState;
import com.vertica.sdk.State.StreamState;
import com.vertica.sdk.StreamWriter;
import com.vertica.sdk.UDParser;
import com.vertica.sdk.UdfException;

public class NumericTextParser extends UDParser {

    private String separator; // Holds column separator character

    // List of strings that we accept as digits.
    private List<String> numbers = Arrays.asList("zero", "one",
            "two", "three", "four", "five", "six", "seven",
            "eight", "nine");

    // Hold information about the last rejected row.
    private String rejectedReason;
    private String rejectedRow;

    // Constructor gets the separator character from the Factory's prepare()
    // method.
    public NumericTextParser(String sepparam) {
        super();
        this.separator = sepparam;
    }

    // Called to perform the actual work of parsing. Gets a buffer of bytes
    // to turn into tuples.
    @Override
    public StreamState process(ServerInterface srvInterface, DataBuffer input,
            InputState input_state) throws UdfException, DestroyInvocation {

        int i=input.offset; // Current position in the input buffer
        // Flag to indicate whether we just found the end of a row.
        boolean lastCharNewline = false;
        // Buffer to hold the row of data being read.
        StringBuffer line = new StringBuffer();

        //Continue reading until end of buffer.
        for(; i < input.buf.length; i++){
            // Loop through input until we find a linebreak: marks end of row
            char inchar = (char) input.buf[i];
            // Note that this isn't a robust way to find rows. It should
            // accept a user-defined row separator. Also, the following
            // assumes ASCII line break metheods, which isn't a good idea
            // in the UTF world. But it is good enough for this simple example.
            if (inchar != '\n' && inchar != '\r') {
                // Keep adding to a line buffer until a full row of data is read
                line.append(inchar);
                lastCharNewline = false; // Last character not a new line
            } else {
```

```
            // Found a line break. Process the row.
            lastCharNewline = true; // indicate we got a complete row
            // Update the position in the input buffer. This is updated
            // whether the row is successfully processed or not.
            input.offset = i+1;
            // Call procesRow to extract values and write tuples to the
            // output. Returns false if there was an error.
            if (!processRow(line)) {
                // Processing row failed. Save bad row to rejectedRow field
                // and return to caller indicating a rejected row.
                rejectedRow = line.toString();
                // Update position where we processed the data.
                return StreamState.REJECT;
            }
            line.delete(0, line.length()); // clear row buffer
        }
    }

    // At this point, process() has finished processing the input buffer.
    // There are two possibilities: need to get more data
    // from the input stream to finish processing, or there is
    // no more data to process. If at the end of the input stream and
    // the row was not terminated by a linefeed, it may need
    // to process the last row.

    if (input_state == InputState.END_OF_FILE && lastCharNewline) {
        // End of input and it ended on a newline. Nothing more to do
        return StreamState.DONE;
    } else if (input_state == InputState.END_OF_FILE && !lastCharNewline) {
        // At end of input stream but didn't get a final newline. Need to
        // process the final row that was read in, then exit for good.
        if (line.length() == 0) {
            // Nothing to process. Done parsing.
            return StreamState.DONE;
        }
        // Need to parse the last row, not terminated by a linefeed. This
        // can occur if the file being read didn't have a final line break.
        if (processRow(line)) {
            return StreamState.DONE;
        } else {
            // Processing last row failed. Save bad row to rejectedRow field
            // and return to caller indicating a rejected row.
            rejectedRow = line.toString();
            // Tell Vertica the entire buffer was processed so it won't
            // call again to have the line processed.
            input.offset = input.buf.length;
            return StreamState.REJECT;
        }
    } else {
        // Stream is not fully read, so tell Vertica to send more. If
        // process() didn't get a complete row before it hit the end of the
        // input buffer, it will end up re-processing that segment again
        // when more data is added to the buffer.
        return StreamState.INPUT_NEEDED;
    }
}

// Breaks a row into columns, then parses the content of the
// columns. Returns false if there was an error parsing the
// row, in which case it sets the rejected row to the input
// line. Returns true if the row was successfully read.
private boolean processRow(StringBuffer line)
```

```
private boolean processRow(StringBuffer line)
                    throws UdfException, DestroyInvocation {
    String[] columns = line.toString().split(Pattern.quote(separator));
    // Loop through the columns, decoding their contents
    for (int col = 0; col < columns.length; col++) {
        // Call decodeColumn to extract value from this column
        Integer colval = decodeColumn(columns[col]);
        if (colval == null) {
            // Could not parse one of the columns. Indicate row should
            // be rejected.
            return false;
        }
        // Column parsed OK. Write it to the output. writer is a field
        // provided by the parent class. Since this parser only accepts
        // integers, there is no need to verify that data type of the parsed
        // data matches the data type of the column being written. In your
        // UDParsers, you may want to perform this verification.
        writer.setLong(col,colval);
    }
    // Done with the row of data. Advance output to next row.

    // Note that this example does not verify that all of the output columns
    // have values assigned to them. If there are missing values at the
    // end of a row, they get automatically get assigned a default value
    // (0 for integers). This isn't a robust solution. Your UDParser
    // should perform checks here to handle this situation and set values
    // (such as null) when appropriate.
    writer.next();
    return true; // Successfully processed the row.
}

// Gets a string with text numerals, i.e. "One Two Five Seven" and turns
// it into an integer value, i.e. 1257. Returns null if the string could not
// be parsed completely into numbers.
private Integer decodeColumn(String text) {
    int value = 0; // Hold the value being parsed.

    // Split string into individual words. Eat extra spaces.
    String[] words = text.toLowerCase().trim().split("\\s+");

    // Loop through the words, matching them against the list of
    // digit strings.
    for (int i = 0; i < words.length; i++) {
        if (numbers.contains(words[i])) {
            // Matched a digit. Add the it to the value.
            int digit = numbers.indexOf(words[i]);
            value = (value * 10) + digit;
        } else {
            // The string didn't match one of the accepted string values
            // for digits. Need to reject the row. Set the rejected
            // reason string here so it can be incorporated into the
            // rejected reason object.
            //
            // Note that this example does not handle null column values.
            // In most cases, you want to differentiate between an
            // unparseable column value and a missing piece of input
            // data. This example just rejects the row if there is a missing
            // column value.
            rejectedReason = String.format(
                    "Could not parse '%s' into a digit",words[i]);
            return null;
        }
```

```
        }
        return value;
    }

    // Vertica calls this method if the parser rejected a row of data
    // to find out what went wrong and add to the proper logs. Just gathers
    // information stored in fields and returns it in an object.
    @Override
    public RejectedRecord getRejectedRecord() throws UdfException {
        return new RejectedRecord(rejectedReason,rejectedRow.toCharArray(),
                rejectedRow.length(), "\n");
    }
```

ParserFactory implementation

The following code implements the parser factory.

NumericTextParser accepts a single optional parameter named separator . This parameter is defined in the getParameterType() method, and the plan() method stores its value. NumericTextParser outputs only integer values. Therefore, if the output table contains a column whose data type is not integer, the getParserReturnType() method throws an exception.

```
package com.myCompany.UDParser;

import java.util.regex.Pattern;

import com.vertica.sdk.ParamReader;
import com.vertica.sdk.ParamWriter;
import com.vertica.sdk.ParserFactory;
import com.vertica.sdk.PerColumnParamReader;
import com.vertica.sdk.PlanContext;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.SizedColumnTypes;
import com.vertica.sdk.UDParser;
import com.vertica.sdk.UdfException;
import com.vertica.sdk.VerticaType;

public class NumericTextParserFactory extends ParserFactory {

    // Called once on the initiator host to check the parameters and set up the
    // context data that hosts performing processing will need later.
    @Override
    public void plan(ServerInterface srvInterface,
            PerColumnParamReader perColumnParamReader,
            PlanContext planCtxt) {

        String separator = "|"; // assume separator is pipe character

        // See if a parameter was given for column separator
        ParamReader args = srvInterface.getParamReader();
        if (args.containsParameter("separator")) {
            separator = args.getString("separator");
            if (separator.length() > 1) {
                throw new UdfException(0,
                        "Separator parameter must be a single character");
            }
            if (Pattern.quote(separator).matches("[a-zA-Z]")) {
                throw new UdfException(0,
                        "Separator parameter cannot be a letter");
            }
        }
```

```
    // Save separator character in the Plan Data
    ParamWriter context = planCtxt.getWriter();
    context.setString("separator", separator);
  }

  // Define the data types of the output table that the parser will return.
  // Mainly, this just ensures that all of the columns in the table which
  // is the target of the data load are integer.
  @Override
  public void getParserReturnType(ServerInterface srvInterface,
          PerColumnParamReader perColumnParamReader,
          PlanContext planCtxt,
          SizedColumnTypes argTypes,
          SizedColumnTypes returnType) {

    // Get access to the output table's columns
    for (int i = 0; i < argTypes.getColumnCount(); i++ ) {
      if (argTypes.getColumnType(i).isInt()) {
          // Column is integer... add it to the output
          returnType.addInt(argTypes.getColumnName(i));
      } else {
          // Column isn't an int, so throw an exception.
          // Technically, not necessary since the
          // UDx framework will automatically error out when it sees a
          // Discrepancy between the type in the target table and the
          // types declared by this method. Throwing this exception will
          // provide a clearer error message to the user.
          String message = String.format(
              "Column %d of output table is not an Int", i + 1);
          throw new UdfException(0, message);
      }
    }
  }

  // Instantiate the UDParser subclass named NumericTextParser. Passes the
  // separator characetr as a paramter to the constructor.
  @Override
  public UDParser prepare(ServerInterface srvInterface,
      PerColumnParamReader perColumnParamReader, PlanContext planCtxt,
      SizedColumnTypes returnType) throws UdfException {
    // Get the separator character from the context
    String separator = planCtxt.getReader().getString("separator");
    return new NumericTextParser(separator);
  }

  // Describe the parameters accepted by this parser.
  @Override
  public void getParameterType(ServerInterface srvInterface,
          SizedColumnTypes parameterTypes) {
    parameterTypes.addVarchar(1, "separator");
  }
```

Java example: JSON parser

The JSON Parser consumes a stream of JSON objects. Each object must be well formed and on a single line in the input. Use line breaks to delimit the objects. The parser uses the field names as keys in a map, which become column names in the table. You can find the code for this example in /opt/vertica/packages/flextable/examples. This directory also contains an example data file.

This example uses the setRowFromMap() method to write data.

Loading and using the example

Load the library and define the JSON parser, using the third-party library (gson-2.2.4.jar) as follows. See the comments in JsonParser.java for a download URL:

```
=> CREATE LIBRARY json
-> AS '/opt/vertica/packages/flextable/examples/java/output/json.jar'
-> DEPENDS '/opt/vertica/bin/gson-2.2.4.jar' language 'java';
CREATE LIBRARY

=> CREATE PARSER JsonParser AS LANGUAGE 'java'
-> NAME 'com.vertica.flex.JsonParserFactory' LIBRARY json;
CREATE PARSER FUNCTION
```

You can now define a table and then use the JSON parser to load data into it, as follows:

```
=> CREATE TABLE mountains(name varchar(64), type varchar(32), height integer);
CREATE TABLE

=> COPY mountains FROM '/opt/vertica/packages/flextable/examples/mountains.json'
-> WITH PARSER JsonParser();
-[ RECORD 1 ]--
Rows Loaded | 2

=> SELECT * from mountains;
-[ RECORD 1 ]--------
name   | Everest
type   | mountain
height | 29029
-[ RECORD 2 ]--------
name   | Mt St Helens
type   | volcano
height |
```

The data file contains a value (hike_safety) that was not loaded because the table definition did not include that column. The data file follows:

```
{ "name": "Everest", "type":"mountain", "height": 29029, "hike_safety": 34.1  }
{ "name": "Mt St Helens", "type": "volcano", "hike_safety": 15.4 }
```

Implementation

The following code shows the process() method from JsonParser.java. The parser attempts to read the input into a Map. If the read is successful, the JSON Parser calls setRowFromMap() :

```
    @Override
    public StreamState process(ServerInterface srvInterface, DataBuffer input,
            InputState inputState) throws UdfException, DestroyInvocation {
        clearReject();
        StreamWriter output = getStreamWriter();

        while (input.offset < input.buf.length) {
            ByteBuffer lineBytes = consumeNextLine(input, inputState);

            if (lineBytes == null) {
                return StreamState.INPUT_NEEDED;
            }

            String lineString = StringUtils.newString(lineBytes);

            try {
                Map<String,Object> map = gson.fromJson(lineString, parseType);

                if (map == null) {
                    continue;
                }

                output.setRowFromMap(map);
                // No overrides needed, so just call next() here.
                output.next();
            } catch (Exception ex) {
                setReject(lineString, ex);
                return StreamState.REJECT;
            }
        }
    }
```

The factory, JsonParserFactory.java, instantiates and returns a parser in the prepare() method. No additional setup is required.

C++ example: delimited parser and chunker

The ExampleDelimitedUDChunker class divides an input at delimiter characters. You can use this chunker with any parser that understands delimited input. ExampleDelimitedParser is a ContinuousUDParser subclass that uses this chunker.

Loading and using the example

Load and use the example as follows.

```
=> CREATE LIBRARY ExampleDelimitedParserLib AS '/home/dbadmin/EDP.so';

=> CREATE PARSER ExampleDelimitedParser AS
   LANGUAGE 'C++' NAME 'DelimitedParserFrameworkExampleFactory'
   LIBRARY ExampleDelimitedParserLib;

=> COPY t FROM stdin WITH PARSER ExampleDelimitedParser();
0
1
2
3
4
5
6
7
8
9
\.
```

Chunker implementation

This chunker supports apportioned load. The alignPortion() method finds the beginning of the first complete record in the current portion and aligns the input buffer with it. The record terminator is passed as an argument and set in the constructor.

```
StreamState ExampleDelimitedUDChunker::alignPortion(
        ServerInterface &srvInterface,
        DataBuffer &input, InputState state)
{
  /* find the first record terminator.  Its record belongs to the previous portion */
  void *buf = reinterpret_cast<void *>(input.buf + input.offset);
  void *term = memchr(buf, recordTerminator, input.size - input.offset);

  if (term) {
    /* record boundary found.  Align to the start of the next record */
    const size_t chunkSize = reinterpret_cast<size_t>(term) - reinterpret_cast<size_t>(buf);
    input.offset += chunkSize
        + sizeof(char) /* length of record terminator */;

    /* input.offset points at the start of the first complete record in the portion */
    return DONE;
  } else if (state == END_OF_FILE || state == END_OF_PORTION) {
    return REJECT;
  } else {
    VIAssert(state == START_OF_PORTION || state == OK);
    return INPUT_NEEDED;
  }
```

The process() method has to account for chunks that span portion boundaries. If the previous call was at the end of a portion, the method set a flag. The code begins by checking for and handling that condition. The logic is similar to that of alignPortion() , so the example calls it to do part of the division.

```
StreamState ExampleDelimitedUDChunker::process(
        ServerInterface &srvInterface,
        DataBuffer &input,
        InputState input_state)
{
  const size_t termLen = 1;
  const char *terminator = &recordTerminator;

  if (pastPortion) {
    /*
     * Previous state was END_OF_PORTION, and the last chunk we will produce
     * extends beyond the portion we started with, into the next portion.
     * To be consistent with alignPortion(), that means finding the first
     * record boundary, and setting the chunk to be at that boundary.
     * Fortunately, this logic is identical to aligning the portion (with
     * some slight accounting for END_OF_FILE)!
     */
    const StreamState findLastTerminator = alignPortion(srvInterface, input);

    switch (findLastTerminator) {
      case DONE:
        return DONE;
      case INPUT_NEEDED:
        if (input_state == END_OF_FILE) {
          /* there is no more input where we might find a record terminator */
          input.offset = input.size;
          return DONE;
        }
        return INPUT_NEEDED;
      default:
        VIAssert("Invalid return state from alignPortion()");
    }
    return findLastTerminator;
  }
```

Now the method looks for the delimiter. If the input began at the end of a portion, it sets the flag.

```
  size_t ret = input.offset, term_index = 0;
  for (size_t index = input.offset; index < input.size; ++index) {
    const char c = input.buf[index];
    if (c == terminator[term_index]) {
      ++term_index;
      if (term_index == termLen) {
        ret = index + 1;
        term_index = 0;
      }
      continue;
    } else if (term_index > 0) {
      index -= term_index;
    }

    term_index = 0;
  }

  if (input_state == END_OF_PORTION) {
    /*
     * Regardless of whether or not a record was found, the next chunk will extend
     * into the next portion.
     */
    pastPortion = true;
  }
```

Finally, process() moves the input offset and returns.

```
    // if we were able to find some rows, move the offset to point at the start of the next (potential) row, or end of block
    if (ret > input.offset) {
        input.offset = ret;
        return CHUNK_ALIGNED;
    }

    if (input_state == END_OF_FILE) {
        input.offset = input.size;
        return DONE;
    }

    return INPUT_NEEDED;
}
```

Factory implementation

The file ExampleDelimitedParser.cpp defines a factory that uses this UDChunker . The chunker supports apportioned load, so the factory implements isChunkerApportionable() :

```
    virtual bool isChunkerApportionable(ServerInterface &srvInterface) {
        ParamReader params = srvInterface.getParamReader();
        if (params.containsParameter("disable_chunker") && params.getBoolRef("d\
isable_chunker")) {
            return false;
        } else {
            return true;
        }
    }
```

The prepareChunker() method creates the chunker:

```
    virtual UDChunker* prepareChunker(ServerInterface &srvInterface,
                                      PerColumnParamReader &perColumnParamReade\
r,
                                      PlanContext &planCtxt,
                                      const SizedColumnTypes &returnType)
    {
        ParamReader params = srvInterface.getParamReader();
        if (params.containsParameter("disable_chunker") && params.getBoolRef("d\
isable_chunker")) {
            return NULL;
        }

        std::string recordTerminator("\n");

        ParamReader args(srvInterface.getParamReader());
        if (args.containsParameter("record_terminator")) {
            recordTerminator = args.getStringRef("record_terminator").str();
        }

        return vt_createFuncObject<ExampleDelimitedUDChunker>(srvInterface.allo\
cator,
                recordTerminator[0]);
    }
```

Python example: complex types JSON parser

The following example details a UDParser that takes a JSON object and parses it into complex types. For this example, the parser assumes the input data are arrays of rows with two integer fields. The input records should be separated by newline characters. If any row fields aren't specified by the JSON input, the function parses those fields as NULL.

The source code for this UDParser also contains a factory method for parsing rows that have an integer and an array of integer fields. The implementation of the parser is independent of the return type in the factory, so you can create factories with different return types that all point to the ComplexJsonParser() class in the prepare() method. The complete source code is in /opt/vertica/sdk/examples/python/UDParsers.py .

Loading and using the example

Load the library and create the parser as follows:

```
=> CREATE OR REPLACE LIBRARY UDParsers AS '/home/dbadmin/examples/python/UDParsers.py' LANGUAGE 'Python';

=> CREATE PARSER ComplexJsonParser AS LANGUAGE 'Python' NAME 'ArrayJsonParserFactory' LIBRARY UDParsers;
```

You can now define a table and then use the JSON parser to load data into it, for example:

```
=> CREATE TABLE orders (a bool, arr array[row(a int, b int)]);
CREATE TABLE

=> COPY orders (arr) FROM STDIN WITH PARSER ComplexJsonParser();
[]
[{"a":1, "b":10}]
[{"a":1, "b":10}, {"a":null, "b":10}]
[{"a":1, "b":10},{"a":10, "b":20}]
[{"a":1, "b":10}, {"a":null, "b":null}]
[{"a":1, "b":2}, {"a":3, "b":4}, {"a":5, "b":6}, {"a":7, "b":8}, {"a":9, "b":10}, {"a":11, "b":12}, {"a":13, "b":14}]
\.

=> SELECT * FROM orders;
a |                                 arr
--+-----------------------------------------------------------------------
  | []
  | [{"a":1,"b":10}]
  | [{"a":1,"b":10},{"a":null,"b":10}]
  | [{"a":1,"b":10},{"a":10,"b":20}]
  | [{"a":1,"b":10},{"a":null,"b":null}]
  | [{"a":1,"b":2},{"a":3,"b":4},{"a":5,"b":6},{"a":7,"b":8},{"a":9,"b":10},{"a":11,"b":12},{"a":13,"b":14}]
(6 rows)
```

Setup

All Python UDxs must import the Vertica SDK library. ComplexJsonParser() also requires the json library.

```
import vertica_sdk
import json
```

Factory implementation

The prepare() method instantiates and returns a parser:

```
def prepare(self, srvInterface, perColumnParamReader, planCtxt, returnType):
    return ComplexJsonParser()
```

getParserReturnType() declares that the return type must be an array of rows that each have two integer fields:

```
def getParserReturnType(self, srvInterface, perColumnParamReader, planCtxt, argTypes, returnType):
    fieldTypes = vertica_sdk.SizedColumnTypes.makeEmpty()
    fieldTypes.addInt('a')
    fieldTypes.addInt('b')
    returnType.addArrayType(vertica_sdk.SizedColumnTypes.makeRowType(fieldTypes, 'elements'), 64, 'arr')
```

Parser implementation

The process() method reads in data with an InputBuffer and then splits that input data on the newline character. The method then passes the processed data to the writeRows() method. writeRows() turns each data row into a JSON object, checks the type of that JSON object, and then writes the appropriate value or object to the output.

```
class ComplexJsonParser(vertica_sdk.UDParser):

    leftover = ''

    def process(self, srvInterface, input_buffer, input_state, writer):
        input_buffer.setEncoding('utf-8')

        self.count = 0
        rec = self.leftover + input_buffer.read()
        row_lst = rec.split('\n')
        self.leftover = row_lst[-1]
        self.writeRows(row_lst[:-1], writer)
        if input_state == InputState.END_OF_FILE:
            self.writeRows([self.leftover], writer)
            return StreamState.DONE
        else:
            return StreamState.INPUT_NEEDED

    def writeRows(self, str_lst, writer):
        for s in str_lst:
            stripped = s.strip()
            if len(stripped) == 0:
                return
            elif len(stripped) > 1 and stripped[0:2] == "//":
                continue
            jsonValue = json.loads(stripped)
            if type(jsonValue) is list:
                writer.setArray(0, jsonValue)
            elif jsonValue is None:
                writer.setNull(0)
            else:
                writer.setRow(0, jsonValue)
            writer.next()
```

Load parallelism

Vertica can divide the work of loading data, taking advantage of parallelism to speed up the operation. Vertica supports several types of parallelism:

- Distributed load: Vertica distributes files in a multi-file load to several nodes to load in parallel, instead of loading all of them on a single node. Vertica manages distributed load; you do not need to do anything special in your UDL.
- Cooperative parse: A source being loaded on a single node uses multi-threading to parallelize the parse. Cooperative parse divides a load at execution time, based on how threads are scheduled. You must enable cooperative parse in your parser. See Cooperative parse .
- Apportioned load: Vertica divides a single large file or other single source into segments, which it assigns to several nodes to load in parallel. Apportioned load divides the load at planning time, based on available nodes and cores on each node. You must enable apportioned load in your source and parser. See Apportioned load .

You can support both cooperative parse and apportioned load in the same UDL. Vertica decides which to use for each load operation and might use both. See Combining cooperative parse and apportioned load .

In this section

- Cooperative parse
- Apportioned load
- Combining cooperative parse and apportioned load

Cooperative parse

By default, Vertica parses a data source in a single thread on one database node. You can optionally use *cooperative parse* to parse a source using multiple threads on a node. More specifically, data from a source passes through a *chunker* that groups blocks from the source stream into logical units. These chunks can be parsed in parallel. The chunker divides the input into pieces that can be individually parsed, and the parser then parses them concurrently. Cooperative parse is available only for unfenced UDxs. (See Fenced and unfenced modes .)

To use cooperative parse, a chunker must be able to locate end-of-record markers in the input. Locating these markers might not be possible in all input formats.

Chunkers are created by parser factories. At load time, Vertica first calls the UDChunker to divide the input into chunks and then calls the UDParser to parse each chunk.

You can use cooperative parse and apportioned load independently or together. See Combining cooperative parse and apportioned load .

How Vertica divides a load

When Vertica receives data from a source, it calls the chunker's process() method repeatedly. A chunker is, essentially, a lightweight parser; instead of parsing, the process() method divides the input into chunks.

After the chunker has finished dividing the input into chunks, Vertica sends those chunks to as many parsers as are available, calling the process() method on the parser.

Implementing cooperative parse

To implement cooperative parse, perform the following actions:

- Subclass UDChunker and implement process() .
- In your ParserFactory , implement prepareChunker() to return a UDChunker .

See C++ example: delimited parser and chunker for a UDChunker that also supports apportioned load.

Apportioned load

A parser can use more than one database node to load a single input source in parallel. This approach is referred to as *apportioned load* . Among the parsers built into Vertica, the default (delimited) parser supports apportioned load.

Apportioned load, like cooperative parse, requires an input that can be divided at record boundaries. The difference is that cooperative parse does a sequential scan to find record boundaries, while apportioned load first jumps (seeks) to a given position and then scans. Some formats, like generic XML, do not support seeking.

To use apportioned load, you must ensure that the source is reachable by all participating database nodes. You typically use apportioned load with distributed file systems.

It is possible for a parser to not support apportioned load directly but to have a chunker that supports apportioning.

You can use apportioned load and cooperative parse independently or together. See Combining cooperative parse and apportioned load .

How Vertica apportions a load

If both the parser and its source support apportioning, then you can specify that a single input is to be distributed to multiple database nodes for loading. The SourceFactory breaks the input into portions and assigns them to execution nodes. Each Portion consists of an offset into the input and a size. Vertica distributes the portions and their parameters to the execution nodes. A source factory running on each node produces a UDSource for the given portion.

The UDParser first determines where to start parsing:

- If the portion is the first one in the input, the parser advances to the offset and begins parsing.
- If the portion is not the first, the parser advances to the offset and then scans until it finds the end of a record. Because records can break across portions, parsing begins after the first record-end encountered.

The parser must complete a record, which might require it to read past the end of the portion. The parser is responsible for parsing all records that *begin* in the assigned portion, regardless of where they end. Most of this work occurs within the process() method of the parser.

Sometimes, a portion contains nothing to be parsed by its assigned node. For example, suppose you have a record that begins in portion 1, runs through all of portion 2, and ends in portion 3. The parser assigned to portion 1 parses the record, and the parser assigned to portion 3 starts after that record. The parser assigned to portion 2, however, has no record starting within its portion.

If the load also uses Cooperative parse , then after apportioning the load and before parsing, Vertica divides portions into chunks for parallel loading.

Implementing apportioned load

To implement apportioned load, perform the following actions in the source, the parser, and their factories.

In your SourceFactory subclass:

- Implement isSourceApportionable() and return true .
- Implement plan() to determine portion size, designate portions, and assign portions to execution nodes. To assign portions to particular executors, pass the information using the parameter writer on the plan context (PlanContext::getWriter()).
- Implement prepareUDSources() . Vertica calls this method on each execution node with the plan context created by the factory. This method returns the UDSource instances to be used for this node's assigned portions.
- If sources can take advantage of parallelism, you can implement getDesiredThreads() to request a number of threads for each source. See SourceFactory class for more information about this method.

In your UDSource subclass, implement process() as you would for any other source, using the assigned portion. You can retrieve this portion with getPortion() .

In your ParserFactory subclass:

- Implement isParserApportionable() and return true .
- If your parser uses a UDChunker that supports apportioned load, implement isChunkerApportionable() .

In your UDParser subclass:

- Write your UDParser subclass to operate on portions rather than whole sources. You can do so by handling the stream states PORTION_START and PORTION_END , or by using the ContinuousUDParser API. Your parser must scan for the beginning of the portion, find the first record boundary after that position, and parse to the end of the last record beginning in that portion. Be aware that this behavior might require that the parser read beyond the end of the portion.
- Handle the special case of a portion containing no record start by returning without writing any output.

In your UDChunker subclass, implement alignPortion() . See Aligning Portions .

Example

The SDK provides a C++ example of apportioned load in the ApportionLoadFunctions directory:

- FilePortionSource is a subclass of UDSource .
- DelimFilePortionParser is a subclass of ContinuousUDParser .

Use these classes together. You could also use FilePortionSource with the built-in delimited parser.

The following example shows how you can load the libraries and create the functions in the database:

```
=> CREATE LIBRARY FilePortionSourceLib as '/home/dbadmin/FP.so';

=> CREATE LIBRARY DelimFilePortionParserLib as '/home/dbadmin/Delim.so';

=> CREATE SOURCE FilePortionSource AS
LANGUAGE 'C++' NAME 'FilePortionSourceFactory' LIBRARY FilePortionSourceLib;

=> CREATE PARSER DelimFilePortionParser AS
LANGUAGE 'C++' NAME 'DelimFilePortionParserFactory' LIBRARY DelimFilePortionParserLib;
```

The following example shows how you can use the source and parser to load data:

```
=> COPY t WITH SOURCE FilePortionSource(file='g1/*.dat') PARSER DelimFilePortionParser(delimiter = '|',
   record_terminator = '~');
```

Combining cooperative parse and apportioned load

You can enable both Cooperative parse and Apportioned load in the same parser, allowing Vertica to decide how to load data.

Deciding how to divide a load

Vertica uses apportioned load, where possible, at query-planning time. It decides whether to also use cooperative parse at execution time.

Apportioned load requires SourceFactory support. Given a suitable UDSource , at planning time Vertica calls the isParserApportionable() method on the ParserFactory . If this method returns true , Vertica apportions the load.

If isParserApportionable() returns false but isChunkerApportionable() returns true , then a chunker is available for cooperative parse and that chunker supports apportioned load. Vertica apportions the load.

If neither of these methods returns true , then Vertica does not apportion the load.

At execution time, Vertica first checks whether the load is running in unfenced mode and proceeds only if it is. Cooperative parse is not supported in fenced mode.

If the load is not apportioned, and more than one thread is available, Vertica uses cooperative parse.

If the load is apportioned, and exactly one thread is available, Vertica uses cooperative parse if and only if the parser is not apportionable. In this case, the chunker is apportionable but the parser is not.

If the load is apportioned, and more than one thread is available, and the chunker is apportionable, Vertica uses cooperative parse.

If Vertica uses cooperative parse but prepareChunker() does not return a UDChunker instance, Vertica reports an error.

Executing apportioned, cooperative loads

If a load uses both apportioned load and cooperative parse, Vertica uses the SourceFactory to break the input into portions. It then assigns the portions to execution nodes. See How Vertica Apportions a Load .

On the execution node, Vertica calls the chunker's alignPortion() method to align the input with portion boundaries. (This step is skipped for the first portion, which by definition is already aligned at the beginning.) This step is necessary because a parser using apportioned load sometimes has to read beyond the end of the portion, so a chunker needs to find the end point.

After aligning the portion, Vertica calls the chunker's process() method repeatedly. See How Vertica Divides a Load .

The chunks found by the chunker are then sent to the parser's process() method for processing in the usual way.

Continuous load

The ContinuousUDSource , ContinuousUDFilter , and ContinuousUDParser classes allow you to write and process data as needed instead of having to iterate through the data. The Python API does not support continuous load.

Each class includes the following functions:

- initialize() - Invoked before run() . You can optionally override this function to perform setup and initialization.
- run() - Processes the data.
- deinitialize() - Invoked after run() has returned. You can optionally override this function to perform tear-down and destruction.

Do not override the setup() , process() , and destroy() functions that are inherited from parent classes.

You can use the yield() function to yield control back to the server during idle or busy loops so the server can check for status changes or query cancellations.

These three classes use associated ContinuousReader and ContinuousWriter classes to read input data and write output data.

ContinuousUDSource API (C++)

The ContinuousUDSource class extends UDSource and adds the following methods for extension by subclasses:

```
virtual void initialize(ServerInterface &srvInterface);

virtual void run();

virtual void deinitialize(ServerInterface &srvInterface);
```

ContinuousUDFilter API (C++)

The ContinuousUDFilter class extends UDFilter and adds the following methods for extension by subclasses:

```
virtual void initialize(ServerInterface &srvInterface);

virtual void run();

virtual void deinitialize(ServerInterface &srvInterface);
```

ContinuousUDParser API

C++

Java

The ContinuousUDParser class extends UDParser and adds the following methods for extension by subclasses:

```
virtual void initialize(ServerInterface &srvInterface);

virtual void run();

virtual void deinitialize(ServerInterface &srvInterface);
```

See the API documentation for additional utility methods.

Buffer classes

Buffer classes are used as handles to the raw data stream for all UDL functions. The C++ and Java APIs use a single DataBuffer class for both input and output. The Python API has two classes, InputBuffer and OutputBuffer .

DataBuffer API (C++, java)

C++

Java

The DataBuffer class has a pointer to a buffer and size, and an offset indicating how much of the stream has been consumed.

```
/**
* A contiguous in-memory buffer of char *
*/
  struct DataBuffer {
   /// Pointer to the start of the buffer
   char * buf;

   /// Size of the buffer in bytes
   size_t size;

   /// Number of bytes that have been processed by the UDL
   size_t offset;
};
```

InputBuffer and OutputBuffer APIs (python)

The Python InputBuffer and OutputBuffer classes replace the DataBuffer class in the C++ and Java APIs.

InputBuffer class

The InputBuffer class decodes and translates raw data streams depending on the specified encoding. Python natively supports a wide range of languages and codecs . The InputBuffer is an argument to the process() method for both UDFilters and UDParsers. A user interacts with the UDL's data stream by calling methods of the InputBuffer

If you do not specify a value for setEncoding() , Vertica assumes a value of NONE.

```
class InputBuffer:
    def getSize(self):
        ...
    def getOffset(self):
        ...

    def setEncoding(self, encoding):
        """
        Set the encoding of the data contained in the underlying buffer
        """
        pass

    def peek(self, length = None):
        """
        Copy data from the input buffer into Python.
        If no encoding has been specified, returns a Bytes object containing raw data.
        Otherwise, returns data decoded into an object corresponding to the specified encoding
        (for example, 'utf-8' would return a string).
        If length is None, returns all available data.
        If length is not None then the length of the returned object is at most what is requested.
        This method does not advance the buffer offset.
        """
        pass

    def read(self, length = None):
        """
        See peek().
        This method does the same thing as peek(), but it also advances the
        buffer offset by the number of bytes consumed.
        """
        pass

        # Advances the DataBuffer offset by a number of bytes equal to the result
        # of calling "read" with the same arguments.
        def advance(self, length = None):
        """
        Advance the buffer offset by the number of bytes indicated by
        the length and encoding arguments.  See peek().
    Returns the new offset.
        """
        pass
```

OutputBuffer class

The OutputBuffer class encodes and outputs data from Python to Vertica. The OutputBuffer is an argument to the process() method for both UDFilters and UDParsers. A user interacts with the UDL's data stream by calling methods of the OutputBuffer to manipulate and encode data.

The write() method transfers all data from the Python client to Vertica. The output buffer can accept any size object. If a user writes an object to the OutputBuffer larger than Vertica can immediately process, Vertica stores the overflow. During the next call to process() ,Vertica checks for leftover data. If there is any, Vertica copies it to the DataBuffer before determining whether it needs to call process() from the Python UDL.

If you do not specify a value for setEncoding() , Vertica assumes a value of NONE.

```
class OutputBuffer:
def setEncoding(self, encoding):
"""
Specify the encoding of the data which will be written to the underlying buffer.
"""
pass
def write(self, data):
"""
Transfer bytes from the data object into Vertica.
If an encoding was specified via setEncoding(), the data object will be converted to bytes using the specified encoding.
Otherwise, the data argument is expected to be a Bytes object and is copied to the underlying buffer.
"""
pass
```

Apache Hadoop integration

Apache™ Hadoop™, like Vertica, uses a cluster of nodes for distributed processing. The primary component of interest is HDFS, the Hadoop Distributed File System.

You can use Vertica with HDFS in several ways:

- You can import HDFS data into locally-stored ROS files.
- You can access HDFS data in place using external tables. You can define the tables yourself or get schema information from Hive, a Hadoop component.
- You can use HDFS as a storage location for ROS files.
- You can export data from Vertica to share with other Hadoop components using a Hadoop columnar format. See File export for more information.

Hadoop file paths are expressed as URLs in the webhdfs or hdfs URL scheme. For more about using these schemes, see HDFS file system .

Hadoop distributions

Vertica can be used with Hadoop distributions from Hortonworks, Cloudera, and MapR. See Hadoop integrations for the specific versions that are supported.

If you are using Cloudera, you can manage your Vertica cluster using Cloudera Manager. See Integrating with Cloudera Manager .

If you are using MapR, see Integrating Vertica with the MapR distribution of Hadoop .

WebHDFS requirement

By default, if you use a URL in the hdfs scheme, Vertica treats it as webhdfs . If you instead use the (deprecated) LibHDFS++ library, you must still have a WebHDFS service available. LibHDFS++ does not support some WebHDFS features, such as encryption zones, wire encryption, or writes, and falls back to WebHDFS when needed.

For some uses, such as Eon Mode communal storage, you must use WebHDFS directly with the webhdfs scheme.

> **Deprecated**
> Support for LibHDFS++ is deprecated. You can set the HDFSUseWebHDFS configuration parameter to 0 (disabled) to use LibHDFS++ until it is removed.

In this section

- Cluster layout
- Configuring HDFS access
- Accessing kerberized HDFS data
- Using HDFS storage locations
- Using the HCatalog Connector
- Integrating with Cloudera Manager
- Integrating Vertica with the MapR distribution of Hadoop
- Hive primer for Vertica integration

Cluster layout

Vertica supports two cluster architectures for Hadoop integration. Which architecture you use affects the decisions you make about integration with HDFS. These options might also be limited by license terms.

- You can co-locate Vertica on some or all of your Hadoop nodes. Vertica can then take advantage of data locality.
- You can build a Vertica cluster that is separate from your Hadoop cluster. In this configuration, Vertica can fully use each of its nodes; it does not share resources with Hadoop.

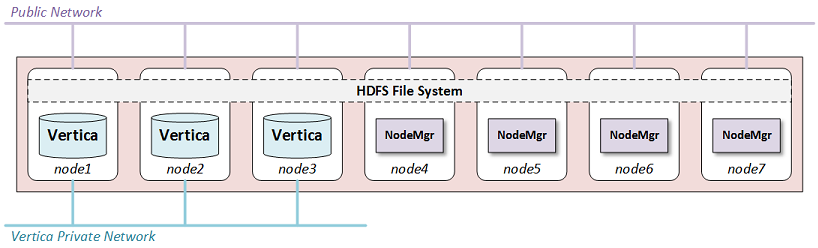
With either architecture, if you are using the hdfs scheme to read ORC or Parquet files, you must do some additional configuration. See Configuring HDFS access .

In this section

- Co-located clusters
- Configuring Hadoop for co-located clusters
- Configuring rack locality
- Separate clusters

Co-located clusters

With co-located clusters, Vertica is installed on some or all of your Hadoop nodes. The Vertica nodes use a private network in addition to the public network used by all Hadoop nodes, as the following figure shows:



You might choose to place Vertica on all of your Hadoop nodes or only on some of them. If you are using HDFS Storage Locations you should use at least three Vertica nodes, the minimum number for K-safety in an Enterprise Mode database .

Using more Vertica nodes can improve performance because the HDFS data needed by a query is more likely to be local to the node.

You can place Hadoop and Vertica clusters within a single rack, or you can span across many racks and nodes. If you do not co-locate Vertica on every node, you can improve performance by co-locating it on at least one node in each rack. See Configuring rack locality .

Normally, both Hadoop and Vertica use the entire node. Because this configuration uses shared nodes, you must address potential resource contention in your configuration on those nodes. See Configuring Hadoop for co-located clusters for more information. No changes are needed on Hadoop-only nodes.

Hardware recommendations

Hadoop clusters frequently do not have identical provisioning requirements or hardware configurations. However, Vertica nodes should be equivalent in size and capability, per the best-practice standards recommended in Platform and hardware requirements and recommendations .

Because Hadoop cluster specifications do not always meet these standards, Vertica recommends the following specifications for Vertica nodes in your Hadoop cluster.

Specifications for...	**Recommendation**
Processor	For best performance, run: • Two-socket servers with 8–14 core CPUs, clocked at or above 2.6 GHz for clusters over 10 TB • Single-socket servers with 8–12 cores clocked at or above 2.6 GHz for clusters under 10 TB

Memory	Distribute the memory appropriately across all memory channels in the server: • **Minimum** —8 GB of memory per physical CPU core in the server • **High-performance applications** —12–16 GB of memory per physical core • **Type** —at least DDR3-1600, preferably DDR3-1866
Storage	Read/write: • **Minimum** —40 MB/s per physical core of the CPU • **For best performance** —60–80 MB/s per physical core Storage post RAID: Each node should have 1–9 TB. For a production setting, Vertica recommends RAID 10. In some cases, RAID 50 is acceptable. Because Vertica performs heavy compression and encoding, SSDs are not required. In most cases, a RAID of more, less-expensive HDDs performs just as well as a RAID of fewer SSDs. If you intend to use RAID 50 for your data partition, you should keep a spare node in every rack, allowing for manual failover of a Vertica node in the case of a drive failure. A Vertica node recovery is faster than a RAID 50 rebuild. Also, be sure to never put more than 10 TB compressed on any node, to keep node recovery times at an acceptable rate.
Network	10 GB networking in almost every case. With the introduction of 10 GB over cat6a (Ethernet), the cost difference is minimal.

Configuring Hadoop for co-located clusters

If you are co-locating Vertica on any HDFS nodes, there are some additional configuration requirements.

Hadoop configuration parameters

For best performance, set the following parameters with the specified minimum values:

Parameter	Minimum Value
HDFS block size	512MB
Namenode Java Heap	1GB
Datanode Java Heap	2GB

WebHDFS

Hadoop has two services that can provide web access to HDFS:

- WebHDFS
- httpFS

For Vertica, you must use the WebHDFS service.

YARN

The YARN service is available in newer releases of Hadoop. It performs resource management for Hadoop clusters. When co-locating Vertica on YARN-managed Hadoop nodes you must make some changes in YARN.

Vertica recommends reserving at least 16GB of memory for Vertica on shared nodes. Reserving more will improve performance. How you do this depends on your Hadoop distribution:

- If you are using Hortonworks, create a "Vertica" node label and assign this to the nodes that are running Vertica.
- If you are using Cloudera, enable and configure static service pools.

Consult the documentation for your Hadoop distribution for details. Alternatively, you can disable YARN on the shared nodes.

Hadoop balancer

The Hadoop Balancer can redistribute data blocks across HDFS. For many Hadoop services, this feature is useful. However, for Vertica this can reduce performance under some conditions.

If you are using HDFS storage locations, the Hadoop load balancer can move data away from the Vertica nodes that are operating on it, degrading performance. This behavior can also occur when reading ORC or Parquet files if Vertica is not running on all Hadoop nodes. (If you are using separate Vertica and Hadoop clusters, all Hadoop access is over the network, and the performance cost is less noticeable.)

To prevent the undesired movement of data blocks across the HDFS cluster, consider excluding Vertica nodes from rebalancing. See the Hadoop documentation to learn how to do this.

Replication factor

By default, HDFS stores three copies of each data block. Vertica is generally set up to store two copies of each data item through K-Safety. Thus, lowering the replication factor to 2 can save space and still provide data protection.

To lower the number of copies HDFS stores, set HadoopFSReplication, as explained in Troubleshooting HDFS storage locations .

Disk space for Non-HDFS use

You also need to reserve some disk space for non-HDFS use. To reserve disk space using Ambari, set dfs.datanode.du.reserved to a value in the hdfs-site.xml configuration file.

Setting this parameter preserves space for non-HDFS files that Vertica requires.

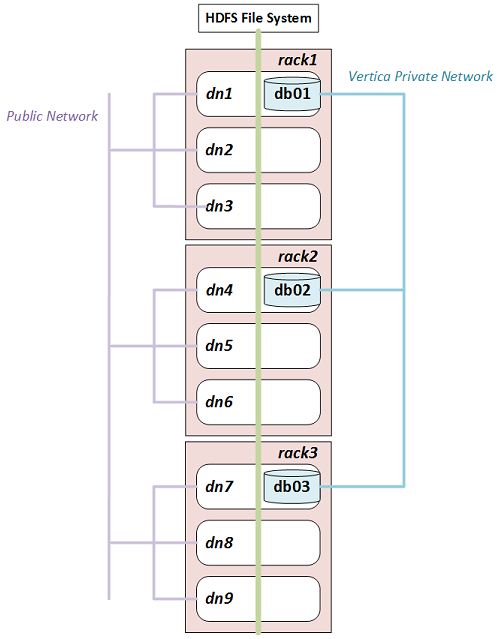
Configuring rack locality

> **Note**
> This feature is supported only for reading ORC and Parquet data on co-located clusters. It is only meaningful on Hadoop clusters that span multiple racks.

When possible, when planning a query Vertica automatically uses database nodes that are co-located with the HDFS nodes that contain the data. Moving query execution closer to the data reduces network latency and can improve performance. This behavior, called node locality, requires no additional configuration.

When Vertica is co-located on only a subset of HDFS nodes, sometimes there is no database node that is co-located with the data. However, performance is usually better if a query uses a database node in the same rack. If configured with information about Hadoop rack structure, Vertica attempts to use a database node in the same rack as the data to be queried.

For example, the following diagram illustrates a Hadoop cluster with three racks each containing three data nodes. (Typical production systems have more data nodes per rack.) In each rack, Vertica is co-located on one node.



If you configure rack locality, Vertica uses db01 to query data on dn1, dn2, or dn3, and uses db02 and db03 for data on rack2 and rack3 respectively. Because HDFS replicates data, any given data block can exist in more than one rack. If a data block is replicated on dn2, dn3, and dn6, for example, Vertica uses either db01 or db02 to query it.

Hadoop components are rack-aware, so configuration files describing rack structure already exist in the Hadoop cluster. To use this information in Vertica, configure fault groups that describe this rack structure. Vertica uses fault groups in query planning.

Configuring fault groups

Vertica uses Fault groups to describe physical cluster layout. Because your database nodes are co-located on HDFS nodes, Vertica can use the information about the physical layout of the HDFS cluster.

> **Tip**
> For best results, ensure that each Hadoop rack contains at least one co-located Vertica node.

Hadoop stores its cluster-layout data in a topology mapping file in HADOOP_CONF_DIR. On HortonWorks the file is typically named topology_mappings.data. On Cloudera it is typically named topology.map. Use the data in this file to create an input file for the fault-group script. For more information about the format of this file, see Creating a fault group input file .

Following is an example topology mapping file for the cluster illustrated previously:

```
[network_topology]
dn1.example.com=/rack1
10.20.41.51=/rack1
dn2.example.com=/rack1
10.20.41.52=/rack1
dn3.example.com=/rack1
10.20.41.53=/rack1
dn4.example.com=/rack2
10.20.41.71=/rack2
dn5.example.com=/rack2
10.20.41.72=/rack2
dn6.example.com=/rack2
10.20.41.73=/rack2
dn7.example.com=/rack3
10.20.41.91=/rack3
dn8.example.com=/rack3
10.20.41.92=/rack3
dn9.example.com=/rack3
10.20.41.93=/rack3
```

From this data, you can create the following input file describing the Vertica subset of this cluster:

```
/rack1 /rack2 /rack3
/rack1 = db01
/rack2 = db02
/rack3 = db03
```

This input file tells Vertica that the database node "db01" is on rack1, "db02" is on rack2, and "db03" is on rack3. In creating this file, ignore Hadoop data nodes that are not also Vertica nodes.

After you create the input file, run the fault-group tool:

```
$ python /opt/vertica/scripts/fault_group_ddl_generator.py dbName input_file > fault_group_ddl.sql
```

The output of this script is a SQL file that creates the fault groups. Execute it following the instructions in Creating fault groups .

You can review the new fault groups with the following statement:

```
=> SELECT member_name,node_address,parent_name FROM fault_groups
INNER JOIN nodes ON member_name=node_name ORDER BY parent_name;

 member_name  | node_address  | parent_name
--------------------------+--------------+-------------
db01          | 10.20.41.51   | /rack1
db02          | 10.20.41.71   | /rack2
db03          | 10.20.41.91   | /rack3

(3 rows)
```

Working with multi-level racks

A Hadoop cluster can use multi-level racks. For example, /west/rack-w1, /west/rack-2, and /west/rack-w3 might be served from one data center, while /east/rack-e1, /east/rack-e2, and /east/rack-e3 are served from another. Use the following format for entries in the input file for the fault-group script:

```
/west /east
/west = /rack-w1 /rack-w2 /rack-w3
/east = /rack-e1 /rack-e2 /rack-e3
/rack-w1 = db01
/rack-w2 = db02
/rack-w3 = db03
/rack-e1 = db04
/rack-e2 = db05
/rack-e3 = db06
```

Do not create entries using the full rack path, such as /west/rack-w1.

Auditing results

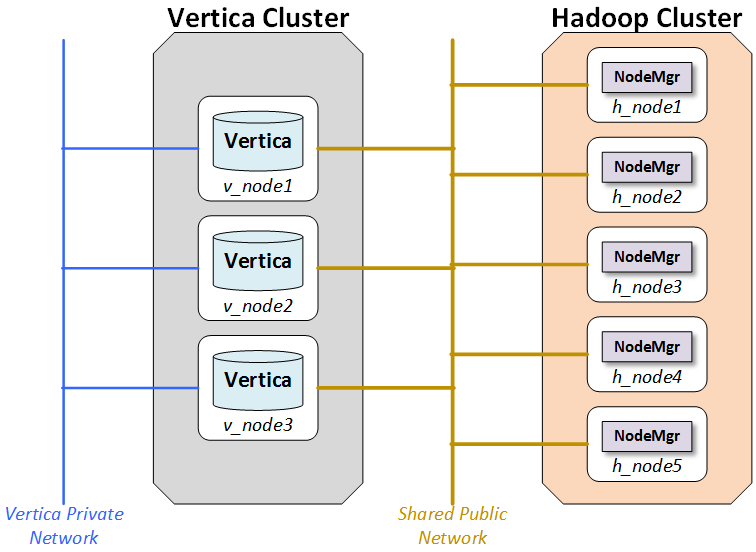
To see how much data can be loaded with rack locality, use EXPLAIN with the query and look for statements like the following in the output:

```
100% of ORC data including co-located data can be loaded with rack locality.
```

Separate clusters

With separate clusters, a Vertica cluster and a Hadoop cluster share no nodes. You should use a high-bandwidth network connection between the two clusters.

The following figure illustrates the configuration for separate clusters::



Network

The network is a key performance component of any well-configured cluster. When Vertica stores data to HDFS it writes and reads data across the network.

The layout shown in the figure calls for two networks, and there are benefits to adding a third:

- Database Private Network: Vertica uses a private network for command and control and moving data between nodes in support of its database functions. In some networks, the command and control and passing of data are split across two networks.
- Database/Hadoop Shared Network: Each Vertica node must be able to connect to each Hadoop data node and the NameNode. Hadoop best practices generally require a dedicated network for the Hadoop cluster. This is not a technical requirement, but a dedicated network improves Hadoop performance. Vertica and Hadoop should share the dedicated Hadoop network.
- Optional Client Network: Outside clients may access the clustered networks through a client network. This is not an absolute requirement, but the use of a third network that supports client connections to either Vertica or Hadoop can improve performance. If the configuration does not support a client network, than client connections should use the shared network.

Hadoop configuration parameters

For best performance, set the following parameters with the specified minimum values:

Parameter	**Minimum Value**
HDFS block size	512MB
Namenode Java Heap	1GB
Datanode Java Heap	2GB

Configuring HDFS access

Vertica uses information from the Hadoop cluster configuration to support reading data (COPY or external tables). In Eon Mode, it also uses this information to access communal storage on HDFS. Vertica nodes therefore must have access to certain Hadoop configuration files.

For both co-located and separate clusters that use Kerberos authentication, configure Vertica for Kerberos as explained in Configure Vertica for Kerberos Authentication .

Vertica requires access to the WebHDFS service and ports on all name nodes and data nodes. For more information about WebHDFS ports, see HDFS Ports in the Cloudera documentation .

Accessing Hadoop configuration files

Your Vertica nodes need access to certain Hadoop configuration files:

- If Vertica is co-located on HDFS nodes, then those configuration files are already present.
- If Vertica is running on a separate cluster, you must copy the required files to all database nodes. A simple way to do so is to configure your Vertica nodes as Hadoop edge nodes. Client applications run on *edge nodes* ; from Hadoop's perspective, Vertica is a client application. You can use Ambari or Cloudera Manager to configure edge nodes. For more information, see the documentation from your Hadoop vendor.

Verify that the value of the HadoopConfDir configuration parameter (see Hadoop parameters) includes a directory containing the core-site.xml and hdfs-site.xml files. If you do not set a value, Vertica looks for the files in /etc/hadoop/conf. For all Vertica users, the directory is accessed by the Linux user under which the Vertica server process runs.

Vertica uses several properties defined in these configuration files. These properties are listed in HDFS file system .

Using a cluster with high availability NameNodes

If your Hadoop cluster uses High Availability (HA) Name Nodes, verify that the dfs.nameservices parameter and the individual name nodes are defined in hdfs-site.xml .

Using more than one Hadoop cluster

In some cases, a Vertica cluster requires access to more than one HDFS cluster. For example, your business might use separate HDFS clusters for separate regions, or you might need data from both test and deployment clusters.

To support multiple clusters, perform the following steps:

1. Copy the configuration files from all HDFS clusters to your database nodes. You can place the copied files in any location readable by Vertica. However, as a best practice, you should place them all in the same directory tree, with one subdirectory per HDFS cluster. The locations must be the same on all database nodes.
2. Set the HadoopConfDir configuration parameter. The value is a colon-separated path containing the directories for all of your HDFS clusters.
3. Use an explicit name node or name service in the URL when creating an external table or copying data. Do not use hdfs:/// because it could be ambiguous. For more information about URLs, see HDFS file system .

Vertica connects directly to a name node or name service; it does not otherwise distinguish among HDFS clusters. Therefore, names of HDFS name nodes and name services must be globally unique.

Verifying the configuration

Use the VERIFY_HADOOP_CONF_DIR function to verify that Vertica can find configuration files in HadoopConfDir.

Use the HDFS_CLUSTER_CONFIG_CHECK function to test access through the hdfs scheme.

For more information about testing your configuration, see Verifying HDFS configuration .

Updating configuration files

If you update the configuration files after starting Vertica, use the following statement to refresh them:

```
=> SELECT CLEAR_HDFS_CACHES();
```

The CLEAR_HDFS_CACHES function also flushes information about which name node is active in a High Availability (HA) Hadoop cluster. Therefore, the first request after calling this function is slow, because the initial connection to the name node can take more than 15 seconds.

In this section

- Verifying HDFS configuration
- Troubleshooting reads from HDFS

Verifying HDFS configuration

Use the EXTERNAL_CONFIG_CHECK function to test access to HDFS. This function calls several others. If you prefer to test individual components, or if some tests do not apply to your configuration, you can instead call the functions individually. For example, if you are not using the HCatalog Connector then you do not need to call that function. The functions are:

- KERBEROS_CONFIG_CHECK : tests the Vertica keytab and the user's Kerberos credential.
- HADOOP_IMPERSONATION_CONFIG_CHECK : shows the delegation tokens that are in use. This function does not test them.
- HDFS_CLUSTER_CONFIG_CHECK : tests access to the HDFS clusters found in HadoopConfDir, including using Kerberos and impersonation (delegation tokens).
- HCATALOGCONNECTOR_CONFIG_CHECK : tests HCatalog Connector access to HiveServer2.

To run all tests, call EXTERNAL_CONFIG_CHECK with no arguments:

```
=> SELECT EXTERNAL_CONFIG_CHECK();
```

To test only some authorities, nameservices, or Hive schemas, pass a single string argument. The format is a comma-separated list of "key=value" pairs, where keys are "authority", "nameservice", and "schema". The value is passed to all of the sub-functions; see those reference pages for details on how values are interpreted.

The following example tests the configuration of only the nameservice named "ns1":

```
=> SELECT EXTERNAL_CONFIG_CHECK('nameservice=ns1');
```

Troubleshooting reads from HDFS

You might encounter the following issues when accessing data in HDFS.

Queries using [web]hdfs:/// show unexpected results

If you are using the /// shorthand to query external tables and see unexpected results, such as production data in your test cluster, verify that HadoopConfDir is set to the value you expect. The HadoopConfDir configuration parameter defines a path to search for the Hadoop configuration files that Vertica needs to resolve file locations. The HadoopConfDir parameter can be set at the session level, overriding the permanent value set in the database.

To debug problems with /// URLs, try replacing the URLs with ones that use an explicit nameservice or name node. If the explicit URL works, then the problem is with the resolution of the shorthand. If the explicit URL also does not work as expected, then the problem is elsewhere (such as your nameservice).

Queries take a long time to run when using HA

The High Availability Name Node feature in HDFS allows a name node to fail over to a standby name node. The dfs.client.failover.max.attempts configuration parameter (in hdfs-site.xml) specifies how many attempts to make when failing over. Vertica uses a default value of 4 if this parameter is not set. After reaching the maximum number of failover attempts, Vertica concludes that the HDFS cluster is unavailable and aborts the operation.

Vertica uses the dfs.client.failover.sleep.base.millis and dfs.client.failover.sleep.max.millis parameters to decide how long to wait between retries. Typical ranges are 500 milliseconds to 15 seconds, with longer waits for successive retries.

A second parameter, ipc.client.connect.retry.interval , specifies the time to wait between attempts, with typical values being 10 to 20 seconds.

Cloudera and Hortonworks both provide tools to automatically generate configuration files. These tools can set the maximum number of failover attempts to a much higher number (50 or 100). If the HDFS cluster is unavailable (all name nodes are unreachable), Vertica can appear to hang for an extended period (minutes to hours) while trying to connect.

Failover attempts are logged in the QUERY_EVENTS system table. The following example shows how to query this table to find these events:

```
=> SELECT event_category, event_type, event_description, operator_name,
   event_details, count(event_type) AS count
   FROM query_events
   WHERE event_type ilike 'WEBHDFS FAILOVER RETRY'
   GROUP BY event_category, event_type, event_description, operator_name, event_details;
-[ RECORD 1 ]-----+----------------------------------------
event_category    | EXECUTION
event_type        | WEBHDFS FAILOVER RETRY
event_description | WebHDFS Namenode failover and retry.
operator_name     | WebHDFS FileSystem
event_details     | WebHDFS request failed on ns
count             | 4
```

You can either wait for Vertica to complete or abort the connection, or set the dfs.client.failover.max.attempts parameter to a lower value.

WebHDFS error when using LibHDFS++

When creating an external table or loading data and using the hdfs scheme, you might see errors from WebHDFS failures. Such errors indicate that Vertica was not able to use the hdfs scheme and fell back to webhdfs , but that the WebHDFS configuration is incorrect.

First verify the value of the HadoopConfDir configuration parameter, which can be set at the session level. Then verify that the HDFS configuration files found there have the correct WebHDFS configuration for your Hadoop cluster. See Configuring HDFS access for information about use of these files. See your Hadoop documentation for information about WebHDFS configuration.

Vertica places too much load on the name node (LibHDFS++)

Large HDFS clusters can sometimes experience heavy load on the name node when clients, including Vertica, need to locate data. If your name node is sensitive to this load and if you are using LibHDFS++, you can instruct Vertica to distribute metadata about block locations to its nodes so that they do not have to contact the name node as often. Distributing this metadata can degrade database performance somewhat in deployments where the name node isn't contended. This performance effect is because the data must be serialized and distributed.

If protecting your name node from load is more important than query performance, set the EnableHDFSBlockInfoCache configuration parameter to 1 (true). Usually this applies to large HDFS clusters where name node contention is already an issue.

This setting applies to access through LibHDFS++ (hdfs scheme). Sometimes LibHDFS++ falls back to WebHDFS, which does not use this setting. If you have enabled this setting and you are still seeing high traffic on your name node from Vertica, check the QUERY_EVENTS system table for LibHDFS++ UNSUPPORTED OPERATION events.

Kerberos authentication errors

Kerberos authentication can fail even though a ticket is valid if Hadoop expires tickets frequently. It can also fail due to clock skew between Hadoop and Vertica nodes. For details, see Troubleshooting Kerberos authentication .

Accessing kerberized HDFS data

If your Hortonworks or Cloudera Hadoop cluster uses Kerberos authentication to restrict access to HDFS, Vertica must be granted access. If you use Kerberos authentication for your Vertica database, and database users have HDFS access, then you can configure Vertica to use the same principals to authenticate. However, not all Hadoop administrators want to grant access to all database users, preferring to manage access in other ways. In addition, not all Vertica databases use Kerberos.

Vertica provides the following options for accessing Kerberized Hadoop clusters:

- Using Vertica Kerberos principals.
- Using Hadoop proxy users combined with Vertica users.
- Using a user-specified delegation token which grants access to HDFS. Delegation tokens are issued by the Hadoop cluster.

Proxy users and delegation tokens both use a session parameter to manage identities. Vertica need not be Kerberized when using either of these methods.

Vertica does not support Kerberos for MapR.

To test your Kerberos configuration, see Verifying HDFS configuration .

In this section

- Using Kerberos with Vertica
- Proxy users and delegation tokens
- Token expiration

Using Kerberos with Vertica

If you use Kerberos for your Vertica cluster and your principals have access to HDFS, then you can configure Vertica to use the same credentials for HDFS.

Vertica authenticates with Hadoop in two ways that require different configurations:

- **User Authentication** —On behalf of the user, by passing along the user's existing Kerberos credentials. This method is also called user impersonation. Actions performed on behalf of particular users, like executing queries, generally use user authentication.
- **Vertica Authentication** —On behalf of system processes that access ROS data or the catalog, by using a special Kerberos credential stored in a keytab file.

> **Note**
> Vertica and Hadoop must use the same Kerberos server or servers (KDCs).

Vertica can interact with more than one Kerberos realm. To configure multiple realms, see Multi-realm Support .

Vertica attempts to automatically refresh Hadoop tokens before they expire. See Token expiration .

User authentication

To use Vertica with Kerberos and Hadoop, the client user first authenticates with one of the Kerberos servers (Key Distribution Center, or KDC) being used by the Hadoop cluster. A user might run kinit or sign in to Active Directory, for example.

A user who authenticates to a Kerberos server receives a Kerberos ticket. At the beginning of a client session, Vertica automatically retrieves this ticket. Vertica then uses this ticket to get a Hadoop token, which Hadoop uses to grant access. Vertica uses this token to access HDFS, such as when executing a query on behalf of the user. When the token expires, Vertica automatically renews it, also renewing the Kerberos ticket if necessary.

The user must have been granted permission to access the relevant files in HDFS. This permission is checked the first time Vertica reads HDFS data.

Vertica can use multiple KDCs serving multiple Kerberos realms, if proper cross-realm trust has been set up between realms.

Vertica authentication

Automatic processes, such as the Tuple Mover or the processes that access Eon Mode communal storage, do not log in the way users do. Instead, Vertica uses a special identity (principal) stored in a keytab file on every database node. (This approach is also used for Vertica clusters that use Kerberos but do not use Hadoop.) After you configure the keytab file, Vertica uses the principal residing there to automatically obtain and maintain a Kerberos ticket, much as in the client scenario. In this case, the client does not interact with Kerberos.

Each Vertica node uses its own principal; it is common to incorporate the name of the node into the principal name. You can either create one keytab per node, containing only that node's principal, or you can create a single keytab containing all the principals and distribute the file to all nodes. Either way, the node uses its principal to get a Kerberos ticket and then uses that ticket to get a Hadoop token.

When creating HDFS storage locations Vertica uses the principal in the keytab file, not the principal of the user issuing the CREATE LOCATION statement. The HCatalog Connector sometimes uses the principal in the keytab file, depending on how Hive authenticates users.

Configuring users and the keytab file

If you have not already configured Kerberos authentication for Vertica, follow the instructions in Configure Vertica for Kerberos authentication . Of particular importance for Hadoop integration:

1. Create one Kerberos principal per node.
2. Place the keytab files in the same location on each database node and set configuration parameter KerberosKeytabFile to that location.
3. Set KerberosServiceName to the name of the principal. (See Inform Vertica about the Kerberos principal .)

If you are using the HCatalog Connector, follow the additional steps in Configuring security in the HCatalog Connector documentation.

If you are using HDFS storage locations, give all node principals read and write permission to the HDFS directory you will use as a storage location.

Proxy users and delegation tokens

An alternative to granting HDFS access to individual Vertica users is to use delegation tokens, either directly or with a proxy user. In this configuration, Vertica accesses HDFS on behalf of some other (Hadoop) user. The Hadoop users need not be Vertica users at all.

In Vertica, you can either specify the name of the Hadoop user to act on behalf of (doAs), or you can directly use a Kerberos delegation token that you obtain from HDFS (Bring Your Own Delegation Token). In the doAs case, Vertica obtains a delegation token for that user, so both approaches ultimately use delegation tokens to access files in HDFS.

Use the HadoopImpersonationConfig session parameter to specify a user or delegation token to use for HDFS access. Each session can use a different user and can use either doAs or a delegation token. The value of HadoopImpersonationConfig is a set of JSON objects.

To use delegation tokens of either type (more specifically, when HadoopImpersonationConfig is set), you must access HDFS through WebHDFS.

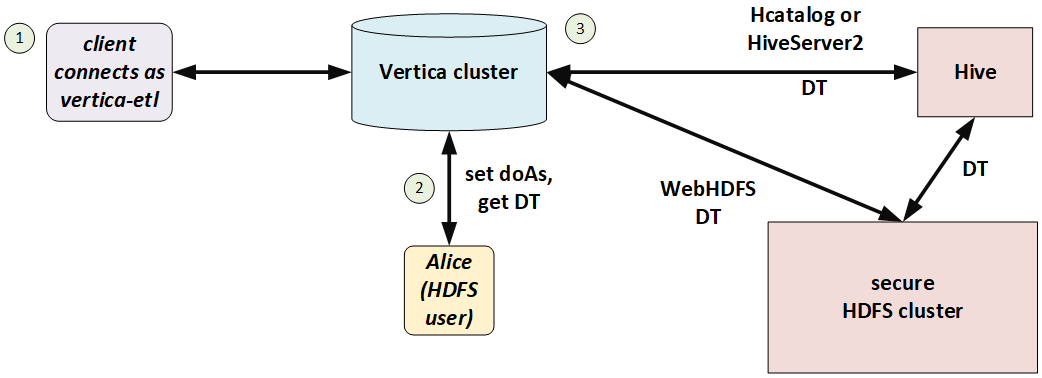
In this section

- User impersonation (doAs)
- Bring your own delegation token
- Getting a HiveServer2 delegation token
- HadoopImpersonationConfig format

User impersonation (doAs)

You can use user impersonation to access data in an HDFS cluster from Vertica. This approach is called "doAs" (for "do as") because Vertica uses a single proxy user on behalf of another (Hadoop) user. The impersonated Hadoop user does not need to also be a Vertica user.

In the following illustration, Alice is a Hadoop user but not a Vertica user. She connects to Vertica as the proxy user, vertica-etl. In her session, Vertica obtains a delegation token (DT) on behalf of the doAs user (Alice), and uses that delegation token to access HDFS.



You can use doAs with or without Kerberos, so long as HDFS and Vertica match. If HDFS uses Kerberos then Vertica must too.

User configuration

The Hadoop administrator must create a proxy user and allow it to access HDFS on behalf of other users. Set values in core-site.xml as in the following example:

```
<name>hadoop.proxyuser.vertica-etl.users</name>
<value>*</value>
<name>hadoop.proxyuser.vertica-etl.hosts</name>
<value>*</value>
```

In Vertica, create a corresponding user.

Session configuration

To make requests on behalf of a Hadoop user, first set the HadoopImpersonationConfig session parameter to specify the user and HDFS cluster. Vertica will access HDFS as that user until the session ends or you change the parameter.

The value of this session parameter is a collection of JSON objects. Each object specifies an HDFS cluster and a Hadoop user. For the cluster, you can specify either a name service or an individual name node. If you are using HA name node, then you must either use a name service or specify all name nodes. HadoopImpersonationConfig format describes the full JSON syntax.

The following example shows access on behalf of two different users. The users "stephanie" and "bob" are Hadoop users, not Vertica users. "vertica-etl" is a Vertica user.

```
$ vsql -U vertica-etl

=> ALTER SESSION SET
  HadoopImpersonationConfig = '[{"nameservice":"hadoopNS", "doAs":"stephanie"}]';
=> COPY nation FROM 'webhdfs:///user/stephanie/nation.dat';

=> ALTER SESSION SET
  HadoopImpersonationConfig = '[{"nameservice":"hadoopNS", "doAs":"bob"}, {"authority":"hadoop2:50070", "doAs":"rob"}]';
=> COPY nation FROM 'webhdfs:///user/bob/nation.dat';
```

Vertica uses Hadoop delegation tokens, obtained from the name node, to impersonate Hadoop users. In a long-running session, a token could expire. Vertica attempts to renew tokens automatically; see Token expiration .

Testing the configuration

You can use the HADOOP_IMPERSONATION_CONFIG_CHECK function to test your HDFS delegation tokens and HCATALOGCONNECTOR_CONFIG_CHECK to test your HCatalog Connector delegation token.

Bring your own delegation token

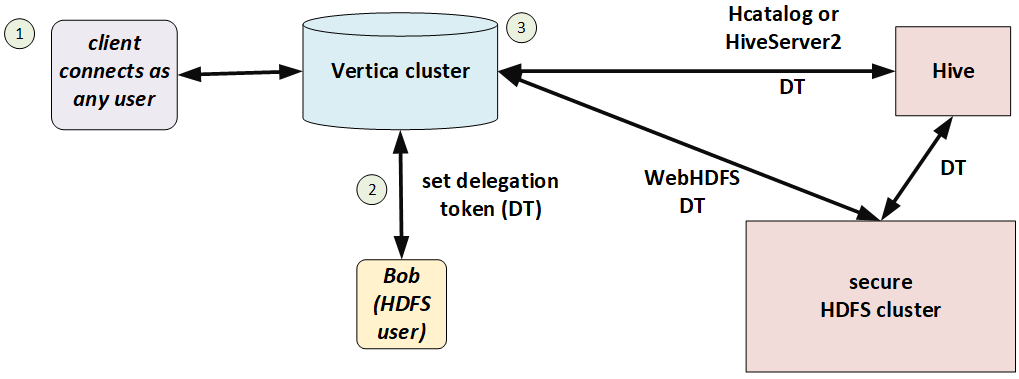
Instead of creating a proxy user and giving it access to HDFS for use with doAs, you can give Vertica a Hadoop delegation token to use. You must obtain this delegation token from the Hadoop name node. In this model, security is handled entirely on the Hadoop side, with Vertica just passing along a token. Vertica may or may not be Kerberized.

A typical workflow is:

- In an ETL front end, a user submits a query.
- The ETL system uses authentication and authorization services to verify that the user has sufficient permission to run the query.
- The ETL system requests a delegation token for the user from the name node.
- The ETL system makes a client connection to Vertica, sets the delegation token for the session, and runs the query.

When using a delegation token, clients can connect as any Vertica user. No proxy user is required.

In the following illustration, Bob has a Hadoop-issued delegation token. He connects to Vertica and Vertica uses that delegation token to access files in HDFS.



Session configuration

Set the HadoopImpersonationConfig session parameter to specify the delegation token and HDFS cluster. Vertica will access HDFS using that delegation token until the session ends, the token expires, or you change the parameter.

The value of this session parameter is a collection of JSON objects. Each object specifies a delegation token ("token") in WebHDFS format and an HDFS name service or name node. HadoopImpersonationConfig format describes the full JSON syntax.

The following example shows access on behalf of two different users. The users "stephanie" and "bob" are Hadoop users, not Vertica users. "dbuser1" is a Vertica user with no special privileges.

```
$ vsql -U dbuser1

=> ALTER SESSION SET
  HadoopImpersonationConfig ='[{"authority":"hadoop1:50070","token":"JAAGZGJldGwxBmRiZXRsMQCKAWDXJgB9igFg-
zKEfY4gao4BmhSJYtXiWqrhBHbbUn4VScNg58HWQxJXRUJIREZTIGRlbGVnYXRpb24RMTAuMjAuMTAwLjU0OjgwMjA"}]';
=> COPY nation FROM 'webhdfs:///user/stephanie/nation.dat';

=> ALTER SESSION SET
  HadoopImpersonationConfig ='[{"authority":"hadoop1:50070","token":"HgADdG9tA3RvbQCKAWDXJgAoigFg-
zKEKI4gaI4BmhRoOUpq_jPxrVhZ1NSMnodAQnhUthJXRUJIREZTIGRlbGVnYXRpb24RMTAuMjAuMTAwLjU0OjgwMjA"}]';
=> COPY nation FROM 'webhdfs:///user/bob/nation.dat';
```

You can use the WebHDFS REST API to get delegation tokens:

```
$ curl -s --noproxy "*" --negotiate -u: -X GET "http://hadoop1:50070/webhdfs/v1/?op=GETDELEGATIONTOKEN"
```

Vertica does not, and cannot, renew delegation tokens when they expire. You must either keep sessions shorter than token lifetime or implement a renewal scheme.

Delegation tokens and the HCatalog Connector

HiveServer2 uses a different format for delegation tokens. To use the HCatalog Connector, therefore, you must set two delegation tokens, one as usual (authority) and one for HiveServer2 (schema). The HCatalog Connector uses the schema token to access metadata and the authority token to access data. The schema name is the same Hive schema you specified in CREATE HCATALOG SCHEMA. The following example shows how to use these two delegation tokens.

```
$ vsql -U dbuser1

-- set delegation token for user and HiveServer2
=> ALTER SESSION SET
  HadoopImpersonationConfig='[
    {"nameservice":"hadoopNS","token":"JQAHcmVsZWFzZQdyZWxlYXNlAIoBYVJKrYSKAWF2VzGEjgmzj_IUCIrI9b8Dqu6awFTHk5nC-
fHB8xsSV0VCSERGUyBkZWxlZ2F0aW9uETEwLjIwLjQyLjEwOTo4MDIw"},

{"schema":"access","token":"UwAHcmVsZWFzZQdyZWxlYXNlL2hpdmUvZW5nLWc5LTEwMC52ZXJ0aWNhY29ycC5jb21AVkVSVElDQUNPUlAuQ09NigFhU
kmyTooBYXZWNk4BjgETFKN2xPURn19Yq9tf-0nekoD51TZvFUhJVkVfREVMRUdBVElPTl9UT0tFThZoaXZlc2VydmVyMkNsaWVudFRva2Vu"}]';

-- uses HiveServer2 token to get metadata
=> CREATE HCATALOG SCHEMA access WITH hcatalog_schema 'access';

-- uses both tokens
=> SELECT * FROM access.t1;

--uses only HiveServer2 token
=> SELECT * FROM hcatalog_tables;
```

HiveServer2 does not provide a REST API for delegation tokens like WebHDFS does. See Getting a HiveServer2 delegation token for some tips.

Testing the configuration

You can use the HADOOP_IMPERSONATION_CONFIG_CHECK function to test your HDFS delegation tokens and HCATALOGCONNECTOR_CONFIG_CHECK to test your HCatalog Connector delegation token.

Getting a HiveServer2 delegation token

To acccess Hive metadata using HiveServer2, you need a special delegation token. (See Bring your own delegation token .) HiveServer2 does not provide an easy way to get this token, unlike the REST API that grants HDFS (data) delegation tokens.

The following utility code shows a way to get this token. You will need to modify this code for your own cluster; in particular, change the value of the connectURL static.

```
import java.io.FileWriter;
import java.io.PrintStream;
import java.io.PrintWriter;
import java.io.StringWriter;
```

```
import java.io.Writer;
import java.security.PrivilegedExceptionAction;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hive.shims.Utils;
import org.apache.hadoop.security.UserGroupInformation;
import org.apache.hive.jdbc.HiveConnection;
import org.json.simple.JSONArray;
import org.json.simple.JSONObject;

public class JDBCTest {
  public static final String driverName = "org.apache.hive.jdbc.HiveDriver";
  public static String connectURL =
"jdbc:hive2://node2.cluster0.example.com:2181,node1.cluster0.example.com:2181,node3.cluster0.example.com:2181/;serviceDiscoveryMode=zooKeeper;zoo
eeperNamespace=hiveserver2";
  public static String schemaName = "hcat";
  public static String verticaUser = "condor";
  public static String proxyUser = "condor-2";
  public static String krb5conf = "/home/server/kerberos/krb5.conf";
  public static String realm = "EXAMPLE.COM";
  public static String keytab = "/home/server/kerberos/kt.keytab";

  public static void main(String[] args) {
    if (args.length < 7) {
      System.out.println(
          "Usage: JDBCTest <jdbc_url> <hive_schema> <kerberized_user> <proxy_user> <krb5_conf> <krb_realm> <krb_keytab>");
      System.exit(1);
    }
    connectURL = args[0];
    schemaName = args[1];
    verticaUser = args[2];
    proxyUser = args[3];
    krb5conf = args[4];
    realm = args[5];
    keytab = args[6];

    System.out.println("connectURL: " + connectURL);
    System.out.println("schemaName: " + schemaName);
    System.out.println("verticaUser: " + verticaUser);
    System.out.println("proxyUser: " + proxyUser);
    System.out.println("krb5conf: " + krb5conf);
    System.out.println("realm: " + realm);
    System.out.println("keytab: " + keytab);
    try {
      Class.forName("org.apache.hive.jdbc.HiveDriver");
      System.out.println("Found HiveServer2 JDBC driver");
    } catch (ClassNotFoundException e) {
      System.out.println("Couldn't find HiveServer2 JDBC driver");
    }
    try {
      Configuration conf = new Configuration();
      System.setProperty("java.security.krb5.conf", krb5conf);
      conf.set("hadoop.security.authentication", "kerberos");
      UserGroupInformation.setConfiguration(conf);
      dtTest();
    } catch (Throwable e) {
      Writer stackString = new StringWriter();
      e.printStackTrace(new PrintWriter(stackString));
      System.out.println(e);
```

```
    System.out.println(e);
    System.out.printf("Error occurred when connecting to HiveServer2 with [%s]: %s\n%s\n",
        new Object[] { connectURL, e.getMessage(), stackString.toString() });
  }
 }

 private static void dtTest() throws Exception {
  UserGroupInformation user = UserGroupInformation.loginUserFromKeytabAndReturnUGI(verticaUser + "@" + realm, keytab);
  user.doAs(new PrivilegedExceptionAction() {
   public Void run() throws Exception {
    System.out.println("In doas: " + UserGroupInformation.getLoginUser());
    Connection con = DriverManager.getConnection(JDBCTest.connectURL);
    System.out.println("Connected to HiveServer2");
    JDBCTest.showUser(con);
    System.out.println("Getting delegation token for user");
    String token = ((HiveConnection) con).getDelegationToken(JDBCTest.proxyUser, "hive/_HOST@" + JDBCTest.realm);
    System.out.println("Got token: " + token);
    System.out.println("Closing original connection");
    con.close();

    System.out.println("Setting delegation token in UGI");
    Utils.setTokenStr(Utils.getUGI(), token, "hiveserver2ClientToken");
    con = DriverManager.getConnection(JDBCTest.connectURL + ";auth=delegationToken");
    System.out.println("Connected to HiveServer2 with delegation token");
    JDBCTest.showUser(con);
    con.close();

    JDBCTest.writeDTJSON(token);

    return null;
   }
  });
 }

 private static void showUser(Connection con) throws Exception {
  String sql = "select current_user()";
  Statement stmt = con.createStatement();
  ResultSet res = stmt.executeQuery(sql);
  StringBuilder result = new StringBuilder();
  while (res.next()) {
   result.append(res.getString(1));
  }
  System.out.println("\tcurrent_user: " + result.toString());
 }

 private static void writeDTJSON(String token) {
  JSONArray arr = new JSONArray();
  JSONObject obj = new JSONObject();
  obj.put("schema", schemaName);
  obj.put("token", token);
  arr.add(obj);
  try {
   FileWriter fileWriter = new FileWriter("hcat_delegation.json");
   fileWriter.write(arr.toJSONString());
   fileWriter.flush();
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}
```

Following is an example call and its output:

```
$ java -cp hs2token.jar JDBCTest 'jdbc:hive2://test.example.com:10000/default;principal=hive/_HOST@EXAMPLE.COM' "default" "testuser" "test"
"/etc/krb5.conf" "EXAMPLE.COM" "/test/testuser.keytab"
connectURL: jdbc:hive2://test.example.com:10000/default;principal=hive/_HOST@EXAMPLE.COM
schemaName: default
verticaUser: testuser
proxyUser: test
krb5conf: /etc/krb5.conf
realm: EXAMPLE.COM
keytab: /test/testuser.keytab
Found HiveServer2 JDBC driver
log4j:WARN No appenders could be found for logger (org.apache.hadoop.metrics2.lib.MutableMetricsFactory).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
In doas: testuser@EXAMPLE.COM (auth:KERBEROS)
Connected to HiveServer2
      current_user: testuser
Getting delegation token for user
Got token:
JQAEdGVzdARoaXZlB3JlbGVhc2WKAWgvBOwzigFoUxFwMwKOAgMUHfqJ5ma7_27LiePN8C7MxJ682bsVSElWRV9ERUxFR0FUSU9OX1RPS0VOFmhpd
mVzZXJ2ZXIyQ2xpZW50VG9rZW4
Closing original connection
Setting delegation token in UGI
Connected to HiveServer2 with delegation token
      current_user: testuser
```

HadoopImpersonationConfig format

The value of the HadoopImpersonationConfig session parameter is a set of one or more JSON objects. Each object describes one doAs user or delegation token for one Hadoop destination. You must use WebHDFS, not LibHDFS++, to use impersonation.

Syntax

```
[ { ("doAs" | "token"): value,
   ("nameservice" | "authority" | "schema"): value} [,...]
]
```

Properties

doAs	The name of a Hadoop user to impersonate.
token	A delegation token to use for HDFS access.
nameservice	A Hadoop name service. All access to this name service uses the doAs user or delegation token.
authority	A name node authority. All access to this authority uses the doAs user or delegation token. If the name node fails over to another name node, the doAs user or delegation token does *not* automatically apply to the failover name node. If you are using HA name node, use nameservice instead of authority or include objects for every name node.
schema	A Hive schema, for use with the HCatalog Connector. Vertica uses this object's doAs user or token to access Hive metadata only. For data access you must also specify a name service or authority object, just like for all other data access.

Examples

In the following example of doAs, Bob is a Hadoop user and vertica-etl is a Kerberized proxy user.

```
$ kinit vertica-etl -kt /home/dbadmin/vertica-etl.keytab
$ vsql -U vertica-etl

=> ALTER SESSION SET
  HadoopImpersonationConfig = '[{"nameservice":"hadoopNS", "doAs":"Bob"}]';
=> COPY nation FROM 'webhdfs:///user/bob/nation.dat';
```

In the following example, the current Vertica user (it doesn't matter who that is) uses a Hadoop delegation token. This token belongs to Alice, but you never specify the user name here. Instead, you use it to get the delegation token from Hadoop.

```
$ vsql -U dbuser1

=> ALTER SESSION SET
   HadoopImpersonationConfig ='[{"nameservice":"hadoopNS","token":"JAAGZGJldGwxBmRiZXRsMQCKAWDXJgB9igFg-
zKEfY4gao4BmhSJYtXiWqrhBHbbUn4VScNg58HWQxJXRUJIREZTIGRlbGVnYXRpb24RMTAuMjAuMTAwLjU0OjgwMjA"}]';
=> COPY nation FROM 'webhdfs:///user/alice/nation.dat';
```

In the following example, "authority" specifies the (single) name node on a Hadoop cluster that does not use high availability.

```
$ vsql -U dbuser1

=> ALTER SESSION SET
   HadoopImpersonationConfig ='[{"authority":"hadoop1:50070", "doAs":"Stephanie"}]';
=> COPY nation FROM 'webhdfs://hadoop1:50070/user/stephanie/nation.dat';
```

To access data in Hive you need to specify two delegation tokens. The first, for a name service or authority, is for data access as usual. The second is for the HiveServer2 metadata for the schema. HiveServer2 requires a delegation token in WebHDFS format. The schema name is the Hive schema you specify with CREATE HCATALOG SCHEMA.

```
$ vsql -U dbuser1

-- set delegation token for user and HiveServer2
=> ALTER SESSION SET
   HadoopImpersonationConfig='[
     {"nameservice":"hadoopNS","token":"JQAHcmVsZWFzZQdyZWxlYXNlAIoBYVJKrYSKAWF2VzGEjgmzj_IUCIrI9b8Dqu6awFTHk5nC-
fHB8xsSV0VCSERGUyBkZWxlZ2F0aW9uETEwLjIwLjQyLjEwOTo4MDIw"},

{"schema":"access","token":"UwAHcmVsZWFzZQdyZWxlYXNlL2hpdmUvZW5nLWc5LTEwMC52ZXJ0aWNhY29ycC5jb21AVkVSVElDQUNPUlAuQ09NigFhU
kmyTooBYXZWNk4BjgETFKN2xPURn19Yq9tf-0nekoD51TZvFUhJVkVfREVMRUdBVElPTl9UT0tFThZoaXZlc2VydmVyMkNsaWVudFRva2Vu"}]';

-- uses HiveServer2 token to get metadata
=> CREATE HCATALOG SCHEMA access WITH hcatalog_schema 'access';

-- uses both tokens
=> SELECT * FROM access.t1;

--uses only HiveServer2 token
=> SELECT * FROM hcatalog_tables;
```

Each object in the HadoopImpersonationConfig collection specifies one connection to one Hadoop cluster. You can add as many connections as you like, including to more than one Hadoop cluster. The following example shows delegation tokens for two different Hadoop clusters. Vertica uses the correct token for each cluster when connecting.

```
$ vsql -U dbuser1

=> ALTER SESSION SET
   HadoopImpersonationConfig ='[
    {"nameservice":"productionNS","token":"JAAGZGJldGwxBmRiZXRsMQCKAWDXJgB9igFg-
zKEfY4gao4BmhSJYtXiWqrhBHbbUn4VScNg58HWQxJXRUJIREZTIGRlbGVnYXRpb24RMTAuMjAuMTAwLjU0OjgwMjA"},
    {"nameservice":"testNS", "token":"HQAHcmVsZWFzZQdyZWxlYXNlAIoBYVJKrYSKAWF2VzGEjgmzj_IUCIrI9b8Dqu6awFTHk5nC-
fHB8xsSV0VCSERGUyBkZWxlZ2F0aW9uETEwLjIwLjQyLjEwOTo4MDIw"}]';

=> COPY clicks FROM 'webhdfs://productionNS/data/clickstream.dat';
=> COPY testclicks FROM 'webhdfs://testNS/data/clickstream.dat';
```

Token expiration

Vertica uses Hadoop tokens when using Kerberos tickets (Using Kerberos with Vertica) or doAs (User impersonation (doAs)). Vertica attempts to automatically refresh Hadoop tokens before they expire, but you can also set a minimum refresh frequency if you prefer. Use the HadoopFSTokenRefreshFrequency configuration parameter to specify the frequency in seconds:

```
=> ALTER DATABASE exampledb SET HadoopFSTokenRefreshFrequency = '86400';
```

If the current age of the token is greater than the value specified in this parameter, Vertica refreshes the token before accessing data stored in HDFS.

Vertica does not refresh delegation tokens (Bring your own delegation token).

Using HDFS storage locations

Vertica stores data in its native format, ROS, in storage locations. You can place storage locations on the local Linux file system or in HDFS. If you place storage locations on HDFS, you must perform additional configuration in HDFS to be able to manage them. These are in addition to the requirements in Vertica for managing storage locations and backup/restore .

If you use HDFS storage locations, the HDFS data must be available when you start Vertica. Your HDFS cluster must be operational, and the ROS files must be present. If you moved data files, or they are corrupted, or your HDFS cluster is not responsive, Vertica cannot start.

In this section

- Hadoop configuration for backup and restore
- Removing HDFS storage locations

Hadoop configuration for backup and restore

If your Vertica cluster uses storage locations on HDFS, and you want to be able to back up and restore those storage locations using vbr , you must enable snapshotting in HDFS.

The Vertica backup script uses HDFS's snapshotting feature to create a backup of HDFS storage locations. A directory must allow snapshotting before HDFS can take a snapshot. Only a Hadoop superuser can enable snapshotting on a directory. Vertica can enable snapshotting automatically if the database administrator is also a Hadoop superuser.

If HDFS is unsecured, the following instructions apply to the database administrator account, usually dbadmin. If HDFS uses Kerberos security, the following instructions apply to the principal stored in the Vertica keytab file, usually vertica. The instructions below use the term "database account" to refer to this user.

We recommend that you make the database administrator or principal a Hadoop superuser. If you are not able to do so, you must enable snapshotting on the directory before configuring it for use by Vertica.

The steps you need to take to make the Vertica database administrator account a superuser depend on the distribution of Hadoop you are using. Consult your Hadoop distribution's documentation for details.

Manually enabling snapshotting for a directory

If you cannot grant superuser status to the database account, you can instead enable snapshotting of each directory manually. Use the following command:

```
$ hdfs dfsadmin -allowSnapshot path
```

Issue this command for each directory on each node. Remember to do this each time you add a new node to your HDFS cluster.

Nested snapshottable directories are not allowed, so you cannot enable snapshotting for a parent directory to automatically enable it for child directories. You must enable it for each individual directory.

Additional requirements for Kerberos

If HDFS uses Kerberos, then in addition to granting the keytab principal access, you must give Vertica access to certain Hadoop configuration files. See Configuring Kerberos .

Testing the database account's ability to make HDFS directories snapshottable

After making the database account a Hadoop superuser, verify that the account can set directories snapshottable:

1. Log into the Hadoop cluster as the database account (dbadmin by default).
2. Determine a location in HDFS where the database administrator can create a directory. The /tmp directory is usually available. Create a test HDFS directory using the command:
   ```
   $ hdfs dfs -mkdir /path/testdir
   ```
3. Make the test directory snapshottable using the command:

```
$ hdfs dfsadmin -allowSnapshot /path/testdir
```

The following example demonstrates creating an HDFS directory and making it snapshottable:

```
$ hdfs dfs -mkdir /tmp/snaptest
$ hdfs dfsadmin -allowSnapshot /tmp/snaptest
Allowing snaphot on /tmp/snaptest succeeded
```

Removing HDFS storage locations

The steps to remove an HDFS storage location are similar to standard storage locations:

1. Remove any existing data from the HDFS storage location by using SET_OBJECT_STORAGE_POLICY to change each object's storage location. Alternatively, you can use CLEAR_OBJECT_STORAGE_POLICY . Because the Tuple Mover runs infrequently, set the *enforce-storage-move* parameter to true to make the change immediately.
2. Retire the location on each host that has the storage location defined using RETIRE_LOCATION . Set *enforce-storage-move* to true .
3. Drop the location on each node using DROP_LOCATION .
4. Optionally remove the snapshots and files from the HDFS directory for the storage location.
5. Perform a full database backup.

For more information about changing storage policies, changing usage, retiring locations, and dropping locations, see Managing storage locations .

Important
If you have backed up the data in the HDFS storage location you are removing, you must perform a full database backup after you remove the location. If you do not and restore the database to a backup made before you removed the location, the location's data is restored.

Removing storage location files from HDFS

Dropping an HDFS storage location does not automatically clean the HDFS directory that stored the location's files. Any snapshots of the data files created when backing up the location are also not deleted. These files consume disk space on HDFS and also prevent the directory from being reused as an HDFS storage location. Vertica cannot create a storage location in a directory that contains existing files or subdirectories.

You must log into the Hadoop cluster to delete the files from HDFS. An alternative is to use some other HDFS file management tool.

Removing backup snapshots

HDFS returns an error if you attempt to remove a directory that has snapshots:

```
$ hdfs dfs -rm -r -f -skipTrash /user/dbadmin/v_vmart_node0001
rm: The directory /user/dbadmin/v_vmart_node0001 cannot be deleted since
/user/dbadmin/v_vmart_node0001 is snapshottable and already has snapshots
```

The Vertica backup script creates snapshots of HDFS storage locations as part of the backup process. If you made backups of your HDFS storage location, you must delete the snapshots before removing the directories.

HDFS stores snapshots in a subdirectory named .snapshot . You can list the snapshots in the directory using the standard HDFS ls command:

```
$ hdfs dfs -ls /user/dbadmin/v_vmart_node0001/.snapshot
Found 1 items
drwxrwx---   - dbadmin supergroup          0 2014-09-02 10:13 /user/dbadmin/v_vmart_node0001/.snapshot/s20140902-101358.629
```

To remove snapshots, use the command:

```
$ hdfs dfs -removeSnapshot directory snapshotname
```

The following example demonstrates the command to delete the snapshot shown in the previous example:

```
$ hdfs dfs -deleteSnapshot /user/dbadmin/v_vmart_node0001 s20140902-101358.629
```

You must delete each snapshot from the directory for each host in the cluster. After you have deleted the snapshots, you can delete the directories in the storage location.

Important
Each snapshot's name is based on a timestamp down to the millisecond. Nodes independently create their own snapshots. They do not synchronize snapshot creation, so their snapshot names differ. You must list each node's snapshot directory to learn the names of the snapshots it contains.

See Apache's HDFS Snapshot documentation for more information about managing and removing snapshots.

Removing the storage location directories

You can remove the directories that held the storage location's data by either of the following methods:

- Use an HDFS file manager to delete directories. See your Hadoop distribution's documentation to determine if it provides a file manager.
- Log into the Hadoop Name Node using the database administrator's account and use HDFS's rmr command to delete the directories. See Apache's File System Shell Guide for more information.

The following example uses the HDFS rmr command from the Linux command line to delete the directories left behind in the HDFS storage location directory /user/dbamin . It uses the -skipTrash flag to force the immediate deletion of the files:

```
$ hdfsp dfs -ls /user/dbadmin
Found 3 items
drwxrwx---   - dbadmin supergroup          0 2014-08-29 15:11 /user/dbadmin/v_vmart_node0001
drwxrwx---   - dbadmin supergroup          0 2014-08-29 15:11 /user/dbadmin/v_vmart_node0002
drwxrwx---   - dbadmin supergroup          0 2014-08-29 15:11 /user/dbadmin/v_vmart_node0003

$ hdfs dfs -rmr -skipTrash /user/dbadmin/*
Deleted /user/dbadmin/v_vmart_node0001
Deleted /user/dbadmin/v_vmart_node0002
Deleted /user/dbadmin/v_vmart_node0003
```

Using the HCatalog Connector

The Vertica HCatalog Connector lets you access data stored in Apache's Hive data warehouse software the same way you access it within a native Vertica table.

If your files are in the Optimized Columnar Row (ORC) or Parquet format and do not use complex types, the HCatalog Connector creates an external table and uses the ORC or Parquet reader instead of using the Java SerDe. See ORC and PARQUET for more information about these readers.

The HCatalog Connector performs predicate pushdown to improve query performance. Instead of reading all data across the network to evaluate a query, the HCatalog Connector moves the evaluation of predicates closer to the data. Predicate pushdown applies to Hive partition pruning, ORC stripe pruning, and Parquet row-group pruning. The HCatalog Connector supports predicate pushdown for the following predicates: >, >=, =, <>, <=, <.

In this section

- Overview
- How the HCatalog Connector works
- HCatalog Connector requirements
- Installing the Java runtime on your Vertica cluster
- Configuring Vertica for HCatalog
- Configuring security
- Defining a schema using the HCatalog Connector
- Querying Hive tables using HCatalog Connector
- Viewing Hive schema and table metadata
- Synchronizing an HCatalog schema or table with a local schema or table
- Data type conversions from Hive to Vertica
- Using nonstandard SerDes
- Troubleshooting HCatalog Connector problems

Overview

There are several Hadoop components that you need to understand to use the HCatalog connector:

- Apache Hive lets you query data stored in a Hadoop Distributed File System (HDFS) the same way you query data stored in a relational database. Behind the scenes, Hive uses a set of serializer and deserializer (SerDe) classes to extract data from files stored in HDFS and break it into columns and rows. Each SerDe handles data files in a specific format. For example, one SerDe extracts data from comma-separated data files while another interprets data stored in JSON format.
- Apache HCatalog is a component of the Hadoop ecosystem that makes Hive's metadata available to other Hadoop components (such as Pig).
- HiveServer2 makes HCatalog and Hive data available via JDBC. Through it, a client can make requests to retrieve data stored in Hive, as well as information about the Hive schema. HiveServer2 can use authorization services (Sentry or Ranger). HiverServer2 can use Hive LLAP (Live Long And Process).

The Vertica HCatalog Connector lets you transparently access data that is available through HiveServer2. You use the connector to define a schema in Vertica that corresponds to a Hive database or schema. When you query data within this schema, the HCatalog Connector transparently extracts and formats the data from Hadoop into tabular data. Vertica supports authorization services and Hive LLAP.

> **Note**
> You can use the WebHCat service instead of HiveServer2, but performance is usually better with HiveServer2. Support for WebHCat is deprecated. To use WebHCat, set the HCatalogConnectorUseHiveServer2 configuration parameter to 0. See Hadoop parameters . WebHCat does not support authorization services.

HCatalog connection features

The HCatalog Connector lets you query data stored in Hive using the Vertica native SQL syntax. Some of its main features are:

- The HCatalog Connector always reflects the current state of data stored in Hive.
- The HCatalog Connector uses the parallel nature of both Vertica and Hadoop to process Hive data. The result is that querying data through the HCatalog Connector is often faster than querying the data directly through Hive.
- Because Vertica performs the extraction and parsing of data, the HCatalog Connector does not significantly increase the load on your Hadoop cluster.
- The data you query through the HCatalog Connector can be used as if it were native Vertica data. For example, you can execute a query that joins data from a table in an HCatalog schema with a native table.

HCatalog Connector considerations

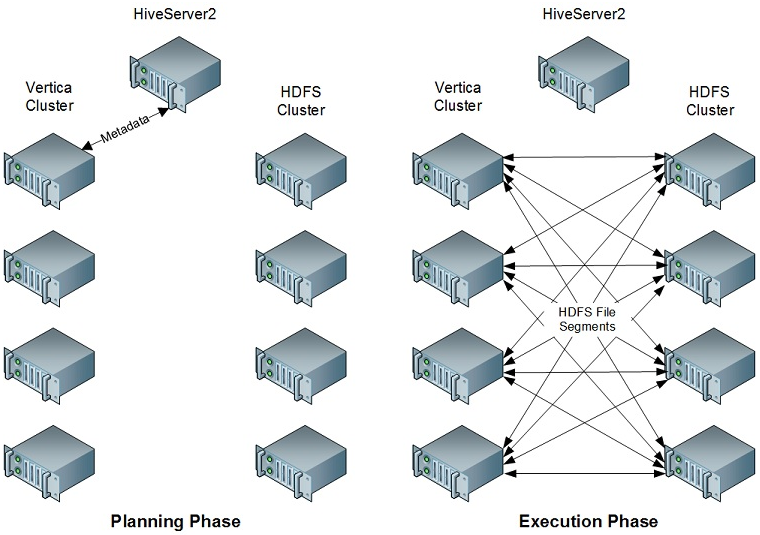
There are a few things to keep in mind when using the HCatalog Connector:

- Hive's data is stored in flat files in a distributed file system, requiring it to be read and deserialized each time it is queried. This deserialization causes Hive performance to be much slower than that of Vertica. The HCatalog Connector has to perform the same process as Hive to read the data. Therefore, querying data stored in Hive using the HCatalog Connector is much slower than querying a native Vertica table. If you need to perform extensive analysis on data stored in Hive, you should consider loading it into Vertica. Vertica optimization often makes querying data through the HCatalog Connector faster than directly querying it through Hive.
- If Hive uses Kerberos security, the HCatalog Connector uses the querying user's credentials in queries by default. If Hive uses Sentry or Ranger to enforce security, then you must either disable this behavior in Vertica by setting EnableHCatImpersonation to 0 or grant users access to the underlying data in HDFS. (Sentry supports ACL synchronization to automatically grant access.) Alternatively, you can specify delegation tokens for data and metadata access. See Configuring security .
- Hive supports complex data types such as lists, maps, and structs that Vertica does not support. Columns containing these data types are converted to a JSON representation of the data type and stored as a VARCHAR. See Data type conversions from Hive to Vertica .

> **Note**
> The HCatalog Connector is read-only. It cannot insert data into Hive.

How the HCatalog Connector works

When planning a query that accesses data from a Hive table, the Vertica HCatalog Connector on the initiator node contacts HiveServer2 (or WebHCat) in your Hadoop cluster to determine if the table exists. If it does, the connector retrieves the table's metadata from the metastore database so the query planning can continue. When the query executes, all nodes in the Vertica cluster directly retrieve the data necessary for completing the query from HDFS. They then use the Hive SerDe classes to extract the data so the query can execute. When accessing data in ORC or Parquet format, the HCatalog Connector uses Vertica's internal readers for these formats instead of the Hive SerDe classes.



This approach takes advantage of the parallel nature of both Vertica and Hadoop. In addition, by performing the retrieval and extraction of data directly, the HCatalog Connector reduces the impact of the query on the Hadoop cluster.

For files in the Optimized Columnar Row (ORC) or Parquet format that do not use complex types, the HCatalog Connector creates an external table and uses the ORC or Parquet reader instead of using the Java SerDe. You can direct these readers to access custom Hive partition locations if Hive used them when writing the data. By default these extra checks are turned off to improve performance.

HCatalog Connector requirements

Before you can use the HCatalog Connector, both your Vertica and Hadoop installations must meet the following requirements.

Vertica requirements

All of the nodes in your cluster must have a Java Virtual Machine (JVM) installed. You must use the same Java version that the Hadoop cluster uses. See Installing the Java Runtime on Your Vertica Cluster .

You must also add certain libraries distributed with Hadoop and Hive to your Vertica installation directory. See Configuring Vertica for HCatalog .

Hadoop requirements

Your Hadoop cluster must meet several requirements to operate correctly with the Vertica Connector for HCatalog:

- It must have Hive, HiveServer2, and HCatalog installed and running. See Apache's HCatalog page for more information.
- The HiveServer2 server and all of the HDFS nodes that store HCatalog data must be directly accessible from all of the hosts in your Vertica database. Verify that any firewall separating the Hadoop cluster and the Vertica cluster will pass HiveServer2, metastore database, and HDFS traffic.
- The data that you want to query must be in an internal or external Hive table.
- If a table you want to query uses a non-standard SerDe, you must install the SerDe's classes on your Vertica cluster before you can query the data. See Using nonstandard SerDes .

Installing the Java runtime on your Vertica cluster

The HCatalog Connector requires a 64-bit Java Virtual Machine (JVM). The JVM must support Java 6 or later, and must be the same version as the one installed on your Hadoop nodes.

> **Note**
> If your Vertica cluster is configured to execute User Defined Extensions (UDxs) written in Java, it already has a correctly-configured JVM installed. See Developing user-defined extensions (UDxs) for more information.

Installing Java on your Vertica cluster is a two-step process:

1. Install a Java runtime on all of the hosts in your cluster.
2. Set the JavaBinaryForUDx configuration parameter to tell Vertica the location of the Java executable.

Installing a Java runtime

For Java-based features, Vertica requires a 64-bit Java 6 (Java version 1.6) or later Java runtime. Vertica supports runtimes from either Oracle or OpenJDK . You can choose to install either the Java Runtime Environment (JRE) or Java Development Kit (JDK), since the JDK also includes the JRE.

Many Linux distributions include a package for the OpenJDK runtime. See your Linux distribution's documentation for information about installing and configuring OpenJDK.

To install the Oracle Java runtime, see the Java Standard Edition (SE) Download Page . You usually run the installation package as root in order to install it. See the download page for instructions.

Once you have installed a JVM on each host, ensure that the java command is in the search path and calls the correct JVM by running the command:

```
$ java -version
```

This command should print something similar to:

```
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)
```

Note
Any previously installed Java VM on your hosts may interfere with a newly installed Java runtime. See your Linux distribution's documentation for instructions on configuring which JVM is the default. Unless absolutely required, you should uninstall any incompatible version of Java before installing the Java 6 or Java 7 runtime.

Setting the JavaBinaryForUDx configuration parameter

The JavaBinaryForUDx configuration parameter tells Vertica where to look for the JRE to execute Java UDxs. After you have installed the JRE on all of the nodes in your cluster, set this parameter to the absolute path of the Java executable. You can use the symbolic link that some Java installers create (for example /usr/bin/java). If the Java executable is in your shell search path, you can get the path of the Java executable by running the following command from the Linux command line shell:

```
$ which java
/usr/bin/java
```

If the java command is not in the shell search path, use the path to the Java executable in the directory where you installed the JRE. Suppose you installed the JRE in /usr/java/default (which is where the installation package supplied by Oracle installs the Java 1.6 JRE). In this case the Java executable is /usr/java/default/bin/java .

You set the configuration parameter by executing the following statement as a database superuser :

```
=> ALTER DATABASE DEFAULT SET PARAMETER JavaBinaryForUDx = '/usr/bin/java';
```

See ALTER DATABASE for more information on setting configuration parameters.

To view the current setting of the configuration parameter, query the CONFIGURATION_PARAMETERS system table:

```
=> \x
Expanded display is on.
=> SELECT * FROM CONFIGURATION_PARAMETERS WHERE parameter_name = 'JavaBinaryForUDx';
-[ RECORD 1 ]----------------+-----------------------------------------------------------
node_name                    | ALL
parameter_name               | JavaBinaryForUDx
current_value                | /usr/bin/java
default_value                |
change_under_support_guidance | f
change_requires_restart      | f
description                  | Path to the java binary for executing UDx written in Java
```

Once you have set the configuration parameter, Vertica can find the Java executable on each node in your cluster.

Note
Since the location of the Java executable is set by a single configuration parameter for the entire cluster, you must ensure that the Java executable is installed in the same path on all of the hosts in the cluster.

Configuring Vertica for HCatalog

Before you can use the HCatalog Connector, you must add certain Hadoop and Hive libraries to your Vertica installation. You must also copy the Hadoop configuration files that specify various connection properties. Vertica uses the values in those configuration files to make its own connections to Hadoop.

You need only make these changes on one node in your cluster. After you do this you can install the HCatalog connector.

Copy Hadoop libraries and configuration files

Vertica provides a tool, hcatUtil, to collect the required files from Hadoop. This tool copies selected libraries and XML configuration files from your Hadoop cluster to your Vertica cluster. This tool might also need access to additional libraries:

- If you plan to use Hive to query files that use Snappy compression, you need access to the Snappy native libraries, libhadoop*.so and libsnappy*.so.
- If you plan to use Hive to query files that use LZO compression, you need access to the hadoop-lzo-*.jar and libgplcompression.so* libraries. In core-site.xml you must also edit the io.compression.codecs property to include com.hadoop.compression.lzo.LzopCodec .
- If you plan to use a JSON SerDe with a Hive table, you need access to its library. This is the same library that you used to configure Hive; for example:

```
hive> add jar /home/release/json-serde-1.3-jar-with-dependencies.jar;



hive> create external table nationjson (id int,name string,rank int,text string)
    ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
    LOCATION '/user/release/vt/nationjson';
```

- If you are using any other libraries that are not standard across all supported Hadoop versions, you need access to those libraries.

If any of these cases applies to you, do one of the following:

- Include the path(s) in the path you specify as the value of --hcatLibPath, or
- Copy the file(s) to a directory already on that path.

If Vertica is not co-located on a Hadoop node, you should do the following:

1. Copy /opt/vertica/packages/hcat/tools/hcatUtil to a Hadoop node and run it there, specifying a temporary output directory. Your Hadoop, HIVE, and HCatalog lib paths might be different. In newer versions of Hadoop the HCatalog directory is usually a subdirectory under the HIVE directory, and Cloudera creates a new directory for each revision of the configuration files. Use the values from your environment in the following command:

   ```
   hcatUtil --copyJars
       --hadoopHiveHome="$HADOOP_HOME/lib;$HIVE_HOME/lib;/hcatalog/dist/share"
        --hadoopHiveConfPath="$HADOOP_CONF_DIR;$HIVE_CONF_DIR;$WEBHCAT_CONF_DIR"
       --hcatLibPath="/tmp/hadoop-files"
   ```

   If you are using Hive LLAP, specify the hive2 directories.
2. Verify that all necessary files were copied:

   ```
   hcatUtil --verifyJars --hcatLibPath=/tmp/hadoop-files
   ```
3. Copy that output directory (/tmp/hadoop-files, in this example) to /opt/vertica/packages/hcat/lib on the Vertica node you will connect to when installing the HCatalog connector. If you are updating a Vertica cluster to use a new Hadoop cluster (or a new version of Hadoop), first remove all JAR files in /opt/vertica/packages/hcat/lib except vertica-hcatalogudl.jar.
4. Verify that all necessary files were copied:

   ```
   hcatUtil --verifyJars --hcatLibPath=/opt/vertica/packages/hcat
   ```

If Vertica is co-located on some or all Hadoop nodes, you can do this in one step on a shared node. Your Hadoop, HIVE, and HCatalog lib paths might be different; use the values from your environment in the following command:

```
hcatUtil --copyJars
    --hadoopHiveHome="$HADOOP_HOME/lib;$HIVE_HOME/lib;/hcatalog/dist/share"
    --hadoopHiveConfPath="$HADOOP_CONF_DIR;$HIVE_CONF_DIR;$WEBHCAT_CONF_DIR"
    --hcatLibPath="/opt/vertica/packages/hcat/lib"
```

The hcatUtil script has the following arguments:

-c, --copyJars	Copy the required JAR files from hadoopHiveHome and configuration files from hadoopHiveConfPath.
-v, --verifyJars	Verify that the required files are present in hcatLibPath. Check the output of hcatUtil for error and warning messages.
--hadoopHiveHome= *"value1;value2;..."*	Paths to the Hadoop, Hive, and HCatalog home directories. Separate directories by semicolons (;). Enclose paths in double quotes. **Note** Always place $HADOOP_HOME on the path before $HIVE_HOME. In some Hadoop distributions, these two directories contain different versions of the same library.
--hadoopHiveConfPath= *"value1;value2;..."*	Paths of the following configuration files: • hive-site.xml • core-site.xml • yarn-site.xml • webhcat-site.xml (optional with the default configuration; required if you use WebHCat instead of HiveServer2) • hdfs-site.xml Separate directories by semicolons (;). Enclose paths in double quotes. In previous releases of Vertica this parameter was optional under some conditions. It is now required.
--hcatLibPath= *"value"*	Output path for the libraries and configuration files. On a Vertica node, use /opt/vertica/packages/hcat/lib. If you have previously run hcatUtil with a different version of Hadoop, first remove the old JAR files from the output directory (all except vertica-hcatalogudl.jar).

After you have copied the files and verified them, install the HCatalog connector.

Install the HCatalog Connector

On the same node where you copied the files from hcatUtil, install the HCatalog connector by running the install.sql script. This script resides in the ddl/ folder under your HCatalog connector installation path. This script creates the library and VHCatSource and VHCatParser.

Note
The data that was copied using hcatUtil is now stored in the database. If you change any of those values in Hadoop, you need to rerun hcatUtil and install.sql. The following statement returns the names of the libraries and configuration files currently being used:

```
=> SELECT dependencies FROM user_libraries WHERE lib_name='VHCatalogLib';
```

Now you can create HCatalog schema parameters, which point to your existing Hadoop services, as described in Defining a schema using the HCatalog Connector .

Upgrading to a new version of Vertica

After upgrading to a new version of Vertica, perform the following steps:

1. Uninstall the HCatalog Connector using the uninstall.sql script. This script resides in the ddl/ folder under your HCatalog connector installation path.
2. Delete the contents of the hcatLibPath directory except for vertica-hcatalogudl.jar .
3. Rerun hcatUtil.
4. Reinstall the HCatalog Connector using the install.sql script.

For more information about upgrading Vertica, see Upgrade Vertica .

Additional options for Hadoop columnar file formats

When reading Hadoop columnar file formats (ORC or Parquet), the HCatalog Connector attempts to use the built-in readers. When doing so, it uses WebHDFS by default. You can use the deprecated LibHDFS++ library by using the hdfs URI scheme and using ALTER DATABASE to set HDFSUseWebHDFS to 0. This setting applies to all HDFS access, not just the HCatalog Connector.

In either case, you must perform the configuration described in Configuring HDFS access .

Configuring security

You can use any of the security options described in Accessing kerberized HDFS data to access Hive data. This topic describes additional steps needed specifically for using the HCatalog Connector.

If you use Kerberos from Vertica, the HCatalog Connector can use an authorization service (Sentry or Ranger). If you use delegation tokens, you must manage authorization yourself.

Kerberos

You can use Kerberos from Vertica as described in Using Kerberos with Vertica .

How you configure the HCatalog Connector depends on how Hive manages authorization.

- If Hive uses Sentry to manage authorization, and if Sentry uses ACL synchronization, then the HCatalog Connector must access HDFS as the current user. Verify that the EnableHCatImpersonation configuration parameter is set to 1 (the default). ACL synchronization automatically provides authorized users with read access to the underlying HDFS files.
- If Hive uses Sentry without ACL synchronization, then the HCatalog Connector must access HDFS data as the Vertica principal. (The user still authenticates and accesses metadata normally.) Set the EnableHCatImpersonation configuration parameter to 0. The Vertica principal must have read access to the underlying HDFS files.
- If Hive uses Ranger to manage authorization, and the Vertica users have read access to the underlying HDFS files, then you can use user impersonation. Verify that the EnableHCatImpersonation configuration parameter is set to 1 (the default). You can, instead, disable user impersonation and give the Vertica principal read access to the HDFS files.
- If Hive uses either Sentry or Ranger, the HCatalog Connector must use HiveServer2 (the default). WebHCat does not support authorization services.
- If Hive does not use an authorization service, or if you are connecting to Hive using WebHCat instead of HiveServer2, then the HCatalog Connector accesses Hive as the current user. Verify that EnableHCatImpersonation is set to 1. All users must have read access to the underlying HDFS files.

In addition, in your Hadoop configuration files (core-site.xml in most distributions), make sure that you enable all Hadoop components to impersonate the Vertica user. The easiest way to do so is to set the proxyuser property using wildcards for all users on all hosts and in all groups. Consult your Hadoop documentation for instructions. Make sure you set this property before running hcatUtil (see Configuring Vertica for HCatalog).

Delegation tokens

You can use delegation tokens for a session as described in Bring your own delegation token . When using the HCatalog Connector you specify two delegation tokens, one for the data and one for the metadata. The metadata token is tied to a Hive schema. See HadoopImpersonationConfig format for information about how to specify these two delegation tokens.

Verifying security configuration

To verify that the HCatalog Connector can access Hive data, use the HCATALOGCONNECTOR_CONFIG_CHECK function.

For more information about testing your configuration, see Verifying HDFS configuration .

Defining a schema using the HCatalog Connector

After you set up the HCatalog Connector, you can use it to define a schema in your Vertica database to access the tables in a Hive database. You define the schema using the CREATE HCATALOG SCHEMA statement.

When creating the schema, you must supply the name of the schema to define in Vertica. Other parameters are optional. If you do not supply a value, Vertica uses default values. Vertica reads some default values from the HDFS configuration files; see Configuration Parameters .

To create the schema, you must have read access to all Hive data. Verify that the user creating the schema has been granted access, either directly or through an authorization service such as Sentry or Ranger. The dbadmin user has no automatic special privileges.

After you create the schema, you can change many parameters using the ALTER HCATALOG SCHEMA statement.

After you define the schema, you can query the data in the Hive data warehouse in the same way you query a native Vertica table. The following example demonstrates creating an HCatalog schema and then querying several system tables to examine the contents of the new schema. See Viewing Hive schema and table metadata for more information about these tables.

```
=> CREATE HCATALOG SCHEMA hcat WITH HOSTNAME='hcathost' PORT=9083
   HCATALOG_SCHEMA='default' HIVESERVER2_HOSTNAME='hs.example.com'
   SSL_CONFIG='/etc/hadoop/conf/ssl-client.xml' HCATALOG_USER='admin';
CREATE SCHEMA
=> \x
Expanded display is on.

=> SELECT * FROM v_catalog.hcatalog_schemata;
-[ RECORD 1 ]----------------+-----------------------------------------
schema_id                    | 45035996273748224
schema_name                  | hcat
schema_owner_id              | 45035996273704962
schema_owner                 | admin
create_time                  | 2017-12-05 14:43:03.353404-05
hostname                     | hcathost
port                         | -1
hiveserver2_hostname         | hs.example.com
webservice_hostname          |
webservice_port              | 50111
webhdfs_address              | hs.example.com:50070
hcatalog_schema_name         | default
ssl_config                   | /etc/hadoop/conf/ssl-client.xml
hcatalog_user_name           | admin
hcatalog_connection_timeout  | -1
hcatalog_slow_transfer_limit | -1
hcatalog_slow_transfer_time  | -1
custom_partitions            | f

=> SELECT * FROM v_catalog.hcatalog_table_list;
-[ RECORD 1 ]------+------------------
table_schema_id    | 45035996273748224
table_schema       | hcat
hcatalog_schema    | default
table_name         | nation
hcatalog_user_name | admin
-[ RECORD 2 ]------+------------------
table_schema_id    | 45035996273748224
table_schema       | hcat
hcatalog_schema    | default
table_name         | raw
hcatalog_user_name | admin
-[ RECORD 3 ]------+------------------
table_schema_id    | 45035996273748224
table_schema       | hcat
hcatalog_schema    | default
table_name         | raw_rcfile
hcatalog_user_name | admin
-[ RECORD 4 ]------+------------------
table_schema_id    | 45035996273748224
table_schema       | hcat
hcatalog_schema    | default
table_name         | raw_sequence
hcatalog_user_name | admin
```

Configuration parameters

The HCatalog Connector uses the following values from the Hadoop configuration files if you do not override them when creating the schema.

File	Properties
hive-site.xml	hive.server2.thrift.bind.host (used for HIVESERVER2_HOSTNAME)

	hive.server2.thrift.port
	hive.server2.transport.mode
	hive.server2.authentication
	hive.server2.authentication.kerberos.principal
	hive.server2.support.dynamic.service.discovery
	hive.zookeeper.quorum (used as HIVESERVER2_HOSTNAME if dynamic service discovery is enabled)
	hive.zookeeper.client.port
	hive.server2.zookeeper.namespace
	hive.metastore.uris (used for HOSTNAME and PORT)
ssl-client.xml	ssl.client.truststore.location
	ssl.client.truststore.password

Using partitioned data

Hive supports partitioning data, as in the following example:

```
hive> create table users (name varchar(64), address string, city varchar(64))
   partitioned by (state varchar(64)) stored as orc;
```

Vertica takes advantage of partitioning information for formats that provide it in metadata (ORC and Parquet). Queries can skip irrelevant partitions entirely (partition pruning), and Vertica does not need to materialize partition columns. For more about how to use partitions in Vertica, see Partitioned file paths . For more information about how to create compatible partitions in Hive, see Hive primer for Vertica integration .

By default Hive stores partition information under the path for the table definition, which might be local to Hive. However, a Hive user can choose to store partition information elsewhere, such as in a shared location like S3, as in the following example:

```
hive> alter table users add partition (state='MA')
   location 's3a://DataLake/partitions/users/state=MA';
```

During query execution, therefore, Vertica must query Hive for the location of a table's partition information.

Because the additional Hive queries can be expensive, Vertica defaults to looking for partition information only in Hive's default location. If your Hive table specified a custom location, use the CUSTOM_PARTITIONS parameter:

```
=> CREATE HCATALOG SCHEMA hcat WITH HOSTNAME='hcathost' PORT=9083
   HCATALOG_SCHEMA='default' HIVESERVER2_HOSTNAME='hs.example.com'
   SSL_CONFIG='/etc/hadoop/conf/ssl-client.xml' HCATALOG_USER='admin'
   CUSTOM_PARTITIONS='yes';
```

Vertica can access partition locations in the S3 and S3a schemes, and does not support the S3n scheme.

The HIVE_CUSTOM_PARTITIONS_ACCESSED system table records all custom partition locations that have been used in queries.

Using the HCatalog Connector with WebHCat

By default the HCatalog Connector uses HiveServer2 to access Hive data. If you are instead using WebHCat, set the HCatalogConnectorUseHiveServer2 configuration parameter to 0 before creating the schema as in the following example.

```
=> ALTER DATABASE DEFAULT SET PARAMETER HCatalogConnectorUseHiveServer2 = 0;
=> CREATE HCATALOG SCHEMA hcat WITH WEBSERVICE_HOSTNAME='webhcat.example.com';
```

If you have previously used WebHCat, you can switch to using HiveServer2 by setting the configuration parameter to 1 and using ALTER HCATALOG SCHEMA to set HIVESERVER2_HOSTNAME. You do not need to remove the WebHCat values; the HCatalog Connector uses the value of HCatalogConnectorUseHiveServer2 to determine which parameters to use.

Querying Hive tables using HCatalog Connector

Once you have defined the HCatalog schema, you can query data from the Hive database by using the schema name in your query.

```
=> SELECT * from hcat.messages limit 10;
 messageid |   userid   |        time         |             message
-----------+------------+---------------------+---------------------------------
         1 | nPfQ1ayhi  | 2013-10-29 00:10:43 | hymenaeos cursus lorem Suspendis
         2 | N7svORIoZ  | 2013-10-29 00:21:27 | Fusce ad sem vehicula morbi
         3 | 4VvzN3d    | 2013-10-29 00:32:11 | porta Vivamus condimentum
         4 | heojkmTmc  | 2013-10-29 00:42:55 | lectus quis imperdiet
         5 | coROws3OF  | 2013-10-29 00:53:39 | sit eleifend tempus a aliquam mauri
         6 | oDRP1i     | 2013-10-29 01:04:23 | risus facilisis sollicitudin sceler
         7 | AU7a9Kp    | 2013-10-29 01:15:07 | turpis vehicula tortor
         8 | ZJWg185DkZ | 2013-10-29 01:25:51 | sapien adipiscing eget Aliquam tor
         9 | E7ipAsYC3  | 2013-10-29 01:36:35 | varius Cum iaculis metus
        10 | kStCv      | 2013-10-29 01:47:19 | aliquam libero nascetur Cum mal
(10 rows)
```

Since the tables you access through the HCatalog Connector act like Vertica tables, you can perform operations that use both Hive data and native Vertica data, such as a join:

```
=> SELECT u.FirstName, u.LastName, d.time, d.Message from UserData u
-> JOIN hcat.messages d ON u.UserID = d.UserID LIMIT 10;
FirstName | LastName |        time         |                Message
----------+----------+---------------------+---------------------------------
Whitney   | Kerr     | 2013-10-29 00:10:43 | hymenaeos cursus lorem Suspendis
Troy      | Oneal    | 2013-10-29 00:32:11 | porta Vivamus condimentum
Renee     | Coleman  | 2013-10-29 00:42:55 | lectus quis imperdiet
Fay       | Moss     | 2013-10-29 00:53:39 | sit eleifend tempus a aliquam mauri
Dominique | Cabrera  | 2013-10-29 01:15:07 | turpis vehicula tortor
Mohammad  | Eaton    | 2013-10-29 00:21:27 | Fusce ad sem vehicula morbi
Cade      | Barr     | 2013-10-29 01:25:51 | sapien adipiscing eget Aliquam tor
Oprah     | Mcmillan | 2013-10-29 01:36:35 | varius Cum iaculis metus
Astra     | Sherman  | 2013-10-29 01:58:03 | dignissim odio Pellentesque primis
Chelsea   | Malone   | 2013-10-29 02:08:47 | pede tempor dignissim Sed luctus
(10 rows)
```

Viewing Hive schema and table metadata

When using Hive, you access metadata about schemas and tables by executing statements written in HiveQL (Hive's version of SQL) such as SHOW TABLES . When using the HCatalog Connector, you can get metadata about the tables in the Hive database through several Vertica system tables.

There are four system tables that contain metadata about the tables accessible through the HCatalog Connector:

- HCATALOG_SCHEMATA lists all of the schemas that have been defined using the HCatalog Connector.
- HCATALOG_TABLE_LIST contains an overview of all of the tables available from all schemas defined using the HCatalog Connector. This table only shows the tables that the user querying the table can access. The information in this table is retrieved using a single call to HiveServer2 for each schema defined using the HCatalog Connector, which means there is a little overhead when querying this table.
- HCATALOG_TABLES contains more in-depth information than HCATALOG_TABLE_LIST.
- HCATALOG_COLUMNS lists metadata about all of the columns in all of the tables available through the HCatalog Connector. As for HCATALOG_TABLES, querying this table results in one call to HiveServer2 per table, and therefore can take a while to complete.

The following example demonstrates querying the system tables containing metadata for the tables available through the HCatalog Connector.

```
=> CREATE HCATALOG SCHEMA hcat WITH hostname='hcathost'
-> HCATALOG_SCHEMA='default' HCATALOG_DB='default' HCATALOG_USER='hcatuser';
CREATE SCHEMA
=> SELECT * FROM HCATALOG_SCHEMATA;
-[ RECORD 1 ]--------+---------------------------
schema_id            | 45035996273864536
schema_name          | hcat
schema_owner_id      | 45035996273704962
schema_owner         | dbadmin
```

```
create_time          | 2013-11-05 10:19:54.70965-05
hostname             | hcathost
port                 | 9083
webservice_hostname  | hcathost
webservice_port      | 50111
hcatalog_schema_name | default
hcatalog_user_name   | hcatuser
metastore_db_name    | hivemetastoredb

=> SELECT * FROM HCATALOG_TABLE_LIST;
-[ RECORD 1 ]------+------------------
table_schema_id    | 45035996273864536
table_schema       | hcat
hcatalog_schema    | default
table_name         | hcatalogtypes
hcatalog_user_name | hcatuser
-[ RECORD 2 ]------+------------------
table_schema_id    | 45035996273864536
table_schema       | hcat
hcatalog_schema    | default
table_name         | tweets
hcatalog_user_name | hcatuser
-[ RECORD 3 ]------+------------------
table_schema_id    | 45035996273864536
table_schema       | hcat
hcatalog_schema    | default
table_name         | messages
hcatalog_user_name | hcatuser
-[ RECORD 4 ]------+------------------
table_schema_id    | 45035996273864536
table_schema       | hcat
hcatalog_schema    | default
table_name         | msgjson
hcatalog_user_name | hcatuser

=> -- Get detailed description of a specific table
=> SELECT * FROM HCATALOG_TABLES WHERE table_name = 'msgjson';
-[ RECORD 1 ]---------+-----------------------------------------------------------
table_schema_id       | 45035996273864536
table_schema          | hcat
hcatalog_schema       | default
table_name            | msgjson
hcatalog_user_name    | hcatuser
min_file_size_bytes   |
total_number_files    | 10
location              | hdfs://hive.example.com:8020/user/exampleuser/msgjson
last_update_time      |
output_format         | org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
last_access_time      |
max_file_size_bytes   |
is_partitioned        | f
partition_expression  |
table_owner           |
input_format          | org.apache.hadoop.mapred.TextInputFormat
total_file_size_bytes | 453534
hcatalog_group        |
permission            |

=> -- Get list of columns in a specific table
=> SELECT * FROM HCATALOG_COLUMNS WHERE table_name = 'hcatalogtypes'
-> ORDER BY ordinal_position;
```

```
-[ RECORD 1 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | intcol
hcatalog_data_type       | int
data_type                | int
data_type_id             | 6
data_type_length         | 8
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 1
-[ RECORD 2 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | floatcol
hcatalog_data_type       | float
data_type                | float
data_type_id             | 7
data_type_length         | 8
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 2
-[ RECORD 3 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | doublecol
hcatalog_data_type       | double
data_type                | float
data_type_id             | 7
data_type_length         | 8
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 3
-[ RECORD 4 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | charcol
hcatalog_data_type       | string
data_type                | varchar(65000)
data_type_id             | 9
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
```

```
datetime_precision       |
interval_precision       |
ordinal_position         | 4
-[ RECORD 5 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | varcharcol
hcatalog_data_type       | string
data_type                | varchar(65000)
data_type_id             | 9
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 5
-[ RECORD 6 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | boolcol
hcatalog_data_type       | boolean
data_type                | boolean
data_type_id             | 5
data_type_length         | 1
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 6
-[ RECORD 7 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | timestampcol
hcatalog_data_type       | string
data_type                | varchar(65000)
data_type_id             | 9
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 7
-[ RECORD 8 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | varbincol
hcatalog_data_type       | binary
data_type                | varbinary(65000)
data_type_id             | 17
data_type_length         | 65000
```

```
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 8
-[ RECORD 9 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | bincol
hcatalog_data_type       | binary
data_type                | varbinary(65000)
data_type_id             | 17
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 9
```

Synchronizing an HCatalog schema or table with a local schema or table

Querying data from an HCatalog schema can be slow due to Hive performance issues. This slow performance can be especially annoying when you want to examine the structure of the tables in the Hive database. Getting this information from Hive requires you to query the HCatalog schema's metadata using the HCatalog Connector.

To avoid this performance problem you can use the SYNC_WITH_HCATALOG_SCHEMA function to create a snapshot of the HCatalog schema's metadata within a Vertica schema. You supply this function with the name of a pre-existing Vertica schema, typically the one created through CREATE HCATALOG SCHEMA, and a Hive schema available through the HCatalog Connector. You must have permission both in Vertica to write the data and in Hive and HDFS to read it.

> **Note**
> To synchronize a schema, you must have read permission for the underlying files in HDFS. If Hive uses Sentry to manage authorization, then you can use ACL synchronization to manage HDFS access. Otherwise, the user of this function must have read access in HDFS.

The function creates a set of external tables within the Vertica schema that you can then use to examine the structure of the tables in the Hive database. Because the metadata in the Vertica schema is local, query planning is much faster. You can also use standard Vertica statements and system-table queries to examine the structure of Hive tables in the HCatalog schema.

> **Caution**
> The SYNC_WITH_HCATALOG_SCHEMA function overwrites tables in the Vertica schema whose names match a table in the HCatalog schema. Do not use the Vertica schema to store other data.

When SYNC_WITH_HCATALOG_SCHEMA creates tables in Vertica, it matches Hive's STRING and BINARY types to Vertica's VARCHAR(65000) and VARBINARY(65000) types. You might want to change these lengths, using ALTER TABLE SET DATA TYPE, in two cases:

- If the value in Hive is larger than 65000 bytes, increase the size and use LONG VARCHAR or LONG VARBINARY to avoid data truncation. If a Hive string uses multi-byte encodings, you must increase the size in Vertica to avoid data truncation. This step is needed because Hive counts string length in characters while Vertica counts it in bytes.
- If the value in Hive is much smaller than 65000 bytes, reduce the size to conserve memory in Vertica.

The Vertica schema is just a snapshot of the HCatalog schema's metadata. Vertica does not synchronize later changes to the HCatalog schema with the local schema after you call SYNC_WITH_HCATALOG_SCHEMA. You can call the function again to re-synchronize the local schema to the HCatalog schema. If you altered column data types, you will need to repeat those changes because the function creates new external tables.

By default, SYNC_WITH_HCATALOG_SCHEMA does not drop tables that appear in the local schema that do not appear in the HCatalog schema. Thus, after the function call the local schema does not reflect tables that have been dropped in the Hive database since the previous call. You can change this behavior by supplying the optional third Boolean argument that tells the function to drop any table in the local schema that does not correspond to a table in the HCatalog schema.

Instead of synchronizing the entire schema, you can synchronize individual tables by using SYNC_WITH_HCATALOG_SCHEMA_TABLE . If the table already exists in Vertica the function overwrites it. If the table is not found in the HCatalog schema, this function returns an error. In all other respects this function behaves in the same way as SYNC_WITH_HCATALOG_SCHEMA.

If you change the settings of any HCatalog Connector configuration parameters (Hadoop parameters), you must call this function again.

Examples

The following example demonstrates calling SYNC_WITH_HCATALOG_SCHEMA to synchronize the HCatalog schema in Vertica with the metadata in Hive. Because it synchronizes the HCatalog schema directly, instead of synchronizing another schema with the HCatalog schema, both arguments are the same.

```
=> CREATE HCATALOG SCHEMA hcat WITH hostname='hcathost' HCATALOG_SCHEMA='default'
   HCATALOG_USER='hcatuser';
CREATE SCHEMA
=> SELECT sync_with_hcatalog_schema('hcat', 'hcat');
sync_with_hcatalog_schema
-----------------------------------------
Schema hcat synchronized with hcat
tables in hcat = 56
tables altered in hcat = 0
tables created in hcat = 56
stale tables in hcat = 0
table changes erred in hcat = 0
(1 row)

=> -- Use vsql's \d command to describe a table in the synced schema

=> \d hcat.messages
List of Fields by Tables
 Schema  |  Table  | Column |      Type      | Size  | Default | Not Null | Primary Key | Foreign Key
---------+---------+--------+----------------+-------+---------+----------+-------------+-------------
hcat     | messages | id     | int            |     8 |         | f        | f           |
hcat     | messages | userid | varchar(65000) | 65000 |         | f        | f           |
hcat     | messages | "time" | varchar(65000) | 65000 |         | f        | f           |
hcat     | messages | message | varchar(65000) | 65000 |        | f        | f           |
(4 rows)
```

This example shows synchronizing with a schema created using CREATE HCATALOG SCHEMA. Synchronizing with a schema created using CREATE SCHEMA is also supported.

You can query tables in the local schema that you synchronized with an HCatalog schema. However, querying tables in a synchronized schema isn't much faster than directly querying the HCatalog schema, because SYNC_WITH_HCATALOG_SCHEMA only duplicates the HCatalog schema's metadata. The data in the table is still retrieved using the HCatalog Connector.

Data type conversions from Hive to Vertica

The data types recognized by Hive differ from the data types recognized by Vertica. The following table lists how the HCatalog Connector converts Hive data types into data types compatible with Vertica.

Hive Data Type	**Vertica Data Type**
TINYINT (1-byte)	TINYINT (8-bytes)
SMALLINT (2-bytes)	SMALLINT (8-bytes)
INT (4-bytes)	INT (8-bytes)

BIGINT (8-bytes)	BIGINT (8-bytes)
BOOLEAN	BOOLEAN
FLOAT (4-bytes)	FLOAT (8-bytes)
DECIMAL (precision, scale)	DECIMAL (precision, scale)
DOUBLE (8-bytes)	DOUBLE PRECISION (8-bytes)
CHAR (length in characters)	CHAR (length in bytes)
VARCHAR (length in characters)	VARCHAR (length in bytes), if length <= 65000 LONG VARCHAR (length in bytes), if length > 65000
STRING (2 GB max)	VARCHAR (65000)
BINARY (2 GB max)	VARBINARY (65000)
DATE	DATE
TIMESTAMP	TIMESTAMP
LIST/ARRAY	VARCHAR (65000) containing a JSON-format representation of the list.
MAP	VARCHAR (65000) containing a JSON-format representation of the map.
STRUCT	VARCHAR (65000) containing a JSON-format representation of the struct.

Data-width handling differences between Hive and Vertica

The HCatalog Connector relies on Hive SerDe classes to extract data from files on HDFS. Therefore, the data read from these files are subject to Hive's data width restrictions. For example, suppose the SerDe parses a value for an INT column into a value that is greater than 2^{32} -1 (the maximum value for a 32-bit integer). In this case, the value is rejected even if it would fit into a Vertica's 64-bit INTEGER column because it cannot fit into Hive's 32-bit INT.

Hive measures CHAR and VARCHAR length in characters and Vertica measures them in bytes. Therefore, if multi-byte encodings are being used (like Unicode), text might be truncated in Vertica.

Once the value has been parsed and converted to a Vertica data type, it is treated as native data. This treatment can result in some confusion when comparing the results of an identical query run in Hive and in Vertica. For example, if your query adds two INT values that result in a value that is larger than 2^{32} -1, the value overflows its 32-bit INT data type, causing Hive to return an error. When running the same query with the same data in Vertica using the HCatalog Connector, the value will probably still fit within Vertica's 64-int value. Thus the addition is successful and returns a value.

Using nonstandard SerDes

Hive stores its data in unstructured flat files located in the Hadoop Distributed File System (HDFS). When you execute a Hive query, it uses a set of serializer and deserializer (SerDe) classes to extract data from these flat files and organize it into a relational database table. For Hive to be able to extract data from a file, it must have a SerDe that can parse the data the file contains. When you create a table in Hive, you can select the SerDe to be used for the table's data.

Hive has a set of standard SerDes that handle data in several formats such as delimited data and data extracted using regular expressions. You can also use third-party or custom-defined SerDes that allow Hive to process data stored in other file formats. For example, some commonly-used third-party SerDes handle data stored in JSON format.

The HCatalog Connector directly fetches file segments from HDFS and uses Hive's SerDes classes to extract data from them. The Connector includes all Hive's standard SerDes classes, so it can process data stored in any file that Hive natively supports. If you want to query data from a Hive table that uses a custom SerDe, you must first install the SerDe classes on the Vertica cluster.

Determining which SerDe you need

If you have access to the Hive command line, you can determine which SerDe a table uses by using Hive's SHOW CREATE TABLE statement. This statement shows the HiveQL statement needed to recreate the table. For example:

```
hive> SHOW CREATE TABLE msgjson;
OK
CREATE EXTERNAL TABLE msgjson(
messageid int COMMENT 'from deserializer',
userid string COMMENT 'from deserializer',
time string COMMENT 'from deserializer',
message string COMMENT 'from deserializer')
ROW FORMAT SERDE
'org.apache.hadoop.hive.contrib.serde2.JsonSerde'
STORED AS INPUTFORMAT
'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
'hdfs://hivehost.example.com:8020/user/exampleuser/msgjson'
TBLPROPERTIES (
'transient_lastDdlTime'='1384194521')
Time taken: 0.167 seconds
```

In the example, ROW FORMAT SERDE indicates that a special SerDe is used to parse the data files. The next row shows that the class for the SerDe is named org.apache.hadoop.hive.contrib.serde2.JsonSerde .You must provide the HCatalog Connector with a copy of this SerDe class so that it can read the data from this table.

You can also find out which SerDe class you need by querying the table that uses the custom SerDe. The query will fail with an error message that contains the class name of the SerDe needed to parse the data in the table. In the following example, the portion of the error message that names the missing SerDe class is in bold.

```
=> SELECT * FROM hcat.jsontable;
ERROR 3399:  Failure in UDx RPC call InvokePlanUDL(): Error in User Defined
Object [VHCatSource], error code: 0
com.vertica.sdk.UdfException: Error message is [
org.apache.hcatalog.common.HCatException : 2004 : HCatOutputFormat not
initialized, setOutput has to be called. Cause : java.io.IOException:
java.lang.RuntimeException:
MetaException(message:org.apache.hadoop.hive.serde2.SerDeException
SerDe com.cloudera.hive.serde.JSONSerDe does not exist) ] HINT If error
message is not descriptive or local, may be we cannot read metadata from hive
metastore service thrift://hcathost:9083 or HDFS namenode (check
UDxLogs/UDxFencedProcessesJava.log in the catalog directory for more information)
at com.vertica.hcatalogudl.HCatalogSplitsNoOpSourceFactory
.plan(HCatalogSplitsNoOpSourceFactory.java:98)
at com.vertica.udxfence.UDxExecContext.planUDSource(UDxExecContext.java:898)
. . .
```

Installing the SerDe on the Vertica cluster

You usually have two options to getting the SerDe class file the HCatalog Connector needs:

- Find the installation files for the SerDe, then copy those over to your Vertica cluster. For example, there are several third-party JSON SerDes available from sites like Google Code and GitHub. You may find the one that matches the file installed on your Hive cluster. If so, then download the package and copy it to your Vertica cluster.
- Directly copy the JAR files from a Hive server onto your Vertica cluster. The location for the SerDe JAR files depends on your Hive installation. On some systems, they may be located in /usr/lib/hive/lib .

Wherever you get the files, copy them into the /opt/vertica/packages/hcat/lib directory on every node in your Vertica cluster.

Important

If you add a new host to your Vertica cluster, remember to copy every custom SerDer JAR file to it.

Troubleshooting HCatalog Connector problems

You may encounter the following issues when using the HCatalog Connector.

Connection errors

The HCatalog Connector can encounter errors both when you define a schema and when you query it. The types of errors you get depend on which CREATE HCATALOG SCHEMA parameters are incorrect. Suppose you have incorrect parameters for the metastore database, but correct parameters for HiveServer2. In this case, HCatalog-related system table queries succeed, while queries on the HCatalog schema fail. The following example demonstrates creating an HCatalog schema with the correct default HiveServer2 information. However, the port number for the metastore database is incorrect.

```
=> CREATE HCATALOG SCHEMA hcat2 WITH hostname='hcathost'
-> HCATALOG_SCHEMA='default' HCATALOG_USER='hive' PORT=1234;
CREATE SCHEMA
=> SELECT * FROM HCATALOG_TABLE_LIST;
-[ RECORD 1 ]------+--------------------
table_schema_id    | 45035996273864536
table_schema       | hcat2
hcatalog_schema    | default
table_name         | test
hcatalog_user_name | hive

=> SELECT * FROM hcat2.test;
ERROR 3399:  Failure in UDx RPC call InvokePlanUDL(): Error in User Defined
 Object [VHCatSource], error code: 0
com.vertica.sdk.UdfException: Error message is [
org.apache.hcatalog.common.HCatException : 2004 : HCatOutputFormat not
initialized, setOutput has to be called. Cause : java.io.IOException:
MetaException(message:Could not connect to meta store using any of the URIs
provided. Most recent failure: org.apache.thrift.transport.TTransportException:
java.net.ConnectException:
Connection refused
at org.apache.thrift.transport.TSocket.open(TSocket.java:185)
at org.apache.hadoop.hive.metastore.HiveMetaStoreClient.open(
HiveMetaStoreClient.java:277)
. . .
```

To resolve these issues, you must drop and recreate the schema or alter the schema to correct the parameters. If you still have issues, determine whether there are connectivity issues between your Vertica cluster and your Hadoop cluster. Such issues can include a firewall that prevents one or more Vertica hosts from contacting the HiveServer2, metastore, or HDFS hosts.

UDx failure when querying data: error 3399

You might see an error message when querying data (as opposed to metadata like schema information). This might be accompanied by a ClassNotFoundException in the log. This can happen for the following reasons:

- You are not using the same version of Java on your Hadoop and Vertica nodes. In this case you need to change one of them to match the other.
- You have not used hcatUtil to copy all Hadoop and Hive libraries and configuration files to Vertica, or you ran hcatutil and then changed your version of Hadoop or Hive.
- You upgraded Vertica to a new version and did not rerun hcatutil and reinstall the HCatalog Connector.
- The version of Hadoop you are using relies on a third-party library that you must copy manually.
- You are reading files with LZO compression and have not copied the libraries or set the io.compression.codecs property in core-site.xml .
- You are reading Parquet data from Hive, and columns were added to the table after some data was already present in the table. Adding columns does not update existing data, and the ParquetSerDe provided by Hive and used by the HCatalog Connector does not handle this case. This error is due to a limitation in Hive and there is no workaround.
- The query is taking too long and is timing out. If this is a frequent problem, you can increase the value of the UDxFencedBlockTimeout configuration parameter. See General parameters .

If you did not copy the libraries or configure LZO compression, follow the instructions in Configuring Vertica for HCatalog .

If the Hive jars that you copied from Hadoop are out of date, you might see an error message like the following:

```
ERROR 3399: Failure in UDx RPC call InvokePlanUDL(): Error in User Defined Object [VHCatSource],
error code: 0 Error message is [ Found interface org.apache.hadoop.mapreduce.JobContext, but
class was expected ]
HINT hive metastore service is thrift://localhost:13433 (check UDxLogs/UDxFencedProcessesJava.log
in the catalog directory for more information)
```

This error usually signals a problem with hive-hcatalog-core jar . Make sure you have an up-to-date copy of this file. Remember that if you rerun hcatUtil you also need to re-create the HCatalog schema.

You might also see a different form of this error:

```
ERROR 3399: Failure in UDx RPC call InvokePlanUDL(): Error in User Defined Object [VHCatSource],
error code: 0 Error message is [ javax/servlet/Filter ]
```

This error can be reported even if hcatUtil reports that your libraries are up to date. The javax.servlet.Filter class is in a library that some versions of Hadoop use but that is not usually part of the Hadoop installation directly. If you see an error mentioning this class, locate servlet-api-*.jar on a Hadoop node and copy it to the hcat/lib directory on all database nodes. If you cannot locate it on a Hadoop node, locate and download it from the Internet. (This case is rare.) The library version must be 2.3 or higher.

After you have copied the jar to the hcat/lib directory, reinstall the HCatalog connector as explained in Configuring Vertica for HCatalog .

Authentication error when querying data

You might have successfully created a schema using CREATE HCATALOG SCHEMA but get errors at query time such as the following:

```
=> SELECT * FROM hcat.clickdata;
ERROR 6776:  Failed to glob [hdfs:///user/hive/warehouse/click-*.parquet]
because of error: hdfs:///user/hive/warehouse/click-12214.parquet: statAll failed;
error: AuthenticationFailed
```

You might see this error if Hive uses an authorization service. If the permissions on the underlying files in HDFS match those in the authorization service, then Vertica must use user impersonation when accessing that data. To enable user impersonation, set the EnableHCatImpersonation configuration parameter to 1.

Vertica uses the database principal to access HDFS. Therefore, if EnableHCatImpersonation is 0, the Vertica database principal must have access to the data inside the hive warehouse on HDFS. If it does not, you might see the following error:

```
=> SELECT * FROM hcat.salesdata;
ERROR 6776:  Failed to glob [vertica-hdfs:///hive/warehouse/sales/*/*]
because of error: vertica-hdfs:///hive/warehouse/sales/: listStat failed;
error: Permission denied: user=vertica, access=EXECUTE, inode="/hive/warehouse/sales":hdfs:hdfs:d---------
```

The URL scheme in this error message has been changed from hdfs to vertica-hdfs . This is an internal scheme and is not valid in URLs outside of Vertica. You cannot use this scheme when specifying paths in HDFS.

Differing results between Hive and Vertica queries

Sometimes, running the same query on Hive and on Vertica through the HCatalog Connector can return different results. There are a few common causes of this problem.

This discrepancy is often caused by the differences between the data types supported by Hive and Vertica. See Data type conversions from Hive to Vertica for more information about supported data types.

If Hive string values are being truncated in Vertica, this might be caused by multi-byte character encodings in Hive. Hive reports string length in characters, while Vertica records it in bytes. For a two-byte encoding such as Unicode, you need to double the column size in Vertica to avoid truncation.

Discrepancies can also occur if the Hive table uses partition columns of types other than string.

If the Hive table stores partition data in custom locations instead of the default, you will see different query results if you do not specify the CUSTOM_PARTITIONS parameter when creating the Vertica schema. See Using Partitioned Data . Further, even when using custom partitions, there are differences between Vertica and Hive:

- If a custom partition is unavailable at query time, Hive ignores it and returns the rest of the results. Vertica reports an error.
- If a partition is altered in Hive to use a custom location after table creation, Hive reads data from only the new location while Vertica reads data from both the default and the new locations.
- If a partition value and its corresponding directory name disagree, Hive uses the value in its metadata while Vertica uses the value in the directory name.

The following example illustrates these differences. A table in Hive, partitioned on the state column, starts with the following data:

```
hive> select * from inventory;
+--------------+----------------+----------------+--+
| inventory.ID | inventory.quant | inventory.state |
+--------------+----------------+----------------+--+
| 2            | 300            | CA             |
| 4            | 400            | CA             |
| 5            | 500            | DC             |
| 1            | 100            | PA             |
| 2            | 200            | PA             |
+--------------+----------------+----------------+--+
```

The partition for one state, CA, is then moved. This directory contains some data already. Note that the query now returns different results for the CA partitions.

```
hive> ALTER TABLE inventory PARTITION (state='CA') SET LOCATION 'hdfs:///partitions/inventory/state=MD';
hive> select * from inventory;
+--------------+----------------+----------------+--+
| inventory.ID | inventory.quant | inventory.state |
+--------------+----------------+----------------+--+
| 20           | 30000          | CA             |
| 40           | 40000          | CA             |
| 5            | 500            | DC             |
| 1            | 100            | PA             |
| 2            | 200            | PA             |
+--------------+----------------+----------------+--+
5 rows selected (0.399 seconds)
```

The CA partitions were moved to a directory named state=MD. Vertica reports the state for the first two rows, the ones in the new partition location, as MD because of the directory name. It also reports the CA values from the original location in addition to the new one:

```
=> SELECT * FROM hcat.inventory;
 ID | quant | state
----+----------+------
 20 | 30000 | MD
 40 | 40000 | MD
  2 | 300   | CA
  4 | 400   | CA
  1 | 100   | PA
  2 | 200   | PA
  5 | 500   | DC
(7 rows)
```

HCatalog Connector installation fails on MapR

If you mount a MapR file system as an NFS mount point and then install the HCatalog Connector, it could fail with a message like the following:

```
ROLLBACK 2929: Couldn't create new UDx side process,
failed to get UDx side process info from zygote: Broken pipe
```

This might be accompanied by an error like the following in dbLog :

```
java.io.IOException: Couldn't get lock for /home/dbadmin/node02_catalog/UDxLogs/UDxFencedProcessesJava.log
    at java.util.logging.FileHandler.openFiles(FileHandler.java:389)
    at java.util.logging.FileHandler.<init>(FileHandler.java:287)
    at com.vertica.udxfence.UDxLogger.setup(UDxLogger.java:78)
    at com.vertica.udxfence.UDxSideProcess.go(UDxSideProcess.java:75)
    ...
```

This error occurs if you locked your NFS mount point when creating it. Locking is the default. If you use the HCatalog Connector with MapR mounted as an NFS mount point, you must create the mount point with the -o nolock option. For example:

```
sudo mount -o nolock -t nfs MaprCLDBserviceHostname:/mapr/ClusterName/vertica/$(hostname -f)/ vertica
```

You can use the HCatalog Connector with MapR without mounting the MapR file system. If you mount the MapR file system, you must do so without a lock.

Excessive query delays

Network issues or high system loads on the HiveServer2 server can cause long delays while querying a Hive database using the HCatalog Connector. While Vertica cannot resolve these issues, you can set parameters that limit how long Vertica waits before canceling a query on an HCatalog schema. You can set these parameters globally using Vertica configuration parameters. You can also set them for specific HCatalog schemas in the CREATE HCATALOG SCHEMA statement. These specific settings override the settings in the configuration parameters.

The HCatConnectionTimeout configuration parameter and the CREATE HCATALOG SCHEMA statement's HCATALOG_CONNECTION_TIMEOUT parameter control how many seconds the HCatalog Connector waits for a connection to the HiveServer2 server. A value of 0 (the default setting for the configuration parameter) means to wait indefinitely. If the server does not respond by the time this timeout elapses, the HCatalog Connector breaks the connection and cancels the query. If you find that some queries on an HCatalog schema pause excessively, try setting this parameter to a timeout value, so the query does not hang indefinitely.

The HCatSlowTransferTime configuration parameter and the CREATE HCATALOG SCHEMA statement's HCATALOG_SLOW_TRANSFER_TIME parameter specify how long the HCatlog Connector waits for data after making a successful connection to the server. After the specified time has elapsed, the HCatalog Connector determines whether the data transfer rate from the server is at least the value set in the HCatSlowTransferLimit configuration parameter (or by the CREATE HCATALOG SCHEMA statement's HCATALOG_SLOW_TRANSFER_LIMIT parameter). If it is not, then the HCatalog Connector terminates the connection and cancels the query.

You can set these parameters to cancel queries that run very slowly but do eventually complete. However, query delays are usually caused by a slow connection rather than a problem establishing the connection. Therefore, try adjusting the slow transfer rate settings first. If you find the cause of the issue is connections that never complete, you can alternately adjust the Linux TCP socket timeouts to a suitable value instead of relying solely on the HCatConnectionTimeout parameter.

SerDe errors

Errors can occur if you attempt to query a Hive table that uses a nonstandard SerDe. If you have not installed the SerDe JAR files on your Vertica cluster, you receive an error similar to the one in the following example:

```
=> SELECT * FROM hcat.jsontable;
ERROR 3399:  Failure in UDx RPC call InvokePlanUDL(): Error in User Defined
Object [VHCatSource], error code: 0
com.vertica.sdk.UdfException: Error message is [
org.apache.hcatalog.common.HCatException : 2004 : HCatOutputFormat not
initialized, setOutput has to be called. Cause : java.io.IOException:
java.lang.RuntimeException:
MetaException(message:org.apache.hadoop.hive.serde2.SerDeException
SerDe com.cloudera.hive.serde.JSONSerDe does not exist) ] HINT If error
message is not descriptive or local, may be we cannot read metadata from hive
metastore service thrift://hcathost:9083 or HDFS namenode (check
UDxLogs/UDxFencedProcessesJava.log in the catalog directory for more information)
at com.vertica.hcatalogudl.HCatalogSplitsNoOpSourceFactory
.plan(HCatalogSplitsNoOpSourceFactory.java:98)
at com.vertica.udxfence.UDxExecContext.planUDSource(UDxExecContext.java:898)
. . .
```

In the error message, you can see that the root cause is a missing SerDe class (shown in bold). To resolve this issue, install the SerDe class on your Vertica cluster. See Using nonstandard SerDes for more information.

This error may occur intermittently if just one or a few hosts in your cluster do not have the SerDe class.

Integrating with Cloudera Manager

The Cloudera distribution of Hadoop includes Cloudera Manager, a web-based tool for managing a Hadoop cluster. Cloudera Manager can manage any service for which a service description is available, including Vertica.

You can use Cloudera Manager to start, stop, and monitor individual database nodes or the entire database. You can manage both co-located and separate Vertica clusters—Cloudera can manage services on nodes that are not part of the Hadoop cluster.

You must install and configure your Vertica database before proceeding; you cannot use Cloudera Manager to create the database.

Installing the service

> **Note**
> Because the service has to send the database password over the network, you should enable encryption on your Hadoop cluster before

proceeding.

A Cloudera Service Description (CSD) file describes a service that Cloudera can manage. The Vertica CSD is in /opt/vertica/share/CSD on a database node.

To install the Vertica CSD, follow these steps:

1. On a Vertica node, follow the instructions in VerticaAPIKey to generate an API key. You need this key to finish the installation of the CSD.
2. On the Hadoop node that hosts Cloudera Manager, copy the CSD file into /opt/cloudera/csd.
3. Restart Cloudera Manager:
   ```
   $ service cloudera-scm-server restart
   ```
4. In a web browser, go to Cloudera Manager and restart the Cloudera Management Service.
5. If your Vertica cluster is separate from your Hadoop cluster (not co-located on it): Use Cloudera Manager to add the hosts for your database nodes. If your cluster is co-located, skip this step.
6. Use Cloudera Manager to add the Vertica service.
7. On the "Role Assignment" page, select the hosts that are database nodes.
8. On the "Configuration" page, specify values for the following fields:
   - database name
   - agent port (accept the default if you're not sure)
   - API key
   - database user to run as (usually dbadmin) and password

About the agent

When you manage Vertica through Cloudera Manager, you are actually interacting with the Vertica Agent, not the database directly. The Agent runs on all database nodes and interacts with the database on your behalf. Management Console uses the same agent. Most of the time this extra indirection is transparent to you.

A Cloudera-managed service contains one or more roles. In this case the service is "Vertica" and the single role is "Vertica Node".

Available operations

Cloudera Manager shows two groups of operations. Service-level operations apply to the service on all nodes, while role-level operations apply only to a single node.

You can perform the following service-level operations on all nodes:

- Start: Starts the agent and, if it is not already running, the database.
- Stop: Stops the database and agent.
- Restart: Calls Stop and then Start.
- Add Role Instances: Adds new database nodes to Cloudera Manager. The nodes must already be part of the Vertica cluster, and the hosts must already be known to Cloudera Manager.
- Enter Maintenance Mode: Suppresses health alerts generated by Cloudera Manager.
- Exit Maintenance Mode: Resumes normal reporting.
- Update Memory Pool Size: Applies memory-pool settings from the Static Service Pools configuration page.

You can perform all of these operations except Add Role Instances on individual nodes as role-level operations.

Managing memory pools

Cloudera Manager allows you to change resource allocations, such as memory and CPU, for the nodes it manages. If you are using co-located clusters, centrally managing resources can simplify your cluster management. If you are using separate Hadoop and Vertica clusters, you might prefer to manage Vertica separately as described in Managing the database .

Use the Cloudera Manager "Static Service Pools" configuration page to configure resource allocations. The "Vertica Memory Pool" value, specified in GB, is the maximum amount of memory to allocate to the database on each node. If the configuration page includes "Cgroup Memory Hard Limit", set it to the same value as "Vertica Memory Pool".

After you have set these values, you can use the "Update Memory Pool Size" operation to apply the value to the managed nodes. This operation is equivalent to ALTER RESOURCE POOL GENERAL MAXMEMORYSIZE. Configuration changes in "Static Service Pools" do not take effect in Vertica until you perform this operation.

Uninstalling the service

To uninstall the Vertica CSD, follow these steps:

1. Stop the Vertica service and then remove it from Cloudera Manager.
2. Remove the CSD file from /opt/cloudera/csd.
3. From the command line, restart the Cloudera Manager server.
4. In Cloudera Manager, restart the Cloudera Management Service.

Integrating Vertica with the MapR distribution of Hadoop

MapR is a distribution of Apache Hadoop produced by MapR Technologies that extends the standard Hadoop components with its own features. Vertica can integrate with MapR in the following ways:

- You can read data from MapR through an NFS mount point. After you mount the MapR file system as an NFS mount point, you can use CREATE EXTERNAL TABLE AS COPY or COPY to access the data as if it were on the local file system. This option provides the best performance for reading data.
- You can use the HCatalog Connector to read Hive data. Do not use the HCatalog Connector with ORC or Parquet data in MapR for performance reasons. Instead, mount the MapR file system as an NFS mount point and create external tables without using the Hive schema. For more about reading Hive data, see Using the HCatalog Connector .
- You can create a storage location to store data in MapR using the native Vertica format (ROS). Mount the MapR file system as an NFS mount point and then use CREATE LOCATION ...ALL NODES SHARED to create a storage location. (CREATE LOCATION does not support NFS mount points in general, but does support them for MapR.)

> **Note**
> If you create a Vertica database and place its initial storage location on MapR, Vertica designates the storage location for both DATA and TEMP usage. Vertica does not support TEMP storage locations on MapR, so after you create the location, you must alter it to store only DATA files. See Altering location use . Ensure that you have a TEMP location on the Linux file system.

Other Vertica integrations for Hadoop are not available for MapR.

For information on mounting the MapR file system as an NFS mount point, see Accessing Data with NFS and Configuring Vertica Analytics Platform with MapR on the MapR website. In particular, you must configure MapR to add Vertica as a MapR service.

Examples

In the following examples, the MapR file system has been mounted as /mapr.

The following statement creates an external table from ORC data:

```
=> CREATE EXTERNAL TABLE t (a1 INT, a2 VARCHAR(20))
  AS COPY FROM '/mapr/data/file.orc' ORC;
```

The following statement creates an external table from Parquet data and takes advantage of partition pruning (see Partitioned file paths):

```
=> CREATE EXTERNAL TABLE t2 (id int, name varchar(50), created date, region varchar(50))
  AS COPY FROM '/mapr/*/*/*' PARTITION COLUMNS created, region PARQUET();
```

The following statement loads ORC data from MapR into Vertica:

```
=> COPY t FROM '/mapr/data/*.orc' ON ANY NODE ORC;
```

The following statements create a storage location to hold ROS data in the MapR file system:

```
=> CREATE LOCATION '/mapr/my.cluster.com/data' SHARED USAGE 'DATA' LABEL 'maprfs';

=> SELECT ALTER_LOCATION_USE('/mapr/my.cluster.com/data', '', 'DATA');
```

Hive primer for Vertica integration

You can use Hive to export data for use by Vertica. Decisions you make when doing the Hive export affect performance in Vertica.

Tuning ORC stripes and Parquet rowgroups

Vertica can read ORC and Parquet files generated by any Hive version. However, newer Hive versions store more metadata in these files. This metadata is used by both Hive and Vertica to prune values and to read only the required data. Use the latest Hive version to store data in these formats. ORC and Parquet are fully forward- and backward-compatible. To get the best performance, use Hive 0.14 or later.

The ORC format splits a table into groups of rows called stripes and stores column-level metadata in each stripe. The Parquet format splits a table into groups of rows called rowgroups and stores column-level metadata in each rowgroup. Each stripe/rowgroup's metadata is used during predicate evaluation to determine whether the values from this stripe need to be read. Large stripes usually yield better performance, so set the stripe size to at least 256M.

Hive writes ORC stripes and Parquet rowgroups to HDFS, which stores data in HDFS blocks distributed among multiple physical data nodes. Accessing an HDFS block requires opening a separate connection to the corresponding data node. It is advantageous to ensure that an ORC stripe or Parquet rowgroup does not span more than one HDFS block. To do so, set the HDFS block size to be larger than the stripe/rowgroup size. Setting HDFS block size to 512M is usually sufficient.

Hive provides three compression options: None, Snappy, and Zlib. Use Snappy or Zlib compression to reduce storage and I/O consumption. Usually, Snappy is less CPU-intensive but can yield lower compression ratios compared to Zlib.

Storing data in sorted order can improve data access and predicate evaluation performance. Sort table columns based on the likelihood of their occurrence in query predicates; columns that most frequently occur in comparison or range predicates should be sorted first.

Partitioning tables is a very useful technique for data organization. Similarly to sorting tables by columns, partitioning can improve data access and predicate evaluation performance. Vertica supports Hive-style partitions and partition pruning.

The following Hive statement creates an ORC table with stripe size 256M and Zlib compression:

```
hive> CREATE TABLE customer_visits (
      customer_id bigint,
      visit_num int,
      page_view_dt date)
   STORED AS ORC tblproperties("orc.compress"="ZLIB",
      "orc.stripe.size"="268435456");
```

The following statement creates a Parquet table with stripe size 256M and Zlib compression:

```
hive> CREATE TABLE customer_visits (
      customer_id bigint,
      visit_num int,
      page_view_dt date)
   STORED AS PARQUET tblproperties("parquet.compression"="ZLIB",
      "parquet.stripe.size"="268435456");
```

Creating sorted files in Hive

Unlike Vertica, Hive does not store table columns in separate files and does not create multiple projections per table with different sort orders. For efficient data access and predicate pushdown, sort Hive table columns based on the likelihood of their occurrence in query predicates. Columns that most frequently occur in comparison or range predicates should be sorted first.

Data can be inserted into Hive tables in a sorted order by using the ORDER BY or SORT BY keywords. For example, to insert data into the ORC table "customer_visit" from another table "visits" with the same columns, use these keywords with the INSERT INTO command:

```
hive> INSERT INTO TABLE customer_visits
      SELECT * from visits
      ORDER BY page_view_dt;
```

```
hive> INSERT INTO TABLE customer_visits
      SELECT * from visits
      SORT BY page_view_dt;
```

The difference between the two keywords is that ORDER BY guarantees global ordering on the entire table by using a single MapReduce reducer to populate the table. SORT BY uses multiple reducers, which can cause ORC or Parquet files to be sorted by the specified column(s) but not be globally sorted. Using the latter keyword can increase the time taken to load the file.

You can combine clustering and sorting to sort a table globally. The following table definition adds a hint that data is inserted into this table bucketed by customer_id and sorted by page_view_dt:

```
hive> CREATE TABLE customer_visits_bucketed (
      customer_id bigint,
      visit_num int,
      page_view_dt date)
   CLUSTERED BY (page_view_dt)
   SORTED BY (page_view_dt)INTO 10 BUCKETS
   STORED AS ORC;
```

When inserting data into the table, you must explicitly specify the clustering and sort columns, as in the following example:

```
hive> INSERT INTO TABLE customer_visits_bucketed
   SELECT * from visits
   DISTRIBUTE BY page_view_dt
   SORT BY page_view_dt;
```

The following statement is equivalent:

```
hive> INSERT INTO TABLE customer_visits_bucketed
   SELECT * from visits
   CLUSTER BY page_view_dt;
```

Both of the above commands insert data into the customer_visits_bucketed table, globally sorted on the page_view_dt column.

Partitioning Hive tables

Table partitioning in Hive is an effective technique for data separation and organization, as well as for reducing storage requirements. To partition a table in Hive, include it in the PARTITIONED BY clause:

```
hive> CREATE TABLE customer_visits (
      customer_id bigint,
      visit_num int)
   PARTITIONED BY (page_view_dt date)
   STORED AS ORC;
```

Hive does not materialize partition column(s). Instead, it creates subdirectories of the following form:

```
path_to_table/partition_column_name=value/
```

When the table is queried, Hive parses the subdirectories' names to materialize the values in the partition columns. The value materialization in Hive is a plain conversion from a string to the appropriate data type.

Inserting data into a partitioned table requires specifying the value(s) of the partition column(s). The following example creates two partition subdirectories, "customer_visits/page_view_dt=2016-02-01" and "customer_visits/page_view_dt=2016-02-02":

```
hive> INSERT INTO TABLE customer_visits
   PARTITION (page_view_dt='2016-02-01')
   SELECT customer_id, visit_num from visits
   WHERE page_view_dt='2016-02-01'
   ORDER BY page_view_dt;

hive> INSERT INTO TABLE customer_visits
   PARTITION (page_view_dt='2016-02-02')
   SELECT customer_id, visit_num from visits
   WHERE page_view_dt='2016-02-02'
   ORDER BY page_view_dt;
```

Each directory contains ORC files with two columns, customer_id and visit_num.

Example: a partitioned, sorted ORC table

Suppose you have data stored in CSV files containing three columns: customer_id, visit_num, page_view_dtm:

```
1,123,2016-01-01
33,1,2016-02-01
2,57,2016-01-03
...
```

The goal is to create the following Hive table:

```
hive> CREATE TABLE customer_visits (
        customer_id bigint,
        visit_num int)
   PARTITIONED BY (page_view_dt date)
   STORED AS ORC;
```

To achieve this, perform the following steps:

1. Copy or move the CSV files to HDFS.
2. Define a textfile Hive table and copy the CSV files into it:

   ```
   hive> CREATE TABLE visits (
           customer_id bigint,
           visit_num int,
           page_view_dt date)
      ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
      STORED AS TEXTFILE;

   hive> LOAD DATA INPATH path_to_csv_files INTO TABLE visits;
   ```

3. For each unique value in page_view_dt, insert the data into the target table while materializing page_view_dt as page_view_dtm:

   ```
   hive> INSERT INTO TABLE customer_visits
         PARTITION (page_view_dt='2016-01-01')
         SELECT customer_id, visit_num FROM visits
         WHERE page_view_dt='2016-01-01'
         ORDER BY page_view_dt;
   ...
   ```

   This operation inserts data from visits.customer_id into customer_visits.customer_id, and from visits.visit_num into customer_visits.visit_num. These two columns are stored in generated ORC files. Simultaneously, values from visits.page_view_dt are used to create partitions for the partition column customer_visits.page_view_dt, which is not stored in the ORC files.

Data modification in Hive

Hive is well-suited for reading large amounts of write-once data. Its optimal usage is loading data in bulk into tables and never modifying the data. In particular, for data stored in the ORC and Parquet formats, this usage pattern produces large, globally (or nearly globally) sorted files.

Periodic addition of data to tables (known as “trickle load”) is likely to produce many small files. The disadvantage of this is that Vertica has to access many more files during query planning and execution. These extra access can result in longer query-processing time. The major performance degradation comes from the increase in the number of file seeks on HDFS.

Hive can also modify underlying ORC or Parquet files without user involvement. If enough records in a Hive table are modified or deleted, for example, Hive deletes existing files and replaces them with newly-created ones. Hive can also be configured to automatically merge many small files into a few larger files.

When new tables are created, or existing tables are modified in Hive, you must manually synchronize Vertica to keep it up to date. The following statement synchronizes the Vertica schema "hcat" after a change in Hive:

```
=> SELECT sync_with_hcatalog_schema('hcat_local', 'hcat');
```

Schema evolution in Hive

Hive supports two kinds of schema evolution:

1. New columns can be added to existing tables in Hive. Vertica automatically handles this kind of schema evolution. The old records display NULLs for the newer columns.
2. The type of a column for a table can be modified in Hive. Vertica does not support this kind of schema evolution.

The following example demonstrates schema evolution through new columns. In this example, hcat.parquet.txt is a file with the following values:

```
-1|0.65|0.65|6|'b'
```

```
hive> create table hcat.parquet_tmp (a int, b float, c double, d int, e varchar(4))
    row format delimited fields terminated by '|' lines terminated by '\n';

hive> load data local inpath 'hcat.parquet.txt' overwrite into table
    hcat.parquet_tmp;

hive> create table hcat.parquet_evolve (a int) partitioned by (f int) stored as
    parquet;
hive> insert into table hcat.parquet_evolve partition (f=1) select a from
    hcat.parquet_tmp;
hive> alter table hcat.parquet_evolve add columns (b float);
hive> insert into table hcat.parquet_evolve partition (f=2) select a, b from
    hcat.parquet_tmp;
hive> alter table hcat.parquet_evolve add columns (c double);
hive> insert into table hcat.parquet_evolve partition (f=3) select a, b, c from
    hcat.parquet_tmp;
hive> alter table hcat.parquet_evolve add columns (d int);
hive> insert into table hcat.parquet_evolve partition (f=4) select a, b, c, d from
    hcat.parquet_tmp;
hive> alter table hcat.parquet_evolve add columns (e varchar(4));
hive> insert into table hcat.parquet_evolve partition (f=5) select a, b, c, d, e
    from hcat.parquet_tmp;
hive> insert into table hcat.parquet_evolve partition (f=6) select a, b, c, d, e
    from hcat.parquet_tmp;

=> SELECT * from hcat_local.parquet_evolve;

 a |        b         |  c   | d | e | f
---+------------------+------+---+---+---
-1 |                  |      |   |   | 1
-1 | 0.649999976158142 |      |   |   | 2
-1 | 0.649999976158142 | 0.65 |   |   | 3
-1 | 0.649999976158142 | 0.65 | 6 |   | 4
-1 | 0.649999976158142 | 0.65 | 6 | b | 5
-1 | 0.649999976158142 | 0.65 | 6 | b | 6
(6 rows)
```

Apache Kafka integration

Vertica provides a high-performance mechanism for integrating with Apache Kafka, an open-source distributed real-time streaming platform. Because Vertica can both consume data from and produce data for Kafka, you can use Vertica as part of an automated analytics workflow: Vertica can retrieve data from Kafka, perform analytics on the data, and then send the results back to Kafka for consumption by other applications.

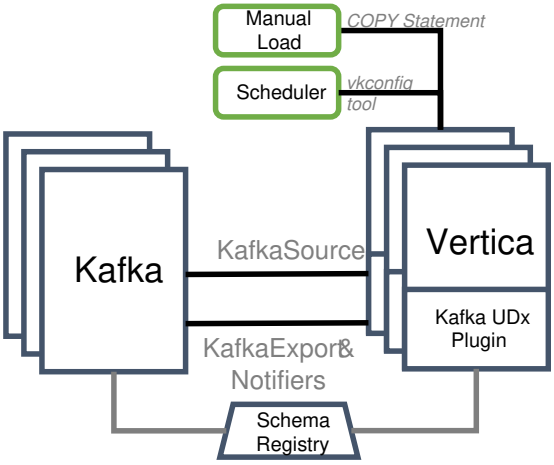
Prerequisites

- Install Vertica .
- Configure the database .
- Install and configure Apache Kafka .

Architecture overview

The Vertica and Kafka integration provides the following features:

- A UDx library containing functions that load and parse data from Kafka topics into Vertica
- A job scheduler that uses the UDL library to continuously consume data from Kafka with exactly-once semantics
- Push-based notifiers that send data collector messages from Vertica to Kafka
- A KafkaExport function that sends Vertica data to Kafka



Vertica as a Kafka consumer

A Kafka consumer reads messages written to Kafka by other data streams. Because Vertica can read messages from Kafka, you can store and analyze data from any application that sends data to Kafka without configuring each individual application to connect to Vertica. Vertica provides tools to automatically or manually consume data loads from Kafka.

Manual loads

Manually load a finite amount of data from Kafka by directly executing a COPY statement. This is useful if you want to analyze, test, or perform additional processing on a set of messages.

For more information, see Consuming data from Kafka .

Automatic loads

Automatically load data from Kafka with a job scheduler. A scheduler constantly loads data and ensures that each Kafka message is loaded exactly once.

You must install Java 8 on each Vertica node that runs the scheduler. For more information, see Automatically consume data from Kafka with the scheduler .

Vertica as a Kafka producer

A Kafka producer sends data to Kafka, which is then available to Kafka consumers for processing. You can send the following types of data to Kafka:

- Vertica anayltics results. Use KafkaExport to export Vertica tables and queries.
- Health and performance data from Data Collector tables. Create push-based notifiers to send this data for consumption for third-party monitoring tools.
- Ad hoc messages. Use NOTIFY to signal that tasks such as stored procedures are complete.

For more information, see Producing data for Kafka .

In this section

- Data streaming integration terms
- Configuring Vertica and Kafka
- Vertica Eon Mode and Kafka
- Consuming data from Kafka
- Avro Schema Registry
- Producing data for Kafka
- TLS/SSL encryption with Kafka
- Authenticating with Kafka using SASL
- Troubleshooting Kafka integration issues
- vkconfig script options
- Kafka function reference
- Data streaming schema tables

Data streaming integration terms

Vertica uses the following terms to describe its streaming feature. These are general terms, which may differ from each specific streaming platform's terminology.

Terminology

Host
A data streaming server.

Source
A feed of messages in a common category which streams into the same Vertica target tables. In Apache Kafka, a source is known as a topic.

Partition
Unit of parallelism within data streaming. Data streaming splits a source into multiple partitions, which can each be served in parallel to consumers such as a Vertica database. Within a partition, messages are usually ordered chronologically.

Offset
An index into a partition. This index is the position within an ordered queue of messages, not an index into an opaque byte stream.

Message
A unit of data within data streaming. The data is typically in JSON or Avro format. Messages are loaded as rows into Vertica tables, and are uniquely identified by their source, partition, and offset.

Data loader terminology

Scheduler
An external tool that schedules data loads from a streaming data source into Vertica.

Microbatch
A microbatch represents a single segment of a data load from a streaming data source. It encompasses all of the information the scheduler needs to perform a load from a streaming data source into Vertica.

Frame
The window of time during which a Scheduler executes microbatches to load data. This window controls the duration of each COPY statement the scheduler runs as a part of the microbatch. During the frame, the scheduler gives an active microbatch from each source an opportunity to load data. It gives priority to microbatches that need more time to load data based on the history of previous microbatches.

Stream
A feed of messages that is identified by a source and partition.

The offset uniquely identifies the position within a particular source-partition stream.

Lane
A thread within a job scheduler instance that issues microbatches to perform the load.

The number of lanes available is based on the PlannedConcurrency of the job scheduler's resource pool. Multiple lanes allow the scheduler to run microbatches for different sources in parallel during a frame.

Configuring Vertica and Kafka

Both Vertica and Kafka have settings you can use to optimize your streaming data loads. The topics in this section explain these settings.

In this section
- Kafka and Vertica configuration settings
- Directly setting Kafka library options
- Configuring Vertica for Apache Kafka version 0.9 and earlier

Kafka and Vertica configuration settings

You can fine-tune configuration settings for Vertica and Kafka that optimize performance.

The Vertica and Kafka integration uses rdkafka version 0.11.6. Unless otherwise noted, the following configuration settings use the default librdkafka configuration properties values. For instructions on how to override the default values, see Directly setting Kafka library options .

Vertica producer settings

These settings change how Vertica produces data for Kafka with the KafkaExport function and notifiers .

queue.buffering.max.messages
Size of the Vertica producer queue. If Vertica generates too many messages too quickly, the queue can fill, resulting in dropped messages. Increasing this value consumes more memory, but reduces the chance of lost messages.

Defaults :
- KafkaExport: 1000
- Notifiers: 10000

queue.buffering.max.ms

Frequency with which Vertica flushes the producer message queue. Lower values decrease latency at the cost of throughput. Higher values increase throughput, but can cause the producer queue (set by queue.buffering.max.messages) to fill more frequently, resulting in dropped messages.

Default: 100 ms

message.max.bytes

Maximum size of a Kafka protocol request message batch. This value should be the same on your sources, brokers, and producers.

message.send.max.retries

Number of attempts the producer makes to deliver the message to a broker. Higher values increase the chance of success.

retry.backoff.ms

Interval Vertica waits before resending a failed message.

request.required.acks

Number of broker replica acknowledgments Kafka requires before it considers message delivery successful. Requiring acknowledgments increases latency. Removing acknowledgments increases the risk of message loss.

request.timeout.ms

Interval that the producer waits for a response from the broker. Broker response time is affected by server load and the number of message acknowledgments you require.
Higher values increase latency.

compression.type

Compression algorithm used to encode data before sending it to a broker. Compression helps to reduce the network footprint of your Vertica producers and increase disk utilization. Vertica supports gzip and snappy .

Kafka broker settings

Kafka brokers receive messages from producers and distribute them among Kafka consumers. Configure these settings on the brokers themselves. These settings function independently of your producer and consumer settings. For detailed information on Apache Kafka broker settings, refer to the Apache Kafka documentation .

message.max.bytes

Maximum size of a Kafka protocol request message batch.This value should be the same on your sources, brokers, and producers.

num.io.threads

Number of network threads the broker uses to receive and process requests. More threads can increase your concurrency.

num.network.threads

Number of network threads the broker uses to accept network requests. More threads can increase your concurrency.

Vertica consumer settings

The following settings changes how Vertica acts when it consumes data from Kafka. You can set this value using the kafka_conf parameter on the KafkaSource UDL when directly executing a COPY statement. For schedulers, use the --message_max_bytes settings in the scheduler tool .

message.max.bytes

Maximum size of a Kafka protocol request message batch. Set this value to a high enough value to prevent the overhead of fetching batches of messages interfering with loading data. Defaults to 24MB for newly-created load specs.

Directly setting Kafka library options

Vertica relies on the open source rdkafka library to communicate with Apache Kafka. This library contains many options for controlling how Vertica and Kafka interact. You set the most common rdkafka library options through the settings in the vkconfig utility and the Kafka integration functions such as KafkaSource.

There are some rdkafka settings that cannot be directly set from within the Vertica. Under normal circumstances, you do not need to change them. However, if you find that you need to set a specific rdkafka setting that is not directly available from Vertica, you can directly pass options to the rdkafka library through the kafka_conf options.

The kafka_conf argument is supported when using a scheduler to load data from Kafka. You can set the values in the following ways (listed in order of lower to higher precedence):

- The Linux environment variable VERTICA_RDKAFKA_CONF set on the host where you run the vkconfig utility.

- The Linux environment variable VERTICA_RDKAFKA_CONF_ *KAFKA_CLUSTER* set on the host where you run the vkconfig utility. The *KAFKA_CLUSTER* portion of the variable name is the name of a Kafka cluster you have defined using vkconfig's cluster utility . The settings in this environment variable only affect the specific Kafka cluster you name in *KAFKA_CLUSTER* .
- The --kafka_conf option of the vkconfig utility. This option can be set in the cluster , source , launch , and sync tools. Note that the setting only applies to each vkconfig utility call—it does not carry over to other vkconfig utility calls. For example, if you need to supply an option to the cluster and source tool, you must supply the kafka_conf option to both of them.

> **Note**
> Using an environment variable to set your rdkafka options helps to keep your settings consistent. It is easy to forget to set the --kafka_conf option for each call to the vkconfig script.

All of these options cascade, so setting an option using the --kafka_conf argument to the cluster tool overrides the same option that was set in the environment variables.

You can also directly set rdkafka options when directly calling KafkaExport, KafkaSource, and several other Kafka integration functions. These functions accept a parameter named kafka_conf .

The kafka_conf option settings

The kafka_conf vkconfig option accepts a JSON object with settings in the following formats:

- One or more option/value pairs:
  ```
  --kafka_conf '{"option1":value1[, "option2":value2...]}'
  ```
- A single option with multiple values:
  ```
  --kafka_conf '{"option1":"value1[;value2...]"}'
  ```

Vertica provides the kafka_conf_secret parameter to pass sensitive configuration settings. This parameter accepts values in the same format as kafka_conf. Values passed to kafka_conf_secret are not logged or stored in system tables.

See the rdkafka project on GitHub for a list of the configuration options supported by the rdkafka library.

Important

Arbitrarily setting options via kafka_conf can result in errors or unpredictable behavior. If you encounter a problem loading messages after setting an rdkafka option using the kafka_conf option, roll back your change to see if that was the source of the problem.

To prevent confusion, never set options via the kafka_conf parameter that can be set directly through scheduler options. For example, do not use the kafka_conf option to set Kafka's message.max.bytes setting. Instead, use the load-spec tool's --message-max-bytes option.

Example

The following example demonstrates disabling rdkafka's api.version.request option when manually loading messages using KafkaSource. You should always disable this option when accessing Kafka cluster running version 0.9 or earlier. See Configuring Vertica for Apache Kafka version 0.9 and earlier for more information.

```
=> CREATE FLEX TABLE iot_data();
CREATE TABLE
=> COPY public.iot_data SOURCE KafkaSource(stream='iot_json|0|-2',
                                brokers='kafka-01.example.com:9092',
                                stop_on_eof=True,
                                kafka_conf='{"api.version.request":false}')
      PARSER KafkaJSONParser();
 Rows Loaded
-------------
      5000
(1 row)
```

This example demonstrates setting two options with a JSON object when calling the cluster tool. It disables the api.version.request option and enables CRC checks of messages from Kafka using the check.crcs option:

```
$ vkconfig cluster --create --cluster StreamCluster1 \
                   --hosts kafka01.example.com:9092,kafka02.example.com:9092 \
                   --conf myscheduler.config \
                   --kafka_conf '{"api.version.request":false, "check.crcs":true}'
```

The following example demonstrates setting the same options using an environment variable:

```
$ export VERTICA_RDKAFKA_CONF='{"api.version.request":false, "check.crcs":true}'
$ vkconfig cluster --create --cluster StreamCluster1 \
                   --hosts kafka01.example.com:9092,kafka02.example.com:9092 \
                   --conf myscheduler.config
```

Important
Setting the check.crc option is just an example. Vertica does not suggest you enable the CRC check in your schedulers under normal circumstances. It adds additional overhead and can result in slower performance.

Configuring Vertica for Apache Kafka version 0.9 and earlier

Apache Kafka version 0.10 introduced a new feature that allows consumers to determine which version of the Kafka API the Kafka brokers support. Consumers that support this feature send an initial API version query to the Kafka broker to determine the API version to use when communicating with them. Kafka brokers running version 0.9.0 or earlier cannot respond to the API query. If the consumer does not receive a reply from the Kafka broker within a timeout period (set to 10 seconds by default) the consumer can assume the Kafka broker is running Kafka version 0.9.0 or earlier.

The Vertica integration with Kafka supports this API query feature starting in version 9.1.1. This API check can cause problems if you are connecting Vertica to a Kafka cluster running 0.9.0 or earlier. You may notice poorer performance loading messages from Kafka and may experience errors as the 10 second API request timeout can cause parts of the Kafka integration feature to time out and report errors.

For example, if you run the vkconfig source utility to configure a source on a Kafka 0.9 cluster, you may get the following error:

```
$ vkconfig source --create --conf weblog.conf --cluster kafka_weblog --source web_hits
Exception in thread "main" com.vertica.solutions.kafka.exception.ConfigurationException:
ERROR: [[Vertica][VJDBC](5861) ERROR: Error calling processPartition() in
User Function KafkaListTopics at [/data/build-
centos6/qb/buildagent/workspace/jenkins2/PrimaryBuilds/build_master/build/udx/supported/kafka/KafkaUtil.cpp:173],
error code: 0, message: Error getting metadata: [Local: Broker transport failure]]
at com.vertica.solutions.kafka.model.StreamSource.validateConfiguration(StreamSource.java:184)
at com.vertica.solutions.kafka.model.StreamSource.setFromMapAndValidate(StreamSource.java:130)
at com.vertica.solutions.kafka.model.StreamModel.<init>(StreamModel.java:89)
at com.vertica.solutions.kafka.model.StreamSource.<init>(StreamSource.java:39)
at com.vertica.solutions.kafka.cli.SourceCLI.getNewModel(SourceCLI.java:53)
at com.vertica.solutions.kafka.cli.SourceCLI.getNewModel(SourceCLI.java:15)
at com.vertica.solutions.kafka.cli.CLI.run(CLI.java:56)
at com.vertica.solutions.kafka.cli.CLI._main(CLI.java:132)
at com.vertica.solutions.kafka.cli.SourceCLI.main(SourceCLI.java:25)
Caused by: java.sql.SQLNonTransientException: [Vertica][VJDBC](5861)
ERROR: Error calling processPartition() in User Function KafkaListTopics at [/data/build-
centos6/qb/buildagent/workspace/jenkins2/PrimaryBuilds/build_master/build/udx/supported/kafka/KafkaUtil.cpp:173],
error code: 0, message: Error getting metadata: [Local: Broker transport failure]
at com.vertica.util.ServerErrorData.buildException(Unknown Source)
                    . . .
```

Disabling the API version request when using the streaming job scheduler

To avoid these problems, you must disable the API version request feature in the rdkafka library that Vertica uses to communicate with Kafka. The easiest way to disable this setting when using the scheduler is to set a Linux environment variable on the host on which you run the vkconfig script. Using the environment variable ensures that each call to vkconfig includes the setting to disable the API version request. The vkconfig script checks two variables:

- VERTICA_RDKAFKA_CONF applies to all Kafka clusters that the scheduler running on the host communicates with.
- VERTICA_RDKAFKA_CONF_ *CLUSTER_NAME* applies to just the Kafka cluster named *CLUSTER_NAME* . Use this variable if your scheduler communicates with several Kafka clusters, some of which are not running version 0.9 or earlier.

If you set both variables, the cluster-specific one takes precedence. This feature is useful if most of the clusters your scheduler connects to are running Kafka 0.9 or earlier, but a few run 0.10 or later. This case, you can disable the API version check for most clusters using the VERTICA_RDKAFKA_CONF variable and re-enable the check for specific clusters using VERTICA_RDKAFKA_CONF_ *CLUSTER_NAME* .

If just a few of your Kafka clusters run version 0.9 or earlier, you can just set the cluster-specific variables for them, and leave the default values in place for the majority of your clusters.

The contents of the environment variable is a JSON object that tells the rdkafka library whether to query the Kafka cluster for the API version it supports:

```
'{"api.version.request":false}'
```

To set the environment variable in BASH, use the export command:

```
$ export VERTICA_RDKAFKA_CONF='{"api.version.request":false}'
```

If you wanted the setting to just affect a cluster named kafka_weblog, the command is:

```
$ export VERTICA_RDKAFKA_CONF_kafka_weblog='{"api.version.request":false}'
```

You can add the command to any of the common user environment configuration files such as ~/.bash_profile or the system-wide files such as /etc/profile . You must ensure that the same setting is used for all users who may run the vkconfig script, including any calls made by daemon processes such as init. You can also directly include this command in any script you use to set configure or start your scheduler.

Disabling the API version request when directly loading messages

You can disable the API request when you directly call KafkaSource to load messages by using the kafka_conf parameter:

```
=> CREATE FLEX TABLE iot_data();
CREATE TABLE
=> COPY public.iot_data SOURCE KafkaSource(stream='iot_json|0|-2',
                                brokers='kafka-01.example.com:9092',
                                stop_on_eof=True,
                                kafka_conf='{"api.version.request":false}')
      PARSER KafkaJSONParser();
 Rows Loaded
-------------
      5000
(1 row)
```

See also

- Directly setting Kafka library options
- Automatically consume data from Kafka with the scheduler
- Manually consume data from Kafka

Vertica Eon Mode and Kafka

You can use the Vertica integration with Apache Kafka when Vertica is running in Eon Mode . For Eon Mode running in a cloud environment, consider using a larger frame duration for schedulers in clusters. Cloud infrastructures lead to larger latencies overall, especially when storage layers are separated from compute layers. If you do not account for the added latency, you can experience a lower data throughput, as more of the frame's time is lost to overhead caused by the cloud environment. Using a longer frame length can also help prevent Vertica from creating many small ROS containers when loading data from Kafka. The trade off when you use a larger frame length is that your data load experiences more latency. See Choosing a frame duration for more information.

Consuming data from Kafka

A Kafka consumer subscribes to one or more topics managed by a Kafka cluster. Each topic is a data stream, an unbounded dataset that is represented as an ordered sequence of messages. Vertica can manually or automatically consume Kafka topics to perform analytics on your streaming data.

Manually consume data

Manually consume data from Kafka with a COPY statement that calls a KafkaSource function and parser. Manual loads are helpful when you want to:

- Populate a table one time with the messages currently in Kafka.
- Analyze a specific set of messages. You can choose the subset of data to load from the Kafka stream.

- Explore the data in a Kafka stream before you set up a scheduler to continuously stream the data into Vertica.
- Control the data load in ways not possible with the scheduler. For example, you cannot perform business logic or custom rejection handling during the data load from Kafka because the scheduler does not support additional processing during its transactions. Instead, you can periodically run a transaction that executes a COPY statement to load data from Kafka, and then perform additional processing.

For a detailed example, see Manually consume data from Kafka .

Automatically consume data

Automatically consume streaming data from Kafka into Vertica with a scheduler , a command-line tool that loads data as it arrives. The scheduler loads data in segments defined by a microbatch, a unit of work that processes the partitions of a single Kafka topic for a specified duration of time. You can manage scheduler configuration and options using the vkconfig tool .

For details, see Automatically consume data from Kafka with the scheduler .

Monitoring consumption

You must monitor message consumption to ensure that Kafka and Vertica are communicating effectively. You can use native Kafka tools to monitor consumer groups, or you can use vkconfig tool to view consumption details.

For additional information, see Monitoring message consumption .

Parsing data with Kafka filters

Your data stream might encode data that the Kafka parser functions cannot parse by default. Use Kafka filters to delimit messages in your stream to improve data consumption.

For details, see Parsing custom formats .

In this section

- Manually consume data from Kafka
- Automatically consume data from Kafka with the scheduler
- Monitoring message consumption
- Parsing custom formats

Manually consume data from Kafka

You can manually load streaming data from Kafka into Vertica using a COPY statement, just as you can load a finite set of data from a file or other source. Unlike a standard data source, Kafka data arrives continuously as a stream of messages that you must parse before loading into Vertica. Use Kafka functions in the COPY statement to prepare the data stream.

This example incrementally builds a COPY statement that manually loads JSON-encoded data from a Kafka topic named web_hits. The web_hits topic streams server logs of web site requests.

For information about loading data into Vertica, see Data load .

Creating the target table

To determine the target table schema, you must identify the message structure. The following is a sample of the web_hits stream:

```
{"url": "list.jsp", "ip": "144.177.38.106", "date": "2017/05/02 20:56:00",
"user-agent": "Mozilla/5.0 (compatible; MSIE 6.0; Windows NT 6.0; Trident/5.1)"}
{"url": "search/wp-content.html", "ip": "215.141.172.28", "date": "2017/05/02 20:56:01",
"user-agent": "Opera/9.53.(Windows NT 5.2; sl-SI) Presto/2.9.161 Version/10.00"}
```

This topic streams JSON-encoded data. Because JSON data is inconsistent and might contain unpredictable added values, store this data stream in a flex table. Flex tables dynamically accept additional fields that appear in the data.

The following statement creates a flex table named web_table to store the data stream:

```
=> CREATE FLEX TABLE web_table();
```

To begin the COPY statement, add the web_table as the target table:

```
COPY web_table
```

For more information about flex tables, see Flex tables .

> **Note**
> If you are copying data containing default values into a flex table, you must identify the default value column as __raw__ . For more information,

see Bulk Loading Data into Flex Tables .

Defining KafkaSource

The source of your COPY statement is always KafkaSource . KafkaSource accepts details about the data stream, Kafka brokers, and additional processing options to continuously load data until an end condition is met.

Stream details

The stream parameter defines the data segment that you want to load from one or more topic partitions. Each Kafka topic splits its messages into different partitions to get scalable throughput. Kafka keeps a backlog of messages for each topic according to rules set by the Kafka administrator. You can choose to load some or all of the messages in the backlog, or just load the currently streamed messages.

For each partition, the stream parameter requires the topic name, topic partition, and the partition offset as a list delimited by a pipe character (|). Optionally, you can provide and end offset as an end condition to stop loading from the data stream:

'stream=' *topic_name* | *partition* | *start_offset* [| *end_offset*]'

To load the entire backlog from a single partition of the web_hits topic, use the SOURCE keyword to append KafkaSource with the following stream parameter values:

```
COPY ...
SOURCE KafkaSource(stream='web_hits|0|-2', ...
```

In the previous example:

- web_hits is the name of the topic to load data from.
- 0 is the topic partition to load data from. Topic partitions are 0-indexed, and web_hits contains only one partition.
- -2 loads the entire backlog. This is a special offset value that tells KafkaSource to start loading at the earliest available message offset.

Loading multiple partitions

This example loads from only one partition, but it is important to understand how to load from multiple partitions in a single COPY statement.

To load from additional partitions in the same topic, or even additional topics, supply a comma-separated list of topic name, partition number, and offset values delimited by pipe characters. For example, the following stream argument loads the entire message backlog from partitions 0 through 2 of the web_hits topic:

```
KafkaSource(stream='web_hits|0|-2,web_hits|1|-2,web_hits|2|-2'...
```

Note
While you can load messages from different Kafka topics in the same COPY statement, you must ensure the data from the different topics is compatible with the target table's schema. The schema is less of a concern if you are loading data into a flex table, which can accommodate almost any data you want to load.

When you load multiple partitions in the same COPY statement, you can set the executionparallelism parameter to define the number of threads created for the COPY statement. Ideally, you want to use one thread per partition. You can choose to not specify a value and let Vertica determine the number of threads based on the number of partitions and the resources available in the resource pool. In this example, there is only one partition, so there's no need for additional threads to load data.

Note
The EXECUTIONPARALLELISM setting on the resource pool assigned is the upper limit on the number of threads your COPY statement can use. Setting executionparallelism on the KafkaSource function call to a value that is higher than that of the resource pool's EXECUTIONPARALLELISM setting does not increase the number of threads Vertica uses beyond the limits of the resource pool.

Adding the Kafka brokers

KafkaSource requires the host names (or IP addresses) and port numbers of the brokers in your Kafka cluster. The Kafka brokers are the service Vertica accesses in order to retrieve the Kafka data. In this example, the Kafka cluster has one broker named kafka01.example.com, running on port 9092. Append the brokers parameter and value to the COPY statement:

```
COPY ...
SOURCE KafkaSource(stream='web_hits|0|-2',
                   brokers='kafka01.example.com:9092', ...
```

Choosing the end condition

Because data continuously arrives from Kafka, manual loads from Kafka require that you define an end condition that indicates when to stop loading data. In addition to the end offset described in Stream Details , you can choose to:

- Copy as much data as possible for a set duration of time.
- Load data until no new data arrives within a timeout period.
- Load all available data, and not wait for any further data to arrive.

This example runs COPY for 10000 milliseconds (10 seconds) to get a sample of the data. If the COPY statement is able to load the entire backlog of data in under 10 seconds, it spends the remaining time loading streaming data as it arrives. This values is set in the duration parameter. Append the duration value to complete the KafkaSource definition:

```
COPY ...
SOURCE KafkaSource(stream='web_hits|0|-2',
                   brokers='kafka01.example.com:9092',
                   duration=interval '10000 milliseconds')
```

If you start a long-duration COPY statement from Kafka and need to stop it, you can call one of the functions that closes its session, such as CLOSE_ALL_SESSIONS .

Important
The duration that you set for your data load is not exact. The duration controls how long the KafkaSource process runs.

Selecting a parser

Kafka does not enforce message formatting on its data streams. Messages are often in Avro or JSON format, but they could be in any format. Your COPY statement usually uses one of three Kafka-specific parsers:

- KafkaParser
- KafkaJSONParser
- KafkaAvroParser

Because the Kafka parsers can recognize record boundaries in streaming data, the other parsers (such as the Flex parsers) are not directly compatible with the output of KafkaSource. You must alter the KafkaSource output using filters before other parsers can process the data. See Parsing custom formats for more information.

In this example, the data in the web_hits is encoded in JSON format, so it uses the KafkaJSONParser. This value is set in the COPY statement's PARSER clause:

```
COPY ...
SOURCE ...
PARSER KafkaJSONParser()
```

Storing rejected data

Vertica saves raw Kafka messages that the parser cannot parse to a rejects table, along with information on why it was rejected. This table is created by the COPY statement. This example saves rejects to the table named web_hits_rejections. This value is set in the COPY statement's REJECTED DATA AS TABLE clause:

```
COPY ...
SOURCE ...
PARSER ...
REJECTED DATA AS TABLE public.web_hits_rejections;
```

Loading the data stream into Vertica

The following steps load JSON data from the web_hits topic for 10 seconds using the COPY statement that was incrementally built in the previous sections:

1. Execute the COPY statement:

```
=> COPY web_table
  SOURCE KafkaSource(stream='web_hits|0|-2',
                     brokers='kafka01.example.com:9092',
                     duration=interval '10000 milliseconds')
  PARSER KafkaJSONParser()
  REJECTED DATA AS TABLE public.web_hits_rejections;
 Rows Loaded
-------------
         800
(1 row)
```

2. Compute the flex table keys:

```
=> SELECT compute_flextable_keys('web_table');
              compute_flextable_keys
-----------------------------------------------------
 Please see public.web_table_keys for updated keys
(1 row)
```

   For additional details, see Computing flex table keys .

3. Query web_table_keys to return the keys:

```
=> SELECT * FROM web_table_keys;
  key_name  | frequency | data_type_guess
------------+-----------+-----------------
 date       |       800 | Timestamp
 user_agent |       800 | Varchar(294)
 ip         |       800 | Varchar(30)
 url        |       800 | Varchar(88)
(4 rows)
```

4. Query web_table to return the data loaded from the web_hits Kafka topic:

```
=> SELECT date, url, ip FROM web_table LIMIT 10;
       date         |                url                 |       ip
--------------------+------------------------------------+-----------------
 2021-10-15 02:33:31 | search/index.htm                   | 192.168.210.61
 2021-10-17 16:58:27 | wp-content/about.html              | 10.59.149.131
 2021-10-05 09:10:06 | wp-content/posts/category/faq.html | 172.19.122.146
 2021-10-01 08:05:39 | blog/wp-content/home.jsp           | 192.168.136.207
 2021-10-10 07:28:39 | main/main.jsp                      | 172.18.192.9
 2021-10-22 12:41:33 | tags/categories/about.html         | 10.120.75.17
 2021-10-17 09:41:09 | explore/posts/main/faq.jsp         | 10.128.39.196
 2021-10-13 06:45:36 | category/list/home.jsp             | 192.168.90.200
 2021-10-27 11:03:50 | category/posts/posts/index.php     | 10.124.166.226
 2021-10-26 01:35:12 | categories/search/category.htm     | 192.168.76.40
(10 rows)
```

Automatically consume data from Kafka with the scheduler

Vertica offers a scheduler that loads streamed messages from one or more Kafka topics. Automatically loading streaming data has a number of advantages over manually using COPY:

- The streamed data automatically appears in your database. The frequency with which new data appears in your database is governed by the scheduler's frame duration.
- The scheduler provides an exactly-once consumption process. The schedulers manage offsets for you so that each message sent by Kafka is consumed once.
- You can configure backup schedulers to provide high-availability. Should the primary scheduler fail for some reason, the backup scheduler automatically takes over loading data.
- The scheduler manages resources for the data load. You control its resource usage through the settings on the resource pool you assign to it . When loading manually, you must take into account the resources your load consumes.

There are a few drawbacks to using a scheduler which may make it unsuitable for your needs. You may find that schedulers do not offer the flexibility you need for your load process. For example, schedulers cannot perform business logic during the load transaction. If you need to perform this sort of processing, you are better off creating your own load process. This process would periodically run COPY statements to load data from Kafka. Then it

would perform the business logic processing you need before committing the transaction.

For information on job scheduler requirements, refer to Apache Kafka integrations .

What the job scheduler does

The scheduler is responsible for scheduling loads of data from Kafka. The scheduler's basic unit of processing is a frame, which is a period of time. Within each frame, the scheduler assigns a slice of time for each active microbatch to run. Each microbatch is responsible for loading data from a single source. Once the frame ends, the scheduler starts the next frame. The scheduler continues this process until you stop it.

The anatomy of a scheduler

Each scheduler has several groups of settings, each of which control an aspect of the data load. These groups are:

- The scheduler itself, which defines the configuration schema, frame duration, and resource pool.
- Clusters, which define the hosts in the Kafka cluster that the scheduler contacts to load data. Each scheduler can contain multiple clusters, allowing you to load data from multiple Kafka clusters with a single scheduler.
- Sources, which define the Kafka topics and partitions in those topics to read data from.
- Targets, which define the tables in Vertica that will receive the data. These tables can be traditional Vertica database tables, or they can be flex tables.
- Load specs, which define setting Vertica uses while loading the data. These settings include the parsers and filters Vertica needs to use to load the data. For example, if you are reading a Kafka topic that is in Avro format, your load spec needs to specify the Avro parser and schema.
- Microbatches, which represent an individual segment of a data load from a Kafka stream. They combine the definitions for your cluster, source, target, and load spec that you create using the other vkconfig tools. The scheduler uses all of the information in the microbatch to execute COPY statements using the KafkaSource UDL function to transfer data from Kafka to Vertica. The statistics on each microbatch's load is stored in the stream_microbatch_history table.

The vkconfig script

You use a Linux command-line script named vkconfig to create, configure, and run schedulers. This script is installed on your Vertica hosts along with the Vertica server in the following path:

```
/opt/vertica/packages/kafka/bin/vkconfig
```

> **Note**
> You can install and use the vkconfig utility on a non-Vertica host. You may want to do this if:
>
> - You do not want the scheduler to use Vertica host resources.
> - You want users who do not have shell accounts on the Vertica hosts to be able to set up and alter schedulers.
>
> The easiest way to install vkconfig on a host is to install the Vertica server RPM. You must use the RPM that matches the version of Vertica installed on your database cluster. Do not create a database after installing the RPM. The vkconfig utility and its associated files will be in the /opt/vertica/packages/kafka/bin directory on the host.

The vkconfig script contains multiple tools. The first argument to the vkconfig script is always the tool you want to use. Each tool performs one function, such as changing one group of settings (such as clusters or sources) or starting and stopping the scheduler. For example, to create or configure a scheduler, you use the command:

```
$ /opt/vertica/packages/kafka/bin/vkconfig scheduler other options...
```

What happens when you create a scheduler

When you create a new scheduler, the vkconfig script takes the following steps:

- Creates a new Vertica schema using the name you specified for the scheduler. You use this name to identify the scheduler during configuration.
- Creates the tables needed to manage the Kafka data load in the newly-created schema. See Data streaming schema tables for more information.

Validating schedulers

When you create or configure a scheduler, it validates the following settings:

- Confirms that all brokers in the specified cluster exist.
- Connects to the specified host or hosts and retrieves the list of all brokers in the Kafka cluster. Getting this list always ensures that the scheduler has an up-to-date list of all the brokers. If the host is part of a cluster that has already been defined, the scheduler cancels the configuration.
- Confirms that the specified source exists. If the source no longer exists, the source is disabled.
- Retrieves the number of partitions in the source. If the number of partitions retrieved from the source is different than the partitions value saved by the scheduler, Vertica updates the scheduler with the number of partitions retrieved from the source in the cluster.

You can disable validation using the --validation-type option in the vkconfig script's scheduler tool. See Scheduler tool options for more information.

Synchronizing schedulers

By default, the scheduler automatically synchronizes its configuration and source information with Kafka host clusters. You can configure the synchronization interval using the --config-refresh scheduler utility option. Each interval, the scheduler:

- Checks for updates to the scheduler's configuration by querying its settings in its Vertica configuration schema.
- Performs all of the checks listed in Validating Schedulers .

You can configure synchronization settings using the --auto-sync option using the vkconfig script's scheduler tool. Scheduler tool options for details.

Launching a scheduler

You use the vkconfig script's launch tool to launch a scheduler.

When you launch a scheduler, it collects data from your sources, starting at the specified offset. You can view the stream_microbatch_history table to see what the scheduler is doing at any given time.

To learn how to create, configure, and launch a scheduler, see Setting up a scheduler in this guide.

You can also choose to bypass the scheduler. For example, you might want to do a single load with a specific range of offsets. For more information, see Manually consume data from Kafka in this guide.

If the Vertica cluster goes down, the scheduler attempts to reconnect and fails. You must relaunch the scheduler when the cluster is restarted.

Managing a running scheduler

When you launch a scheduler from the command line, it runs in the foreground. It will run until you kill it (or the host shuts down). Usually, you want to start the scheduler as a daemon process that starts it when the host operating system starts, or after the Vertica database has started.

You shut down a running scheduler using the vkconfig script's shutdown tool. See Shutdown tool options for details.

You can change most of a scheduler's settings (adding or altering clusters, sources, targets, and microbatches for example) while it is running. The scheduler automatically acts on the configuration updates.

Launching multiple job schedulers for high availability

For high availability, you can launch two or more identical schedulers that target the same configuration schema. You differentiate the schedulers using the launch tool's --instance-name option (see Launch tool options).

The active scheduler loads data and maintains an S lock on the stream_lock table. The scheduler not in use remains in stand-by mode until the active scheduler fails or is disabled. If the active scheduler fails, the backup scheduler immediately obtains the lock on the stream_lock table, and takes over loading data from Kafka where the failed scheduler left off.

Managing messages rejected during automatic loads

Vertica rejects messages during automatic loads using the parser definition, which is required in the microbatch load spec .

The scheduler creates a rejection table to store rejected messages for each microbatch automatically. To manually specify a rejection table, use the --rejection-schema and --rejection-table microbatch utility options when creating the microbatch. Query the stream_microbatches table to return the rejection schema and table for a microbatch.

For additional details about how Vertica handles rejected data, see Handling messy data .

Passing options to the scheduler's JVM

The scheduler uses a Java Virtual Machine to connect to Vertica via JDBC. You can pass command-line options to the JVM through a Linux environment variable named VKCONFIG_JVM_OPTS. This option is useful when configuring a scheduler to use TLS/SSL encryption when connecting to Vertica. See Configuring your scheduler for TLS connections for more information.

Viewing schedulers from the MC

You can view the status of Kafka jobs from the MC. For more information, refer to Viewing load history .

In this section

- Setting up a scheduler
- Choosing a frame duration
- Managing scheduler resources and performance
- Using connection load balancing with the Kafka scheduler
- Limiting loads using offsets
- Updating schedulers after Vertica upgrades

Setting up a scheduler

You set up a scheduler using the Linux command line. Usually you perform the configuration on the host where you want your scheduler to run. It can be one of your Vertica hosts, or a separate host where you have installed the vkconfig utility (see The vkconfig Script for more information).

Follow these steps to set up and start a scheduler to stream data from Kafka to Vertica:

1. Create a Config File (Optional)
2. Add the Kafka Bin Directory to Your Path (Optional)
3. Create a Resource Pool for Your Scheduler
4. Create the Scheduler
5. Create a Cluster
6. Create a Data Table
7. Create a Source
8. Create a Target
9. Create a Load-Spec
10. Create a Microbatch
11. Launch the Scheduler

These steps are explained in the following sections. These sections will use the example of loading web log data (hits on a web site) from Kafka into a Vertica table.

Create a config file (optional)

Many of the arguments you supply to the vkconfig script while creating a scheduler do not change. For example, you often need to pass a username and password to Vertica to authorize the changes to be made in the database. Adding the username and password to each call to vkconfig is tedious and error-prone.

Instead, you can pass the vkconfig utility a configuration file using the --conf option that specifies these arguments for you. It can save you a lot of typing and frustration.

The config file is a text file with a *keyword = value* pair on each line. Each keyword is a vkconfig command-line option, such as the ones listed in Common vkconfig script options .

The following example shows a config file named weblog.conf that will be used to define a scheduler named weblog_sched. This config file is used throughout the rest of this example.

```
# The configuraton options for the weblog_sched scheduler.
username=dbadmin
password=mypassword
dbhost=vertica01.example.com
dbport=5433
config-schema=weblog_sched
```

Add the vkconfig directory to your path (optional)

The vkconfig script is located in the /opt/vertica/packages/kafka/bin directory. Typing this path for each call to vkconfig is tedious. You can add vkconfig to your search path for your current Linux session using the following command:

```
$ export PATH=/opt/vertica/packages/kafka/bin:$PATH
```

For the rest of your session, you are able to call vkconfig without specifying its entire path:

```
$ vkconfig
Invalid tool
Valid options are scheduler, cluster, source, target, load-spec, microbatch, sync, launch,
shutdown, help
```

If you want to make this setting permanent, add the export statement to your ~/.profile file. The rest of this example assumes that you have added this directory to your shell's search path.

Create a resource pool for your scheduler

Vertica recommends that you always create a resource pool specifically for each scheduler. Schedulers assume that they have exclusive use of their assigned resource pool. Using a separate pool for a scheduler lets you fine-tune its impact on your Vertica cluster's performance. You create resource pools with CREATE RESOURCE POOL .

The following resource pool settings play an important role when creating your scheduler's resource pool:

- PLANNEDCONCURRENCY determines the number of microbatches (COPY statements) that the scheduler sends to Vertica simultaneously.
- EXECUTIONPARALLELISM determines the maximum number of threads that each node creates to process a microbatch's partitions.
- QUEUETIMEOUT provides manual control over resource timings. Set this to 0 to allow the scheduler to manage timings.

See Managing scheduler resources and performance for detailed information about these settings and how to fine-tune a resource pool for your scheduler.

The following CREATE RESOURCE POOL statement creates a resource pool that loads 1 microbatch and processes 1 partition:

```
=> CREATE RESOURCE POOL weblog_pool
   MEMORYSIZE '10%'
   PLANNEDCONCURRENCY 1
   EXECUTIONPARALLELISM 1
   QUEUETIMEOUT 0;
```

If you do not create and assign a resource pool for your scheduler, it uses a portion of the GENERAL resource pool. Vertica recommends that you do not use the GENERAL pool for schedulers in production environments. This fallback to the GENERAL pool is intended as a convenience when you test your scheduler configuration. When you are ready to deploy your scheduler, create a resource pool that you tuned to its specific needs. Each time that you start a scheduler that is using the GENERAL pool, the vkconfig utility displays a warning message.

Not allocating enough resources to your schedulers can result in errors. For example, you might get OVERSHOT DEADLINE FOR FRAME errors if the scheduler is unable to load data from all topics in a data frame.

See Resource pool architecture for more information about resource pools.

Create the scheduler

Vertica includes a default scheduler named stream_config. You can use this scheduler or create a new scheduler using the vkconfig script's scheduler tool with the --create and --config-schema options:

```
$ vkconfig scheduler --create --config-schema scheduler_name --conf conf_file
```

The --create and --config-schema options are the only ones required to add a scheduler with default options. This command creates a new schema in Vertica that holds the scheduler's configuration. See What Happens When You Create a Scheduler for details on the creation of the scheduler's schema.

You can use additional configuration parameters to further customize your scheduler. See Scheduler tool options for more information.

The following example:

- Creates a scheduler named weblog_sched using the -- config-schema option.
- Grants privileges to configure and run the scheduler to the Vertica user named kafka_user with the --operator option. The dbadmin user must specify additional privileges separately.
- Specifies a frame duration of seven minutes with the --frame-duration option. For more information about picking a frame duration, see Choosing a frame duration .
- Sets the resource pool that the scheduler uses to the weblog_pool created earlier:

```
$ vkconfig scheduler --create --config-schema weblog_sched --operator kafka_user \
  --frame-duration '00:07:00' --resource-pool weblog_pool --conf weblog.conf
```

> **Note**
> Technically, the previous example doesn't need to supply the --config-schema argument because it is set in the weblog.conf file. It appears in this example for clarity. There's no harm in supplying it on the command line as well as in the configuration file, as long as the values match. If they do not match, the value given on the command line takes priority.

Create a cluster

You must associate at least one Kafka cluster with your scheduler. Schedulers can access more than one Kafka cluster. To create a cluster, you supply a name for the cluster and host names and ports the Kafka cluster's brokers.

When you create a cluster, the scheduler attempts to validate it by connecting to the Kafka cluster. If it successfully connects, the scheduler automatically retrieves the list of all brokers in the cluster. Therefore, you do not have to list every single broker in the --hosts parameter.

The following example creates a cluster named kafka_weblog, with two Kafka broker hosts: kafka01 and kafka03 in the example.com domain. The Kafka brokers are running on port 9092.

```
$ vkconfig cluster --create --cluster kafka_weblog \
 --hosts kafka01.example.com:9092,kafka03.example.com:9092 --conf weblog.conf
```

See Cluster tool options for more information.

Create a source

Next, create at least one source for your scheduler to read. The source defines the Kafka topic the scheduler loads data from as well as the number of partitions the topic contains.

To create and associate a source with a configured scheduler, use the source tool. When you create a source, Vertica connects to the Kafka cluster to verify that the topic exists. So, before you create the source, make sure that the topic already exists in your Kafka cluster. Because Vertica verifies the existence of the topic, you must supply the previously-defined cluster name using the --cluster option.

The following example creates a source for the Kafka topic named web_hits on the cluster created in the previous step. This topic has a single partition.

```
$ vkconfig source --create --cluster kafka_weblog --source web_hits --partitions 1 --conf weblog.conf
```

Note

The --partitions parameter is the number of partitions to load, not a list of individual partitions. For example, if you set this parameter to 3, the scheduler will load data from partitions 0, 1, and 2.

See Source tool options for more information.

Create a data table

Before you can create a target for your scheduler, you must create a target table in your Vertica database. This is the table Vertica uses to store the data the scheduler loads from Kafka. You must decide which type of table to create for your target:

- A standard Vertica database table, which you create using the CREATE TABLE statement. This type of table stores data efficiently. However, you must ensure that its columns match the data format of the messages in Kafka topic you are loading. You cannot load complex types of data into a standard Vertica table.
- A flex table, which you create using CREATE FLEXIBLE TABLE . A flex table is less efficient than a standard Vertica database table. However, it is flexible enough to deal with data whose schema varies and changes. It also can load most complex data types that (such as maps and lists) that standard Vertica tables cannot.

Important

Avoid having columns with primary key restrictions in your target table. The scheduler stops loading data if it encounters a row that has a value which violates this restriction. If you must have a primary key restricted column, try to filter out any redundant values for that column in the streamed data before is it loaded by the scheduler.

The data in this example is in a set format, so the best table to use is a standard Vertica table. The following example creates a table named web_hits to hold four columns of data. This table is located in the public schema.

```
=> CREATE TABLE web_hits (ip VARCHAR(16), url VARCHAR(256), date DATETIME, user_agent VARCHAR(1024));
```

Note

You do not need to create a rejection table to store rejected messages. The scheduler creates the rejection table automatically.

Create a target

Once you have created your target table, you can create your scheduler's target. The target tells your scheduler where to store the data it retrieves from Kafka. This table must exist when you create your target. You use the vkconfig script's target tool with the --target-schema and --target_table options to specify the Vertica target table's schema and name. The following example adds a target for the table created in the previous step.

```
$ vkconfig target --create --target-schema public --target-table web_hits --conf weblog.conf
```

See Target tool options for more information.

Create a load spec

The scheduler's load spec provides parameters that Vertica uses when parsing the data loaded from Kafka. The most important option is --parser which sets the parser that Vertica uses to parse the data. You have three parser options:

- KafkaAvroParser for data in Avro format.
- KafkaJSONParser for data in JSON format.
- KafkaParser to load each message into a single VARCHAR field. See Parsing custom formats for more information.

In this example, the data being loaded from Kafka is in JSON format. The following command creates a load spec named weblog_load and sets the parser to KafkaJSONParser.

```
$ vkconfig load-spec --create --parser KafkaJSONParser --load-spec weblog_load --conf weblog.conf
```

See Load spec tool options for more information.

Create a microbatch

The microbatch combines all of the settings added to the scheduler so far to define the individual COPY statements that the scheduler uses to load data from Kafka.

The following example uses all of the settings created in the previous examples to create a microbatch called weblog.

```
$ vkconfig microbatch --create --microbatch weblog --target-schema public --target-table web_hits \
      --add-source web_hits --add-source-cluster kafka_weblog --load-spec weblog_load \
      --conf weblog.conf
```

For microbatches that might benefit from a reduced transaction size, consider using the --max-parallelism option when creating the microbatch. This option splits a single microbatch with multiple partitions into the specified number of simultaneous COPY statements consisting of fewer partitions.

See Microbatch tool options for more information about --max-parallelism and other options.

Launch the scheduler

Once you've created at least one microbatch, you can run your scheduler. You start your scheduler using the launch tool, passing it the name of the scheduler's schema. The scheduler begins scheduling microbatch loads for every enabled microbatch defined in its schema.

The following example launches the weblog scheduler defined in the previous steps. It uses the nohup command to prevent the scheduler being killed when the user logs out, and redirects stdout and stderr to prevent a nohup.out file from being created.

```
$ nohup vkconfig launch --conf weblog.conf >/dev/null 2>&1 &
```

Important

Vertica does not recommend specifying a password on the command line. Passwords on the command line can be exposed by the system's list of processes, which shows the command line for each process. Instead, put the password in a configuration file. Make sure the configuration file's permissions only allow it to be read by the user.

See Launch tool options for more information.

Checking that the scheduler is running

Once you have launched your scheduler, you can verify that it is running by querying the stream_microbatch_history table in the scheduler's schema. This table lists the results of each microbatch the scheduler has run.

For example, this query lists the microbatch name, the start and end times of the microbatch, the start and end offset of the batch, and why the batch ended. The results are ordered to start from when the scheduler was launched:

```
=> SELECT microbatch, batch_start, batch_end, start_offset,
        end_offset, end_reason
        FROM weblog_sched.stream_microbatch_history
        ORDER BY batch_start DESC LIMIT 10;

 microbatch |       batch_start          |         batch_end          | start_offset | end_offset |  end_reason
------------+----------------------------+----------------------------+--------------+------------+--------------
 weblog     | 2017-10-04 09:30:19.100752 | 2017-10-04 09:30:20.455739 |           -2 |      34931 | END_OF_STREAM
 weblog     | 2017-10-04 09:30:49.161756 | 2017-10-04 09:30:49.873389 |        34931 |      34955 | END_OF_STREAM
 weblog     | 2017-10-04 09:31:19.25731  | 2017-10-04 09:31:22.203173 |        34955 |      35274 | END_OF_STREAM
 weblog     | 2017-10-04 09:31:49.299119 | 2017-10-04 09:31:50.669889 |        35274 |      35555 | END_OF_STREAM
 weblog     | 2017-10-04 09:32:19.43153  | 2017-10-04 09:32:20.7519   |        35555 |      35852 | END_OF_STREAM
 weblog     | 2017-10-04 09:32:49.397684 | 2017-10-04 09:32:50.091675 |        35852 |      36142 | END_OF_STREAM
 weblog     | 2017-10-04 09:33:19.449274 | 2017-10-04 09:33:20.724478 |        36142 |      36444 | END_OF_STREAM
 weblog     | 2017-10-04 09:33:49.481563 | 2017-10-04 09:33:50.068068 |        36444 |      36734 | END_OF_STREAM
 weblog     | 2017-10-04 09:34:19.661624 | 2017-10-04 09:34:20.639078 |        36734 |      37036 | END_OF_STREAM
 weblog     | 2017-10-04 09:34:49.612355 | 2017-10-04 09:34:50.121824 |        37036 |      37327 | END_OF_STREAM
(10 rows)
```

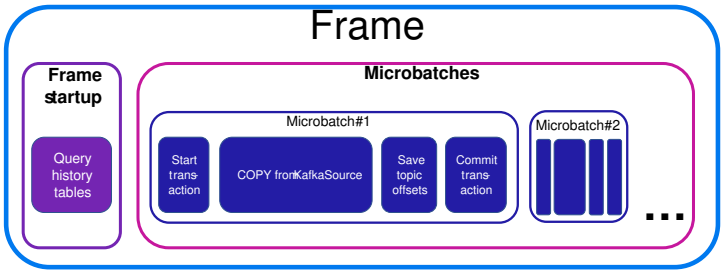
Choosing a frame duration

One key setting for your scheduler is its frame duration. The duration sets the amount of time the scheduler has to run all of the microbatches you have defined for it. This setting has significant impact on how data is loaded from Apache Kafka.

What happens during each frame

To understand the right frame duration, you first need to understand what happens during each frame.

The frame duration is split among the microbatches you add to your scheduler. In addition, there is some overhead in each frame that takes some time away from processing the microbatches. Within each microbatch, there is also some overhead which reduces the time the microbatch spends loading data from Kafka. The following diagram shows roughly how each frame is divided:



As you can see, only a portion of the time in the frame is spent actually loading the streaming data.

How the scheduler prioritizes microbatches

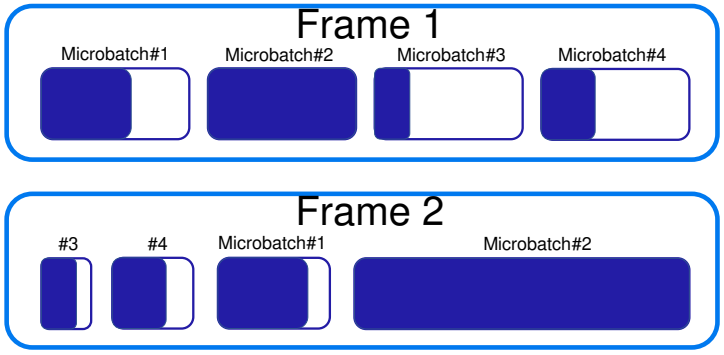
To start with, the scheduler evenly divides the time in the frame among the microbatches. It then runs each microbatch in turn.

In each microbatch, the bulk of the time is dedicated to loading data using a COPY statement. This statement loads data using the KafkaSource UDL. It runs until one of two conditions occurs:

- It reaches the ends of the data streams for the topics and partitions you defined for the microbatch. In this case, the microbatch completes processing early.
- It reaches a timeout set by the scheduler.

As the scheduler processes frames, it notes which microbatches finish early. It then schedules them to run first in the next frame. Arranging the microbatches in this manner lets the scheduler allocate more of the time in the frame to the microbatches that are spending the most time loading data (and perhaps have not had enough time to reach the end of their data streams).

For example, consider the following diagram. During the first frame, the scheduler evenly divides the time between the microbatches. Microbatch #2 uses all of the time allocated to it (as indicated by the filled-in area), while the other microbatches do not. In the next frame, the scheduler rearranges the microbatches so that microbatches that finished early go first. It also allocates less time to the microbatches that ran for a shorter period. Assuming these microbatches finish early again, the scheduler is able to give the rest of the time in the frame to microbatch #2. This shifting of priorities continues while the scheduler runs. If one topic sees a spike in traffic, the scheduler compensates by giving the microbatch reading from that topic more time.



What happens if the frame duration is too short

If you make the scheduler's frame duration too short, microbatches may not have enough time to load all of the data in the data streams they are responsible for reading. In the worst case, a microbatch could fall further behind when reading a high-volume topic during each frame. If left unaddressed, this issue could result in messages never being loaded, as they age out of the data stream before the microbatch has a chance to read them.

In extreme cases, the scheduler may not be able to run each microbatch during each frame. This problem can occur if the frame duration is so short that much of is spent in overhead tasks such committing data and preparing to run microbatches. The COPY statement that each microbatch runs to load data from Kafka has a minimum duration of 1 second. Add to this the overhead of processing data loads. In general, if the frame duration is shorter than 2 seconds times the number of microbatches in the scheduler, then some microbatches may not get a chance to run in each frame.

If the scheduler runs out of time during a frame to run each microbatch, it compensates during the next frame by giving priority to the microbatches that didn't run in the prior frame. This strategy makes sure each microbatch gets a chance to load data. However, it cannot address the root cause of the problem. Your best solution is to increase the frame duration to give each microbatch enough time to load data during each frame.

What happens if the frame duration is too long

One downside of a long frame duration is increased data latency. This latency is the time between when Kafka sends data out and when that data becomes available for queries in your database. A longer frame duration means that there is more time between each execution of a microbatch. That translates into more time between the data in your database being updated.

Depending on your application, this latency may not be important. When determining your frame duration, consider whether having the data potentially delayed up to the entire length of the frame duration will cause an issue.

Another issue to consider when using a long frame duration is the time it takes to shut down the scheduler. The scheduler does not shut down until the current COPY statement completes. Depending on the length of your frame duration, this process might take a few minutes.

The minimum frame duration

At a minimum, allocate two seconds for each microbatch you add to your scheduler. The vkconfig utility warns you if your frame duration is shorter than this lower limit. In most cases, you want your frame duration to be longer. Two seconds per microbatch leaves little time for data to actually load.

Balancing frame duration requirements

To determine the best frame duration for your deployment, consider how sensitive you are to data latency. If you are not performing time-sensitive queries against the data streaming in from Kafka, you can afford to have the default 5 minute or even longer frame duration. If you need a shorter data latency, then consider the volume of data being read from Kafka. High volumes of data, or data that has significant spikes in traffic can cause problems if you have a short frame duration.

Using different schedulers for different needs

Suppose you are loading streaming data from a few Kafka topics that you want to query with low latency and other topics that have a high volume but which you can afford more latency. Choosing a "middle of the road" frame duration in this situation may not meet either need. A better solution is to use multiple schedulers: create one scheduler with a shorter frame duration that reads just the topics that you need to query with low latency. Then create another scheduler that has a longer frame duration to load data from the high-volume topics.

For example, suppose you are loading streaming data from an Internet of Things (IOT) sensor network via Kafka into Vertica. You use the most of this data to periodically generate reports and update dashboard displays. Neither of these use cases are particularly time sensitive. However, three of the topics you are loading from do contain time-sensitive data (system failures, intrusion detection, and loss of connectivity) that must trigger immediate alerts.

In this case, you can create one scheduler with a frame duration of 5 minutes or more to read most of the topics that contain the non-critical data. Then create a second scheduler with a frame duration of at least 6 seconds (but preferably longer) that loads just the data from the three time-sensitive topics. The volume of data in these topics is hopefully low enough that having a short frame duration will not cause problems.

Managing scheduler resources and performance

Your scheduler's performance is impacted by the number of microbatches in your scheduler, partitions in each microbatch, and nodes in your Vertica cluster. Use resource pools to allocate a subset of system resources for your scheduler, and fine-tune those resources to optimize automatic loads into Vertica.

The following sections provide details about scheduler resource pool configurations and processing scenarios.

Schedulers and resource pools

Vertica recommends that you always create a resource pool specifically for each scheduler. Schedulers assume that they have exclusive use of their assigned resource pool. Using a separate pool for a scheduler lets you fine-tune its impact on your Vertica cluster's performance. You create resource pools with CREATE RESOURCE POOL .

If you do not create and assign a resource pool for your scheduler, it uses a portion of the GENERAL resource pool. Vertica recommends that you do not use the GENERAL pool for schedulers in production environments. This fallback to the GENERAL pool is intended as a convenience when you test your scheduler configuration. When you are ready to deploy your scheduler, create a resource pool that you tuned to its specific needs. Each time that you start a scheduler that is using the GENERAL pool, the vkconfig utility displays a warning message.

Not allocating enough resources to your schedulers can result in errors. For example, you might get OVERSHOT DEADLINE FOR FRAME errors if the scheduler is unable to load data from all topics in a data frame.

See Resource pool architecture for more information about resource pools.

Key resource pool settings

A microbatch is a unit of work that processes the partitions of a single Kafka topic within the duration of a frame . The following resource pool settings play an important role in how Vertica loads microbatches and processes partitions:

- PLANNEDCONCURRENCY determines the number of microbatches (COPY statements) the scheduler sends to Vertica simultaneously. At the start of each frame, the scheduler creates the number of scheduler threads specified by PLANNEDCONCURRENCY. Each scheduler thread connects to Vertica and loads one microbatch at a time. If there are more microbatches than scheduler threads, the scheduler queues the extra microbatches and loads them as threads become available.
- EXECUTIONPARALLELISM determines the maximum number of threads each node creates to process a microbatch's partitions. When a microbatch is loaded into Vertica, its partitions are distributed evenly among the nodes in the cluster. During each frame, a node creates a maximum of one thread for each partition. Each thread reads from one partition at a time until processing completes, or the frame ends. If there are more partitions than threads across all nodes, remaining partitions are processed as threads become available.
- QUEUETIMEOUT provides manual control over resource timings. Set the resource pool parameter QUEUETIMEOUT to 0 to allow the scheduler to manage timings. After all of the microbatches are processed, the scheduler waits for the remainder of the frame to process the next microbatch. A properly sized configuration includes rest time to plan for traffic surges. See Choosing a frame duration for information about the impacts of frame duration size.

For example, the following CREATE RESOURCE POOL statement creates a resource pool named weblogs_pool that loads 2 microbatches simultaneously. Each node in the Vertica cluster creates 10 threads per microbatch to process partitions:

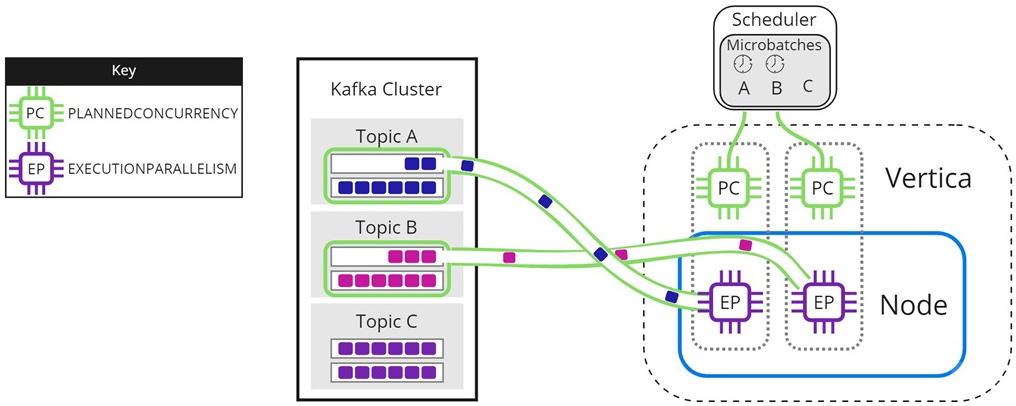
```
=> CREATE RESOURCE POOL weblogs_pool
   MEMORYSIZE '10%'
   PLANNEDCONCURRENCY 2
   EXECUTIONPARALLELISM 10
   QUEUETIMEOUT 0;
```

For a three-node Vertica cluster, weblogs_pool provides resources for each node to create up to 10 threads to process partitions, or 30 total threads per microbatch.

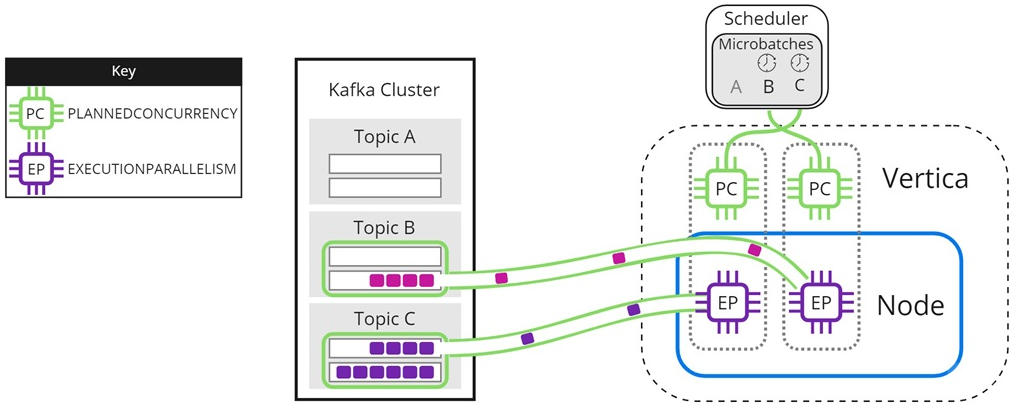
Loading multiple microbatches concurrently

In some circumstances, you might have more microbatches in your scheduler than available PLANNEDCONCURRENCY . The following images illustrate how the scheduler loads microbatches into a single Vertica node when there are not enough scheduler threads to load each microbatch simultaneously. The resource pool's PLANNEDCONCURRENCY (PC) is set to 2, but the scheduler must load three microbatches: A, B, and C. For simplicity, EXECUTIONPARALLELISM (EP) is set to 1.

To begin, the scheduler loads microbatch A and microbatch B while microbatch C waits:



When either microbatch finishes loading, the scheduler loads any remaining microbatches. In the following image, microbatch A is completely loaded into Vertica. The scheduler continues to load microbatch B, and uses the newly available scheduler thread to load microbatch C:

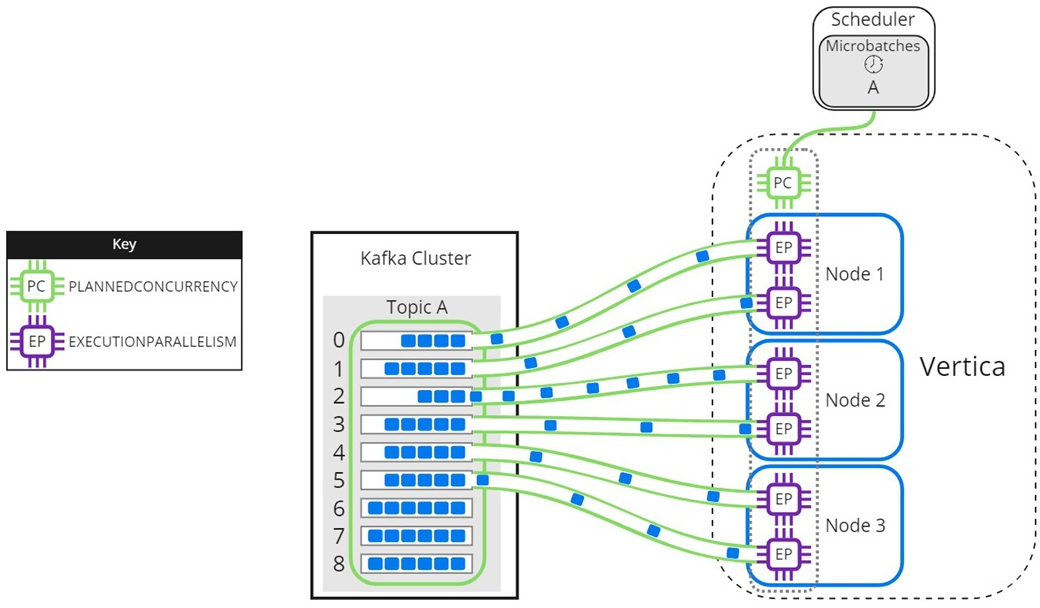


The scheduler continues sending data until all microbatches are loaded into Vertica, or the frame ends.

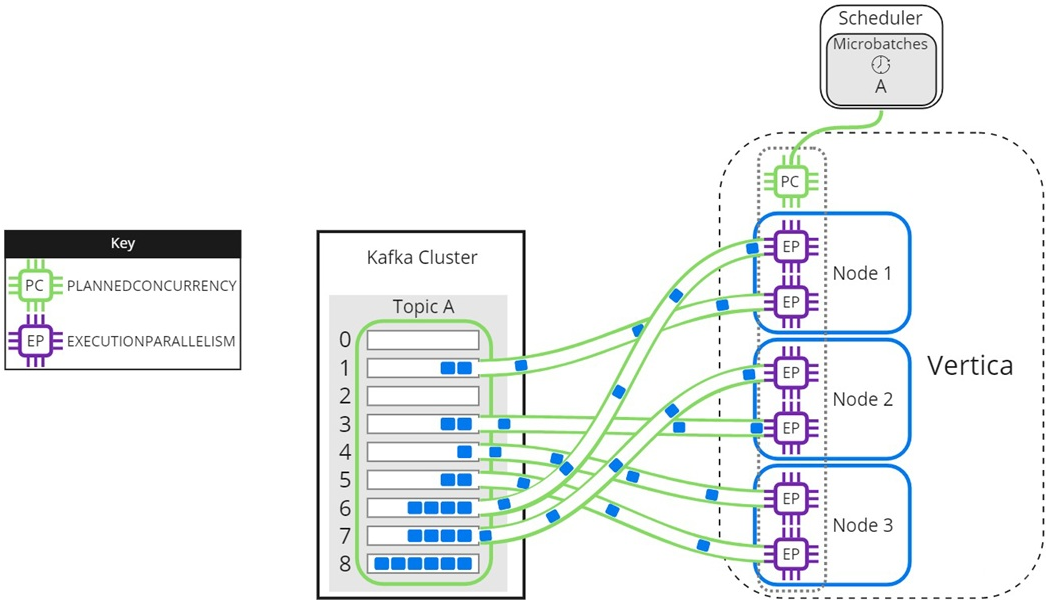
Experiment with PLANNEDCONCURRENCY to optimize performance. Note that setting it too high might create too many connections at the beginning of each frame, resulting in scalability stress on Vertica or Kafka. Setting PLANNEDCONCURRENCY too low does not take full advantage of the multiprocessing power of Vertica.

Parallel processing within Vertica

The resource pool setting EXECUTIONPARALLELISM limits the number of threads each Vertica node creates to process partitions. The following image illustrates how a three-node Vertica cluster processes a topic with nine partitions, when there is not enough EXECUTIONPARALLELISM to create one thread per partition. The partitions are distributed evenly among Node 1, Node 2, and Node 3 in the Vertica cluster. The scheduler's resource pool has PLANNEDCONCURRENCY (PC) set to 1 and EXECUTIONPARALLELISM (EP) set to 2, so each node creates a maximum of 2 threads when the scheduler loads microbatch A. Each thread reads from one partition at a time. Partitions that are not assigned a thread must wait for processing:



As threads finish processing their assigned partitions, the remaining partitions are distributed to threads as they become available:



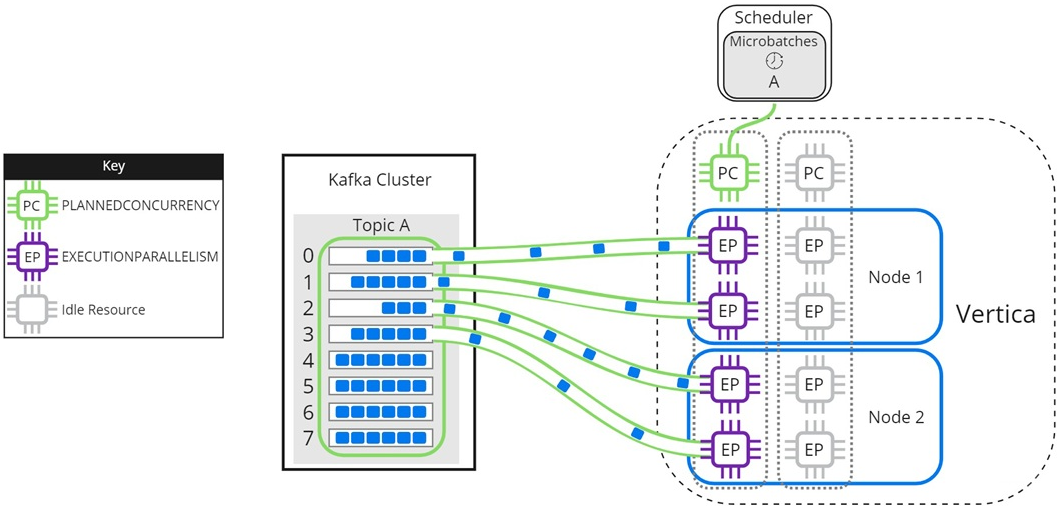
When setting the EXECUTIONPARALLELISM on your scheduler's resource pool, consider the number of partitions across all microbatches in the scheduler.

Loading partitioned topics concurrently

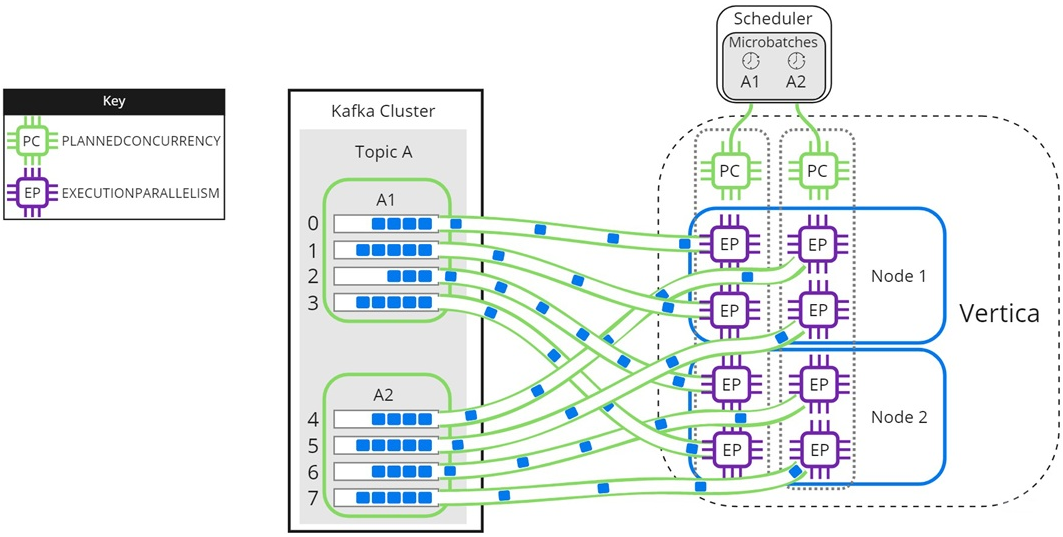
Single topics with multiple partitions might benefit from increased parallel loading or a reduced transaction size. The --max-parallelism microbatch utility option enables you to dynamically split a topic with multiple partitions into multiple, load-balanced microbatches that each consist of a subset of the original microbatch's partitions. The scheduler loads the dynamically split microbatches simultaneously using the PLANNEDCONCURRENCY available in its resource pool.

The EXECUTIONPARALLELISM setting in the scheduler's resource pool determines the maximum number of threads each node creates to process its portion of a single microbatch's partitions. Splitting a microbatch enables each node to create more threads for the same unit of work. When there is enough PLANNEDCONCURRENCY and the number of partitions assigned per node is greater than the EXECUTIONPARALLELISM setting in the scheduler's resource pool, use --max-parallelism to split the microbatch and create more threads per node to process more partitions in parallel.

The following image illustrates how a two-node Vertica cluster loads and processes microbatch A using a resource pool with PLANNEDCONCURRENCY (PC) set to 2, and EXECUTIONPARALLELISM (EP) set to 2. Because the scheduler is loading only one microbatch, there is 1 scheduler thread left unused. Each node creates 2 threads per scheduler thread to process its assigned partitions:



Setting microbatch A's --max-parallelism option to 2 enables the scheduler to dynamically split microbatch A into 2 smaller microbatches, A1 and A2. Because there are 2 available scheduler threads, the subset microbatches are loaded into Vertica simultaneously. Each node creates 2 threads per scheduler thread to process partitions for microbatches A1 and A2:



Use --max-parallelism to prevent bottlenecks in microbatches consisting of high-volume Kafka topics. It also provides faster loads for microbatches that require additional processing, such as text indexing.

Using connection load balancing with the Kafka scheduler

You supply the scheduler with the name of a Vertica node in the --dbhost option or the dbhost entry in your configuration file. The scheduler connects to this node to initiate all of the statements it executes to load data from Kafka. For example, each time it executes a microbatch, the scheduler connects to the same node to run the COPY statement. Having a single node act as the initiator node for all of the scheduler's actions can affect the performance of the node, and in turn the database as a whole.

To avoid a single node becoming a bottleneck, you can use connection load balancing to spread the load of running the scheduler's statements across multiple nodes in your database. Connection load balancing distributes client connections among the nodes in a load balancing group. See About native connection load balancing for an overview of this feature.

Enabling connection load balancing for a scheduler is a two-step process:

1. Choose or create a load balancing policy for your scheduler.
2. Enable load balancing in the scheduler.

Choosing or creating a load balancing policy for the scheduler

A connecting load balancing policy redirects incoming connections in from a specific set of network addresses to a group of nodes. If your database already defines a suitable load balancing policy, you can use it instead of creating one specifically for your scheduler.

If your database does not have a suitable policy, create one. Have your policy redirect connections coming from the IP addresses of hosts running Kafka schedulers to a group of nodes in your database. The group of nodes that you select will act as the initiators for the statements that the scheduler executes.

Important

In an Eon Mode database, only include nodes that are part of a primary subcluster in the scheduler's load balancing group. These nodes are the most efficient for loading data.

The following example demonstrates setting up a load balancing policy for all three nodes in a three-node database. The scheduler runs on node 1 in the database, so the source address range (192.168.110.0/24) of the routing rule covers the IP addresses of the nodes in the database. The last step of the example verifies that connections from the first node (IP address 10.20.110.21) are load balanced.

```
=> SELECT node_name,node_address,node_address_family FROM v_catalog.nodes;
    node_name     | node_address | node_address_family
------------------+--------------+---------------------
 v_vmart_node0001 | 10.20.110.21 | ipv4
 v_vmart_node0002 | 10.20.110.22 | ipv4
 v_vmart_node0003 | 10.20.110.23 | ipv4
(4 rows)


=> CREATE NETWORK ADDRESS node01 ON v_vmart_node0001 WITH '10.20.110.21';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node02 ON v_vmart_node0002 WITH '10.20.110.22';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node03 on v_vmart_node0003 WITH '10.20.110.23';
CREATE NETWORK ADDRESS
```

```
=> CREATE LOAD BALANCE GROUP kafka_scheduler_group WITH ADDRESS node01,node02,node03;
CREATE LOAD BALANCE GROUP
=> CREATE ROUTING RULE kafka_scheduler_rule ROUTE
   '10.20.110.0/24' TO kafka_scheduler_group;
CREATE ROUTING RULE
=> SELECT describe_load_balance_decision('10.20.110.21');
                    describe_load_balance_decision
-------------------------------------------------------------------------------
  Describing load balance decision for address [10.20.110.21]
Load balance cache internal version id (node-local): [2]
Considered rule [kafka_scheduler_rule] source ip filter [10.20.110.0/24]...
input address matches this rule
Matched to load balance group [kafka_scheduler_group] the group has policy [ROUNDROBIN]
number of addresses [3]
(0) LB Address: [10.20.110.21]:5433
(1) LB Address: [10.20.110.22]:5433
(2) LB Address: [10.20.110.23]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [10.20.110.23] port [5433]

(1 row)
```

Important

Be careful if you run your scheduler on a Vertica node and have either set its dbhost name to localhost or are not specifying a value (which means dbhost defaults to localhost). Connections to localhost use the loopback IP address 127.0.0.1 instead of the node's primary network address. If you create a load balancing routing rule that redirects incoming connections from the node's IP address range, it will not apply to connections made using localhost. The best solution is to use the node's IP address or FQDN as the dbhost setting.

If your scheduler connects to Vertica from an IP address that no routing rule applies to, you will see messages similar to the following in the vertica.log:

```
[Session] <INFO> Load balance request from client address ::1 had decision:
   Classic load balancing considered, but either the policy was NONE or no target was
   available. Details: [NONE or invalid]
 <LOG> @v_vmart_node0001: 00000/5789: Connection load balance request
   refused by server
```

Enabling load balancing in the scheduler

Clients must opt-in to load balancing for Vertica to apply the connection load balancing policies to the connection. For example, you must pass the -C flag to the vsql command for your interactive session to be load balanced.

The scheduler uses the Java JDBC library to connect to Vertica. To have the scheduler opt-in to load balancing, you must set the JDBC library's ConnectionLoadBalance option to 1. See Load balancing in JDBC for details.

Use the vkconfig script's --jdbc-opt option, or add the jdbc-opt option to your configuration file to set the ConnectionLoadBalance option. For example, to start the scheduler from the command line using a configuration file named weblog.conf, use the command:

```
$ nohup vkconfig launch --conf weblog.conf --jdbc-opt ConnectionLoadBalance=1 >/dev/null 2>&1 &
```

To permanently enable load balancing, you can add the load balancing option to your configuration file. The following example shows the weblog.conf file from the example in Setting up a scheduler configured to use connection load balancing.

```
username=dbadmin
password=mypassword
dbhost=10.20.110.21
dbport=5433
config-schema=weblog_sched
jdbc-opt=ConnectionLoadBalance=1
```

You can check whether the scheduler's connections are being load balanced by querying the SESSIONS table:

```
=> SELECT node_name, user_name, client_label FROM V_MONITOR.SESSIONS;
   node_name     | user_name |              client_label
-----------------+-----------+-----------------------------------------
 v_vmart_node0001 | dbadmin   | vkstream_weblog_sched_leader_persistent_4
 v_vmart_node0001 | dbadmin   |
 v_vmart_node0002 | dbadmin   | vkstream_weblog_sched_lane-worker-0_0_8
 v_vmart_node0003 | dbadmin   | vkstream_weblog_sched_VDBLogger_0_9
(4 rows)
```

In the client_labels column, the scheduler's connections have labels starting with vkstream (the row without a client label is an interactive session). You can see that the three connections the scheduler has opened all go to different nodes.

Limiting loads using offsets

Kafka maintains a user-configurable backlog of messages. By default, a newly-created scheduler reads all of the messages in a Kafka topic, including all of the messages in the backlog, not just the messages that are streamed out after the scheduler starts. Often, this is what you want.

In some cases, however, you may want to stream just a section of a source into a table. For example, suppose you want to analyze the web traffic of your e-commerce site starting at specific date and time. However, your Kafka topic contains web access records from much further back in time than you want to analyze. In this case, you can use an offset to stream just the data you want into Vertica for analysis.

Another common use case is when you have already loaded data some from Kafka manually (see Manually consume data from Kafka). Now you want to stream all of the newly-arriving data. By default, your scheduler ill reload all of the previously loaded data (assuming it is still available from Kafka). You can use an offset to tell your scheduler to start automatically loading data at the point where your manual data load left off.

Configuring a scheduler to start streaming from an offset

The vkconfig script's microbatch tool has an --offset option that lets you specify the index of the message in the source where you want the scheduler to begin loading. This option accepts a comma-separated list of index values. You must supply one index value for each partition in the source unless you use the --partition option. This option lets you choose the partitions the offsets apply to. The scheduler cannot be running when you set an offset in the microbatch.

If your microbatch defines more than one cluster, use the --cluster option to select which one the offset option applies to. Similarly, if your microbatch has more than one source, you must select one using the --source option.

For example, suppose you want to load just the last 1000 messages from a source named web_hits. To make things easy, suppose the source contains just a single partition, and the microbatch defines just a single cluster and single topic.

Your first task is to determine the current offset of the end of the stream. You can do this on one of the Kafka nodes by calling the GetOffsetShell class with the time parameter set to -1 (the end of the topic):

```
$ path to kafka/bin/kafka-run-class.sh kafka.tools.GetOffsetShell \
                                --broker-list kafka01:9092,kafka03:9092 --time -1 \
                                --topic web_hits

{metadata.broker.list=kafka01:9092,kafka03:9092, request.timeout.ms=1000,
 client.id=GetOffsetShell, security.protocol=PLAINTEXT}
web_hits:0:8932
```

You can also use GetOffsetShell to find the offset in the stream that occurs before a timestamp.

In the above example, the web_hits topic's single partition has an ending offset of 8932. If we want to load the last 1000 messages from the source, we need to set the microbatch's offset to 8932 - 1001 or 7931.

> **Note**
> The start of an offset is inclusive in the Vertica COPY statement. Kafka's native starting offset is exclusive. Therefore, you must add one to the offset to get the correct number of messages.

With the offset calculated, you are ready to set it in the microbatch's configuration. The following example:

- Shuts down the scheduler whose configuration information stored in the weblog.conf file.
- Sets the starting offset using the microbatch utility.
- Restarts the scheduler.

```
$ vkconfig shutdown --conf weblog.conf
$ vkconfig microbatch --microbatch weblog --update --conf weblog.conf --offset 7931
$ nohup vkconfig launch --conf weblog.conf >/dev/null 2>&1 &
```

If the target table was empty or truncated before the scheduler started, it will have 1000 rows in the table in it (until more messages are streamed through the source):

```
=> select count(*) from web_hits;
 count
-------
  1000
(1 row)
```

> **Note**
> The last example assumes that the offset values for the last 1000 messages in the Kafka topic were assigned consecutively. This assumption is not always be true. A Kafka topic can have gaps in its offset numbering for a variety of reasons. Offsets refer to the key value assigned to a message by Kafka, not its position in the topic.

Updating schedulers after Vertica upgrades

A scheduler is only compatible with the version of Vertica that created it. Between Vertica versions, the scheduler's configuration schema or the UDx function the scheduler calls may change. After you upgrade Vertica, you must update your schedulers to account for these changes.

When you upgrade Vertica to a new major version or service pack, use the vkconfig scheduler tool's --upgrade option to update your scheduler. If you do not update a scheduler, you receive an error message if you try to launch it. For example:

```
$ nohup vkconfig launch --conf weblog.conf >/dev/null 2>&1 &
com.vertica.solutions.kafka.exception.FatalException: Configured scheduler schema and current
scheduler configuration schema version do not match. Upgrade configuration by running:
vkconfig scheduler --upgrade
    at com.vertica.solutions.kafka.scheduler.StreamCoordinator.assertVersion(StreamCoordinator.java:64)
    at com.vertica.solutions.kafka.scheduler.StreamCoordinator.run(StreamCoordinator.java:125)
    at com.vertica.solutions.kafka.Launcher.run(Launcher.java:205)
    at com.vertica.solutions.kafka.Launcher.main(Launcher.java:258)
Scheduler instance failed. Check log file. Check log file.
$ vkconfig scheduler --upgrade --conf weblog.conf
Checking if UPGRADE necessary...
UPGRADE required, running UPGRADE...
UPGRADE completed successfully, now the scheduler configuration schema version is v8.1.1
$ nohup vkconfig launch --conf weblog.conf >/dev/null 2>&1 &
```

Monitoring message consumption

You can monitor the progress of your data streaming from Kafka several ways:

- Monitoring the consumer groups to which Vertica reports its progress. This technique is best if the tools you want to use to monitor your data load work with Kafka.
- Use the monitoring APIs built into the vkconfig tool. These APIs report the configuration and consumption of your streaming scheduler in JSON format. These APIs are useful if you are developing your own monitoring scripts, or your monitoring tools can consume status information in JSON format.

In this section

- Monitoring Vertica message consumption with consumer groups
- Getting configuration and statistics information from vkconfig

Monitoring Vertica message consumption with consumer groups

Apache Kafka has a feature named consumer groups that helps distribute message consumption loads across sets of consumers. When using consumer groups, Kafka evenly divides up messages based on the number of consumers in the group. Consumers report back to the Kafka broker which messages it read successfully. This reporting helps Kafka to manage message offsets in the topic's partitions, so that no consumer in the group is sent the same message twice.

Vertica does not rely on Kafka's consumer groups to manage load distribution or preventing duplicate loads of messages. The streaming job scheduler manages topic partition offsets on its own.

Even though Vertica does not need consumer groups to manage offsets, it does report back to the Kafka brokers which messages it consumed. This feature lets you use third-party tools to monitor the Vertica cluster's progress as it loads messages. By default, Vertica reports its progress to a consumer group named vertica- *databaseName* , where *databaseName* is the name of the Vertica database. You can change the name of the consumer group that Vertica reports its progress to when defining a scheduler or during manual loads of data. Third party tools can query the Kafka brokers to monitor the Vertica cluster's progress when loading data.

For example, you can use Kafka's kafka-consumer-groups.sh script (located in the bin directory of your Kafka installation) to view the status of the Vertica consumer group. The following example demonstrates listing the consumer groups available defined in the Kafka cluster and showing the details of the Vertica consumer group:

```
$ cd /path/to/kafka/bin
$ ./kafka-consumer-groups.sh --list --bootstrap-server localhost:9092
Note: This will not show information about old Zookeeper-based consumers.

vertica-vmart
$ ./kafka-consumer-groups.sh --describe --group vertica-vmart \
  --bootstrap-server localhost:9092
Note: This will not show information about old Zookeeper-based consumers.

Consumer group 'vertica-vmart' has no active members.

TOPIC                PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG        CONSUMER-ID                           HOST
CLIENT-ID
web_hits             0          24500           30000           5500       -                                     -               -
```

From the output, you can see that Vertica reports its consumption of messages back to the vertica-vmart consumer group. This group is the default consumer group when Vertica has the example VMart database loaded. The second command lists the topics being consumed by the vertica-vmart consumer group. You can see that the Vertica cluster has read 24500 of the 30000 messages in the topic's only partition. Later, running the same command will show the Vertica cluster's progress:

```
$ cd /path/to/kafka/bin
$ ./kafka-consumer-groups.sh --describe --group vertica-vmart \
   --bootstrap-server localhost:9092
Note: This will not show information about old Zookeeper-based consumers.

Consumer group 'vertica-vmart' has no active members.

TOPIC                PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG        CONSUMER-ID                           HOST
CLIENT-ID
web_hits             0          30000           30000           0          -
```

Changing the consumer group where Vertica reports its progress

You can change the consumer group that Vertica reports its progress to when consuming messages.

Changing for automatic loads with the scheduler

When using a scheduler, you set the consumer group by setting the --consumer-group-id argument to the vkconfig script's scheduler or microbatch utilities. For example, suppose you want the example scheduler shown in Setting up a scheduler to report its consumption to the consumer group name vertica-database. Then you could use the command:

```
$ /opt/vertica/packages/kafka/bin/vkconfig microbatch --update \
   --conf weblog.conf --microbatch weblog --consumer-group-id vertica-database
```

When the scheduler begins loading data, it will start updating the new consumer group. You can see this on a Kafka node using kafka-consumer-groups.sh .

Use the --list option to return the consumer groups:

```
$ /path/to/kafka/bin/kafka-consumer-groups.sh --list --bootstrap-server localhost:9092
Note: This will not show information about old Zookeeper-based consumers.

vertica-database
vertica-vmart
```

Use the --describe and --group options to return details about a specific consumer group:

```
$ /path/to/kafka/bin/kafka-consumer-groups.sh --describe --group vertica-database \
                                --bootstrap-server localhost:9092
Note: This will not show information about old Zookeeper-based consumers.

Consumer group 'vertica-database' has no active members.

TOPIC                  PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG        CONSUMER-ID                                HOST
CLIENT-ID
web_hits               0          30300           30300           0          -                                          -                    -
```

Changing for manual loads

To change the consumer group when manually loading data, use the group_id parameter of KafkaSource function:

```
=> COPY web_hits SOURCE KafkaSource(stream='web_hits|0|-2',
                                    brokers='kafka01.example.com:9092',
                                    stop_on_eof=True,
                                    group_id='vertica_database')
                PARSER KafkaJSONParser();
 Rows Loaded
-------------
       50000
(1 row)
```

Using consumer group offsets when loading messages

You can choose to have your scheduler, manual load, or custom loading script start loading messages from the consumer group's offset. To load messages from the last offset stored in the consumer group, use the special -3 offset.

Automatic load with the scheduler example

To instruct your scheduler to load messages from the consumer group's saved offset, use the vkconfig script microbatch tool's --offset argument.

1. Stop the scheduler using the shutdown command and the configuration file that you used to create the scheduler:
   ```
   $ /opt/vertica/packages/kafka/bin/vkconfig microbatch shutdown --conf weblog.conf
   ```
2. Set the microbatch --offset option to -3:
   ```
   $ /opt/vertica/packages/kafka/bin/vkconfig microbatch --update --conf weblog.conf --microbatch weblog --offset -3
   ```

This sets the offset to -3 for all topic partitions that your scheduler reads from. The scheduler begins the next load with the consumer group's saved offset, and all subsequent loads use the offset saved in stream_microbatch_history .

Manual load example

This example loads messages from the web_hits topic that has one partition consisting of 51,000 messages. For details about manual loads with KafkaSource , see Manually consume data from Kafka .

1. The first COPY statement creates a consumer group named vertica_manual, and loads the first 50,000 messages from the first partition in the web_hits topic:

```
=> COPY web_hits
  SOURCE KafkaSource(stream='web_hits|0|0|50000',
                     brokers='kafka01.example.com:9092',
                     stop_on_eof=True,
                     group_id='vertica_manual')
  PARSER KafkaJSONParser()
  REJECTED DATA AS TABLE public.web_hits_rejections;
 Rows Loaded
-------------
       50000
(1 row)
```

2. The next COPY statement passes -3 as the start_offset stream parameter to load from the consumer group's saved offset:

```
=> COPY web_hits
  SOURCE KafkaSource(stream='web_hits|0|-3',
                     brokers='kafka01.example.com:9092',
                     stop_on_eof=True,
                     group_id='vertica_manual')
  PARSER KafkaJSONParser()
  REJECTED DATA AS TABLE public.web_hits_rejections;
 Rows Loaded
-------------
        1000
(1 row)
```

Disabling consumer group reporting

Vertica reports the offsets of the messages it consumes to Kafka by default. If you do not specifically configure a consumer group for Vertica, it still reports its offsets to a consumer group named vertica_ *database-name* (where *database-name* is the name of the database Vertica is currently running).

If you want to completely disable having Vertica report its consumption back to Kafka, you can set the consumer group to an empty string or NULL. For example:

```
=> COPY web_hits SOURCE KafkaSource(stream='web_hits|0|-2',
                     brokers='kafka01.example.com:9092',
                     stop_on_eof=True,
                     group_id=NULL)
            PARSER KafkaJsonParser();
 Rows Loaded
-------------
       60000
(1 row)
```

Getting configuration and statistics information from vkconfig

The vkconfig tool has two features that help you examine your scheduler's configuration and monitor your data load:

- The vkconfig tools that configure your scheduler (scheduler, cluster, source, target, load-spec, and microbatch) have a --read argument that has them output their current settings in the scheduler.
- The vkconfig statistics tool lets you get statistics on your microbatches. You can filter the microbatch records based on a date and time range, cluster, partition, and other criteria.

Both of these features output their data in JSON format. You can use third-party tools that can consume JSON data or write your own scripts to process the configuration and statics data.

You can also access the data provided by these vkconfig options by querying the configuration tables in the scheduler's schema. However, you may find these options easier to use as they do not require you to connect to the Vertica database.

Getting configuration information

You pass the --read option to vkconfig's configuration tools to get the current settings for the options that the tool can set. This output is in JSON format. This example demonstrates getting the configuration information from the scheduler and cluster tools for the scheduler defined in the weblog.conf configuration file:

```
$ vkconfig scheduler --read --conf weblog.conf
{"version":"v9.2.0", "frame_duration":"00:00:10", "resource_pool":"weblog_pool",
 "config_refresh":"00:05:00", "new_source_policy":"FAIR",
 "pushback_policy":"LINEAR", "pushback_max_count":5, "auto_sync":true,
 "consumer_group_id":null}

$ vkconfig cluster --read --conf weblog.conf
{"cluster":"kafka_weblog", "hosts":"kafak01.example.com:9092,kafka02.example.com:9092"}
```

The --read option lists all of values created by the tool in the scheduler schema. For example, if you have defined multiple targets in your scheduler, the --read option lists all of them.

```
$ vkconfig target --list --conf weblog.conf
{"target_schema":"public", "target_table":"health_data"}
{"target_schema":"public", "target_table":"iot_data"}
{"target_schema":"public", "target_table":"web_hits"}
```

You can filter the --read option output using the other arguments that the vkconfig tools accept. For example, in the cluster tool, you can use the --host argument to limit the output to just show clusters that contain a specific host. These arguments support LIKE-predicate wildcards, so you can match partial values. See LIKE for more information about using wildcards.

The following example demonstrates how you can filter the output of the --read option of the cluster tool using the --host argument. The first call shows the unfiltered output. The second call filters the output to show only those clusters that start with "kafka":

```
$ vkconfig cluster --read --conf weblog.conf
{"cluster":"some_cluster", "hosts":"host01.example.com"}
{"cluster":"iot_cluster",
 "hosts":"kafka-iot01.example.com:9092,kafka-iot02.example.com:9092"}
{"cluster":"weblog",
 "hosts":"web01.example.com.com:9092,web02.example.com:9092"}
{"cluster":"streamcluster1",
 "hosts":"kafka-a-01.example.com:9092,kafka-a-02.example.com:9092"}
{"cluster":"test_cluster",
 "hosts":"test01.example.com:9092,test02.example.com:9092"}

$ vkconfig cluster --read --conf weblog.conf --hosts kafka%
{"cluster":"iot_cluster",
 "hosts":"kafka-iot01.example.com:9092,kafka-iot02.example.com:9092"}
{"cluster":"streamcluster1",
 "hosts":"kafka-a-01.example.com:9092,kafka-a-02.example.com:9092"}
```

See the Cluster tool options , Load spec tool options , Microbatch tool options , Scheduler tool options , Target tool options , and Source tool options for more information.

Getting streaming data load statistics

The vkconfig script's statistics tool lets you view the history of your scheduler's microbatches. You can filter the results using any combination of the following criteria:

- The name of the microbatch
- The Kafka cluster that was the source of the data load
- The name of the topic
- The partition within the topic
- The Vertica schema and table targeted by the data load
- A date and time range
- The latest microbatches

See Statistics tool options for all of the options available in this tool.

This example gets the last two microbatches that the scheduler ran:

```
$ vkconfig statistics --last 2 --conf weblog.conf
{"microbatch":"weblog", "target_schema":"public", "target_table":"web_hits",
 "source_name":"web_hits", "source_cluster":"kafka_weblog", "source_partition":0,
 "start_offset":73300, "end_offset":73399, "end_reason":"END_OF_STREAM",
 "end_reason_message":null, "partition_bytes":19588, "partition_messages":100,
 "timeslice":"00:00:09.807000", "batch_start":"2018-11-02 13:22:07.825295",
 "batch_end":"2018-11-02 13:22:08.135299", "source_duration":"00:00:00.219619",
 "consecutive_error_count":null, "transaction_id":45035996273976123,
 "frame_start":"2018-11-02 13:22:07.601", "frame_end":null}
{"microbatch":"weblog", "target_schema":"public", "target_table":"web_hits",
 "source_name":"web_hits", "source_cluster":"kafka_weblog", "source_partition":0,
 "start_offset":73200, "end_offset":73299, "end_reason":"END_OF_STREAM",
 "end_reason_message":null, "partition_bytes":19781, "partition_messages":100,
 "timeslice":"00:00:09.561000", "batch_start":"2018-11-02 13:21:58.044698",
 "batch_end":"2018-11-02 13:21:58.335431", "source_duration":"00:00:00.214868",
 "consecutive_error_count":null, "transaction_id":45035996273976095,
 "frame_start":"2018-11-02 13:21:57.561", "frame_end":null}
```

This example gets the microbatches from the source named web_hits between 13:21:00 and 13:21:20 on November 2nd 2018:

```
$ vkconfig statistics --source "web_hits" --from-timestamp \
        "2018-11-02 13:21:00" --to-timestamp "2018-11-02 13:21:20"  \
        --conf weblog.conf
{"microbatch":"weblog", "target_schema":"public", "target_table":"web_hits",
 "source_name":"web_hits", "source_cluster":"kafka_weblog", "source_partition":0,
 "start_offset":72800, "end_offset":72899, "end_reason":"END_OF_STREAM",
 "end_reason_message":null, "partition_bytes":19989, "partition_messages":100,
 "timeslice":"00:00:09.778000", "batch_start":"2018-11-02 13:21:17.581606",
 "batch_end":"2018-11-02 13:21:18.850705", "source_duration":"00:00:01.215751",
 "consecutive_error_count":null, "transaction_id":45035996273975997,
 "frame_start":"2018-11-02 13:21:17.34", "frame_end":null}
{"microbatch":"weblog", "target_schema":"public", "target_table":"web_hits",
 "source_name":"web_hits", "source_cluster":"kafka_weblog", "source_partition":0,
 "start_offset":72700, "end_offset":72799, "end_reason":"END_OF_STREAM",
 "end_reason_message":null, "partition_bytes":19640, "partition_messages":100,
 "timeslice":"00:00:09.857000", "batch_start":"2018-11-02 13:21:07.470834",
 "batch_end":"2018-11-02 13:21:08.737255", "source_duration":"00:00:01.218932",
 "consecutive_error_count":null, "transaction_id":45035996273975978,
 "frame_start":"2018-11-02 13:21:07.309", "frame_end":null}
```

See Statistics tool options for more examples of using this tool.

Parsing custom formats

To process a Kafka data stream, the parser must identify the boundary between each message. Vertica provides Kafka parsers that can identify boundaries for Avro , JSON , and raw data formats, but your data stream might use a custom format. To parse custom formats, Vertica provides filters that insert boundary information in the data stream before it reaches the parser.

Kafka filters

Vertica provides the following filters:

- KafkaInsertDelimiters: Inserts a user-specified delimiter between each message in the data stream. The delimiter can contain any characters and be of any length. This parser uses the following syntax:
  KafkaInsertDelimiters(delimiter = ' *delimiter* ')
- KafkaInsertLengths: Inserts the message length in bytes at the beginning of the message. Vertica writes the length as a 4-byte uint32 value in big-endian network byte order. For example, a 100-byte message is preceded by 0x00000064. This parser uses the following syntax:
  KafkaInsertLengths()

> **Note**
> Because each Kafka filter requires input from a stream source and outputs a non-stream source, you cannot use both Kafka filters in the same COPY statement to process a Kafka data stream.

In addition to one of the Kafka filters, you can include one or more user-defined filters in a single COPY statement. Specify multiple filters as a comma-separated list, and list the Vertica filter first. If you use a non-Kafka parser, you must use at least one filter to prepare the data stream for the parser, or the parser fails and returns an error.

Examples

The following COPY statement loads comma-separated values from two partitions in a topic named iot-data . The load exits after it processes all messages in both partitions. The KafkaInsertDelimiters filter inserts newlines between the Kafka messages to convert them into traditional rows of data. The statement uses the standard COPY parser to delimit CSV values with a comma:

```
=> COPY kafka_iot SOURCE KafkaSource(stream='iot-data|0|-2,iot-data|1|-2',
                                     brokers='kafka01:9092',
                                     stop_on_eof=True)
              FILTER KafkaInsertDelimiters(delimiter = E'\n')
              DELIMITER ',';
 Rows Loaded
-------------
        3430
(1 row)
```

Avro Schema Registry

Vertica supports the use of a Confluent schema registry for Avro schemas with the KafkaAvroParser . By using a schema registry, you enable the Avro parser to parse and decode messages written by the registry and to retrieve schemas stored in the registry. In addition, a schema registry enables Vertica to process streamed data without sending a copy of the schema with each record. Vertica can access a schema registry in the following ways:

- schema ID
- subject and version

> **Note**
> If you use the compatibility config resource in your schema registry, you should specify a value of at least BACKWARD. You may also choose to use a stricter compatibility setting. For more information on installing and configuring a schema registry, refer to the Confluent documentation .

Schema ID Loading

In schema ID based loading, the Avro parser checks the schema ID associated with each message to identify the correct schema to use. A single COPY statement can reference multiple schemas. Because each message is not validated, Vertica recommends that you use a flex table as the target table for schema ID based loading.

The following example shows a COPY statement that refers to a schema registry located on the same host:

```
=> COPY logs source kafkasource(stream='simple|0|0', stop_on_eof=true,
duration=interval '10 seconds') parser
KafkaAvroParser(schema_registry_url='http://localhost:8081/');
```

Subject and Version Loading

In subject and version loading, you specify a subject and version in addition to the schema registry URL. The addition of the subject and version identifies a single schema to use for all messages in the COPY. If any message in the statement is incompatible with the schema, the COPY fails. Because all messages are validated prior to loading, Vertica recommends that you use a standard Vertica table as the target for subject and version loading.

The following example shows a COPY statement that identifies a schema subject and schema version as well as a schema registry.

```
=> COPY t source kafkasource(stream='simpleEvolution|0|0',
stop_on_eof=true, duration=interval '10 seconds') parser
KafkaAvroParser(schema_registry_url='http://repository:8081/schema-repo/',
schema_registry_subject='simpleEvolution-value',schema_registry_version='1')
REJECTED DATA AS TABLE "t_rejects";
```

Producing data for Kafka

In addition to consuming data from Kafka, Vertica can produce data for Kafka. Stream the following data from Vertica for consumption by other Kafka consumers:

- Vertica anayltics results. Use KafkaExport to export Vertica tables and queries.
- Health and performance data from Data Collector tables. Create push-based notifiers to send this data for consumption for third-party monitoring tools.
- Ad hoc messages. Use NOTIFY to signal that tasks such as stored procedures are complete.

In this section

- Producing data using KafkaExport
- Producing Kafka messages using notifiers

Producing data using KafkaExport

The KafkaExport function lets you stream data from Vertica to Kafka. You pass this function three arguments and two or three parameters:

```
SELECT KafkaExport(partitionColumn, keyColumn, valueColumn
   USING PARAMETERS brokers='host[:port][,host...]',
   topic='topicname'
   [,kafka_conf='kafka_configuration_setting']
   [,fail_on_conf_parse_error=Boolean])
OVER (partition_clause) FROM table;
```

The *partitionColumn* and *keyColumn* arguments set the Kafka topic's partition and key value, respectively. You can set either or both of these values to NULL. If you set the partition to NULL, Kafka uses its default partitioning scheme (either randomly assigning partitions if the key value is NULL, or based on the key value if it is not).

The *valueColumn* argument is a LONG VARCHAR containing message data that you want to send to Kafka. Kafka does not impose structure on the message content. Your only restriction on the message format is what the consumers of the data are able to parse.

You are free to convert your data into a string in any way you like. For simple messages (such as a comma-separated list), you can use functions such as CONCAT to assemble your values into a message. If you need a more complex data format, such as JSON, consider writing a UDx function that accepts columns of data and outputs a LONG VARCHAR containing the data in the format you require. See Developing user-defined extensions (UDxs) for more information.

See KafkaExport for detailed information about KafkaExport's syntax.

Export example

This example shows you how to perform a simple export of several columns of a table. Suppose you have the following table containing a simple set of Internet of things (IOT) data:

```
=> SELECT * FROM iot_report LIMIT 10;
 server |        date         |       location        |    id
--------+---------------------+-----------------------+----------
      1 | 2016-10-11 04:09:28 | -14.86058, 112.75848  | 70982027
      1 | 2017-07-02 12:37:48 | -21.42197, -127.17672 | 49494918
      1 | 2017-10-19 14:04:33 | -71.72156, -36.27381  | 94328189
      1 | 2018-07-11 19:35:18 | -9.41315, 102.36866   | 48366610
      1 | 2018-08-30 08:09:45 | 83.97962, 162.83848   |   967212
      2 | 2017-01-20 03:05:24 | 37.17372, 136.14026   | 36670100
      2 | 2017-07-29 11:38:37 | -38.99517, 171.72671  | 52049116
      2 | 2018-04-19 13:06:58 | 69.28989, 133.98275   | 36059026
      2 | 2018-08-28 01:09:51 | -59.71784, -144.97142 | 77310139
      2 | 2018-09-14 23:16:15 | 58.07275, 111.07354   |  4198109
(10 rows)

=> \d iot_report
                                 List of Fields by Tables
 Schema |   Table    |  Column |     Type     | Size | Default | Not Null | Primary Key | Foreign Key
--------+------------+---------+--------------+------+---------+----------+-------------+-------------
 public | iot_report | server  | int          |    8 |         | f        | f           |
 public | iot_report | date    | timestamp    |    8 |         | f        | f           |
 public | iot_report | location | varchar(40) |   40 |         | f        | f           |
 public | iot_report | id      | int          |    8 |         | f        | f           |
(4 rows)
```

You want to send the data in this table to a Kafka topic named iot_results for consumption by other applications. Looking at the data and the structure of the iot_report, you may decide the following:

- The server column is a good match for the partitions in iot_report. There are three partitions in the Kafka topic, and the values in server column are between 1 and 3. Suppose the partition column had a larger range of values (for example, between 1 and 100). Then you could use the modulo operator (%) to coerce the values into the same range as the number of partitions (server % 3).
  A complication with these values is that the values in the server are 1-based (the lowest value in the column is 1). Kafka's partition numbering scheme is zero-based. So, you must adjust the values in the server column by subtracting 1 from them.
- The id column can act as the key. This column has a data type of INTEGER. The KafkaExport function expects the key value to be a VARCHAR. Vertica does not automatically cast INTEGER values to VARCHAR, so you must explicitly cast the value in your function call.
- The consumers of the iot_report topic expect values in comma-separated format. You can combine the values from the date and location columns into a single VARCHAR using nested calls to the CONCAT function.

The final piece of information you need to know is the host names and port numbers of the brokers in your Kafka cluster. In this example, there are two brokers named kafka01 and kafka03, running on port 6667 (the port that Hortonworks clusters use). Once you have all of this information, you are ready to export your data.

The following example shows how you might export the contents of iot_report:

```
=> SELECT KafkaExport(server - 1, id::VARCHAR,
   CONCAT(CONCAT(date, ', '), location)
   USING PARAMETERS brokers='kafka01:6667,kafka03:6667',
   topic='iot_results') OVER (PARTITION BEST) FROM iot_report;
 partition | key | message | failure_reason
-----------+-----+---------+----------------
(0 rows)
```

KafkaExport returned 0 rows which means Vertica was able to send all of your data to Kafka without any errors.

Other things to note about the example:

- The CONCAT function automatically converts the date column's DATETIME value to a VARCHAR for you, so you do not need to explicitly cast it.
- Two nested CONCAT functions are necessary to concatenate the date field with a comma, and the resulting string with the location field.
- Adding a third column to the message field would require *two* additional CONCAT function calls (one to concatenate a comma after the location column, and one to concatenate the additional column's value). Using CONCAT becomes messy after just a few column's worth of data.

On the Kafka side, you will see whatever you sent as the valueColumn (third) argument of the KafkaExport function. In the above example, this is a CSV list. If you started a console consumer for iot_results topic before running the example query, you would see the following output when the query runs:

```
$ /opt/kafka/bin/kafka-console-consumer.sh --topic iot_results --zookeeper localhost
2017-10-10 12:08:33, 78.84883, -137.56584
2017-12-06 16:50:57, -25.33024, -157.91389
2018-01-12 21:27:39, 82.34027, 116.66703
2018-08-19 00:02:18, 13.00436, 85.44815
2016-10-11 04:09:28, -14.86058, 112.75848
2017-07-02 12:37:48, -21.42197, -127.17672
2017-10-19 14:04:33, -71.72156, -36.27381
2018-07-11 19:35:18, -9.41315, 102.36866
2018-08-30 08:09:45, 83.97962, 162.83848
2017-01-20 03:05:24, 37.17372, 136.14026
2017-07-29 11:38:37, -38.99517, 171.72671
2018-04-19 13:06:58, 69.28989, 133.98275
2018-08-28 01:09:51, -59.71784, -144.97142
2018-09-14 23:16:15, 58.07275, 111.07354
```

KafkaExport's return value

KafkaExport outputs any rows that Kafka rejected. For example, suppose you forgot to adjust the partition column to be zero-based in the previous example. Then some of the rows exported to Kafka would specify a partition that does not exist. In this case, Kafka rejects these rows, and KafkaExport reports them in table format:

```
=> SELECT KafkaExport(server, id::VARCHAR,
   CONCAT(CONCAT(date, ', '), location)
   USING PARAMETERS brokers='kafka01:6667,kafka03:6667',
   topic='iot_results') OVER (PARTITION BEST) FROM iot_report;
 partition |   key    |                 message                  |      failure_reason
-----------+----------+------------------------------------------+--------------------------
         3 | 40492866 | 2017-10-10 12:08:33, 78.84883, -137.56584,  | Local: Unknown partition
         3 | 73846006 | 2017-12-06 16:50:57, -25.33024, -157.91389, | Local: Unknown partition
         3 | 45020829 | 2018-01-12 21:27:39, 82.34027, 116.66703,   | Local: Unknown partition
         3 | 27462612 | 2018-08-19 00:02:18, 13.00436, 85.44815,    | Local: Unknown partition
(4 rows)
```

> **Note**
> Another common reason for Kafka rejecting a row is that its message value is longer than Kafka's message.max.bytessetting.

You can capture this output by creating a table to hold the rejects. Then use an INSERT statement to insert KafkaExport's results:

```
=> CREATE TABLE export_rejects (partition INTEGER, key VARCHAR, message LONG VARCHAR, failure_reason VARCHAR);
CREATE TABLE
=> INSERT INTO export_rejects SELECT KafkaExport(server, id::VARCHAR,
   CONCAT(CONCAT(date, ', '), location)
   USING PARAMETERS brokers='kafka01:6667,kafka03:6667',
   topic='iot_results') OVER (PARTITION BEST) FROM iot_report;
 OUTPUT
--------
      4
(1 row)
=> SELECT * FROM export_rejects;
 partition |   key    |                 message                  |      failure_reason
-----------+----------+------------------------------------------+--------------------------
         3 | 27462612 | 2018-08-19 00:02:18, 13.00436, 85.44815    | Local: Unknown partition
         3 | 40492866 | 2017-10-10 12:08:33, 78.84883, -137.56584  | Local: Unknown partition
         3 | 73846006 | 2017-12-06 16:50:57, -25.33024, -157.91389 | Local: Unknown partition
         3 | 45020829 | 2018-01-12 21:27:39, 82.34027, 116.66703   | Local: Unknown partition
(4 rows)
```

Producing Kafka messages using notifiers

You can use notifiers to help you monitor your Vertica database using third-party Kafka-aware tools by producing messages to a Kafka topic. You can directly publish messages (for example, from a SQL script to indicate a long-running query has finished). Notifiers can also automatically send messages when a component in the data collector tables is updated.

In this section

- Creating a Kafka notifier
- Sending individual messages via a Kafka notifier
- Monitoring DC tables with Kafka notifiers

Creating a Kafka notifier

The following procedure creates a Kafka notifier. At a minimum, a notifier defines:

- A unique name.
- A message protocol. This is kafka:// when sending messages to Kafka.
- The server to communicate with. For Kafka, this is the address and port number of a Kafka broker.
- The maximum message buffer size. If the queue of messages to be sent via the notifier exceed this limit, messages are dropped.

You create the notifier with CREATE NOTIFIER . This example creates a notifier named load_progress_notifier that sends messages via the Kafka broker running on kafka01.example.com on port 9092:

```
=> CREATE NOTIFIER load_progress_notifier
   ACTION 'kafka://kafka01.example.com:9092'
   MAXMEMORYSIZE '10M';
```

While not required, it is best practice to create notifiers that use an encrypted connection. The following example creates a notifier that uses an encrypted connection and verifies the Kafka server's certificate with the provided TLS configurations :

```
=> CREATE NOTIFIER encrypted_notifier
   ACTION 'kafka://127.0.0.1:9092'
   MAXMEMORYSIZE '10M'
   TLS CONFIGURATION 'notifier_tls_config'
```

Follow this procedure to create or alter notifiers for Kafka endpoints that use SASL_SSL. Note that you must repeat this procedure whenever you change the TLSMODE, certificates, or CA bundle for a given notifier.

1. Create a TLS Configuration with the desired TLS mode, certificate, and CA certificates.
2. Use CREATE or ALTER to disable the notifier and set the TLS Configuration:
   ```
   => ALTER NOTIFIER encrypted_notifier
      DISABLE
      TLS CONFIGURATION kafka_tls_config;
   ```
3. ALTER the notifier and set the proper rdkafka adapter parameters for SASL_SSL:
   ```
   => ALTER NOTIFIER encrypted_notifier PARAMETERS
      'sasl.username=user;sasl.password=password;sasl.mechanism=PLAIN;security.protocol=SASL_SSL';
   ```
4. Enable the notifier:
   ```
   => ALTER NOTIFIER encrypted_notifier ENABLE;
   ```

Sending individual messages via a Kafka notifier

You can send an individual message via a Kafka notifier using the NOTIFY function. This feature is useful for reporting the progress of SQL scripts such as ETL tasks to third-party reporting tools.

You pass this function three string values:

- The message to send.
- The name of the notifier to send the message.
- The Kafka topic to receive the message.

For example, suppose you want to send the message "Daily load finished" to the vertica_notifications topic of the Kafka cluster defined in the load_progress_notifier notifier created earlier. Then you could execute the following statement:

```
=> SELECT NOTIFY('Daily load finished.',
                 'load_progress_notifier',
                 'vertica_notifications');
 NOTIFY
--------
 OK
(1 row)
```

The message the notifier sends to Kafka is in JSON format. You can see the resulting message by using the console consumer on a Kafka node. For example:

```
$ /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
                                           --from-beginning \
                                           --topic vertica_notifications \
                                           --max-messages 1

{"_db":"vmart","_schema":"v_internal","_table":"dc_notifications",
"channel":"vertica_notifications","message":"Daily load finished.",
"node_name":"v_vmart_node0001","notifier":"load_progress_notifier",
"request_id":2,"session_id":"v_vmart_node0001-463079:0x4ba6f",
"statement_id":-1,"time":"2018-06-19 09:48:42.314181-04",
"transaction_id":45035996275565458,"user_id":45035996273704962,
"user_name":"dbadmin"}

Processed a total of 1 messages
```

Monitoring DC tables with Kafka notifiers

The Vertica Data collector (DC) tables monitor many different database functions. You can have a notifier automatically send a message to a Kafka endpoint when a DC component updates. You can query the DATA_COLLECTOR table to get a list of the DC components.

You configure the notifier to send DC component updates to Kafka using the function SET_DATA_COLLECTOR_NOTIFY_POLICY .

To be notified of failed login attempts, you can create a notifier that sends a notification when the DC component LoginFailures updates. The TLSMODE 'verify-ca' verifies that the server's certificate is signed by a trusted CA.

```
=> CREATE NOTIFIER vertica_stats ACTION 'kafka://kafka01.example.com:9092' MAXMEMORYSIZE '10M' TLSMODE 'verify-ca';
CREATE NOTIFIER
=> SELECT SET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures','vertica_stats', 'vertica_notifications', true);
SET_DATA_COLLECTOR_NOTIFY_POLICY
-----------------------------------
 SET
(1 row)
```

Like the messages sent via the NOTIFY function, the data sent to Kafka from the DC components is in JSON format. The previous example results in messages like the following being sent to the vertica_notifications Kafka topic:

```
{"_db":"vmart","_schema":"v_internal","_table":"dc_login_failures",
"authentication_method":"Reject","client_authentication_name":"",
"client_hostname":"::1","client_label":"","client_os_user_name":"dbadmin",
"client_pid":481535,"client_version":"","database_name":"alice",
"effective_protocol":"3.8","node_name":"v_vmart_node0001",
"reason":"INVALID USER","requested_protocol":"3.8","ssl_client_fingerprint":"",
"time":"2018-06-19 14:51:22.437035-04","user_name":"alice"}
```

Viewing notification policies for a DC component

Use the GET_DATA_COLLECTOR_NOTIFY_POLICY function to list the policies set for a DC component.

```
=> SELECT GET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures');
                GET_DATA_COLLECTOR_NOTIFY_POLICY
-------------------------------------------------------------------------
 Notifiable;  Notifier: vertica_stats; Channel: vertica_notifications
(1 row)
```

Disabling a notification policy

You can call SET_DATA_COLLECTOR_NOTIFY_POLICY function with its fourth argument set to FALSE to disable a notification policy. The following example disables the notify policy for the LoginFailures component:

```
=> SELECT SET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures','vertica_stats', 'vertica_notifications', false);
 SET_DATA_COLLECTOR_NOTIFY_POLICY
----------------------------------
 SET
(1 row)
```

```
=> SELECT GET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures');
 GET_DATA_COLLECTOR_NOTIFY_POLICY
----------------------------------
 Not notifiable;
(1 row)
```

TLS/SSL encryption with Kafka

You can use TLS/SSL encryption between Vertica, your scheduler, and Kakfa. This encryption prevents others from accessing the data that is sent between Kafka and Vertica. It can also verify the identity of all parties involved in data streaming, so no impostor can pose as your Vertica cluster or a Kafka broker.

> **Note**
> Many people often confuse the terms TLS and SSL. SSL is an older encryption protocol that has been largely replace with the newer and more secure TLS standard. However, many people still use the term SSL to refer to encryption between servers and applications , even when that encryption is actually TLS. For example, Java and Kafka use the term SSL exclusively, even when dealing with TLS. This document uses SSL/TLS and SSL interchangeably.

Some common cases where you want to use SSL encryption between Vertica and Kafka are:

- Your Vertica database and Kafka communicate over an insecure network. For example, suppose your Kafka cluster is located in a cloud service and your Vertica cluster is within your internal network. In this case, any data you read from Kafka travels over an insecure connection across the Internet.
- You are required by security policies, laws, or other requirements to encrypt all of your network traffic.

For more information about TLS/SSL encryption in Vertica, see TLS protocol .

Using TLS/SSL between the scheduler and Vertica

The scheduler connects to Vertica the same way other client applications do. There are two ways you can configure Vertica to use SSL/TLS authentication and encryption with clients:

- If Vertica is configured to use SSL/TLS server authentication, you can choose to have your scheduler confirm the identity of the Vertica server.
- If Vertica is configured to use mutual SSL/TLS authentication, you can configure your scheduler identify itself to Vertica as well as have it verify the identity of the Vertica server. Depending on your database's configuration, the Vertica server may require your scheduler to use TLS when connecting. See Client authentication with TLS for more information.

For information on encrypted client connections with Vertica, refer to TLS protocol .

The scheduler runs on a Java Virtual Machine (JVM) and uses JDBC to connect to Vertica. It acts like any other JDBC client when connecting to Vertica. To use TLS/SSL encryption for the scheduler's connection to Vertica, use the Java keystore and truststore mechanism to hold the keys and certificates the scheduler uses to identify itself and Vertica.

- The keystore contains your scheduler's private encryption key and its certificate (public key).
- The truststore contains CAs that you trust. If you enable authentication, the scheduler uses these CAs to verify the identity of the Vertica cluster it connects to. If one of the CAs in the trust store was used to sign the server's certificate, then the Scheduler knows it can trust the identity of the Vertica server.

You can pass options to the JVM that executes the scheduler through the Linux environment variable named VKCONFIG_JVM_OPTS. You add the parameters to this variable that alter the scheduler's JDBC settings (such as the truststore and keystore for the scheduler's JDBC connection). See Step 2: Set the VKCONFIG_JVM_OPTS Environment Variable for an example.

You can also use the --jdbc-url scheduler option to alter the JDBC configuration. See Common vkconfig script options for more information about the scheduler options and JDBC connection properties for more information about the properties they can alter.

Using TLS/SSL between Vertica and Kafka

You can stream data from Kafka into Vertica two ways: manually using a COPY statement and the KafkaSource UD source function, or automatically using the scheduler.

To directly copy data from Kafka via an SSL connection, you set session variables containing an SSL key and certificate. When KafkaSource finds that you have set these variables, it uses the key and certificate to create a secure connection to Kafka. See Kafka TLS/SSL Example Part 4: Loading Data Directly From Kafka for details.

When automatically streaming data from Kafka to Vertica, you configure the scheduler the same way you do to use an SSL connection to Vertica. When the scheduler executes COPY statements to load data from Kafka, it uses its own keystore and truststore to create an SSL connection to Kafka.

To use an SSL connection when producing data from Vertica to Kafka, you set the same session variables you use when directly streaming data from Kafka via an SSL connection. The KafkaExport function uses these variables to establish a secure connection to Kafka.

> **Note**
> Notifiers do not currently support using TLS/SSL connections.

See the Apache Kafka documentation for more information about using SSL/TLS authentication with Kafka .

In this section

- Planning TLS/SSL encryption between Vertica and Kafka
- Configuring your scheduler for TLS connections
- Using TLS/SSL when directly loading data from Kafka
- Configure Kafka for TLS
- Troubleshooting Kafka TLS/SSL connection issues

Planning TLS/SSL encryption between Vertica and Kafka

Some things to consider before you begin configuring TLS/SSL:

- Which connections between the scheduler, Vertica, and Kafka needs to be encrypted? In some cases, you may only need to enable encryption between Vertica and Kafka. This scenario is common when Vertica and the Kafka cluster are on different networks. For example, suppose Kafka is hosted in a cloud provider and Vertica is hosted in your internal network. Then the data must travel across the unsecured Internet between the two. However, if Vertica and the scheduler are both in your local network, you may decide that configuring them to use SSL/TLS is unnecessary. In other cases, you will want all parts of the system to be encrypted. For example, you want to encrypt all traffic when Kafka, Vertica, and the scheduler are all hosted in a cloud provider whose network may not be secure.
- Which connections between the scheduler, Vertica, and Kafka require trust? You can opt to have any of these connections fail if one system cannot verify the identity of another. See Verifying Identities below.
- Which CAs will you be using to sign each certificate? The simplest way to configure the scheduler, Kafka, and Vertica is to use the same CA to sign all of the certificates you will use when setting up TLS/SSL. Using the same root CA to sign the certificates requires you to be able to edit the configuration of Kafka, Vertica, and the scheduler. If you cannot use the same CA to sign all certificates, all truststores must contain the entire chain of CAs used to sign the certificates, all the way up to the root CA. Including the entire chain of trust ensures each system can verify each other's identities.

Verifying identities

Your primary challenge when configuring TLS/SSL encryption between Vertica, the scheduler, and Kafka is making sure the scheduler, Kafka, and Vertica can all verify each other's identity. The most common problem people have encountered when setting up TLS/SSL encryption is ensuring the remote system can verify the authenticity of a system's certificate. The best way to prevent this problem is to ensure that the all systems have their certificates signed by a CA that all of the systems explicitly trust.

When a system attempts to start an encrypted connection with another system, it sends its certificate to the remote system. This certificate can be signed by one or more Certificate Authorities (CA) that help identify the system making the connection. These signatures form a "chain of trust." A certificate is signed by a CA. That CA, in turn, could have been signed by another CA, and so forth. Often, the chain ends with a CA (referred to as the root CA) from a well-known commercial provider of certificates, such as Comodo SSL or DigiCert, that are trusted by default on many platforms such as operating systems and web browsers.

If the remote system finds a CA in the chain that it trusts, it verifies the identity of the system making the connection, and the connection can continue. If the remote system cannot find the signature of a CA it trusts, it may block the connection, depending on its configuration. Systems can be configured to only allow connections from systems whose identity has been verified.

Note
Verifying the identity of another system making a TLE/SSL connection is often referred to as "authentication." Do not confuse this use of authentication with other forms of authentication used with Vertica. For example, TLS/SSL's authentication of a client connection has nothing to do with Vertica user authentication. Even if you successfully establish a TLS/SSL connection to Vertica using a client, Vertica still requires you to provide a user name and password before you can interact with it.

Configuring your scheduler for TLS connections

The scheduler can use TLS for two different connections: the one it makes to Vertica, and the connection it creates when running COPY statements to retrieve data from Kafka. Because the scheduler is a Java application, you supply the TLS key and the certificate used to sign it in a keystore. You also supply a truststore that contains in the certificates that the scheduler should trust. Both the connection to Vertica and to Kafka can use the same keystore and truststore. You can also choose to use separate keystores and truststores for these two connections by setting different JDBC settings for the scheduler. See JDBC connection properties for a list of these settings.

See Kafka TLS-SSL Example Part 5: Configure the Scheduler for detailed steps on configuring your scheduler to use SSL.

Important
If you choose to use a file format other than the standard Java Keystore (JKS) format for your keystore or truststore files, you must use the correct file extension in the filename. For example, suppose you choose to use a keystore and truststore saved in PKCS#12 format. Then your keystore and trustore files must end with the .pfx or .p12 extension.

If the scheduler does not recognize the file's extension (or there is no extension in the file name), it assumes that the file is in JKS format. If the file is not in JKS format, you will see an error message when starting the scheduler, similar to "Failed to create an SSLSocketFactory when setting up TLS: keystore not found."

Note that if the Kafka server's parameter client.ssl.auth is set to none or requested , you do not need to create a keystore.

Using TLS/SSL when directly loading data from Kafka

You can manually load data from Kafka using the COPY statement and the KafkaSource user-defined load function (see Manually consume data from Kafka). To have KafkaSource open a secure connection to Kafka, you must supply it with an SSL key and other information.

When starting, the KafkaSource function checks if several user session variables are defined. These variables contain the SSL key, the certificate used to sign the key, and other information that the function needs to create the SSL connection. See Kafka user-defined session parameters for a list of these variables. If KafkaSource finds these variables are defined, it uses them to create an SSL connection to Kafka.

See Kafka TLS/SSL Example Part 4: Loading Data Directly From Kafka for a step-by-step guide on configuring and using an SSL connection when directly copying data from Kafka.

These variables are also used by the KafkaExport function to establish a secure connection to Kafka when exporting data.

Configure Kafka for TLS

This page covers procedures for configuring TLS connections Vertica, Kafka, and the scheduler.

Note that the following example configures TLS for a Kafka server where ssl.client.auth=required , which requires the following:

- kafka_SSL_Certificate
- kafka_SSL_PrivateKey_secret
- kafka_SSL_PrivateKeyPassword_secret
- A keystore for the Scheduler

If your configuration uses ssl.client.auth=none or ssl.client.auth=requested , these parameters and the scheduler keystore are optional.

Creating certificates for Vertica and clients

The CA certificate in this example is self-signed. In a production environment, you should instead use a trusted CA.

This example uses the same self-signed root CA to sign all of the certificates used by the scheduler, Kafka brokers, and Vertica. If you cannot use the same CA to sign the keys for all of these systems, make sure you include the entire chain of trust in your keystores.

For more information, see Generating TLS certificates and keys .

1. Generate a private key, root.key .

```
$ openssl genrsa -out root.key
Generating RSA private key, 2048 bit long modulus
.....................................................................
.........................+++
.............+++
e is 65537 (0x10001)
```

2. Generate a self-signed CA certificate.

```
$ openssl req -new -x509 -key root.key -out root.crt
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MA
Locality Name (eg, city) []:Cambridge
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Company
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:*.mycompany.com
Email Address []:myemail@mycompany.com
```

3. Restrict to the owner read/write permissions for root.key and root.crt . Grant read permissions to other groups for root.crt .

```
$ ls
root.crt  root.key
$ chmod 600 root.key
$ chmod 644 root.crt
```

4. Generate the server private key, *server.key* .

```
$ openssl genrsa -out server.key
Generating RSA private key, 2048 bit long modulus
.....................................................................+++
.....................................+++
e is 65537 (0x10001)
```

5. Create a certificate signing request (CSR) for your CA. Be sure to set the "Common Name" field to a wildcard (asterisk) so the certificate is accepted for all Vertica nodes in the cluster:

```
$ openssl req -new -key server.key -out server_reqout.txt
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MA
Locality Name (eg, city) []:Cambridge
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Company
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:*.mycompany.com
Email Address []:myemail@mycompany.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []: server_key_password
An optional company name []:
```

6. Sign the server certificate with your CA. This creates the server certificate server.crt .

```
$ openssl x509 -req -in server_reqout.txt -days 3650 -sha1 -CAcreateserial -CA root.crt \
  -CAkey root.key -out server.crt
  Signature ok
  subject=/C=US/ST=MA/L=Cambridge/O=My Company/CN=*.mycompany.com/emailAddress=myemail@mycompany.com
  Getting CA Private Key
```

7. Set the appropriate permissions for the key and certificate.

```
$ chmod 600 server.key
$ chmod 644 server.crt
```

Create a client key and certificate (mutual mode only)

In **Mutual Mode** , clients and servers verify each other's certificates before establishing a connection. The following procedure creates a client key and certificate to present to Vertica. The certificate must be signed by a CA that Vertica trusts.

The steps for this are identical to those above for creating a server key and certificate for Vertica.

```
$ openssl genrsa -out client.key
Generating RSA private key, 2048 bit long modulus
.....................................................................+++
.............................+++
e is 65537 (0x10001)

$ openssl req -new -key client.key -out client_reqout.txt
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MA
Locality Name (eg, city) []:Cambridge
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Company
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:*.mycompany.com
Email Address []:myemail@mycompany.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []: server_key_password
An optional company name []:

$ openssl x509 -req -in client_reqout.txt -days 3650 -sha1 -CAcreateserial -CA root.crt \
  -CAkey root.key -out client.crt
Signature ok
subject=/C=US/ST=MA/L=Cambridge/O=My Company/CN=*.mycompany.com/emailAddress=myemail@mycompany.com
Getting CA Private Key

$ chmod 600 client.key
$ chmod 644 client.crt
```

Set up mutual mode client-server TLS

Configure Vertica for mutual mode

The following keys and certificates must be imported and then distributed to the nodes on your Vertica cluster with TLS Configuration for **Mutual Mode** :

- root.key
- root.crt
- server.key
- server.crt

You can view existing keys and certificates by querying CRYPTOGRAPHIC_KEYS and CERTIFICATES .

1. Import the server and root keys and certificates into Vertica with CREATE KEY and CREATE CERTIFICATE . See Generating TLS certificates and keys for details.

```
=> CREATE KEY imported_key TYPE 'RSA' AS '-----BEGIN PRIVATE KEY-----...-----END PRIVATE KEY-----';
=> CREATE CA CERTIFICATE imported_ca AS '-----BEGIN CERTIFICATE-----...-----END CERTIFICATE-----';
=> CREATE CERTIFICATE imported_cert AS '-----BEGIN CERTIFICATE-----...-----END CERTIFICATE-----';
```

   In this example, \set is used to retrieve the contents of root.key , root.crt , server.key , and server.crt .

```
=> \set ca_cert '''`cat root.crt`'''
=> \set serv_key '''`cat server.key`'''
=> \set serv_cert '''`cat server.crt`'''

=> CREATE CA CERTIFICATE root_ca AS :ca_cert;
CREATE CERTIFICATE
=> CREATE KEY server_key TYPE 'RSA' AS :serv_key;
CREATE KEY
=> CREATE CERTIFICATE server_cert AS :serv_cert;
CREATE CERTIFICATE
```

2. Follow the steps for **Mutual Mode** in Configuring client-server TLS to set the proper TLSMODE and TLS Configuration parameters.

Configure a client for mutual mode

Clients must have their private key, certificate, and CA certificate. The certificate will be presented to Vertica when establishing a connection, and the CA certificate will be used to verify the server certificate from Vertica.

This example configures the vsql client for mutual mode.

1. Create a .vsql directory in the user's home directory.

```
$ mkdir ~/.vsql
```

2. Copy client.key , client.crt , and root.crt to the vsql directory.

```
$ cp client.key client.crt root.crt ~/.vsql
```

3. Log into Vertica with vsql and query the SESSIONS system table to verify that the connection is using mutual mode:

```
$ vsql
Password: user-password
Welcome to vsql, the Vertica Analytic Database interactive terminal.

Type:  \h or \? for help with vsql commands
       \g or terminate with semicolon to execute query
       \q to quit

SSL connection (cipher: DHE-RSA-AES256-GCM-SHA384, bits: 256, protocol: TLSv1.2)

=> select user_name,ssl_state from sessions;
 user_name | ssl_state
-----------+-----------
 dbadmin   | Mutual
(1 row)
```

Configure Kafka for TLS

Configure the Kafka brokers

This procedure configures Kafka to use TLS with client connections. You can also configure Kafka to use TLS to communicate between brokers. However, inter-broker TLS has no impact on establishing an encrypted connection between Vertica and Kafka.

1. Create a truststore file for all of your Kafka brokers, importing your CA certificate. This example uses the self-signed root.crt created above.

```
=> $ keytool -keystore kafka.truststore.jks -alias CARoot -import \
            -file root.crt
Enter keystore password: some_password
Re-enter new password: some_password
Owner: EMAILADDRESS=myemail@mycompany.com, CN=*.mycompany.com, O=MyCompany, L=Cambridge, ST=MA, C=US
Issuer: EMAILADDRESS=myemail@mycompany.com, CN=*.mycompany.com, O=MyCompany, L=Cambridge, ST=MA, C=US
Serial number: c3f02e87707d01aa
Valid from: Fri Mar 22 13:37:37 EDT 2019 until: Sun Apr 21 13:37:37 EDT 2019
Certificate fingerprints:
        MD5:  73:B1:87:87:7B:FE:F1:6E:94:55:FD:AF:5D:D0:C3:0C
        SHA1: C0:69:1C:93:54:21:87:C7:03:93:FE:39:45:66:DE:22:18:7E:CD:94
        SHA256: 23:03:BB:B7:10:12:50:D9:C5:D0:B7:58:97:41:1E:0F:25:A0:DB:
                D0:1E:7D:F9:6E:60:8F:79:A6:1C:3F:DD:D5
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 50 69 11 64 45 E9 CC C5   09 EE 26 B5 3E 71 39 7C  Pi.dE.....&.>q9.
0010: E5 3D 78 16                                        .=x.
]
]

#2: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 50 69 11 64 45 E9 CC C5   09 EE 26 B5 3E 71 39 7C  Pi.dE.....&.>q9.
0010: E5 3D 78 16                                        .=x.
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
```

2. Create a keystore file for the Kafka broker named kafka01 . Each broker's keystore should be unique.
   The keytool command adds the a Subject Alternative Name (SAN) used as a fallback when establishing a TLS connection. Use your Kafka' broker's fully-qualified domain name (FQDN) as the value for the SAN and "What is your first and last name?" prompt.
   In this example, the FQDN is kafka01.example.com . The alias for keytool is set to localhost , so local connections to the broker use TLS.

```
$ keytool -keystore kafka01.keystore.jks -alias localhost -validity 365 -genkey -keyalg RSA \
    -ext SAN=DNS:kafka01.mycompany.com
Enter keystore password: some_password
Re-enter new password: some_password
What is your first and last name?
  [Unknown]:  kafka01.mycompany.com
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]: MyCompany
What is the name of your City or Locality?
  [Unknown]:  Cambridge
What is the name of your State or Province?
  [Unknown]:  MA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Database Admin, OU=MyCompany, O=Unknown, L=Cambridge, ST=MA, C=US correct?
  [no]:  yes

Enter key password for <localhost>
      (RETURN if same as keystore password):
```

3. Export the Kafka broker's certificate. In this example, the certificate is exported as kafka01.unsigned.crt .

```
$ keytool -keystore kafka01.keystore.jks -alias localhost \
            -certreq -file kafka01.unsigned.crt
 Enter keystore password: some_password
```

4. Sign the broker's certificate with the CA certificate.

```
$ openssl x509 -req -CA root.crt -CAkey root.key -in kafka01.unsigned.crt \
          -out kafka01.signed.crt -days 365 -CAcreateserial
Signature ok
subject=/C=US/ST=MA/L=Cambridge/O=Unknown/OU=MyCompany/CN=Database Admin
Getting CA Private Key
```

5. Import the CA certificate into the broker's keystore.

> **Note**
> If you use different CAs to sign the certificates in your environment, you must add the entire chain of CAs you used to sign your certificate to the keystore, all the way up to the root CA. Including the entire chain of trust helps other systems verify the identity of your Kafka broker.

```
$ keytool -keystore kafka01.keystore.jks -alias CARoot -import -file root.crt
Enter keystore password: some_password
Owner: EMAILADDRESS=myemail@mycompany.com, CN=*.mycompany.com, O=MyCompany, L=Cambridge, ST=MA, C=US
Issuer: EMAILADDRESS=myemail@mycompany.com, CN=*.mycompany.com, O=MyCompany, L=Cambridge, ST=MA, C=US
Serial number: c3f02e87707d01aa
Valid from: Fri Mar 22 13:37:37 EDT 2019 until: Sun Apr 21 13:37:37 EDT 2019
Certificate fingerprints:
        MD5:  73:B1:87:87:7B:FE:F1:6E:94:55:FD:AF:5D:D0:C3:0C
        SHA1: C0:69:1C:93:54:21:87:C7:03:93:FE:39:45:66:DE:22:18:7E:CD:94
        SHA256: 23:03:BB:B7:10:12:50:D9:C5:D0:B7:58:97:41:1E:0F:25:A0:DB:D0:1E:7D:F9:6E:60:8F:79:A6:1C:3F:DD:D5
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 50 69 11 64 45 E9 CC C5   09 EE 26 B5 3E 71 39 7C  Pi.dE.....&.>q9.
0010: E5 3D 78 16                                        .=x.
]
]

#2: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 50 69 11 64 45 E9 CC C5   09 EE 26 B5 3E 71 39 7C  Pi.dE.....&.>q9.
0010: E5 3D 78 16                                        .=x.
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
```

6. Import the signed Kafka broker certificate into the keystore.

```
$ keytool -keystore kafka01.keystore.jks -alias localhost \
            -import -file kafka01.signed.crt
Enter keystore password: some_password
Owner: CN=Database Admin, OU=MyCompany, O=Unknown, L=Cambridge, ST=MA, C=US
Issuer: EMAILADDRESS=myemail@mycompany.com, CN=*.mycompany.com, O=MyCompany, L=Cambridge, ST=MA, C=US
Serial number: b4bba9a1828ecaaf
Valid from: Tue Mar 26 12:26:34 EDT 2019 until: Wed Mar 25 12:26:34 EDT 2020
Certificate fingerprints:
        MD5:  17:EA:3E:15:B4:15:E9:93:67:EE:59:C0:4F:D1:4C:01
        SHA1: D5:35:B7:F7:44:7C:D6:B4:56:6F:38:2D:CD:3A:16:44:19:C1:06:B7
        SHA256: 25:8C:46:03:60:A7:4C:10:A8:12:8E:EA:4A:FA:42:1D:A8:C5:FB:65:81:74:CB:46:FD:B1:33:64:F2:A3:46:B0
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 1
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

7. If you are not logged into the Kafka broker for which you prepared the keystore, copy the truststore and keystore to it using scp. If you have already decided where to store the keystore and truststore files in the broker's filesystem, you can directly copy them to their final destination. This example just copies them to the root user's home directory temporarily. The next step moves them into their final location.

```
$ scp kafka.truststore.jks kafka01.keystore.jks root@kafka01.mycompany.com:
root@kafka01.mycompany.com's password: root_password
kafka.truststore.jks                          100% 1048     1.0KB/s   00:00
kafka01.keystore.jks                          100% 3277     3.2KB/s   00:00
```

8. Repeat steps 2 through 7 for the remaining Kafka brokers.

Allow Kafka to read the keystore and truststore

If you did not copy the truststore and keystore to directory where Kafka can read them in the previous step, you must copy them to a final location on the broker. You must also allow the user account you use to run Kafka to read these files. The easiest way to ensure the user's access is to give this user ownership of these files.

In this example, Kafka is run by a Linux user kafka . If you use another user to run Kafka, be sure to set the permissions on the truststore and keystore files appropriately.

1. Log into the Kafka broker as root.
2. Copy the truststore and keystore to a directory where Kafka can access them. There is no set location for these files: you can choose a directory under /etc, or some other location where configuration files are usually stored. This example copies them from root's home directory to Kafka's configuration directory named /opt/kafka/config/. In your own system, this configuration directory may be in a different location depending on how you installed Kafka.
3. Copy the truststore and keystore to a directory where Kafka can access them. There is no set location for these files: you can choose a directory under /etc , or some other location where configuration files are usually stored. This example copies them from root's home directory to Kafka's configuration directory named /opt/kafka/config/ . In your own system, this configuration directory may be in a different location depending on how you installed Kafka.

```
~# cd /opt/kafka/config/
/opt/kafka/config# cp /root/kafka01.keystore.jks /root/kafka.truststore.jks .
```

4. If you aren't logged in as a user account that runs Kafka, change the ownership of the truststore and keystore files. This example changes the ownership from root (which is the user currently logged in) to the kafka user:

```
/opt/kafka/config# ls -l
total 80
...
-rw-r--r-- 1 kafka nogroup 1221 Feb 21  2018 consumer.properties
-rw------- 1 root  root    3277 Mar 27 08:03 kafka01.keystore.jks
-rw-r--r-- 1 root  root    1048 Mar 27 08:03 kafka.truststore.jks
-rw-r--r-- 1 kafka nogroup 4727 Feb 21  2018 log4j.properties
...
/opt/kafka/config# chown kafka kafka01.keystore.jks kafka.truststore.jks
/opt/kafka/config# ls -l
total 80
...
-rw-r--r-- 1 kafka nogroup 1221 Feb 21  2018 consumer.properties
-rw------- 1 kafka root    3277 Mar 27 08:03 kafka01.keystore.jks
-rw-r--r-- 1 kafka root    1048 Mar 27 08:03 kafka.truststore.jks
-rw-r--r-- 1 kafka nogroup 4727 Feb 21  2018 log4j.properties
...
```

5. Repeat steps 1 through 3 for the remaining Kafka brokers.

Configure Kafka to use TLS

With the truststore and keystore in place, your next step is to edit the Kafka's server.properties configuration file to tell Kafka to use TLS/SSL encryption. This file is usually stored in the Kafka config directory. The location of this directory depends on how you installed Kafka. In this example, the file is located in /opt/kafka/config .

When editing the files, be sure you do not change their ownership. The best way to ensure Linux does not change the file's ownership is to use su to become the user account that runs Kafka, assuming you are not already logged in as that user:

```
$ /opt/kafka/config# su -s /bin/bash kafka
```

> **Note**
> The previous command lets you start a shell as the kafka system user even if that user cannot log in.

The server.properties file contains Kafka broker settings in a *property* = *value* format. To configure the Kafka broker to use SSL, alter or add the following property settings:

listeners
Host names and ports on which the Kafka broker listens. If you are not using SSL for connections between brokers, you must supply both a PLANTEXT and SSL option. For example:

listeners=PLAINTEXT:// *hostname* :9092,SSL:// *hostname* :9093

ssl.keystore.location
Absolute path to the keystore file.

ssl.keystore.password
Password for the keystore file.

ssl.key.password
Password for the Kafka broker's key in the keystore. You can make this password different than the keystore password if you choose.

ssl.truststore.location
Location of the truststore file.

ssl.truststore.password
Password to access the truststore.

ssl.enabled.protocols
TLS/SSL protocols that Kafka allows clients to use.

ssl.client.auth
Specifies whether SSL authentication is required or optional. The most secure setting for this setting is required to verify the client's identity.

Important
These settings vary depending on your version of Kafka. Always consult the Apache Kafka documentation for your version of Kafka before making changes to server.properties . In particular, be aware that Kafka version 2.0 and later enables host name verification for clients and inter-broker communications by default.

This example configures Kafka to verify client identities via SSL authentication. It does not use SSL to communicate with other brokers, so the server.properties file defines both SSL and PLAINTEXT listener ports. It does not supply a host name for listener ports which tells Kafka to listen on the default network interface.

The lines added to the kafka01 broker's copy of server.properties for this configuration are:

```
listeners=PLAINTEXT://:9092,SSL://:9093
ssl.keystore.location=/opt/kafka/config/kafka01.keystore.jks
ssl.keystore.password=vertica
ssl.key.password=vertica
ssl.truststore.location=/opt/kafka/config/kafka.truststore.jks
ssl.truststore.password=vertica
ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1
ssl.client.auth=required
```

You must make these changes to the server.properties file on all of your brokers.

After making your changes to your broker's server.properties files, restart Kafka. How you restart Kafka depends on your installation:

- If Kafka is running as part of a Hadoop cluster, you can usually restart it from within whatever interface you use to control Hadoop (such as Ambari).
- If you installed Kafka directly, you can restart it either by directly running the kafka-server-stop.sh and kafka-server-start.sh scripts or via the Linux system's service control commands (such as systemctl). You must run this command on each broker.

Test the configuration
If you have not configured client authentication, you can quickly test whether Kafka can access its keystore by running the command:

```
$ openssl s_client -debug -connect broker_host_name:9093 -tls1
```

If Kafka is able to access its keystore, this command will output a dump of the broker's certificate (exit with CTRL+C):

```
=> # openssl s_client -debug -connect kafka01.mycompany.com:9093 -tls1
CONNECTED(00000003)
write to 0xa4e4f0 [0xa58023] (197 bytes => 197 (0xC5))
0000 - 16 03 01 00 c0 01 00 00-bc 03 01 76 85 ed f0 fe   ...........v....
0010 - 60 60 7e 78 9d d4 a8 f7-e6 aa 5c 80 b9 a7 37 61   ``~x......\...7a
0020 - 8e 04 ac 03 6d 52 86 f5-84 4b 5c 00 00 62 c0 14   ....mR...K\..b..
0030 - c0 0a 00 39 00 38 00 37-00 36 00 88 00 87 00 86   ...9.8.7.6......
0040 - 00 85 c0 0f c0 05 00 35-00 84 c0 13 c0 09 00 33   .......5.......3
0050 - 00 32 00 31 00 30 00 9a-00 99 00 98 00 97 00 45   .2.1.0.........E
0060 - 00 44 00 43 00 42 c0 0e-c0 04 00 2f 00 96 00 41   .D.C.B...../...A
0070 - c0 11 c0 07 c0 0c c0 02-00 05 00 04 c0 12 c0 08   ................
0080 - 00 16 00 13 00 10 00 0d-c0 0d c0 03 00 0a 00 ff   ................
0090 - 01 00 00 31 00 0b 00 04-03 00 01 02 00 0a 00 1c   ...1............
00a0 - 00 1a 00 17 00 19 00 1c-00 1b 00 18 00 1a 00 16   ................
00b0 - 00 0e 00 0d 00 0b 00 0c-00 09 00 0a 00 23 00 00   .............#..
00c0 - 00 0f 00 01 01                                    .....
read from 0xa4e4f0 [0xa53ad3] (5 bytes => 5 (0x5))
0000 - 16 03 01 08 fc                                    .....
           . . .
```

The above method is not conclusive, however; it only tells you if Kafka is able to find its keystore.

The best test of whether Kafka is able to accept TLS connections is to configure the command-line Kafka producer and consumer. In order to configure these tools, you must first create a client keystore. These steps are identical to creating a broker keystore.

> **Note**
> This example assumes that Kafka has a topic named test that you can send test messages to.

1. Create the client keystore:
   ```
   keytool -keystore client.keystore.jks -alias localhost -validity 365 -genkey -keyalg RSA -ext SAN=DNS:fqdn_of_client_system
   ```
2. Respond to the "What is your first and last name?" with the FQDN of the system you will use to run the producer and/or consumer. Answer the rest of the prompts with the details of your organization.
3. Export the client certificate so it can be signed:
   ```
   keytool -keystore client.keystore.jks -alias localhost -certreq -file client.unsigned.cert
   ```
4. Sign the client certificate with the root CA:
   ```
   openssl x509 -req -CA root.crt -CAkey root.key -in client.unsigned.cert -out client.signed.cert \
         -days 365 -CAcreateserial
   ```
5. Add the root CA to keystore:
   ```
   keytool -keystore client.keystore.jks -alias CARoot -import -file root.crt
   ```
6. Add the signed client certificate to the keystore:
   ```
   keytool -keystore client.keystore.jks -alias localhost -import -file client.signed.cert
   ```
7. Copy the keystore to a location where you will use it. For example, you could choose to copy it to the same directory where you copied the keystore for the Kafka broker. If you choose to copy it to some other location, or intend to use some other user to run the command-line clients, be sure to add a copy of the truststore file you created for the brokers. Clients can reuse this truststore file for authenticating the Kafka brokers because the same CA is used to sign all of the certificates. Also set the file's ownership and permissions accordingly.

Next, you must create a properties file (similar to the broker's server.properties file) that configures the command-line clients to use TLS. For a client running on the Kafka broker named kafka01, your configuration file could would look like this:

```
security.protocol=SSL
ssl.truststore.location=/opt/kafka/config/kafka.truststore.jks
ssl.truststore.password=trustore_password
ssl.keystore.location=/opt/kafka/config/client.keystore.jks
ssl.keystore.password=keystore_password
ssl.key.password=key_password
ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1
ssl.client.auth=required
```

This property file assumes the keystore file is located in the Kafka configuration directory.

Finally, you can run the command line producer or consumer to ensure they can connect and process data. You supply these clients the properties file you just created. The following example assumes you stored the properties file in the Kafka configuration directory, and that Kafka is installed in /opt/kafka :

```
~# cd /opt/kafka


/opt/kafka# bin/kafka-console-producer.sh --broker-list kafka01.mycompany.com:9093  \
                              --topic test --producer.config config/client.properties
>test
>test again
>More testing. These messages seem to be getting through!
^D
/opt/kafka# bin/kafka-console-consumer.sh --bootstrap-server kafaka01.mycompany.com:9093  --topic test \
                              --consumer.config config/client.properties --from-beginning
test
test again
More testing. These messages seem to be getting through!
^C
Processed a total of 3 messages
```

Loading data from Kafka

After you configure Kafka to accept TLS connections, verify that you can directly load data from it into Vertica. You should perform this step even if you plan to create a scheduler to automatically stream data.

You can choose to create a separate key and certificate for directly loading data from Kafka. This example re-uses the key and certificate created for the Vertica server in part 2 of this example.

You directly load data from Kafka by using the KafkaSource data source function with the COPY statement (see Manually consume data from Kafka). The KafkaSource function creates the connection to Kafka, so it needs a key, certificate, and related passwords to create an encrypted connection. You pass this information via session parameters. See Kafka user-defined session parameters for a list of these parameters.

The easiest way to get the key and certificate into the parameters is by first reading them into vsql variables . You get their contents by using back quotes to read the file contents via the Linux shell. Then you set the session parameters from the variables. Before setting the session parameters, increase the MaxSessionUDParameterSize session parameter to add enough storage space in the session variables for the key and the certificates. They can be larger than the default size limit for session variables (1000 bytes).

The following example reads the server key and certificate and the root CA from the a directory named /home/dbadmin/SSL . Because the server's key password is not saved in a file, the example sets it in a Linux environment variable named KVERTICA_PASS before running vsql. The example sets MaxSessionUDParameterSize to 100000 before setting the session variables. Finally, it enables TLS for the Kafka connection and streams data from the topic named test.

```
$ export KVERTICA_PASS=server_key_password
$ vsql
Password:
Welcome to vsql, the Vertica Analytic Database interactive terminal.

Type:  \h or \? for help with vsql commands
       \g or terminate with semicolon to execute query
       \q to quit

SSL connection (cipher: DHE-RSA-AES256-GCM-SHA384, bits: 256, protocol: TLSv1.2)

=> \set cert '\''`cat /home/dbadmin/SSL/server.crt`'\''
=> \set pkey '\''`cat /home/dbadmin/SSL/server.key`'\''
=> \set ca '\''`cat /home/dbadmin/SSL/root.crt`'\''
=> \set pass '\''`echo $KVERTICA_PASS`'\''
=> alter session set MaxSessionUDParameterSize=100000;
ALTER SESSION
=> ALTER SESSION SET UDPARAMETER kafka_SSL_Certificate=:cert;
ALTER SESSION
=> ALTER SESSION SET UDPARAMETER kafka_SSL_PrivateKey_secret=:pkey;
ALTER SESSION
=> ALTER SESSION SET UDPARAMETER kafka_SSL_PrivateKeyPassword_secret=:pass;
ALTER SESSION
=> ALTER SESSION SET UDPARAMETER kafka_SSL_CA=:ca;
ALTER SESSION
=> ALTER SESSION SET UDPARAMETER kafka_Enable_SSL=1;
ALTER SESSION
=> CREATE TABLE t (a VARCHAR);
CREATE TABLE
=> COPY t SOURCE KafkaSource(brokers='kafka01.mycompany.com:9093',
                             stream='test|0|-2', stop_on_eof=true,
                             duration=interval '5 seconds')
          PARSER KafkaParser();
 Rows Loaded
-------------
           3
(1 row)

=> SELECT * FROM t;
                            a
-----------------------------------------------------------
 test again
 More testing. These messages seem to be getting through!
 test

(3 rows)
```

Configure the scheduler

The final piece of the configuration is to set up the scheduler to use SSL when communicating with Kafka (and optionally with Vertica). When the scheduler runs a COPY command to get data from Kafka, it uses its own key and certificate to authenticate with Kafka. If you choose to have the scheduler use TLS/SSL to connect to Vertica, it can reuse the same keystore and truststore to make this connection.

Create a truststore and keystore for the scheduler

Because the scheduler is a separate component, it must have its own key and certificate. The scheduler runs in Java and uses the JDBC interface to connect to Vertica. Therefore, you must create a keystore (when ssl.client.auth=required) and truststore for it to use when making a TLS-encrypted connection to Vertica.

Keep in mind that creating a keystore is optional if your Kafka server sets ssl.client.auth to none or requested .

This process is similar to creating the truststores and keystores for Kafka brokers. The main difference is using the the -dname option for keytool to set the Common Name (CN) for the key to a domain wildcard. Using this setting allows the key and certificate to match any host in the network. This option is especially useful if you run multiple schedulers on different servers to provide redundancy. The schedulers can use the same key and certificate, no matter which server they are running on in your domain.

1. Create a truststore file for the scheduler. Add the CA certificate that you used to sign the keystore of the Kafka cluster and Vertica cluster. If you are using more than one CA to sign your certificates, add all of the CAs you used.

```
$ keytool -keystore scheduler.truststore.jks -alias CARoot -import \
            -file root.crt
Enter keystore password: some_password
Re-enter new password: some_password
Owner: EMAILADDRESS=myemail@mycompany.com, CN=*.mycompany.com, O=MyCompany, L=Cambridge, ST=MA, C=US
Issuer: EMAILADDRESS=myemail@mycompany.com, CN=*.mycompany.com, O=MyCompany, L=Cambridge, ST=MA, C=US
Serial number: c3f02e87707d01aa
Valid from: Fri Mar 22 13:37:37 EDT 2019 until: Sun Apr 21 13:37:37 EDT 2019
Certificate fingerprints:
        MD5:  73:B1:87:87:7B:FE:F1:6E:94:55:FD:AF:5D:D0:C3:0C
        SHA1: C0:69:1C:93:54:21:87:C7:03:93:FE:39:45:66:DE:22:18:7E:CD:94
        SHA256: 23:03:BB:B7:10:12:50:D9:C5:D0:B7:58:97:41:1E:0F:25:A0:DB:
                D0:1E:7D:F9:6E:60:8F:79:A6:1C:3F:DD:D5
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 50 69 11 64 45 E9 CC C5   09 EE 26 B5 3E 71 39 7C  Pi.dE.....&.>q9.
0010: E5 3D 78 16                                        .=x.
]
]

#2: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 50 69 11 64 45 E9 CC C5   09 EE 26 B5 3E 71 39 7C  Pi.dE.....&.>q9.
0010: E5 3D 78 16                                        .=x.
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
```

2. Initialize the keystore, passing it a wildcard host name as the Common Name. The alias parameter in this command is important, as you use it later to identify the key the scheduler must use when creating SSL conections:

```
keytool -keystore scheduler.keystore.jks -alias vsched -validity 365 -genkey \
      -keyalg RSA  -dname CN=*.mycompany.com
```

Important
If you choose to use a file format other than the standard Java Keystore (JKS) format for your keystore or truststore files, you must use the correct file extension in the filename. For example, suppose you choose to use a keystore and truststore saved in PKCS#12 format. Then your keystore and trustore files must end with the .pfx or .p12 extension.

If the scheduler does not recognize the file's extension (or there is no extension in the file name), it assumes that the file is in JKS format. If the file is not in JKS format, you will see an error message when starting the scheduler, similar to "Failed to create an SSLSocketFactory when setting up TLS: keystore not found."

3. Export the scheduler's key so you can sign it with the root CA:

```
$ keytool -keystore scheduler.keystore.jks -alias vsched -certreq \
    -file scheduler.unsigned.cert
```

4. Sign the scheduler key with the root CA:

```
$ openssl x509 -req -CA root.crt -CAkey root.key -in scheduler.unsigned.cert \
    -out scheduler.signed.cert -days 365 -CAcreateserial
```

5. Re-import the scheduler key into the keystore:

```
$ keytool -keystore scheduler.keystore.jks -alias localhost -import -file scheduler.signed.cert
```

Set environment variable VKCONFIG_JVM_OPTS

You must pass several settings to the JDBC interface of the Java Virtual Machine (JVM) that runs the scheduler. These settings tell the JDBC driver where to find the keystore and truststore, as well as the key's password. The easiest way to pass in these settings is to set a Linux environment variable named VKCONFIG_JVM_OPTS. As it starts, the scheduler checks this environment variable and passes any properties defined in it to the JVM.

The properties that you need to set are:

- javax.net.ssl.keystore: the absolute path to the keystore file to use.
- javax.net.ssl.keyStorePassword: the password for the scheduler's key.
- javax.net.ssl.trustStore: The absolute path to the truststore file.

The Linux command line to set the environment variable is:

```
export VKCONFIG_JVM_OPTS="$VKCONFIG_JVM_OPTS -Djavax.net.ssl.trustStore=/path/to/truststore \
                -Djavax.net.ssl.keyStore=/path/to/keystore \
                -Djavax.net.ssl.keyStorePassword=keystore_password"
```

Note
The previous command preserves any existing contents of the VKCONFIG_JVM_OPTS variable. If you find the variable has duplicate settings, remove the $VKCONFIG_JVM_OPTS from your statement so you override the existing values in the variable.

For example, suppose the scheduler's truststore and keystore are located in the directory /home/dbadmin/SSL . Then you could use the following command to set the VKCONFIG_JVM_OPTS variable:

```
$ export VKCONFIG_JVM_OPTS="$VKCONFIG_JVM_OPTS \
                -Djavax.net.ssl.trustStore=/home/dbadmin/SSL/scheduler.truststore.jks \
                -Djavax.net.ssl.keyStore=/home/dbadmin/SSL/scheduler.keystore.jks \
                -Djavax.net.ssl.keyStorePassword=key_password"
```

Important
The Java property names are case sensitive.

To ensure that this variable is always set, add the command to the ~/.bashrc or other startup file of the user account that runs the scheduler.

If you require TLS on the JDBC connection to Vertica, add TLSmode=require to the JDBC URL that the scheduler uses. The easiest way to add this is to use the scheduler's --jdbc-url option. Assuming that you use a configuration file for your scheduler, you can add this line to it:

```
--jdbc-url=jdbc:vertica://VerticaHost:portNumber/databaseName?user=username&password=password&TLSmode=require
```

For more information about using the JDBC with Vertica, see Java .

Enable TLS in the scheduler configuration

Lastly, enable TLS. Every time you run vkconfig , you must pass it the following options:

--enable-ssl

true , to enable the scheduler to use SSL when connecting to Kafka.

--ssl-ca-alias

Alias for the CA you used to sign your Kafka broker's keys. This must match the value you supplied to the -alias argument of the keytool command to import the CA into the truststore.

> **Note**
> If you used more than one CA to sign keys, omit this option to import all of the CAs into the truststore.

--ssl-key-alias

Alias assigned to the schedule key. This value must match the value you supplied to the -alias you supplied to the keytool command when creating the scheduler's keystore.

--ssl-key-password

Password for the scheduler key.

See Common vkconfig script options for details of these options. For convenience and security, add these options to a configuration file that you pass to vkconfig. Otherwise, you run the risk of exposing the key password via the process list which can be viewed by other users on the same system. See Configuration File Format for more information on setting up a configuration file.

Add the following to the scheduler configuration file to allow it to use the keystore and truststore and enable TLS when connecting to Vertica:

```
enable-ssl=true
ssl-ca-alias=CAroot
ssl-key-alias=vsched
ssl-key-password=vertica
jdbc-url=jdbc:vertica://VerticaHost:portNumber/databaseName?user=username&password=password&TLSmode=require
```

Start the scheduler

Once you have configured the scheduler to use SSL, start it and verify that it can load data. For example, to start the scheduler with a configuration file named weblog.conf , use the command:

```
$ nohup vkconfig launch --conf weblog.conf >/dev/null 2>&1 &
```

Troubleshooting Kafka TLS/SSL connection issues

After configuring Vertica, Kafka, and your scheduler to use TLS/SSL authentication and encryption, you may encounter issues with data streaming. This section explains some of the more common errors you may encounter, and how to trouble shoot them.

Errors when launching the scheduler

You may see errors like this when launching the scheduler:

```
$ vkconfig launch --conf weblog.conf
java.sql.SQLNonTransientException: com.vertica.solutions.kafka.exception.ConfigurationException:
      No keystore system property found: null
   at com.vertica.solutions.kafka.util.SQLUtilities.getConnection(SQLUtilities.java:181)
   at com.vertica.solutions.kafka.cli.CLIUtil.assertDBConnectionWorks(CLIUtil.java:40)
   at com.vertica.solutions.kafka.Launcher.run(Launcher.java:135)
   at com.vertica.solutions.kafka.Launcher.main(Launcher.java:263)
Caused by: com.vertica.solutions.kafka.exception.ConfigurationException: No keystore system property found: null
   at com.vertica.solutions.kafka.security.KeyStoreUtil.loadStore(KeyStoreUtil.java:77)
   at com.vertica.solutions.kafka.security.KeyStoreUtil.<init>(KeyStoreUtil.java:42)
   at com.vertica.solutions.kafka.util.SQLUtilities.getConnection(SQLUtilities.java:179)
   ... 3 more
```

The scheduler throws these errors when it cannot locate or read the keystore or truststore files. To resolve this issue:

- Verify you have set the VKCONFIG_JVM_OPTS Linux environment variable. Without this variable, the scheduler will not know where to find the truststore and keystore to use when creating TLS/SSL connections. See Step 2: Set the VKCONFIG_JVM_OPTS Environment Variable for more information.
- Verify that the keystore and truststore files are located in the path you set in the VKCONFIG_JVM_OPTS environment variable.
- Verify that the user account that runs the scheduler has read access to the trustore and keystore files.
- Verify that the key password you provide in the scheduler configuration is correct. Note that you must supply the password for the key, not the keystore.

Another possible error message is a failure to set up a TLS Keystore:

```
Exception in thread "main" java.sql.SQLRecoverableException: [Vertica][VJDBC](100024) IOException while communicating with server: java.io.IOException: Failed to create an SSLSocketFactory when setting up TLS: keystore not found.
    at com.vertica.io.ProtocolStream.logAndConvertToNetworkException(Unknown Source)
    at com.vertica.io.ProtocolStream.enableSSL(Unknown Source)
    at com.vertica.io.ProtocolStream.initSession(Unknown Source)
    at com.vertica.core.VConnection.tryConnect(Unknown Source)
    at com.vertica.core.VConnection.connect(Unknown Source)
    . . .
```

This error can be caused by using a keystore or truststore file in a format other than JKS and not supplying the correct file extension. If the scheduler does not recognize the file extension of your keystore or truststore file name, it assumes the file is in JKS format. If the file isn't in this format, the scheduler will exit with the error message shown above. To correct this error, rename the keystore and truststore files to use the correct file extension. For example, if your files are in PKCS 12 filemat, change their file extension to .p12 or .pks .

Data does not load

If you find that scheduler is not loading data into your database, you should first query the stream_microbatch_history table to determine whether the scheduler is executing microbatches, and if so, what their results are. A faulty TLS/SSL configuration usually results in a status of NETWORK_ISSUE:

```
=> SELECT frame_start, end_reason, end_reason_message FROM weblog_sched.stream_microbatch_history;
     frame_start       |  end_reason   | end_reason_message
-------------------------+---------------+--------------------
 2019-04-05 11:35:18.365 | NETWORK_ISSUE |
 2019-04-05 11:35:38.462 | NETWORK_ISSUE |
```

If you suspect an SSL issue, you can verify that Vertica is establishing a connection to Kafka by looking at Kafka's server.log file. Failed SSL connection attempts can appear in this log like this example:

```
java.io.IOException: Unexpected status returned by SSLEngine.wrap, expected
    CLOSED, received OK. Will not send close message to peer.
    at org.apache.kafka.common.network.SslTransportLayer.close(SslTransportLayer.java:172)
    at org.apache.kafka.common.utils.Utils.closeAll(Utils.java:703)
    at org.apache.kafka.common.network.KafkaChannel.close(KafkaChannel.java:61)
    at org.apache.kafka.common.network.Selector.doClose(Selector.java:739)
    at org.apache.kafka.common.network.Selector.close(Selector.java:727)
    at org.apache.kafka.common.network.Selector.pollSelectionKeys(Selector.java:520)
    at org.apache.kafka.common.network.Selector.poll(Selector.java:412)
    at kafka.network.Processor.poll(SocketServer.scala:551)
    at kafka.network.Processor.run(SocketServer.scala:468)
    at java.lang.Thread.run(Thread.java:748)
```

If you do not see errors of this sort, you likely have a network problem between Kafka and Vertica. If you do see these errors, consider the following debugging steps:

- Verify that the configuration of your Kafka cluster is uniform. For example, you may see connection errors if some Kafka nodes are set to require client authentication and others aren't.
- Verify that the Common Names (CN) in the certificates and keys match the host name of the system.
- Verify that the Kafka cluster is accepting connections on the ports and host names you specify in the server.properties file's listeners property. For example, suppose you use IP addresses in this setting, but use host names when defining the cluster in the scheduler's configuration. Then Kafka may reject the connection attempt by Vertica or Vertica may reject the Kafka node's identity.
- If you are using client authentication in Kafka, try turning it off to see if the scheduler can connect. If disabling authentication allows the scheduler to stream data, then you can isolate the problem to client authentication. In this case, review the certificates and CAs of both the Kafka cluster and the scheduler. Ensure that the truststores include all of the CAs used to sign the key, up to and including the root CA.

Avro schema registry and KafkaAvroParser

At minimum, the KafkaAvroParser requires the following parameters to create a TLS connection between the Avro Schema Registry and Vertica:

- schema_registry_url with the https scheme
- schema_registry_ssl_ca_path

If and only if TLS access fails, determine what TLS schema registry information Vertica requires from the following:

- Certificate Authority (CA)

- TLS server certificate
- TLS key

Provide only the necessary TLS schema registry information with KafkaAvroParser parameters. The TLS information must be accessible in the filesystem of each Vertica node that processes Avro data.

The following example shows how to pass these parameters to KafkaAvroParser:

```
KafkaAvroParser(
   schema_registry_url='https://localhost:8081'
   schema_registry_ssl_ca_path='path/to/certificate-authority',
   schema_registry_ssl_cert_path='path/to/tls-server-certificate',
   schema_registry_ssl_key_path='path/to/private-tls-key',
   schema_registry_ssl_key_password_path='path/to/private-tls-key-password'
)
```

Authenticating with Kafka using SASL

Kafka supports using Simple Authentication and Security Layer (SASL) to authenticate producers and consumers. You can use SASL to authenticate Vertica with Kafka when using most of the Kafka-related functions such as KafkaSource .

Vertica supports using the SASL_PLAINTEXT and SASL_SSL protocols with the following authentication mechanisms:

- PLAIN
- SCRAM-SHA-256
- SCRAM-SHA-512

You must configure your Kafka cluster to enable SASL authentication. See the Kafka documentation for your Kafka version to learn how to configure SASL authentication.

> **Note**
> KafkaExport does not support using TLS/SSL with SASL authentication at this time.

To use SASL authentication between Vertica and Kafka, directly set SASL-related configuration options in the rdkafka library using the kafka_conf parameter. Vertica uses this library to connect to Kafka. See Directly setting Kafka library options for more information on directly setting configuration options in the rdkafka library.

Among the relevant configuration options are:

- security.protocol sets the security protocol to use to authenticate with Kafka.
- sasl.mechanism sets the security mechanism.
- sasl.username sets the SASL user to use for authentication.
- sasl.password sets the password to use for SASL authentication.

See the rdkafka configuration documentation for a list of all the SASL-related settings.

The following example demonstrates calling KafkaCheckBrokers using the SASL_PLAINTEXT security protocol:

```
=> SELECT KafkaCheckBrokers(USING PARAMETERS
    brokers='kafka01.example.com:9092',
    kafka_conf='{"sasl.username":"dbadmin", "sasl.mechanism":"PLAIN", "security.protocol":"SASL_PLAINTEXT"}',
    kafka_conf_secret='{"sasl.password":"password"}'
  ) OVER ();
```

This example demonstrates using SASL authentication when copying data from Kafka via an SSL connection. This example assumes that Vertica and Kafka have been configures to use TLS/SSL encryption as described in TLS/SSL encryption with Kafka :

```
=> COPY mytopic_table
    SOURCE KafkaSource(
      stream='mytopic|0|-2',
      brokers='kafka01.example.com:9092',
      stop_on_eof=true,
      kafka_conf='{"sasl.username":"dbadmin", "sasl.password":"pword", "sasl.mechanism":"PLAIN", "security.protocol":"SASL_SSL"}'
    )
    FILTER KafkaInsertDelimiters(delimiter = E'\n')
    DELIMITER ','
    ENCLOSED BY '"';
```

For more information about using SASL with the rfkafka library, see Using SASL with librdkafka on the rdkafka github site.

Troubleshooting Kafka integration issues

The following topics can help you troubleshoot issues integrating Vertica with Apache Kafka.

In this section

- Using kafkacat to troubleshoot Kafka integration issues
- Troubleshooting slow load speeds
- Troubleshooting missing messages

Using kafkacat to troubleshoot Kafka integration issues

Kafkacat is a third-party open-source utility that lets you connect to Kafka from the Linux command line. It uses the same underlying library that the Vertica integration for Apache Kafka uses to connect to Kafka. This shared library makes kafkcat a useful tool for testing and debugging your Vertica integration with Kafka.

You may find kafkacat useful for:

- Testing connectivity between your Vertica and Kafka clusters.
- Examining Kafka data for anomalies that may prevent some of it from loading into Vertica.
- Producing data for test loads into Vertica.
- Listing details about a Kafka topic.

For more information about kafkacat, see its project page at Github .

Running kafkacat on Vertica nodes

The kafkacat utility is bundled in the Vertica install package, so it is available on all nodes of your Vertica cluster in the /opt/vertica/packages/kafka/bin directory. This is the same directory containing the vkconfig utility, so if you have added it to your path, you can use the kafkacat utility without specifying its full path. Otherwise, you can add this path to your shell's environment variable using the command:

```
set PATH=/opt/vertica/packages/kafka/bin:$PATH
```

> **Note**
> On Debian and Ubuntu systems, you must tell kafkacat to use Vertica's own copy of the SSL libraries by setting the LD_LIBRARY_PATH environment variable:
>
> ```
> $ export LD_LIBRARY_PATH=/opt/vertica/lib
> ```
>
> If you do not set this environment variable, the kafkcat utility exits with the error:
>
> ```
> kafkacat: error while loading shared libraries: libcrypto.so.10:
>       cannot open shared object file: No such file or directory
> ```

Executing kafkacat without any arguments gives you a basic help message:

```
$ kafkacat
Error: -b <broker,..> missing

Usage: kafkacat <options> [file1 file2 .. | topic1 topic2 ..]]
kafkacat - Apache Kafka producer and consumer tool
```

https://github.com/edenhill/kafkacat
Copyright (c) 2014-2015, Magnus Edenhill
Version releases/VER_8_1_RELEASE_BUILD_1_555_20170615-4931-g3fb918 (librdkafka releases/VER_8_1_RELEASE_BUILD_1_555_20170615-4931-g3fb918)

General options:
  -C | -P | -L   Mode: Consume, Produce or metadata List
  -G <group-id>   Mode: High-level KafkaConsumer (Kafka 0.9 balanced consumer groups)
                     Expects a list of topics to subscribe to
  -t <topic>   Topic to consume from, produce to, or list
  -p <partition>   Partition
  -b <brokers,..>   Bootstrap broker(s) (host[:port])
  -D <delim>   Message delimiter character:
                     a-z.. | \r | \n | \t | \xNN
                     Default: \n
  -K <delim>   Key delimiter (same format as -D)
  -c <cnt>   Limit message count
  -X list   List available librdkafka configuration properties
  -X prop=val   Set librdkafka configuration property.
                     Properties prefixed with "topic." are
                     applied as topic properties.
  -X dump   Dump configuration and exit.
  -d <dbg1,...>   Enable librdkafka debugging:
                     all,generic,broker,topic,metadata,queue,msg,protocol,cgrp,security,fetch,feature
  -q   Be quiet (verbosity set to 0)
  -v   Increase verbosity
  -V   Print version

Producer options:
  -z snappy|gzip   Message compression. Default: none
  -p -1   Use random partitioner
  -D <delim>   Delimiter to split input into messages
  -K <delim>   Delimiter to split input key and message
  -l   Send messages from a file separated by
                     delimiter, as with stdin.
                     (only one file allowed)
  -T   Output sent messages to stdout, acting like tee.
  -c <cnt>   Exit after producing this number of messages
  -Z   Send empty messages as NULL messages
  file1 file2..   Read messages from files.
                     With -l, only one file permitted.
                     Otherwise, the entire file contents will
                     be sent as one single message.

Consumer options:
  -o <offset>   Offset to start consuming from:
                     beginning | end | stored |
                     <value> (absolute offset) |
                     -<value> (relative offset from end)
  -e   Exit successfully when last message received
  -f <fmt..>   Output formatting string, see below.
                     Takes precedence over -D and -K.
  -D <delim>   Delimiter to separate messages on output
  -K <delim>   Print message keys prefixing the message
                     with specified delimiter.
  -O   Print message offset using -K delimiter
  -c <cnt>   Exit after consuming this number of messages
  -Z   Print NULL messages and keys as "NULL"(instead of empty)
  -u   Unbuffered output

```
Metadata options:
 -t <topic>      Topic to query (optional)


Format string tokens:
 %s              Message payload
 %S              Message payload length (or -1 for NULL)
 %R              Message payload length (or -1 for NULL) serialized
                 as a binary big endian 32-bit signed integer
 %k              Message key
 %K              Message key length (or -1 for NULL)
 %t              Topic
 %p              Partition
 %o              Message offset
 \n \r \t        Newlines, tab
 \xXX \xNNN      Any ASCII character
Example:
 -f 'Topic %t [%p] at offset %o: key %k: %s\n'


Consumer mode (writes messages to stdout):
 kafkacat -b <broker> -t <topic> -p <partition>
or:
 kafkacat -C -b ...

High-level KafkaConsumer mode:
 kafkacat -b <broker> -G <group-id> topic1 top2 ^aregex\d+

Producer mode (reads messages from stdin):
 ... | kafkacat -b <broker> -t <topic> -p <partition>
or:
 kafkacat -P -b ...

Metadata listing:
 kafkacat -L -b <broker> [-t <topic>]
```

Testing connectivity to a Kafka cluster and getting metadata

One basic troubleshooting step you often need to perform is verifying that Vertica nodes can connect to the Kafka cluster. Successfully executing just about any kafkacat command will prove the Vertica node you are logged into is able to reach the Kafka cluster. One simple command you can execute to verify connectivity is to get the metadata for all of the topics the Kafka cluster has defined. The following example demonstrates using kafkacat's metadata listing command to connect to the broker named kafka01 running on port 6667 (the Kafka broker port used by Hortonworks Hadoop clusters).

```
$ kafkacat -L -b kafka01:6667
Metadata for all topics (from broker -1: kafka01:6667/bootstrap):
 2 brokers:
  broker 1001 at kafka03.example.com:6667
  broker 1002 at kafka01.example.com:6667
 4 topics:
  topic "iot-data" with 3 partitions:
    partition 2, leader 1002, replicas: 1002, isrs: 1002
    partition 1, leader 1001, replicas: 1001, isrs: 1001
    partition 0, leader 1002, replicas: 1002, isrs: 1002
  topic "__consumer_offsets" with 50 partitions:
    partition 23, leader 1001, replicas: 1002,1001, isrs: 1001,1002
    partition 41, leader 1001, replicas: 1002,1001, isrs: 1001,1002
    partition 32, leader 1002, replicas: 1002,1001, isrs: 1001,1002
    partition 8, leader 1002, replicas: 1002,1001, isrs: 1001,1002
    partition 17, leader 1001, replicas: 1002,1001, isrs: 1001,1002
    partition 44, leader 1002, replicas: 1002,1001, isrs: 1001,1002
    partition 35, leader 1001, replicas: 1002,1001, isrs: 1001,1002
    partition 26, leader 1002, replicas: 1002,1001, isrs: 1001,1002
```

```
  partition 11, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 29, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 38, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 47, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 20, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 2, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 5, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 14, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 46, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 49, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 40, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 4, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 13, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 22, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 31, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 16, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 7, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 43, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 25, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 34, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 10, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 37, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 1, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 19, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 28, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 45, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 36, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 27, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 9, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 18, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 21, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 48, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 12, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 3, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 30, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 39, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 15, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 42, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 24, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 33, leader 1001, replicas: 1002,1001, isrs: 1001,1002
  partition 6, leader 1002, replicas: 1002,1001, isrs: 1001,1002
  partition 0, leader 1002, replicas: 1002,1001, isrs: 1001,1002
topic "web_hits" with 1 partitions:
  partition 0, leader 1001, replicas: 1001, isrs: 1001
topic "ambari_kafka_service_check" with 1 partitions:
  partition 0, leader 1002, replicas: 1002, isrs: 1002
```

You can also use this output to verify the topics defined by your Kafka cluster, as well as the number of partitions each topic defines. You need this information when copying data between Kafka and Vertica.

Retrieving messages from a Kafka topic

When you are troubleshooting issues with streaming messages from Kafka in Vertica, you often want to look at the raw data that Kafka sent. For example, you may want to verify that the messages are in the format that your expect. Or, you may want to review specific messages to see if some of them weren't in the right format for Vertica to parse. You can use kafkacat to read messages from a topic using its consume command (-C). At the very least, you must pass kafkacat the brokers (-b argument) and the topic you want to read from (-t). You can also choose to read messages from a specific offset (-o) and partition (-p). You will usually also want kafkacat to exit after completing the data read (-e) instead continuing to wait for more messages.

This example gets the last message in the topic named web_hits. The offset argument uses a negative value, which tells kafkacat to read from the end of the topic.

```
$ kafkacat -C -b kafka01:6667 -t web_hits -o -1 -e
{"url": "wp-content/list/search.php", "ip": "132.74.240.52",
"date": "2018/03/28 14:12:34",
"user_agent": "Mozilla/5.0 (iPod; U; CPU iPhone OS 4_2 like
Mac OS X; sl-SI) AppleWebKit/532.22.4 (KHTML, like Gecko) Version/3.0.5
Mobile/8B117 Safari/6532.22.4"}
% Reached end of topic web_hits [0] at offset 54932: exiting
```

You can also read a specific range of messages by specifying an offset and a limit (-c argument). For example, you may want to look at a specific range of data to determine why Vertica could not load it. The following example reads 10 messages from the topic iot-data starting at offset 3280:

```
$ kafkacat -C -b kafka01:6667 -t iot-data -o 3280 -c 10 -e
63680, 19, 24.439323, 26.0128725
43510, 71, 162.319805, -37.4924025
91113, 53, 139.764857, -74.735731
88508, 14, -85.821967, -7.236280
20419, 31, -129.583988, 13.995481
79365, 79, 153.184594, 51.1583485
26283, 25, -168.911020, 35.331027
32111, 56, -157.930451, 82.2676385
56808, 17, 19.603286, -0.7698495
9118, 73, 97.365445, -74.8593245
```

Generating data for a Kafka topic

If you are preparing to stream data from a Kafka topic that is not yet active, you may want a way to stream test messages. You can then verify that the topic's messages load into Vertica without worrying that you will miss actual data.

To send data to Kafka, use kafkacat's produce command (-P). The easiest way to supply it with messages is to pipe them in via STDIN, one message per line. You can choose a specific partition for the data, or have kafkacat randomly assign each message to a random partition by setting the partition number to -1. For example, suppose you have a file named iot-data.csv that you wanted to produce to random partitions of a Kafka topic named iot-data. Then you could use the following command:

```
$ cat iot_data.csv | kafkacat -P -p -1 -b kafka01:6667 -t iot-data
```

Troubleshooting slow load speeds

Here are some potential issues that could cause messages to load slowly from Kafka.

Verify you have disabled the API version check when communicating with Kafka 0.9 or earlier

If your Kafka cluster is running 0.9 or earlier, be sure you have disable the rdkafka library's api.version.request option. If you do not, every Vertica connection to Kafka will pause for 10 seconds until the API version request times out. Depending on the frame size of your load or other timeout settings, this delay can either reduce the throughput of your data loads. It can even totally prevent messages from being loaded. See Configuring Vertica for Apache Kafka version 0.9 and earlier for more information.

Eon Mode and cloud latency

Eon Mode separates compute from storage in a Vertica cluster, which can cause a small amount of latency when Vertica loads and saves data. Cloud computing infrastructure can also cause latency. This latency can eat into the frame duration for your schedulers, causing them to load less data in each frame. For this reason, you should consider increasing frame durations when loading data from Kafka in an Eon Mode database. See Vertica Eon Mode and Kafka for more information.

Troubleshooting missing messages

Kafka producers that emit a high data volume might overwhelm Vertica, possibly resulting in messages expiring in Kafka before the scheduler loads them into Vertica. This is more common when Vertica performs additional processing on the loaded messages, such as text indexing.

If you see that you are missing messages from a topic with multiple partitions, consider configuring the --max-parallelism microbatch utility option . The --max-parallelism option splits a microbatch into multiple subset microbatches. This enables you to use PLANNEDCONCURRENCY available in the scheduler's resource pool to create more scheduler threads for simultaneous loads of a single microbatch. Each node uses the resource pool EXECUTIONPARALLELISM setting to determine the number of threads created to process partitions. Because EXECUTIONPARALLELISM threads are created per scheduler thread, using more PLANNEDCONCURRENCY per microbatch enables you to process more partitions in parallel for a single unit of work.

For details, see Managing scheduler resources and performance .

vkconfig script options

Vertica includes the vkconfig script that lets you configure your schedulers. This script contains multiple tools that set groups of options in the scheduler, as well as starting and shutting it down. You supply the tool you want to use as the first argument in your call to the vkconfig script.

The topics in this section explain each of the tools available in the vkconfig script as well as their options. You can use the options in the Common vkconfig script options topic with any of the utilities. Utility-specific options appear in their respective tables.

In this section

- Common vkconfig script options
- Scheduler tool options
- Cluster tool options
- Source tool options
- Target tool options
- Load spec tool options
- Microbatch tool options
- Launch tool options
- Shutdown tool options
- Statistics tool options
- Sync tool options

Common vkconfig script options

These options are available across the different tools available in the vkconfig script.

--conf ***filename***

A text file containing configuration options for the vkconfig script. See Configuration File Format below.

--config-schema ***schema_name***

The name of the scheduler's Vertica schema. This value is the same as the name of the scheduler. You use this name to identify the scheduler during configuration.

Default:
stream_config

--dbhost ***host name***

The host name or IP address of the Vertica node acting as the initiator node for the scheduler.

Default:
localhost

--dbport ***port_number***

The port to use to connect to a Vertica database.

Default:
5433

--enable-ssl

Enables the vkconfig script to use SSL to connect to Vertica or between Vertica and Kafka. See Configuring your scheduler for TLS connections for more information.

--help

Prints out a help menu listing available options with a description.

--jdbc-opt ***option*** **=** ***value*** **[&** ***option2*** **=** ***value2*** **...]**

One or more options to add to the standard JDBC URL that vkconfig uses to connect to Vertica. Cannot be combined with --jdbc-url .

--jdbc-url ***url***

A complete JDBC URL that vkconfig uses instead of standard JDBC URL string to connect to Vertica.

--password ***password***

Password for the database user.

--ssl-ca-alias ***alias_name***

The alias of the root certificate authority in the truststore. When set, the scheduler loads only the certificate associated with the specified alias. When omitted, the scheduler loads all certificates into the truststore.

--ssl-key-alias *alias_name*

The alias of the key and certificate pairs within the keystore. Must be set when Vertica uses SSL to connect to Kafka.

--ssl-key-password *password*

The password for the SSL key. Must be set when Vertica uses SSL to connect to Kafka.

> **Caution**
> Specifying this option on the command line can expose it to other users logged into the host. Always use a configuration file to set this option.

--username *username*

The Vertica database user used to alter the configuration of the scheduler. This use must have create privileges on the scheduler's schema.

Default:
Current user

--version

Displays the version number of the scheduler.

Configuration file format

You can use a configuration file to store common parameters you use in your calls to the vkconfig utility. The configuration file is a text file containing one option setting per line in the format:

```
option=value
```

You can also include comments in the option file by prefixing them with a hash mark (#).

```
#config.properties:
username=myuser
password=mypassword
dbhost=localhost
dbport=5433
```

You tell vkconfig to use the configuration file using the --conf option:

```
$ /opt/vertica/packages/kafka/bin/vkconfig source --update --conf config.properties
```

You can override any stored parameter from the command line:

```
$ /opt/vertica/packages/kafka/bin/vkconfig source --update --conf config.properties --dbhost otherVerticaHost
```

Examples

These examples show how you can use the shared utility options.

Display help for the scheduler utility:

```
$ vkconfig scheduler --help
This command configures a Scheduler, which can run and load data from configured
sources and clusters into Vertica tables. It provides options for changing the
'frame duration' (time given per set of batches to resolve), as well as the
dedicated Vertica resource pool the Scheduler will use while running.

Available Options:
PARAMETER             #ARGS    DESCRIPTION
conf                  1        Allow the use of a properties file to associate
                               parameter keys and values. This file enables
                               command string reuse and cleaner command strings.
help                  0        Outputs a help context for the given subutility.
version               0        Outputs the current Version of the scheduer.
skip-validation       0        [Depricated] Use --validation-type.
validation-type       1        Determine what happens when there are
                               configuration errors. Accepts: ERROR - errors
                               out, WARN - prints out a message and continues,
                               SKIP - skip running validations
```

		SKIP skip running validations
dbhost	1	The Vertica database hostname that contains metadata and configuration information. The default value is 'localhost'.
dbport	1	The port at the hostname to connect to the Vertica database. The default value is '5433'.
username	1	The user to connect to Vertica. The default value is the current system user.
password	1	The password for the user connecting to Vertica. The default value is empty.
jdbc-url	1	A JDBC URL that can override Vertica connection parameters and provide additional JDBC options.
jdbc-opt	1	Options to add to the JDBC URL used to connect to Vertica ('&'-separated key=value list). Used with generated URL (i.e. not with '--jdbc-url' set).
enable-ssl	1	Enable SSL between JDBC and Vertica and/or Vertica and Kafka.
ssl-ca-alias	1	The alias of the root CA within the provided truststore used when connecting between Vertica and Kafka.
ssl-key-alias	1	The alias of the key and certificate pair within the provided keystore used when connecting between Vertica and Kafka.
ssl-key-password	1	The password for the key used when connecting between Vertica and Kafka. Should be hidden with file access (see --conf).
config-schema	1	The schema containing the configuration details to be used, created or edited. This parameter defines the scheduler. The default value is 'stream_config'.
create	0	Create a new instance of the supplied type.
read	0	Read an instance of the supplied type.
update	0	Update an instance of the supplied type.
delete	0	Delete an instance of the supplied type.
drop	0	Drops the specified configuration schema. CAUTION: this command will completely delete and remove all configuration and monitoring data for the specified scheduler.
dump	0	Dump the config schema query string used to answer this command in the output.
operator	1	Specifies a user designated as an operator for the created configuration. Used with --create.
add-operator	1	Add a user designated as an operator for the specified configuration. Used with --update.
remove-operator	1	Removes a user designated as an operator for the specified configuration. Used with --update.
upgrade	0	Upgrade the current scheduler configuration schema to the current version of this scheduler. WARNING: if upgrading between EXCAVATOR and FRONTLOADER be aware that the Scheduler is not backwards compatible. The upgrade procedure will translate your kafka model into the new stream model.
upgrade-to-schema	1	Used with upgrade: will upgrade the configuration to a new given schema instead of upgrading within the same schema.
fix-config	0	Attempts to fix the configuration (ex: dropped tables) before doing any other updates. Used with --update.
frame-duration	1	The duration of the Scheduler's frame, in

		which every configured Microbatch runs. Default is 300 seconds: '00:05:00'
resource-pool	1	The Vertica resource pool to run the Scheduler on. Default is 'general'.
config-refresh	1	The interval of time between Scheduler configuration refreshes. Default is 5 minutes: '00:05'
new-source-policy	1	The policy for new Sources to be scheduled during a frame. Options are: START, END, and FAIR. Default is 'FAIR'.
pushback-policy	1	
pushback-max-count	1	
auto-sync	1	Automatically update configuration based on metadata from the Kafka cluster
consumer-group-id	1	The Kafka consumer group id to report offsets to.
eof-timeout-ms	1	[DEPRECATED] This option has no effect.

Scheduler tool options

The vkconfig script's scheduler tool lets you configure schedulers that continuously loads data from Kafka into Vertica. Use the scheduler tool to create, update, or delete a scheduler, defined by config-schema . If you do not specify a scheduler, commands apply to the default stream_config scheduler.

Syntax

```
vkconfig scheduler {--create | --read | --update | --drop} other_options...
```

--create

Creates a new scheduler. Cannot be used with --delete , --read , or --update .

--read

Outputs the current setting of the scheduler in JSON format. Cannot be used with --create , --delete , or --update .

--update

Updates the settings of the scheduler. Cannot be used with --create , --delete , or --read .

--drop

Drops the scheduler's schema. Dropping its schema deletes the scheduler. After you drop the scheduler's schema, you cannot recover it.

--add-operator *user_name*

Grants a Vertica user account or role access to use and alter the scheduler. Requires the --update shared utility option.

--auto-sync {TRUE|FALSE}

When TRUE, Vertica automatically synchronizes scheduler source information at the interval specified in --config-refresh .

For details about what the scheduler synchronizes at each interval, see the "Validating Schedulers" and "Synchronizing Schedulers" sections in Automatically consume data from Kafka with the scheduler .

Default: TRUE

--config-refresh *HH:MM:SS*

The interval of time that the scheduler runs before synchronizing its settings and updating its cached metadata (such as changes made by using the --update option).

Default: 00:05:00

--consumer-group-id *id_name*

The name of the Kafka consumer group to which Vertica reports its progress consuming messages. Set this value to disable progress reports to a Kafka consumer group. For details, see Monitoring Vertica message consumption with consumer groups .

Default: vertica_ *database-name*

--dump

When you use this option along with the --read option, vkconfig outputs the Vertica query it would use to retrieve the data, rather than outputting the data itself. This option is useful if you want to access the data from within Vertica without having to go through vkconfig. This option has no effect if not used with --read .

--eof-timeout-ms ***number of milliseconds***

If a COPY command does not receive any messages within the eof-timeout-ms interval, Vertica responds by ending that COPY statement.

See Manually consume data from Kafka for more information.
Default: 1 second

--fix-config

Repairs the configuration and re-creates any missing tables. Valid only with the --update shared configuration option.

--frame-duration ***HH:MM:SS***

The interval of time that all individual frames last with this scheduler. The scheduler must have enough time to run each microbatch (each of which execute a COPY statement). You can approximate the average available time per microbatch using the following equation:

TimePerMicrobatch=(*FrameDuration** *Parallelism*)/*Microbatches*

This is just a rough estimate as there are many factors that impact the amount of time that each microbatch will be able to run.
The vkconfig utility warns you if the time allocated per microbatch is below 2 seconds. You usually should allocate more than two seconds per microbatch to allow the scheduler to load all of the data in the data stream.

> **Note**
> In versions of Vertica earlier than 10.0, the default frame duration was 10 seconds. In version 10.0, this default value was increased to 5 minutes in part to compensate for the removal of WOS. If you created your scheduler with the default frame duration in a version prior to 10.0, the frame duration is not updated to the new default value. In this case, consider adjusting the frame duration manually. See Choosing a frame duration for more information.

Default: 00:05:00

--message_max_bytes ***max_message_size***

Specifies the maximum size, in bytes, of a Kafka protocol batch message.
Default: 25165824

--new-source-policy {FAIR|START|END}

Determines how Vertica allocates resources to the newly added source, one of the following:

- FAIR: Takes the average length of time from the previous batches and schedules itself appropriately.
- START: All new sources start at the beginning of the frame. The batch receives the minimal amount of time to run.
- END: All new sources start at the end of the frame. The batch receives the maximum amount of time to run.

Default: FAIR

--operator ***username***

Allows the dbadmin to grant privileges to a previously created Vertica user or role.

This option gives the specified user all privileges on the scheduler instance and EXECUTE privileges on the libkafka library and all its UDxs. Granting operator privileges gives the user the right to read data off any source in any cluster that can be reached from the Vertica node.

The dbadmin must grant the user separate permission for them to have write privileges on the target tables.

Requires the --create shared utility option. Use the --add-operator option to grant operate privileges after the scheduler has been created.

To revoke privileges, use the --remove-operator option.

--remove-operator ***user_name***

Removes access to the scheduler from a Vertica user account. Requires the --update shared utility option.

--resource-pool ***pool_name***

The resource pool to be used by all queries executed by this scheduler. You must create this pool in advance.

Default: GENERAL pool

> **Note**

> The scheduler can use only one-fourth of GENERAL pool's PLANNEDCONCURRENCY .

--upgrade

Upgrades the existing scheduler and configuration schema to the current Vertica version. The upgraded version of the scheduler is not backwards compatible with earlier versions. To upgrade a scheduler to an alternate schema, use the upgrade-to-schema parameter. See Updating schedulers after Vertica upgrades for more information.

--upgrade-to-schema *schema name*

Copies the scheduler's schema to a new schema specified by *schema name* and then upgrades it to be compatible with the current version of Vertica. Vertica does not alter the old schema. Requires the --upgrade scheduler utility option.

--validation-type {ERROR|WARN|SKIP}

Renamed from --skip-validation , specifies the level of validation performed on the scheduler. Invalid SQL syntax and other errors can cause invalid microbatches. Vertica supports the following validation types:

- ERROR: Cancel configuration or creation if validation fails.
- WARN: Proceed with task if validation fails, but display a warning.
- SKIP: Perform no validation.

For more information on validation, refer to Automatically consume data from Kafka with the scheduler .
Default: ERROR

See Common vkconfig script options for options that are available in all of the vkconfig tools.

Examples

These examples show how you can use the scheduler utility options.

Give a user, Jim, privileges on the StreamConfig scheduler. Specify that you are making edits to the stream_config scheduler with the --config-schema option:

```
$ /opt/vertica/packages/kafka/bin/vkconfig scheduler --update --config-schema stream_config --add-operator Jim
```

Edit the default stream_config scheduler so that every microbatch waits for data for one second before ending:

```
$ /opt/vertica/packages/kafka/bin/vkconfig scheduler --update --eof-timeout-ms 1000
```

Upgrade the scheduler named iot_scheduler_8.1 to a new scheduler named iot_scheduler_9.0 that is compatible with the current version of Vertica:

```
$ /opt/vertica/packages/kafka/bin/vkconfig scheduler --upgrade --config-schema iot_scheduler_8.1 \
                                    --upgrade-to-schema iot_scheduler_9.0
```

Drop the schema scheduler219a:

```
$ /opt/vertica/packages/kafka/bin/vkconfig scheduler --drop --config-schema  scheduler219a --username dbadmin
```

Read the current setting of the options you can set using the scheduler tool for the scheduler defined in weblogs.conf.

```
$ vkconfig scheduler --read --conf weblog.conf
{"version":"v9.2.0", "frame_duration":"00:00:10", "resource_pool":"weblog_pool",
"config_refresh":"00:05:00", "new_source_policy":"FAIR",
"pushback_policy":"LINEAR", "pushback_max_count":5, "auto_sync":true,
"consumer_group_id":null}
```

Cluster tool options

The vkconfig script's cluster tool lets you define the streaming hosts your scheduler connects to.

Syntax

```
vkconfig cluster {--create | --read | --update | --delete} \
    [--cluster cluster_name] [other_options...]
```

--create

Creates a new cluster. Cannot be used with --delete , --read , or --update .

--read

Outputs the settings of all clusters defined in the scheduler. This output is in JSON format. Cannot be used with --create , --delete , or --update .

You can limit the output to specific clusters by supplying one or more cluster names in the --cluster option. You an also limit the output to clusters that contain one or more specific hosts using the --hosts option. Use commas to separate multiple values.
You can use LIKE wildcards in these options. See LIKE for more information about using wildcards.

--update

Updates the settings of *cluster_name* . Cannot be used with --create , --delete , or --read .

--delete

Deletes the cluster *cluster_name* . Cannot be used with --create , --read , or --update .

--dump

When you use this option along with the --read option, vkconfig outputs the Vertica query it would use to retrieve the data, rather than outputting the data itself. This option is useful if you want to access the data from within Vertica without having to go through vkconfig. This option has no effect if not used with --read .

--cluster *cluster_name*

A unique, case-insensitive name for the cluster to operate on. This option is required for --create , --update , and --delete .

--hosts *b1* : *port* [, *b2* : *port* ...]

Identifies the broker hosts that you want to add, edit, or remove from a Kafka cluster. To identify multiple hosts, use a comma delimiter.

--kafka_conf ' *kafka_configuration_setting* '

A JSON-formatted object of option/value pairs to pass directly to the rdkafka library. This is the library Vertica uses to communicate with Kafka. You can use this parameter to directly set configuration options that are not available through the Vertica integration with Kafka. See Directly setting Kafka library options for details.

--kafka_conf_secret ' *kafka_configuration_setting* '

Conceals sensitive configuration data that you must pass directly to the rdkafka library, such as passwords. This parameter accepts settings in the same format as kafka_conf .
Values passed to this parameter are not logged or stored in system tables.

--new-cluster *cluster_name*

The updated name for the cluster. Requires the --update shared utility option.

--validation-type {ERROR|WARN|SKIP}

Specifies the level of validation performed on a created or updated cluster:

- ERROR - Cancel configuration or creation if vkconfig cannot validate that the cluster exists. This is the default setting.
- WARN - Proceed with task if validation fails, but display a warning.
- SKIP - Perform no validation.

Renamed from --skip-validation .

See Common vkconfig script options for options that are available in all of the vkconfig tools.

Examples

This example shows how you can create the cluster, StreamCluster1, and assign two hosts:

```
$ /opt/vertica/packages/kafka/bin/vkconfig cluster --create --cluster StreamCluster1 \
                                    --hosts 10.10.10.10:9092,10.10.10.11:9092
                                    --conf myscheduler.config
```

This example shows how you can list all of the clusters associated with the scheduler defined in the weblogs.conf file:

```
$ vkconfig cluster --read --conf weblog.conf
{"cluster":"kafka_weblog",
"hosts":"kafka01.example.com:9092,kafka02.example.com:9092"}
```

Source tool options

Use the vkconfig script's source tool to create, update, or delete a source.

Syntax

```
vkconfig source {--create | --read | --update | --delete} \
      --source source_name [other_options...]
```

--create

Creates a new source. Cannot be used with --delete , --read , or --update .

--read

Outputs the current setting of the sources defined in the scheduler. The output is in JSON format. Cannot be used with --create , --delete , or --update .

By default this option outputs all of the sources defined in the scheduler. You can limit the output by using the --cluster , --enabled , --partitions , and --source options. The output will only contain sources that match the values in these options. The --enabled option can only have a true or false value. The --source option is case-sensitive.
You can use LIKE wildcards in these options. See LIKE for more information about using wildcards.

--update

Updates the settings of *source_name* . Cannot be used with --create , --delete , or --read .

--delete

Deletes the source named *source_name* . Cannot be used with --create , --read , or --update .

--source *source_name*

Identifies the source to create or alter in the scheduler's configuration. This option is case-sensitive. You can use any name you like for a new source. Most people use the name of the Kafka topic the scheduler loads its data from. This option is required for --create , --update , and --delete .

--cluster *cluster_name*

Identifies the cluster containing the source that you want to create or edit. You must have already defined this cluster in the scheduler.

--dump

When you use this option along with the --read option, vkconfig outputs the Vertica query it would use to retrieve the data, rather than outputting the data itself. This option is useful if you want to access the data from within Vertica without having to go through vkconfig. This option has no effect if not used with --read .

--enabled TRUE|FALSE

When TRUE, the source is available for use.

--new-cluster *cluster_name*

Changes the cluster this source belongs to.

All sources referencing the old cluster source now target this cluster.
Requires: --update and --source options

--new-source *source_name*

Updates the name of an existing source to the name specified by this parameter.

Requires: --update shared utility option

--partitions *count*

Sets the number of partitions in the source.

Default:
The number of partitions defined in the cluster.

Requires: --create and --source options

You must keep this consistent with the number of partitions in the Kafka topic.

Renamed from --num-partitions .

--validation-typERROR|WARN|SKIP}

Controls the validation performed on a created or updated source:

- ERROR - Cancel configuration or creation if vkconfig cannot validate the source. This is the default setting.
- WARN - Proceed with task if validation fails, but display a warning.
- SKIP - Perform no validation.

Renamed from --skip-validation .

See Common vkconfig script options for options that are available in all of the vkconfig tools.

Examples

The following examples show how you can create or update SourceFeed.

Create the source SourceFeed and assign it to the cluster, StreamCluster1 in the scheduler defined by the myscheduler.conf config file:

```
$ /opt/vertica/packages/kafka/bin/vkconfig source --create --source SourceFeed \
                                  --cluster StreamCluster1 --partitions 3
                                  --conf myscheduler.conf
```

Update the existing source SourceFeed to use the existing cluster, StreamCluster2 in the scheduler defined by the myscheduler.conf config file:

```
$ /opt/vertica/packages/kafka/bin/vkconfig source --update --source SourceFeed \
                                  --new-cluster StreamCluster2
                                  --conf myscheduler.conf
```

The following example reads the sources defined in the scheduler defined by the weblogs.conf file.

```
$ vkconfig source --read --conf weblog.conf
{"source":"web_hits", "partitions":1, "src_enabled":true,
"cluster":"kafka_weblog",
"hosts":"kafka01.example.com:9092,kafka02.example.com:9092"}
```

Target tool options

Use the target tool to configure a Vertica table to receive data from your streaming data application.

Syntax

```
vkconfig target {--create | --read | --update | --delete} \
        [--target-table table --table_schema schema] \
        [other_options...]
```

--create

Adds a new target table for the scheduler. Cannot be used with --delete , --read , or --update .

--read

Outputs the targets defined in the scheduler. This output is in JSON format. Cannot be used with --create , --delete , or --update .

By default this option outputs all of the targets defined in the configuration schema. You can limit the output to specific targets by using the --target-schema and --target-table options. The vkconfig script only outputs targets that match the values you set in these options.
You can use LIKE wildcards in these options. See LIKE for more information about using wildcards.

--update

Updates the settings for the targeted table. Use with the --new-target-schema and --new-target-table options. Cannot be used with --create , --delete , or --read .

--delete

Removes the scheduler's association with the target table *table* . Cannot be used with --create , --read , or --update .

--target-table *table*

The existing Vertica table for the scheduler to target. This option is required for --create , --update , and --delete .

--target-schema *schema*

The existing Vertica schema containing the target table. This option is required for --create , --update , and --delete .

--dump

When you use this option along with the --read option, vkconfig outputs the Vertica query it would use to retrieve the data, rather than outputting the data itself. This option is useful if you want to access the data from within Vertica without having to go through vkconfig. This option has no effect if not used with --read .

--new-target-schema *schema_name*

Changes the schema containing the target table to a another existing schema.

Requires: --update option.

--new-target-table *table_name*

Changes the Vertica target table associated with this schema to a another existing table.

Requires: --update option.

--validation-type {ERROR|WARN|SKIP}

Controls validation performed on a created or updated target:

- ERROR - Cancel configuration or creation if vkconfig cannot validate that the table exists. This is the default setting.
- WARN - Creates or updates the target if validation fails, but display a warning.
- SKIP - Perform no validation.

Renamed from --skip-validation .

Important
Avoid having columns with primary key restrictions in your target table. The scheduler stops loading data if it encounters a row that has a value which violates this restriction. If you must have a primary key restricted column, try to filter out any redundant values for that column in the streamed data before is it loaded by the scheduler.

See Common vkconfig script options for options that are available in all of the vkconfig tools.

Examples

This example shows how you can create a target for the scheduler defined in the myscheduler.conf configuration file from public.streamtarget table:

```
$ /opt/vertica/packages/kafka/bin/vkconfig target --create \
      --target-table streamtarget --conf myscheduler.conf
```

This example lists all of the targets in the scheduler defined in the weblogs.conf configuration file.

```
$ vkconfig target --read --conf weblog.conf
{"target_schema":"public", "target_table":"web_hits"}
```

Load spec tool options

The vkconfig script's load spec tool lets you provide parameters for a COPY statement that loads streaming data.

Syntax

```
$ vkconfig load-spec {--create | --read | --update | --delete} \
      [--load-spec spec-name] [other-options...]
```

--create

Creates a new load spec. Cannot be used with --delete , --read , or --update .

--read

Outputs the current settings of the load specs defined in the scheduler. This output is in JSON format. Cannot be used with --create , --delete , or --update .

By default, this option outputs all load specs defined in the scheduler. You can limit the output by supplying a single value or a comma-separated list of values to these options:

- --load-spec
- --filters
- --uds-kv-parameters
- --parser
- --message-max-bytes
- --parser-parameters

The vkconfig script only outputs the configuration of load specs that match the values you supply.

You can use LIKE wildcards in these options. See LIKE for more information about using wildcards.

--update

Updates the settings of *spec-name* . Cannot be used with --create , --delete , or --read .

--delete

Deletes the load spec named *spec-name* . Cannot be used with --create , --read , or --update .

--load-spec *spec-name*

A unique name for copy load spec to operate on. This option is required for --create , --update , and --delete .

--dump

When you use this option along with the --read option, vkconfig outputs the Vertica query it would use to retrieve the data, rather than outputting the data itself. This option is useful if you want to access the data from within Vertica without having to go through vkconfig. This option has no effect if not used with --read .

--filters " *filter-name* "

A Vertica FILTER chain containing all of the UDFilters to use in the COPY statement. For more information on filters, refer to Parsing custom formats .

--message-max-bytes *max-size*

Specifies the maximum size, in bytes, of a Kafka protocol batch message.
Default: 25165824

--new-load-spec *new-name*

A new, unique name for an existing load spec. Requires the --update parameter.

--parser-parameters " *key* = *value* [,...]`"`

A list of parameters to provide to the parser specified in the --parser parameter. When you use a Vertica native parser, the scheduler passes these parameters to the COPY statement where they are in turn passed to the parser.

--parser *parser-name*

Identifies a Vertica UDParser to use with a specified target.This parser is used within the COPY statement that the scheduler runs to load data. If you are using a Vertica native parser, the values supplied to the --parser-parameters option are passed through to the COPY statement.

Default: KafkaParser

--uds-kv-parameters *key* = *value* [,...]

A comma separated list of key value pairs for the user-defined source.

--validation-type {ERROR|WARN|SKIP}

Specifies the validation performed on a created or updated load spec, to one of the following:

- ERROR : Cancel configuration or creation if vkconfig cannot validate the load spec. This is the default setting.
- WARN : Proceed with task if validation fails, but display a warning.
- SKIP : Perform no validation.

Renamed from --skip-validation .

See Common vkconfig script options for options that are available in all of the vkconfig tools.

Examples

These examples show how you can use the Load Spec utility options.

Create load spec Streamspec1 :

```
$ /opt/vertica/packages/kafka/bin/vkconfig load-spec --create --load-spec Streamspec1 --conf myscheduler.conf
```

Rename load spec Streamspec1 to Streamspec2 :

```
$ /opt/vertica/packages/kafka/bin/vkconfig load-spec --update --load-spec Streamspec1 \
                                    --new-load-spec Streamspec2 \
                                    --conf myscheduler.conf
```

Update load spec Filterspec to use the KafkaInsertLengths filter and a custom decryption filter:

```
$ /opt/vertica/packages/kafka/bin/vkconfig load-spec --update --load-spec Filterspec \
                                    --filters "KafkaInsertLengths() DecryptFilter(parameter=Key)" \
                                    --conf myscheduler.conf
```

Read the current settings for load spec streamspec1 :

```
$ vkconfig load-spec --read --load-spec streamspec1 --conf weblog.conf
{"load_spec":"streamspec1", "filters":null, "parser":"KafkaParser",
"parser_parameters":null, "load_method":"TRICKLE", "message_max_bytes":null,
"uds_kv_parameters":null}
```

Microbatch tool options

The vkconfig script's microbatch tool lets you configure a scheduler's microbatches.

Syntax

```
vkconfig microbatch {--create | --read | --update | --delete} \
    [--microbatch microbatch_name] [other_options...]
```

--create

Creates a new microbatch. Cannot be used with --delete , --read , or --update .

--read

Outputs the current settings of all microbatches defined in the scheduler. This output is in JSON format. Cannot be used with --create , --delete , or --update .

You can limit the output to specific microbatches by using the --consumer-group-id , --enabled , --load-spec , --microbatch , --rejection-schema , --rejection-table , --target-schema , --target-table , and --target-columns options. The --enabled option only accepts a true or false value.
You can use LIKE wildcards in these options. See LIKE for more information about using wildcards.

--update

Updates the settings of *microbatch_name* . Cannot be used with --create , --delete , or --read .

--delete

Deletes the microbatch named *microbatch_name* . Cannot be used with --create , --read , or --update .

--microbatch *microbatch_name*

A unique, case insensitive name for the microbatch. This option is required for --create , --update , and --delete .

--add-source-cluster *cluster_name*

The name of a cluster to assign to the microbatch you specify with the --microbatch option. You can use this parameter once per command. You can also use it with --update to add sources to a microbatch. You can only add sources from the same cluster to a single microbatch. Requires --add-source .

--add-source *source_name*

The name of a source to assign to this microbatch. You can use this parameter once per command. You can also use it with --update to add sources to a microbatch. Requires --add-source-cluster .

--cluster *cluster_name*

The name of the cluster to which the --offset option applies. Only required if the microbatch defines more than one cluster or the --source parameter is supplied. Requires the --offset option.

--consumer-group-id *id_name*

The name of the Kafka consumer group to which Vertica reports its progress consuming messages. Set this value to disable progress reports to a Kafka consumer group. For details, see Monitoring Vertica message consumption with consumer groups .
Default: vertica_ *database-name*

--dump

When you use this option along with the --read option, vkconfig outputs the Vertica query it would use to retrieve the data, rather than outputting the data itself. This option is useful if you want to access the data from within Vertica without having to go through vkconfig. This option has no effect if not used with --read .

--enabled TRUE|FALSE

When TRUE, allows the microbatch to execute.

--load-spec *loadspec_name*

The load spec to use while processing this microbatch.

--max-parallelism *max_num_loads*

The maximum number of simultaneous COPY statements created for the microbatch. The scheduler dynamically splits a single microbatch with multiple partitions into *max_num_loads* COPY statements with fewer partitions.

This option allows you to:

- Control the transaction size.
- Optimize your loads according to your scheduler's scheduler's resource pool settings, such as PLANNEDCONCURRENCY .

--new-microbatch *updated_name*

The updated name for the microbatch. Requires the --update option.

--offset ***partition_offset*** **[,...]**

The offset of the message in the source where the microbatch starts its load. If you use this parameter, you must supply an offset value for each partition in the source or each partition you list in the --partition option.

You can use this option to skip some messages in the source or reload previously read messages.
See Special Starting Offset Values below for more information.

Important
You cannot set an offset for a microbatch while the scheduler is running. If you attempt to do so, the vkconfig utility returns an error. Use the shutdown utility to shut the scheduler down before setting an offset for a microbatch.

--partition ***partition*** **[,...]**

One or more partitions to which the offsets given in the --offset option apply. If you supply this option, then the offset values given in the --offset option applies to the partitions you specify. Requires the --offset option.

--rejection-schema ***schema_name***

The existing Vertica schema that contains a table for storing rejected messages.

--rejection-table ***table_name***

The existing Vertica table that stores rejected messages.

--remove-source-cluster ***cluster_name***

The name of a cluster to remove from this microbatch. You can use this parameter once per command. Requires --remove-source .

--remove-source ***source_name***

The name of a source to remove from this microbatch. You can use this parameter once per command. You can also use it with --update to remove multiple sources from a microbatch. Requires --remove-source-cluster .

--source ***source_name***

The name of the source to which the offset in the --offset option applies. Required when the microbatch defines more than one source or the --cluster parameter is given. Requires the --offset option.

--target-columns ***column_expression***

A column expression for the target table, where *column_expression* can be a comma-delimited list of columns or a complete expression.

See the COPY statement Parameters for a description of column expressions.

--target-schema ***schema_name***

The existing Vertica target schema associated with this microbatch.

--target-table ***table_name***

The name of a Vertica table corresponding to the target. This table must belong to the target schema.

--validation-type {ERROR|WARN|SKIP}

Controls the validation performed on a created or updated microbatch:

- ERROR - Cancel configuration or creation if vkconfig cannot validate the microbatch. This is the default setting.
- WARN - Proceed with task if validation fails, but display a warning.
- SKIP - Perform no validation.

Renamed from --skip-validation .

See Common vkconfig script options for options that are available in all of the vkconfig tools.

Special starting offset values

The *start_offset* portion of the stream parameter lets you start loading messages from a specific point in the topic's partition. It also accepts one of two special offset values:

- -2 tells the scheduler to start loading at the earliest available message in the topic's partition. This value is useful when you want to load as many messages as you can from the Kafka topic's partition.
- -3 tells the scheduler to start loading from the consumer group's saved offset. If the consumer group does not have a saved offset, it starts loading from the earliest available message in the topic partition. See Monitoring Vertica message consumption with consumer groups for more information.

Examples

This example shows how you can create the microbatch, mbatch1. This microbatch identifies the schema, target table, load spec, and source for the microbatch:

```
$ /opt/vertica/packages/kafka/bin/vkconfig microbatch --create --microbatch mbatch1 \
                                    --target-schema public \
                                    --target-table BatchTarget \
                                    --load-spec Filterspec \
                                    --add-source SourceFeed \
                                    --add-source-cluster StreamCluster1 \
                                    --conf myscheduler.conf
```

This example demonstrates listing the current settings for the microbatches in the scheduler defined in the weblog.conf configuration file.

```
$ vkconfig microbatch --read --conf weblog.conf
{"microbatch":"weblog", "target_columns":null, "rejection_schema":null,
"rejection_table":null, "enabled":true, "consumer_group_id":null,
"load_spec":"weblog_load", "filters":null, "parser":"KafkaJSONParser",
"parser_parameters":null, "load_method":"TRICKLE", "message_max_bytes":null,
"uds_kv_parameters":null, "target_schema":"public", "target_table":"web_hits",
"source":"web_hits", "partitions":1, "src_enabled":true, "cluster":"kafka_weblog",
"hosts":"kafka01.example.com:9092,kafka02.example.com:9092"}
```

Launch tool options

Use the vkconfig script's launch tool to assign a name to a scheduler instance.

Syntax

```
vkconfig launch [options...]
```

--enable-ssl {true|false}

(Optional) Enables SSL authentication between Kafka and Vertica . For more information, refer to TLS/SSL encryption with Kafka .

--ssl-ca-alias *alias*

The user-defined alias of the root certifying authority you are using to authenticate communication between Vertica and Kafka. This parameter is used only when SSL is enabled.

--ssl-key-alias *alias*

The user-defined alias of the key/certificate pair you are using to authenticate communication between Vertica and Kafka. This parameter is used only when SSL is enabled.

--ssl-key-password *password*

The password used to create your SSL key. This parameter is used only when SSL is enabled.

--instance-name *name*

(Optional) Allows you to name the process running the scheduler. You can use this command when viewing the scheduler_history table, to find which instance is currently running.

--refresh-interval *hours*

(Optional) The time interval at which the connection between Vertica and Kafka is refreshed (24 hours by default).

--kafka_conf ' *kafka_configuration_setting* '

A JSON-formatted object of option/value pairs to pass directly to the rdkafka library. This is the library Vertica uses to communicate with Kafka. You can use this parameter to directly set configuration options that are not available through the Vertica integration with Kafka. See Directly setting Kafka library options for details.

--kafka_conf_secret ' *kafka_configuration_setting* '

Conceals sensitive configuration data that you must pass directly to the rdkafka library, such as passwords. This parameter accepts settings in the same format as kafka_conf .

Values passed to this parameter are not logged or stored in system tables.

See Common vkconfig script options for options that are available in all of the vkconfig tools.

Examples

This example shows how you can launch the scheduler defined in the myscheduler.conf config file and give it the instance name PrimaryScheduler:

```
$ nohup /opt/vertica/packages/kafka/bin/vkconfig launch --instance-name PrimaryScheduler \
 --conf myscheduler.conf >/dev/null 2>&1 &
```

This example shows how you can launch an instance named SecureScheduler with SSL enabled:

```
$ nohup /opt/vertica/packages/kafka/bin/vkconfig launch --instance-name SecureScheduler --enable-SSL true \
                                    --ssl-ca-alias authenticcert --ssl-key-alias ourkey \
                                    --ssl-key-password secret \
                                    --conf myscheduler.conf \
                                    >/dev/null 2>&1 &
```

Shutdown tool options

Use the vkconfig script's shutdown tool to terminate one or all Vertica schedulers running on a host. Always run this command before restarting a scheduler to ensure the scheduler has shutdown correctly.

Syntax

```
vkconfig shutdown [options...]
```

See Common vkconfig script options for options that are available in all vkconfig tools.

Examples

To terminate all schedulers running on a host, use the shutdown command with no options:

```
$ /opt/vertica/packages/kafka/bin/vkconfig shutdown
```

Use the --conf or --config-schema option to specify a scheduler to shut down. The following command terminates the scheduler that was launched with the same --conf myscheduler.conf option:

```
$ /opt/vertica/packages/kafka/bin/vkconfig shutdown --conf myscheduler.conf
```

Statistics tool options

The statistics tool lets you access the history of microbatches that your scheduler has run. This tool outputs the log of the microbatches in JSON format to the standard output. You can use its options to filter the list of microbatches to get just the microbatches that interest you.

> **Note**
> The statistics tool can sometimes produce confusing output if you have altered the scheduler configuration over time. For example, suppose you have microbatch-a target a table. Later, you change the scheduler's configuration so that microbatch-b targets the table. Afterwards, you run the statistics tool and filter the microbatch log based on target table. Then the log output will show entries from both microbatch-a and microbatch-b.

Syntax

```
vkconfig statistics [options]
```

--cluster " *cluster* "[," *cluster2* "...]

Only return microbatches that retrieved data from a cluster whose name matches one in the list you supply.

--dump

Instead of returning microbatch data, return the SQL query that vkconfig would execute to extract the data from the scheduler tables. You can use this option if you want use a Vertica client application to get the microbatch log instead of using vkconfig's JSON output.

--from-timestamp " *timestamp* "

Only return microbatches that began after *timestamp* . The timestamp value is in *yyyy* -[*m*] *m* -[*d*] *d hh* : *mm* : *ss* format.

Cannot be used in conjunction with --last .

--last *number*

Returns the *number* most recent microbatches that meet all other filters. Cannot be used in conjunction with --from-timestamp or --to-timestamp .

--microbatch " *name* "[," *name2* "...]

Only return microbatches whose name matches one of the names in the comma-separated list.

--partition *partition#* [, *partition#2* ...]

Only return microbatches that accessed data from the topic partition that matches ones of the values in the partition list.

--source " *source* "[," *source2* "...]

Only return microbatches that accessed data from a source whose name matches one of the names in the list you supply to this argument.

--target-schema "schema"[,"schema2"...]

Only return microbatches that wrote data to the Vertica schemas whose name matches one of the names in the target schema list argument.

--target-table "table"[,"table2"...]

Only return microbatches that wrote data to Vertica tables whose name match one of the names in the target schema list argument.

--to-timestamp " *timestamp* "

Only return microbatches that began before *timestamp* . The timestamp value is in *yyyy* -[*m*] *m* -[*d*] *d hh* : *mm* : *ss* format.

Cannot be used in conjunction with --last .

See Common vkconfig script options for options that are available in all of the vkconfig tools.

Usage considerations

- You can use LIKE wildcards in the values you supply to the --cluster , --microbatch , --source , --target-schema , and --target-table arguments. This feature lets you match partial strings in the microbatch data. See LIKE for more information about using wildcards.
- The string comparisons for the --cluster , --microbatch , --source , --target-schema , and --target-table arguments are case-insensitive.
- The date and time values you supply to the --from-timestamp and --to-timestamp arguments use the java.sql.timestamp format for parsing the value. This format's parsing can accept values that you may consider invalid and would expect it to reject. For example, if you supply a timestamp of 01-01-2018 24:99:99, the Java timestamp parser silently converts the date to 2018-01-02 01:40:39 instead of returning an error.

Examples

This example gets the last microbatch that the scheduler defined in the weblog.conf file ran:

```
$ /opt/vertica/packages/kafka/bin/vkconfig statistics --last 1 --conf weblog.conf
{"microbatch":"weblog", "target_schema":"public", "target_table":"web_hits",
"source_name":"web_hits", "source_cluster":"kafka_weblog", "source_partition":0,
"start_offset":80000, "end_offset":79999, "end_reason":"END_OF_STREAM",
"end_reason_message":null, "partition_bytes":0, "partition_messages":0,
"timeslice":"00:00:09.793000", "batch_start":"2018-11-06 09:42:00.176747",
"batch_end":"2018-11-06 09:42:00.437787", "source_duration":"00:00:00.214314",
"consecutive_error_count":null, "transaction_id":45035996274513069,
"frame_start":"2018-11-06 09:41:59.949", "frame_end":null}
```

If your scheduler is reading from more than partition, the --last 1 option lists the last microbatch from each partition:

```
$ /opt/vertica/packages/kafka/bin/vkconfig statistics --last 1 --conf iot.conf
{"microbatch":"iotlog", "target_schema":"public", "target_table":"iot_data",
"source_name":"iot_data", "source_cluster":"kafka_iot", "source_partition":0,
"start_offset":-2, "end_offset":-2, "end_reason":"DEADLINE",
"end_reason_message":null, "partition_bytes":0, "partition_messages":0,
"timeslice":"00:00:09.842000", "batch_start":"2018-11-06 12:52:49.387567",
"batch_end":"2018-11-06 12:52:59.400219", "source_duration":"00:00:09.950127",
"consecutive_error_count":null, "transaction_id":45035996274537015,
"frame_start":"2018-11-06 12:52:49.213", "frame_end":null}
{"microbatch":"iotlog", "target_schema":"public", "target_table":"iot_data",
"source_name":"iot_data", "source_cluster":"kafka_iot", "source_partition":1,
"start_offset":1604, "end_offset":1653, "end_reason":"END_OF_STREAM",
"end_reason_message":null, "partition_bytes":4387, "partition_messages":50,
"timeslice":"00:00:09.842000", "batch_start":"2018-11-06 12:52:49.387567",
"batch_end":"2018-11-06 12:52:59.400219", "source_duration":"00:00:00.220329",
"consecutive_error_count":null, "transaction_id":45035996274537015,
"frame_start":"2018-11-06 12:52:49.213", "frame_end":null}
{"microbatch":"iotlog", "target_schema":"public", "target_table":"iot_data",
"source_name":"iot_data", "source_cluster":"kafka_iot", "source_partition":2,
"start_offset":1603, "end_offset":1652, "end_reason":"END_OF_STREAM",
"end_reason_message":null, "partition_bytes":4383, "partition_messages":50,
"timeslice":"00:00:09.842000", "batch_start":"2018-11-06 12:52:49.387567",
"batch_end":"2018-11-06 12:52:59.400219", "source_duration":"00:00:00.318997",
"consecutive_error_count":null, "transaction_id":45035996274537015,
"frame_start":"2018-11-06 12:52:49.213", "frame_end":null}
{"microbatch":"iotlog", "target_schema":"public", "target_table":"iot_data",
"source_name":"iot_data", "source_cluster":"kafka_iot", "source_partition":3,
"start_offset":1604, "end_offset":1653, "end_reason":"END_OF_STREAM",
"end_reason_message":null, "partition_bytes":4375, "partition_messages":50,
"timeslice":"00:00:09.842000", "batch_start":"2018-11-06 12:52:49.387567",
"batch_end":"2018-11-06 12:52:59.400219", "source_duration":"00:00:00.219543",
"consecutive_error_count":null, "transaction_id":45035996274537015,
"frame_start":"2018-11-06 12:52:49.213", "frame_end":null}
```

You can use the --partition argument to get just the partitions you want:

```
$ /opt/vertica/packages/kafka/bin/vkconfig statistics --last 1 --partition 2 --conf iot.conf
{"microbatch":"iotlog", "target_schema":"public", "target_table":"iot_data",
"source_name":"iot_data", "source_cluster":"kafka_iot", "source_partition":2,
"start_offset":1603, "end_offset":1652, "end_reason":"END_OF_STREAM",
"end_reason_message":null, "partition_bytes":4383, "partition_messages":50,
"timeslice":"00:00:09.842000", "batch_start":"2018-11-06 12:52:49.387567",
"batch_end":"2018-11-06 12:52:59.400219", "source_duration":"00:00:00.318997",
"consecutive_error_count":null, "transaction_id":45035996274537015,
"frame_start":"2018-11-06 12:52:49.213", "frame_end":null}
```

If your scheduler reads from more than one source, the --last 1 option outputs the last microbatch from each source:

```
$ /opt/vertica/packages/kafka/bin/vkconfig statistics --last 1 --conf weblog.conf
{"microbatch":"weberrors", "target_schema":"public", "target_table":"web_errors",
"source_name":"web_errors", "source_cluster":"kafka_weblog",
"source_partition":0, "start_offset":10000, "end_offset":9999,
"end_reason":"END_OF_STREAM", "end_reason_message":null,
"partition_bytes":0, "partition_messages":0, "timeslice":"00:00:04.909000",
"batch_start":"2018-11-06 10:58:02.632624",
"batch_end":"2018-11-06 10:58:03.058663", "source_duration":"00:00:00.220618",
"consecutive_error_count":null, "transaction_id":45035996274523991,
"frame_start":"2018-11-06 10:58:02.394", "frame_end":null}
{"microbatch":"weblog", "target_schema":"public", "target_table":"web_hits",
"source_name":"web_hits", "source_cluster":"kafka_weblog", "source_partition":0,
"start_offset":80000, "end_offset":79999, "end_reason":"END_OF_STREAM",
"end_reason_message":null, "partition_bytes":0, "partition_messages":0,
"timeslice":"00:00:09.128000", "batch_start":"2018-11-06 10:58:03.322852",
"batch_end":"2018-11-06 10:58:03.63047", "source_duration":"00:00:00.226493",
"consecutive_error_count":null, "transaction_id":45035996274524004,
"frame_start":"2018-11-06 10:58:02.394", "frame_end":null}
```

You can use wildcards to enable partial matches. This example demonstrates getting the last microbatch for all microbatches whose names end with "log":

```
~$ /opt/vertica/packages/kafka/bin/vkconfig statistics --microbatch "%log" \
                                    --last 1 --conf weblog.conf
{"microbatch":"weblog", "target_schema":"public", "target_table":"web_hits",
"source_name":"web_hits", "source_cluster":"kafka_weblog", "source_partition":0,
"start_offset":80000, "end_offset":79999, "end_reason":"END_OF_STREAM",
"end_reason_message":null, "partition_bytes":0, "partition_messages":0,
"timeslice":"00:00:04.874000", "batch_start":"2018-11-06 11:37:16.17198",
"batch_end":"2018-11-06 11:37:16.460844", "source_duration":"00:00:00.213129",
"consecutive_error_count":null, "transaction_id":45035996274529932,
"frame_start":"2018-11-06 11:37:15.877", "frame_end":null}
```

To get microbatches from a specific period of time, use the --from-timestamp and --to-timestamp arguments. This example gets the microbatches that read from partition #2 between 12:52:30 and 12:53:00 on 2018-11-06 for the scheduler defined in iot.conf.

```
$ /opt/vertica/packages/kafka/bin/vkconfig statistics  --partition 1 \
                    --from-timestamp "2018-11-06 12:52:30" \
                    --to-timestamp "2018-11-06 12:53:00" --conf iot.conf
{"microbatch":"iotlog", "target_schema":"public", "target_table":"iot_data",
"source_name":"iot_data", "source_cluster":"kafka_iot", "source_partition":1,
"start_offset":1604, "end_offset":1653, "end_reason":"END_OF_STREAM",
"end_reason_message":null, "partition_bytes":4387, "partition_messages":50,
"timeslice":"00:00:09.842000", "batch_start":"2018-11-06 12:52:49.387567",
"batch_end":"2018-11-06 12:52:59.400219", "source_duration":"00:00:00.220329",
"consecutive_error_count":null, "transaction_id":45035996274537015,
"frame_start":"2018-11-06 12:52:49.213", "frame_end":null}
{"microbatch":"iotlog", "target_schema":"public", "target_table":"iot_data",
"source_name":"iot_data", "source_cluster":"kafka_iot", "source_partition":1,
"start_offset":1554, "end_offset":1603, "end_reason":"END_OF_STREAM",
"end_reason_message":null, "partition_bytes":4371, "partition_messages":50,
"timeslice":"00:00:09.788000", "batch_start":"2018-11-06 12:52:38.930428",
"batch_end":"2018-11-06 12:52:48.932604", "source_duration":"00:00:00.231709",
"consecutive_error_count":null, "transaction_id":45035996274536981,
"frame_start":"2018-11-06 12:52:38.685", "frame_end":null}
```

This example demonstrates using the --dump argument to get the SQL statement vkconfig executed to retrieve the output from the previous example:

```
$ /opt/vertica/packages/kafka/bin/vkconfig statistics  --dump --partition 1 \
                --from-timestamp "2018-11-06 12:52:30" \
                --to-timestamp "2018-11-06 12:53:00" --conf iot.conf
SELECT microbatch, target_schema, target_table, source_name, source_cluster,
source_partition, start_offset, end_offset, end_reason, end_reason_message,
partition_bytes, partition_messages, timeslice, batch_start, batch_end,
last_batch_duration AS source_duration, consecutive_error_count, transaction_id,
frame_start, frame_end FROM "iot_sched".stream_microbatch_history WHERE
(source_partition = '1') AND (frame_start >= '2018-11-06 12:52:30.0') AND
(frame_start < '2018-11-06 12:53:00.0') ORDER BY frame_start DESC, microbatch,
source_cluster, source_name, source_partition;
```

Sync tool options

The sync utility immediately updates all source definitions by querying the Kafka cluster's brokers defined by the source. By default, it updates all of the sources defined in the target schema. To update just specific sources, use the --source and --cluster options to specify which sources to update.

Syntax

```
vkconfig sync [options...]
```

--source ***source_name***

The name of the source sync. This source must already exist in the target schema.

--cluster ***cluster_name***

Identifies the cluster containing the source that you want to sync. You must have already defined this cluster in the scheduler.

--kafka_conf ' ***kafka_configuration_setting*** **'**

A JSON-formatted object of option/value pairs to pass directly to the rdkafka library. This is the library Vertica uses to communicate with Kafka. You can use this parameter to directly set configuration options that are not available through the Vertica integration with Kafka. See Directly setting Kafka library options for details.

--kafka_conf_secret ' ***kafka_configuration_setting*** **'**

Conceals sensitive configuration data that you must pass directly to the rdkafka library, such as passwords. This parameter accepts settings in the same format as kafka_conf .

Values passed to this parameter are not logged or stored in system tables.

See the Common vkconfig script options for options available in all of the vkconfig tools..

Kafka function reference

This section lists the functions that make up Vertica's Kafka integration feature.

In this section

- KafkaAvroParser
- KafkaCheckBrokers
- KafkaExport
- KafkaJSONParser
- KafkaListManyTopics
- KafkaListTopics
- KafkaOffsets
- KafkaParser
- KafkaSource
- KafkaTopicDetails

KafkaAvroParser

The KafkaAvroParser parses Avro-formatted Kafka messages and loads them into a regular Vertica table or a Vertica flex table.

Syntax

```
KafkaAvroParser(param=value[,...])
```

enforce_length

When set to TRUE, rejects the row if any value is too wide to fit into its column. When using the default setting (FALSE) , the parser truncates any

value that is too wide to fit within the column's maximum width.

reject_on_materialized_type_error

When set to TRUE, rejects the row if it contains a materialized column value that cannot be mapped into the materialized column's data type.

flatten_maps

If set to TRUE, flattens all Avro maps.

flatten_arrays

If set to TRUE, flattens Avro arrays.

flatten_records

If set to TRUE, flattens all Avro records.

external_schema

The schema of the Avro file as a JSON string. If this parameter is not specified, the parser assumes that each message has the schema on it. If you are using a schema registry, do not use this parameter.

codec

The codec in which the Avro file was written. Valid values are:

- default : Data is not compressed and codec is not needed
- deflate : Data is compressed using the deflate codec
- snappy : Snappy compression

> **Note**
> This option is mainly provided for backwards compatibility. You usually have Kafka compress data at the message level, and have KafkaSource decompress the message for you.

with_metadata

If set to TRUE, messages include Avro datum, schema, and object metadata. By default, the KafkaAvroParser parses messages without including schema and metadata. If you enable this parameter, write your messages using the Avro API and confirm they contain only Avro datum. The default value is FALSE.

schema_registry_url

Required, the URL of the Confluent schema registry. This parameter is required to load data based on a schema registry version. If you are using an external schema, do not use this parameter. For more information, refer to Avro Schema Registry .

> **Note**
> TLS connections must use the HTTPS protocol.

schema_registry_ssl_ca_path

Required for TLS connections, the path on the Vertica node's file system to a directory containing one or more hashed certificate authority (CA) certificates that signed the schema registry's server certificate. Each Vertica node must store hashed CA certificates on the same path.

Important
In some circumstances, you might receive a validation error stating that the KafkaAvroParser cannot locate your CA certificate. To correct this error, store your CA certificate in your operating system's default bundle. For example, store the CA certificate in /etc/pki/tls/certs/ca-bundle.crt on Red Hat operating systems.

For details on hashed CA certificates, see Hashed CA Certificates .

schema_registry_ssl_cert_path

Path on the Vertica node's file system to a client certificate issued by a certificate authority (CA) that the schema registry trusts.

schema_registry_ssl_key_path

Path on the Vertica server file system to the private key for the client certificate defined with schema_registry_ssl_cert_path .

schema_registry_ssl_key_password_path

Path on the Vertica server file system to the optional password for the private key defined with schema_registry_ssl_key_path .

schema_registry_subject

In the schema registry, the subject of the schema to use for data loading.

schema_registry_version

In the schema registry, the version of the schema to use for data loading.

key_separator

Sets the character to use as the separator between keys.

Data types

KafkaAvroParser supports the same data types as the favroparser . For details, see Avro data .

Example

The following example demonstrates loading data from Kafka in an Avro format. The statement:

- Loads data into an existing flex table named weather_logs.
- Copies data from the default Kafka broker (running on the local system on port 9092).
- The source is named temperature.
- The source has a single partition.
- The load starts from offset 0.
- The load ends either after 10 seconds or the load reaches the end of the source, whichever occurs first.
- The KafkaAvroParser does not flatten any arrays, maps, or records it finds in the source.
- The schema for the data is provided in the statement as a JSON string. It defines a record type named Weather that contains fields for a station name, a time, and a temperature.
- Rejected rows of data are saved to a table named t_rejects1.

```
=> COPY weather_logs
  SOURCE KafkaSource(stream='temperature|0|0', stop_on_eof=true,
                     duration=interval '10 seconds')
  PARSER KafkaAvroParser(flatten_arrays=False, flatten_maps=False, flatten_records=False,
                         external_schema=E'{"type":"record","name":"Weather","fields":'
                                         '[{"name":"station","type":"string"},'
                                         '{"name":"time","type":"long"},'
                                         '{"name":"temp","type":"int"}]}')
  REJECTED DATA AS TABLE "t_rejects1";
```

Hashed CA certificates

Some parameters like schema_registry_ssl_ca_path require hashed CA certificates rather than the CA certificates themselves. A hashed CA certificate is a symbolic link to the original CA certificate. This symbolic link must have the following naming scheme:

```
CA_hash.0
```

For example, if the hash for ca_cert.pem is 9741086f , the hashed CA certificate would be 9741086f.0 , a symbolic link to ca_cert.pem .

For details, see the OpenSSL 1.1 or 1.0 documentation.

Hashing CA certificates

The procedure for hashing CA certificates varies between versions of openssl . You can find your version of openssl with:

```
$ openssl version
```

For openssl 1.1 or higher, use openssl rehash . For example, if the directory /my_ca_certs/ contains ca_cert.pem , you can hash and symbolically link to it with:

```
$ openssl rehash /my_ca_certs/
```

This adds the hashed CA certificate to the directory:

```
$ ls -l
total 8
lrwxrwxrwx 1 ver ver    8 Mar 13 14:41 9da13359.0 -> ca_cert.pem
-rw-r--r-- 1 ver ver 1245 Mar 13 14:41 ca_cert.pem
```

For openssl 1.0, you can use openssl x509 -hash -noout -in *ca_cert* .pem to retrieve the hash and then create a symbolic link to the CA certificate. For example:

1. Run the following command to retrieve the hash of the CA certificate ca_cert.pem :

```
$ openssl x509 -hash -noout -in /my_ca_certs/ca_cert.pem
9741086f
```

2. Create a symbolic link to /my_ca_certs/ca_cert.pem :

```
$ ln /my_ca_certs/ca_cert.pem /my_ca_certs/9741086f.0
```

This adds the hashed CA certificate to the directory:

```
$ ls -l
total 8
-rw-r--r-- 2 ver ver 1220 Mar 13 13:41 9741086f.0 -> ca_cert.pem
-rw-r--r-- 2 ver ver 1220 Mar 13 13:41 ca_cert.pem
```

KafkaCheckBrokers

Retrieves information about the brokers in a Kafka cluster. This function is intended mainly for internal use—it used by the streaming job scheduler to get the list of brokers in the Kafka cluster. You can call the function to determine which brokers Vertica knows about.

Syntax

```
KafkaCheckBrokers(USING PARAMETERS brokers='hostname:port[,hostname2:port...]'
                  [, kafka_conf='kafka_configuration_setting']
                  [, timeout=timeout_sec])
```

brokers

The host name and port number of a broker in the Kafka cluster used to retrieve the list of brokers. You can supply more than one broker as a comma-separated list. If the list includes brokers from more than one Kafka cluster, the cluster containing the last host in the list is queried.

kafka_conf

A JSON-formatted object of option/value pairs to pass directly to the rdkafka library. This is the library Vertica uses to communicate with Kafka. You can use this parameter to directly set configuration options that are not available through the Vertica integration with Kafka. See Directly setting Kafka library options for details.

kafka_conf_secret

Conceals sensitive configuration data that you must pass directly to the rdkafka library, such as passwords. This parameter accepts settings in the same format as kafka_conf .

Values passed to this parameter are not logged or stored in system tables.

timeout

Integer number of seconds to wait for a response from the Kafka cluster.

Example

```
=> SELECT KafkaCheckBrokers(USING PARAMETERS brokers='kafka01.example.com:9092')
       OVER ();
 broker_id |       hostname      | port
-----------+---------------------+------
         2 | kafka03.example.com | 9092
         1 | kafka02.example.com | 9092
         3 | kafka04.example.com | 9092
         0 | kafka01.example.com | 9092
(4 rows)
```

KafkaExport

Sends Vertica data to Kafka.

If Vertica successfully exports all of the rows of data to Kafka, this function returns zero rows. You can use the output of this function to copy failed messages to a secondary table for evaluation and reprocessing.

Syntax

```
SELECT KafkaExport(partitionColumn, keyColumn, valueColumn
   USING PARAMETERS brokers='host[:port][,host...]',
   topic='topicname'
   [,kafka_conf='kafka_configuration_setting']
   [,fail_on_conf_parse_error=Boolean])
OVER (partition_clause) FROM table;
```

Parameters

partitionColumn

The target partition for the export. If you set this value to NULL, Vertica uses the default partitioning scheme. You can use the partition argument to send messages to partitions that map to Vertica segments.

keyColumn

The user defined key value associated with the valueColumn. Use NULL to skip this argument.

valueColumn

The message itself. The column is a LONG VARCHAR, allowing you to send up to 32MB of data to Kafka. However, Kafka may impose its own limits on message size.

brokers

A string containing a comma-separated list of one or more host names or IP addresses (with optional port number) of brokers in the Kafka cluster.

topic

The Kafka topic to which you are exporting.

kafka_conf

A JSON-formatted object of option/value pairs to pass directly to the rdkafka library. This is the library Vertica uses to communicate with Kafka. You can use this parameter to directly set configuration options that are not available through the Vertica integration with Kafka. See Directly setting Kafka library options for details.

kafka_conf_secret

Conceals sensitive configuration data that you must pass directly to the rdkafka library, such as passwords. This parameter accepts settings in the same format as kafka_conf .

Values passed to this parameter are not logged or stored in system tables.

fail_on_conf_parse_error

Determines whether the function fails when kafka_conf contains incorrectly formatted options and values, or invalid configuration properties.

Default Value : FALSE

For accepted option and value formats, see Directly setting Kafka library options .

For a list of valid configuration properties, see the rdkafka GitHub repository .

Example

The following example converts each row from the iot_report table into a JSON object, and exports it to a Kafka topic named iot-report.

The iot_report table contains the following columns and data:

```
=> SELECT * FROM iot_report;
 server |        date         |        location         |    id
--------+---------------------+-------------------------+----------
      1 | 2016-10-11 04:09:28 |  -14.86058, 112.75848   | 70982027
      1 | 2017-07-02 12:37:48 |  -21.42197, -127.17672  | 49494918
      2 | 2017-01-20 03:05:24 |  37.17372, 136.14026    | 36670100
      2 | 2017-07-29 11:38:37 |  -38.99517, 171.72671   | 52049116
      1 | 2017-10-19 14:04:33 |  -71.72156, -36.27381   | 94328189
(5 rows)
```

To build the KafkaExport function, provide the following values to define the Kafka message:

- *partitionColumn* : Use the server column. Because the server column values are one-based and Kafka partitions are zero-based, subtract 1 from the server value.
- *keyColumn* : Use the id column. This requires that you explicitly cast the id value to a VARCHAR type.

- *valueColumn* : Each message formats the date and location columns as the key/value pairs in the exported JSON object.
  To convert these rows to JSON format, use the ROW function to convert the date and location columns to structured data. Then, pass the ROW function to the TO_JSON function to encode the data structure as a JSON object.

Complete the remaining required function arguments and execute the KafkaExport function. If it succeeds, it returns 0 rows:

```
=> SELECT KafkaExport(server - 1, id::VARCHAR, TO_JSON(ROW(date, location))
                USING PARAMETERS brokers='broker01:9092,broker02:9092',
                topic='iot-results')
  OVER (PARTITION BEST)
  FROM iot_report;
 partition | key | message | failure_reason
-----------+-----+---------+----------------
(0 rows)
```

Use kafkacat to verify that the consumer contains the JSON-formatted data in the iot-results topic:

```
$ /opt/vertica/packages/kafka/bin/kafkacat -C -t iot-results -b broker01:9092,broker02:9092
{"date":"2017-01-20 03:05:24","location":" 37.17372, 136.14026   "}
{"date":"2017-07-29 11:38:37","location":" -38.99517, 171.72671  "}
{"date":"2016-10-11 04:09:28","location":" -14.86058, 112.75848  "}
{"date":"2017-10-19 14:04:33","location":" -71.72156, -36.27381  "}
{"date":"2017-07-02 12:37:48","location":" -21.42197, -127.17672 "}
```

See also

Producing data using KafkaExport

KafkaJSONParser

The KafkaJSONParser parses JSON-formatted Kafka messages and loads them into a regular Vertica table or a Vertica flex table.

Syntax

```
KafkaJSONParser(
    [enforce_length=Boolean]
    [, flatten_maps=Boolean]
    [, flatten_arrays=Boolean]
    [, start_point=string]
    [, start_point_occurrence=integer]
    [, omit_empty_keys=Boolean]
    [, reject_on_duplicate=Boolean]
    [, reject_on_materialized_type_error=Boolean]
    [, reject_on_empty_key=Boolean]
    [, key_separator=char]
    [, suppress_nonalphanumeric_key_chars=Boolean]
    )
```

enforce_length

If set to TRUE, rejects the row if data being loaded is too wide to fit into its column. Defaults to FALSE, which truncates any data that is too wide to fit into its column.

flatten_maps

If set to TRUE, flattens all JSON maps.

flatten_arrays

If set to TRUE, flattens JSON arrays.

start_point

Specifies the key in the JSON data that the parser should parse. The parser only extracts data that is within the value associated with the start_point key. It parses the values of all instances of the start_point key within the data.

start_point_occurrence

Integer value indicating which the occurrence of the key specified by the start_point parameter where the parser should begin parsing. For example, if you set this value to 4, the parser will only begin loading data from the fifth occurrence of the start_point key. Only has an effect if you also supply the start_point parameter.

omit_empty_keys

If set to TRUE, omits any key from the load data that does not have a value set.

reject_on_duplicate

If set to TRUE, rejects the row that contains duplicate key names. Key names are case-insensitive, so the keys "mykey" and "MyKey" are considered duplicates.

reject_on_materialized_type_error

If set to TRUE, rejects the row if the data includes keys matching an existing materialized column and has a key that cannot be mapped into the materialized column's data type.

reject_on_empty_key

If set to TRUE, rejects any row containing a key without a value.

key_separator

A single character to use as the separator between key values instead of the default period (.) character.

suppress_nonalphanumeric_key_chars

If set to TRUE, replaces all non-alphanumeric characters in JSON key values with an underscore (_) character.

See JSON data for more information.

The following example demonstrates loading JSON data from Kafka. The parameters in the statement define to the load to:

- Load data into the pre-existing table named logs.
- The KafkaSource streams the data from a single partition in the source called server_log.
- The Kafka broker for the data load is running on the host named kafka01 on port 9092.
- KafkaSource stops loading data after either 10 seconds or on reaching the end of the stream, whichever happens first.
- The KafkJSONParser flattens any arrays or maps in the JSON data.

```
=> COPY logs SOURCE KafkaSource(stream='server_log|0|0',
                                stop_on_eof=true,
                                duration=interval '10 seconds',
                                brokers='kafka01:9092')
  PARSER KafkaJSONParser(flatten_arrays=True, flatten_maps=True);
```

KafkaListManyTopics

Retrieves information about all topics from a Kafka broker. This function lists all of the topics defined in the Kafka cluster as well the number of partitions it contains and which brokers serve the topic.

Syntax

```
KafkaListManyTopics('broker:port[;...]'
                    [USING PARAMETERS
                       [kafka_conf='kafka_configuration_setting'
                       [, timeout=timeout_sec]])
```

broker

The hostname (or ip address) of a broker in the Kafka cluster

port

The port number on which the broker is running.

kafka_conf

A JSON-formatted object of option/value pairs to pass directly to the rdkafka library. This is the library Vertica uses to communicate with Kafka. You can use this parameter to directly set configuration options that are not available through the Vertica integration with Kafka. See Directly setting Kafka library options for details.

kafka_conf_secret

Conceals sensitive configuration data that you must pass directly to the rdkafka library, such as passwords. This parameter accepts settings in the same format as kafka_conf .

Values passed to this parameter are not logged or stored in system tables.

timeout

Integer number of seconds to wait for a response from the Kafka cluster.

Example

```
=> \x
Expanded display is on.
=> SELECT KafkaListManyTopics('kafka01.example.com:9092')
  OVER (PARTITION AUTO);
-[ RECORD 1 ]--+-------------------------------------------------
brokers        | kafka01.example.com:9092,kafka02.example.com:9092
topic          | __consumer_offsets
num_partitions | 50
-[ RECORD 2 ]--+-------------------------------------------------
brokers        | kafka01.example.com:9092,kafka02.example.com:9092
topic          | iot_data
num_partitions | 1
-[ RECORD 3 ]--+-------------------------------------------------
brokers        | kafka01.example.com:9092,kafka02.example.com:9092
topic          | test
num_partitions | 1
```

KafkaListTopics

Gets the list of topics available from a Kafka broker.

Syntax

```
KafkaListTopics(USING PARAMETERS brokers='hostname:port[,hostname2:port2...]'
                                 [, kafka_conf='kafka_configuration_setting']
                                 [, timeout=timeout_sec])
```

brokers

The host name and port number of the broker to query for a topic list. You can supply more than one broker as a comma-separated list. However, the returned list will only contains the topics served by the last broker in the list.

kafka_conf

A JSON-formatted object of option/value pairs to pass directly to the rdkafka library. This is the library Vertica uses to communicate with Kafka. You can use this parameter to directly set configuration options that are not available through the Vertica integration with Kafka. See Directly setting Kafka library options for details.

kafka_conf_secret

Conceals sensitive configuration data that you must pass directly to the rdkafka library, such as passwords. This parameter accepts settings in the same format as kafka_conf .

Values passed to this parameter are not logged or stored in system tables.

timeout

Integer number of seconds to wait for a response from the Kafka cluster.

Example

```
=> SELECT KafkaListTopics(USING PARAMETERS brokers='kafka1-01.example.com:9092')
        OVER ();
        topic         | num_partitions
----------------------+----------------
 test                 |              1
 iot_data             |              1
 __consumer_offsets   |             50
 vertica_notifications |             1
 web_hits             |              1
(5 rows)
```

KafkaOffsets

The KafkaOffsets user-defined transform function returns load operation statistics generated by the most recent invocation of KafkaSource. Query KafkaOffsets to see the metadata produced by your most recent load operation. You can query KafkaOffsets after each KafkaSource invocation to view information about that load. If you are using the scheduler, you can also view historical load information in the stream_microbatch_history table.

For each load operation, KafkaOffsets returns the following:

- source kafka topic
- source kafka partition
- starting offset
- ending offset
- number of messages loaded
- number of bytes read
- duration of the load operation
- end message
- end reason

The following example demonstrates calling KafkaOffsets to show partition information that was loaded using KafkaSource:

```
=> SELECT kpartition, start_offset, end_offset, msg_count, ending FROM (select KafkaOffsets() over()) AS stats ORDER BY kpartition;
 kpartition | start_offset | end_offset | msg_count |   ending
------------+--------------+------------+-----------+------------
          0 |           -2 |       9999 |      1068 | END_OFFSET
```

The output shows that KafkaSource loaded 1068 messages (rows) from Kafka in a single partition. The KafkaSource ended the data load because it reached the ending offset.

> **Note**
> The values shown in the start_offset column are exclusive (the message with the shown offset was not loaded) and the values in the end_offset column are inclusive (the message with the shown offset was loaded). This is the opposite of the values specified in the KafkaSource's stream parameter. The difference between the inclusiveness of KafkaSource's and KafkaOffset's start and end offsets are based on the needs of the job scheduler. KafkaOffset is primarily intended for the job scheduler's use, so the start and end offset values are defined so the scheduler can easily start streaming from where left off.

KafkaParser

The KafkaParser does not parse data loaded from Kafka. Instead, it passes the messages through as LONG VARCHAR values. Use this parser when you want to load raw Kafka messages into Vertica for further processing. You can use this parser as a catch-all for unsupported formats.

KafkaParser does not take any parameters.

Example

The following example loads raw messages from a Kafka topic named iot-data into a table named raw_iot.

```
=> CREATE TABLE raw_iot(message LONG VARCHAR);
CREATE TABLE
=> COPY raw_iot SOURCE KafkaSource(stream='iot-data|0|-2,iot-data|1|-2,iot-data|2|-2',
                                   brokers='docd01:6667,docd03:6667', stop_on_eof=TRUE)
              PARSER KafkaParser();
 Rows Loaded
-------------
        5000
(1 row)

=> select * from raw_iot limit 10;
              message
-----------------------------------
 10039, 59, -68.951406, -19.270126
 10042, 40, -82.688712, 4.7187705
 10054, 6, -153.805268, -10.5173935
 10054, 71, -135.613150, 58.286458
 10081, 44, 130.288419, -77.344405
 10104, -5, 77.882598, -56.600744
 10132, 87, 103.530616, -69.672863
 10135, 6, -121.420382, 15.3229855
 10166, 77, -179.592211, 42.0477075
 10183, 62, 17.225394, -55.6644765
(10 rows)
```

KafkaSource

The KafkaSource UDL accesses data from a Kafka cluster. All Kafka parsers must use KafkaSource. Messages processed by KafkaSource must be at least one byte in length. KafkaSource writes an error message to vertica.log for zero-length messages.

The output of KafkaSource does not work directly with any of the non-Kafka parsers in Vertica (such as the FCSVPARSER). The KafkaParser produces additional metadata about the stream that parsers need to use in order to correctly parse the data. You must use filters such as KafkaInsertDelimiters to transform the data into a format that can be processed by other parsers. See Parsing custom formats for more an example.

You can cancel a running KafkaSource data load by using a close session function such as CLOSE_ALL_SESSIONS .

Syntax

```
KafkaSource(stream='topic_name|partition|start_offset[|end_offset]'[, param=value [,...] ] )
```

stream

Required. Defines the data to be loaded as a comma-separated list of one or more partitions. Each partition is defined by three required values and one optional value separated by pipe characters (|) :

- *topic_name* : the name of the Kafka topic to load data from. You can read from different Kafka topics in the same stream parameter, with some limitations. See Loading from Multiple Topics in the Same Stream Parameter below for more information.
- *partition* : the partition in the Kafka topic to copy.
- *start_offset* : the offset in the Kafka topic where the load will begin. This offset is inclusive (the message with the offset *start_offset* is loaded). See Special Starting Offset Values below for additional options.
- *end_offset* : the optional offset where the load should end. This offset is exclusive (the message with the offset end_offset will not be loaded). To end a load using *end_offset* , you must supply an ending offset value for all partitions in the stream parameter. Attempting to set an ending offset for some partitions and not set offset values for others results in an error.
  If you do not specify an ending offset, you must supply at least one other ending condition using stop_on_eof or duration .

brokers

A comma-separated list of host:port pairs of the brokers in the Kafka cluster. Vertica recommends running Kafka on a different machine than Vertica.

Default: localhost:9092

duration

An INTERVAL that specifies the duration of the frame. After this specified amount of time, KafkaSource terminates the COPY statements. If this parameter is not set, you must set at least one other ending condition by using stop_on_eof or specify an ending offset instead. See Duration Note below for more information.

executionparallelism

The number of threads to use when loading data. Normally, you set this to an integer value between 1 and the number of partitions the node is loading from. Setting this parameter to a reduced value limits the number of threads used to process any COPY statement. It also increases the throughput of short queries issued in the pool, especially if the queries are executed concurrently.

If you do not specify this parameter, Vertica automatically creates a thread for every partition, up to the limit allowed by the resource pool.
If the value you specify for the KafkaSource is lower than the value specified for the scheduler resource pool, the KafkaSource value applies. This value cannot exceed the value specified for the scheduler's resource pool.

stop_on_eof

Determines whether KafkaSource should terminate the COPY statement after it reaches the end of a file. If this value is not set, you must set at least one other ending condition using duration or by supplying ending offsets instead.

Default: FALSE

group_id

The name of the Kafka consumer group to which Vertica reports its progress consuming messages. Set this value to disable progress reports to a Kafka consumer group. For details, see Monitoring Vertica message consumption with consumer groups .
Default: vertica_ *database-name*

kafka_conf

A JSON-formatted object of option/value pairs to pass directly to the rdkafka library. This is the library Vertica uses to communicate with Kafka. You can use this parameter to directly set configuration options that are not available through the Vertica integration with Kafka. See Directly setting Kafka library options for details.

kafka_conf_secret

Conceals sensitive configuration data that you must pass directly to the rdkafka library, such as passwords. This parameter accepts settings in the same format as kafka_conf .
Values passed to this parameter are not logged or stored in system tables.

fail_on_conf_parse_error

Determines whether the function fails when kafka_conf contains incorrectly formatted options and values, or invalid configuration properties.
Default Value : FALSE

For accepted option and value formats, see Directly setting Kafka library options .

For a list of valid configuration properties, see the rdkafka GitHub repository .

Special starting offset values

The *start_offset* portion of the stream parameter lets you start loading messages from a specific point in the topic's partition. It also accepts one of two special offset values:

- -2 tells KafkaSource to start loading at the earliest available message in the topic's partition. This value is useful when you want to load as many messages as you can from the Kafka topic's partition.
- -3 tells KafkaSource to start loading from the consumer group's saved offset. If the consumer group does not have a saved offset, it starts loading from the earliest available message in the topic partition. See Monitoring Vertica message consumption with consumer groups for more information.

Loading from multiple topics in the same stream parameter

You can load from multiple Kafka topics in a single stream parameter as long as you follow these guidelines:

- The data for the topics must be in the same format because you pass the data from KafkaSource to a single parser. For example, you cannot load data from one topic that is in Avro format and another in JSON format.
- Similarly, you need to be careful if you are loading Avro data and specifying an external schema from a registry. The Avro parser accepts a single schema per data load. If the data from the separate topics have different schemas, then all of the data from one of the topics will be rejected.
- The data in the different topics should have the same (or very similar) schemas, especially if you are loading data into a traditional Vertica table. While you can load data with different schemas into a flex table, there are only a few scenarios where it makes sense to combine dissimilar data into a single table.

Duration note

The duration parameter applies to the length of time that Vertica allows the KafkaSource function to run. It usually reflects the amount of time the overall load statement takes. However, if KafkaSource is loading a large volume of data or the data needs extensive processing and parsing, the overall runtime of the query can exceed the amount of time specified in duration .

Example

The following example demonstrates calling KafkaSource to load data from Kafka into an existing flex table named web_table with the following options:

- The stream is named web_hits which has a single partition.
- The load starts at the earliest message in the stream (identified by passing -2 as the start offset).
- The load ends when it reaches the message with offset 1000.
- The Kafka cluster's brokers are kafka01 and kafka03 in the example.com domain.
- The brokers are listening on port 9092.
- The load ends if it reaches the end of the stream before reaching the message with offset 1000. If you do not supply this option, the connector waits until Kafka sends a message with offset 1000.
- The loaded data is sent to the KafkaJSONParser for processing.

```
=> COPY web_table
   SOURCE KafkaSource(stream='web_hits|0|-2|1000',
                      brokers='kafka01.example.com:9092,kafka03.example.com:9092',
                      stop_on_eof=true)
   PARSER KafkaJSONParser();
 Rows Loaded
-------------
        1000
(1 row)
```

To view details about this load operation, query KafkaOffsets . KafkaOffsets returns metadata about the messages that Vertica consumed from Kafka during the most recent KafkaSource invocation:

```
=> SELECT KafkaOffsets() OVER();
  ktopic  | kpartition | start_offset | end_offset | msg_count | bytes_read |    duration     |  ending  |     end_msg
----------+------------+--------------+------------+-----------+------------+-----------------+----------+-------------------
 web_hits |          0 |            0 |        999 |      1000 |     197027 | 00:00:00.385365 | END_OFFSET | Last message read
(1 row)
```

The msg_count column verifies that Vertica loaded 1000 messages, and the ending column indicates that Vertica stopped consuming messages when it reached the message with the offset 1000.

KafkaTopicDetails

Retrieves information about the specified topic from one or more Kafka brokers. This function lists details about the topic partitions, and the Kafka brokers that serve each partition in the Kafka cluster.

Syntax

```
KafkaTopicDetails(USING PARAMETERS brokers='hostname:port[,hostname2:port2...]'
                                  , topic=topic_name
                                 [, kafka_conf='option=value[;option2=value2...]']
                                 [, timeout=timeout_sec])
```

brokers
: The hostname (or IP address) of a broker in the Kafka cluster.

port
: The port number on which the broker is running.

topic
: Kafka topic that you want details for.

kafka_conf
: A semicolon-delimited list of *option* = *value* pairs to pass directly to the rdkafka library. This is the library Vertica uses to communicate with Kafka. You can use this parameter to directly set configuration options that are not available through the Vertica integration with Kafka. See Directly setting Kafka library options for details.

kafka_conf_secret

Conceals sensitive configuration data that you must pass directly to the rdkafka library, such as passwords. This parameter accepts settings in the same format as kafka_conf .

Values passed to this parameter are not logged or stored in system tables.

timeout

Integer number of seconds to wait for a response from the Kafka cluster.

Example

```
=> SELECT KafkaTopicDetails(USING PARAMETERS brokers='kafka1-01.example.com:9092',topic='iot_data') OVER();
 partition_id | lead_broker | replica_brokers | in_sync_replica_brokers
--------------+-------------+-----------------+-------------------------
            0 |           0 | 0               | 0
            1 |           1 | 1               | 1
            2 |           0 | 0               | 0
            3 |           1 | 1               | 1
            4 |           0 | 0               | 0
(5 rows)
```

Data streaming schema tables

Every time you create a scheduler (--create), Vertica creates a schema for that scheduler with the name you specify or the default stream_config . Each schema has the following tables:

- stream_clusters
- stream_events
- stream_load_specs
- stream_microbatch_history
- stream_microbatch_source_map
- stream_microbatches
- stream_scheduler
- stream_scheduler_history
- stream_sources
- stream_targets

Caution

Vertica recommends that you do not alter these tables except in consultation with support.

In this section

- stream_clusters
- stream_events
- stream_load_specs
- stream_lock
- stream_microbatch_history
- stream_microbatch_source_map
- stream_microbatches
- stream_scheduler
- stream_scheduler_history
- stream_sources
- stream_targets

stream_clusters

This table lists clusters and hosts. You change settings in this table using the vkconfig cluster tool. See Cluster tool options for more information.

Column	Data Type	Description
id	INTEGER	The identification number assigned to the cluster.
cluster	VARCHAR	The name of the cluster.

hosts	VARCHAR	A comma-separated list of hosts associated with the cluster.

Examples

This example shows a cluster and its associated hosts.

```
=> SELECT * FROM stream_config.stream_clusters;

   id    |    cluster     |               hosts
---------+----------------+-----------------------------------
 2250001 | streamcluster1 | 10.10.10.10:9092,10.10.10.11:9092
(1 rows)
```

stream_events

This table logs microbatches and other important events from the scheduler in an internal log table.

This table was renamed from kafka_config.kafka_events.

Column	Data Type	Description
event_time	TIMESTAMP	The time the event was logged.
log_level	VARCHAR	The type of event that was logged. **Valid Values:** • TRACE • DEBUG • FATAL • ERROR • WARN • INFO **Default:** INFO
frame_start	TIMESTAMP	The time when the frame executed.
frame_end	TIMESTAMP	The time when the frame completed.
microbatch	INTEGER	The identification number of the associated microbatch.
message	VARCHAR	A description of the event.
exception	VARCHAR	If this log is in the form of a stack trace, this column lists the exception.

Examples

This example shows typical rows from the stream_events table.

```
=> SELECT * FROM stream_config.stream_events;
-[ RECORD 1 ]-+-------------
event_time    | 2016-07-17 13:28:35.548-04
log_level     | INFO
frame_start   |
frame_end     |
microbatch    |
message       | New leader registered for schema stream_config. New ID: 0, new Host: 10.20.30.40
exception     |
-[ RECORD 2 ]-+-------------
event_time    | 2016-07-17 13:28:45.643-04
log_level     | INFO
frame_start   | 2015-07-17 12:28:45.633
frame_end     | 2015-07-17 13:28:50.701-04
microbatch    |
message       | Generated tuples: test3|2|-2,test3|1|-2,test3|0|-2
exception     |
-[ RECORD 3 ]-+----------------
event_time    | 2016-07-17 14:28:50.701-04
log_level     | INFO
frame_start   | 2016-07-17 13:28:45.633
frame_end     | 2016-07-17 14:28:50.701-04
microbatch    |
message       | Total rows inserted: 0
exception     |
```

stream_load_specs

This table describes user-created load specs. You change the entries in this table using the vkconfig utility's load spec tool .

Column	Data Type	Description
id	INTEGER	The identification number assigned to the cluster.
load_spec	VARCHAR	The name of the load spec.
filters	VARCHAR	A comma-separated list of UDFilters for the scheduler to include in the COPY statement it uses to load data from Kafka.
parser	VARCHAR	A Vertica UDParser to use with a specified target. If you are using a Vertica native parser, parser parameters serve as a COPY statement parameters.
parser_parameters	VARCHAR	A list of parameters to provide to the parser.
load_method	VARCHAR	The COPY load method to use for all loads with this scheduler. **Deprecated** In Vertica 10.0, load methods are no longer used due to the removal of the WOS. The value shown in this column has no effect.
message_max_bytes	INTEGER	The maximum size, in bytes, of a message.
uds_kv_parameters	VARCHAR	A list of parameters that are supplied to the KafkaSource statement. If the value in this column is in the format *key* = *value* , the scheduler it to the COPY statement's KafkaSource call.

Examples

This example shows the load specs that you can use with a Vertica instance.

```
SELECT * FROM stream_config.stream_load_specs;
-[ RECORD 1 ]-----+------------
id                | 1
load_spec         | loadspec2
filters           |
parser            | KafkaParser
parser_parameters |
load_method       | direct
message_max_bytes | 1048576
uds_kv_parameters |
-[ RECORD 2 ]-----+------------
id                | 750001
load_spec         | streamspec1
filters           |
parser            | KafkaParser
parser_parameters |
load_method       | TRICKLE
message_max_bytes | 1048576
uds_kv_parameters |
```

stream_lock

This table is locked by the scheduler. This locks prevent multiple schedulers from running at the same time. The scheduler that locks this table updates it with its own information.

Important
Do not use this table in a serializable transaction that locks this table. Locking this table can interfere with the operation of the scheduler.

Column	Data Type	Description
scheduler_id	INTEGER	A unique ID for the scheduler instance that is currently running.
update_time	TIMESTAMP	The time the scheduler took ownership of writing to the schema.
process_info	VARCHAR	Information about the scheduler process. Currently unused.

Example

```
=> SELECT * FROM weblog_sched.stream_lock;
 scheduler_id |       update_time       | process_info
--------------+-------------------------+--------------
            2 | 2018-11-08 10:12:36.033 |
(1 row)
```

stream_microbatch_history

This table contains a history of every microbatch executed within this scheduler configuration.

Column	Data Type	Description
source_name	VARCHAR	The name of the source.
source_cluster	VARCHAR	The name of the source cluster. The clusters are defined in stream_clusters .
source_partition	INTEGER	The number of the data streaming partition.
start_offset	INTEGER	The starting offset of the microbatch.

end_offset	INTEGER	The ending offset of the microbatch.
end_reason	VARCHAR	An explanation for why the batch ended.The following are valid end reasons: • DEADLINE - The batch ran out of time. • END_OFFSET - The load reached the ending offset specified in the KafkaSource. This reason is never used by the scheduler, as it does specify an end offset. • END_OF_STREAM - There are no messages available to the scheduler or the eof_timeout has been reached. • NETWORK_ERROR - The scheduler could not connect to Kafka. • RESET_OFFSET - The start offset was changed using the --update and --offset parameters to the KafkaSource. This state does not occur during normal scheduler operations. • SOURCE_ISSUE - The Kafka service returned an error. • UNKNOWN - The batch ended for an unknown reason.
end_reason_message	VARCHAR	If the end reason is a network or source issue, this column contains a brief description of the issue.
partition_bytes	INTEGER	The number of bytes transferred from a source partition to a Vertica target table.
partition_messages	INTEGER	The number of messages transferred from a source partition to a Vertica target table.
microbatch_id	INTEGER	The Vertica transaction id for the batch session.
microbatch	VARCHAR	The name of the microbatch.
target_schema	VARCHAR	The name of the target schema.
target_table	VARCHAR	The name of the target table.
timeslice	INTERVAL	The amount of time spent in the KafkaSource operator.
batch_start	TIMESTAMP	The time the batch executed.
batch_end	TIMESTAMP	The time the batch completed.
last_batch_duration	INTERVAL	The length of time required to run the complete COPY statement.
last_batch_parallelism	INTEGER	The number of parallel COPY statements generated to process the microbatch during the last frame.
microbatch_sub_id	INTEGER	The identifier for the COPY statement that processed the microbatch.
consecutive_error_count	INTEGER	(Currently not used.) The number of times a microbatch has encountered an error on an attempt to load. This value increases over multiple attempts.
transaction_id	INTEGER	The identifier for the transaction within the session.
frame_start	TIMESTAMP	The time the frame started. A frame can contain multiple microbatches.
frame_end	TIMESTAMP	The time the frame completed.

Examples

This example shows typical rows from the stream_microbatch_history table.

```
=> SELECT * FROM stream_config.stream_microbatch_history;

-[ RECORD 1 ]--+--------------------------
source_name              | streamsource1
source_cluster           | kafka-1
source_partition         | 0
start_offset             | 196
end_offset               | 196
end_reason               | END_OF_STREAM
partition_bytes          | 0
partition_messages       | 0
microbatch_id            | 1
microbatch               | mb_0
target_schema            | public
target_table             | kafka_flex_0
timeslice                | 00:00:09.892
batch_start              | 2016-07-28 11:31:25.854221
batch_end                | 2016-07-28 11:31:26.357942
last_batch_duration      | 00:00:00.379826
last_batch_parallelism   | 1
microbatch_sub_id        | 0
consecutive_error_count |
transaction_id           | 45035996275130064
frame_start              | 2016-07-28 11:31:25.751
frame_end                |
end_reason_message       |

-[ RECORD 2 ]--+--------------------------
source_name              | streamsource1
source_cluster           | kafka-1
source_partition         | 1
start_offset             | 197
end_offset               | 197
end_reason               | NETWORK_ISSUE
partition_bytes          | 0
partition_messages       | 0
microbatch_id            | 1
microbatch               | mb_0
target_schema            | public
target_table             | kafka_flex_0
timeslice                | 00:00:09.897
batch_start              | 2016-07-28 11:31:45.84898
batch_end                | 2016-07-28 11:31:46.253367
last_batch_duration      | 000:00:00.377796
last_batch_parallelism   | 1
microbatch_sub_id        | 0
consecutive_error_count |
transaction_id           | 45035996275130109
frame_start              | 2016-07-28 11:31:45.751
frame_end                |
end_reason_message       | Local: All brokers are down
```

stream_microbatch_source_map

This table maps microbatches to the their associated sources.

Column	Data Type	Description
microbatch	INTEGER	The identification number of the microbatch.

source	INTEGER	The identification number of the associated source.

Examples

This example shows typical rows from the stream_microbatch table.

```
SELECT * FROM stream_config.stream_microbatch_source_map;
microbatch | source
-----------+--------
         1 |      4
         3 |      2
(2 rows)
```

stream_microbatches

This table contains configuration data related to microbatches.

Column	Data Type	Description
id	INTEGER	The identification number of the microbatch.
microbatch	VARCHAR	The name of the microbatch.
target	INTEGER	The identification number of the target associated with the microbatch.
load_spec	INTEGER	The identification number of the load spec associated with the microbatch.
target_columns	VARCHAR	The table columns associated with the microbatch.
rejection_schema	VARCHAR	The schema that contains the rejection table.
rejection_table	VARCHAR	The table where Vertica stores messages that are rejected by the database.
max_parallelism	INTEGER	The number of parallel COPY statements the scheduler uses to process the microbatch.
enabled	BOOLEAN	When TRUE, the microbatch is enabled for use.
consumer_group_id	VARCHAR	The name of the Kafka consumer group to report loading progress to. This value is NULL if the microbatch reports its progress to the default consumer group for the scheduler. See Monitoring Vertica message consumption with consumer groups for more information.

Examples

This example shows a row from a typical stream_microbatches table.

```
=> select * from weblog_sched.stream_microbatches;
-[ RECORD 1 ]-----+----------
id                | 750001
microbatch        | weberrors
target            | 750001
load_spec         | 2250001
target_columns    |
rejection_schema  |
rejection_table   |
max_parallelism   | 1
enabled           | t
consumer_group_id |
-[ RECORD 2 ]-----+----------
id                | 1
microbatch        | weblog
target            | 1
load_spec         | 1
target_columns    |
rejection_schema  |
rejection_table   |
max_parallelism   | 1
enabled           | t
consumer_group_id | weblog_group
```

stream_scheduler

This table contains metadata related to a single scheduler.

This table was renamed from kafka_config.kafka_scheduler. This table used to contain a column named eof_timeout_ms. It has been removed.

Column	Data Type	Description
version	VARCHAR	The version of the scheduler.
frame_duration	INTERVAL	The length of time of the frame. The default is 00:00:10.
resource_pool	VARCHAR	The resource pool associated with this scheduler.
config_refresh	INTERVAL	The interval of time that the scheduler runs before applying any changes to its metadata, such as, changes made using the --update option. For more information, refer to --config-refresh in Scheduler tool options .
new_source_policy	VARCHAR	When during the frame that the source runs. Set this value with the --new-source-policy in Source tool options . **Valid Values:** • FAIR: Takes the average length of time from the previous batches and schedules itself appropriately. • START: Runs all new sources at the beginning of the frame. In this case, Vertica gives the minimal amount of time to run. • END: Runs all new sources starting at the end of the frame. In this case, Vertica gives the maximum amount of time to run. **Default:** FAIR

pushback_policy	VARCHAR	(Not currently used.) How Vertica handles delays for microbatches that continually fail. **Valid Values:** • FLAT • LINEAR • EXPONENTIAL **Default:** LINEAR
pushback_max_count	INTEGER	(Currently not used.) The maximum number of times a microbatch can fail before Vertica terminates it.
auto_sync	BOOLEAN	When TRUE, the scheduler automatically synchronizes source information with host clusters. For more information, refer to Automatically consume data from Kafka with the scheduler . **Default:** TRUE
consumer_group_id	VARCHAR	The name of the Kafka consumer group to which the scheduler reports its progress in consuming messages. This value is NULL if the scheduler reports to the default consumer group named vertica- *database_name* . See Monitoring Vertica message consumption with consumer groups for more information.

Examples

This example shows a typical row in the stream_scheduler table.

```
=> SELECT * FROM weblog_sched.stream_scheduler;
-[ RECORD 1 ]------+----------------------
version            | v9.2.1
frame_duration     | 00:05:00
resource_pool      | weblog_pool
config_refresh     | 00:05
new_source_policy  | FAIR
pushback_policy    | LINEAR
pushback_max_count | 5
auto_sync          | t
consumer_group_id  | vertica-consumer-group
```

stream_scheduler_history

This table shows the history of launched scheduler instances.

This table was renamed from kafka_config.kafka_scheduler_history.

Column	Data Type	Description
elected_leader_time	TIMESTAMP	The time when this instance took began scheduling operations.
host	VARCHAR	The host name of the machine running the scheduler instance.
launcher	VARCHAR	The name of the currently active scheduler instance. **Default:** NULL
scheduler_id	INTEGER	The identification number of the scheduler.
version	VARCHAR	The version of the scheduler.

Examples

This example shows typical rows from the stream_scheduler_history table.

```
SELECT * FROM stream_config.stream_scheduler_history;
  elected_leader_time   |     host     |     launcher      | scheduler_id | version
------------------------+--------------+-------------------+--------------+---------
2016-07-26 13:19:42.692 | 10.20.100.62 |                   |            0 | v8.0.0
2016-07-26 13:54:37.715 | 10.20.100.62 |                   |            1 | v8.0.0
2016-07-26 13:56:06.785 | 10.20.100.62 |                   |            2 | v8.0.0
2016-07-26 13:56:56.033 | 10.20.100.62 | SchedulerInstance |            3 | v8.0.0
2016-07-26 15:51:20.513 | 10.20.100.62 | SchedulerInstance |            4 | v8.0.0
2016-07-26 15:51:35.111 | 10.20.100.62 | SchedulerInstance |            5 | v8.0.0
  (6 rows)
```

stream_sources

This table contains metadata related to data streaming sources.

This table was formerly named kafka_config.kafka_scheduler.

Column	Data Type	Description
id	INTEGER	The identification number of the source
source	VARCHAR	The name of the source.
cluster	INTEGER	The identification number of the cluster associated with the source.
partitions	INTEGER	The number of partitions in the source.
enabled	BOOLEAN	When TRUE, the source is enabled for use.

Examples

This example shows a typical row from the stream_sources table.

```
select * from  stream_config.stream_sources;
-[ RECORD 1 ]----------
  id         | 1
  source     | SourceFeed1
  cluster    | 1
  partitions | 1
  enabled    | t
-[ RECORD 2 ]----------
  id         | 250001
  source     | SourceFeed2
  cluster    | 1
  partitions | 1
  enabled    | t
```

stream_targets

This table contains the metadata for all Vertica target tables.

The table was formerly named kafka_config.kafka_targets.

Column	Data Type	Description
id	INTEGER	The identification number of the target table
target_schema	VARCHAR	The name of the schema for the target table.
target_table	VARCHAR	The name of the target table.

Examples

This example shows typical rows from the stream_tables table.

```
=> SELECT * FROM stream_config.stream_targets;
-[ RECORD 1 ]-----+--------------------
id                | 1
target_schema     | public
target_table      | stream_flex1
-[ RECORD 2 ]-----+--------------------
id                | 2
target_schema     | public
target_table      | stream_flex2
```

Apache Spark integration

Apache Spark is an open-source, general-purpose cluster-computing framework. See the Apache Spark website for more information. Vertica provides a Spark connector that you install into Spark that lets you transfer data between Vertica and Spark.

Using the connector, you can:

- Copy large volumes of data from Spark DataFrames to Vertica tables. This feature lets you save your Spark analytics in Vertica.
- Copy data from Vertica to Spark RDDs or DataFrames for use with Python, R, Scala and Java. The connector efficiently pushes down column selection and predicate filtering to Vertica before loading the data.

The connector lets you use Spark to preprocess data for Vertica and to use Vertica data in your Spark application. You can even round-trip data from Vertica to Spark—copy data from Vertica to Spark for analysis, and then save the results of that analysis back to Vertica.

See the spark-connector project on GitHub for more information about the connector.

How the connector works

The Spark connector is a library that you incorporate into your Spark applications to read data from and write data to Vertica. When transferring data, the connector uses an intermediate storage location as a buffer between the Vertica and Spark clusters. Using the intermediate storage location lets both Vertica and Spark use all of the nodes in their clusters to transfer data in parallel.

When transferring data from Vertica to Spark, the connector tells Vertica to write the data as Parquet files in the intermediate storage location. As part of this process, the connector pushes down the required columns and any Spark data filters into Vertica as SQL. This push down lets Vertica pre-filter the data so it only copies the data that Spark needs. Once Vertica finishes copying the data, the connector has Spark load it into DataFrames from the intermediate location.

When transferring data from Spark to Vertica, the process is reversed. Spark writes data into the intermediate storage location. It then connects to Vertica and runs a COPY statement to load the data from the intermediate location into a table.

Getting the connector

The Spark connector is an open source project. For the latest information on it, visit the spark-connector project on GitHub. To get an alert when there are updates to the connector, you can log into a GitHub account and click the **Notifications** button on any of the project's pages.

You have three options to get the Spark connector:

- Get the connector from Maven Central . If you use Gradle, Maven, or SBT to manage your Spark applications, you can add a dependency to your project to automatically get the connector and its dependencies and add them into your Spark application. See Getting the Connector from Maven Central below.
- Download a precompiled assembly from the GitHub project's releases page . Once you have downloaded the connector, you must configure Spark to use it. See Deploy the Connector to the Spark Cluster below.
- Clone the connector project and compile it. This option is useful if you want features or bugfixes that have not been released yet. See Compiling the Connector below.

See the Spark connector project on GitHub for detailed instructions on deploying and using the connector.

Getting the connector from maven central

The Vertica Spark connector is available from the Maven Central Repository. Using Maven Central is the easiest method of getting the connector if your build tool supports downloading dependencies from it.

If your Spark project is managed by Gradle, Maven, or SBT you can add the Spark connector to it by listing it as a dependency in its configuration file. You may also have to enable Maven Central if your build tool does not automatically do so.

For example, suppose you use SBT to manage your Spark application. In this case, the Maven Central repository is enabled by default. All you need to do is add the com.vertica.spark dependency to your build.sbt file.

See the com.vertica.spark page on the Maven Repository site for more information on the available Spark connector versions and dependency information for your build system.

Compiling the connector

You may choose to compile the Spark connector if you want to test new features or bugfixes that have not yet been released. Compiling the connector is necessary if you plan on contributing your own features.

To compile the connector, you need:

- The SBT build tool. In order to compile the connector, you must install SBT and all of its dependencies (including a version of the Java SDK). See the SBT documentation for requirements and installation instructions.
- git to clone the Spark connector V2 source from the GitHub.

As a quick overview, executing the following commands on a Linux command line will download the source and compile it into an assembly file:

```
$ git clone https://github.com/vertica/spark-connector.git
$ cd spark-connector/connector
$ sbt assembly
```

Once compiled, the connector is located at target/scala- *n.n* /spark-vertica-connector-assembly- *x.x* .jar . The *n.n* is the currently-supported version of Scala and *x.x* is the current version of the Spark connector.

See the CONTRIBUTING document in the connector's GitHub project for detailed requirements and compiling instructions.

Once you compile the connector, you must deploy it to your Spark cluster. See the next section for details.

Deploy the connector to the Spark cluster

If you downloaded the Spark connector from GitHub or compiled it yourself, you must deploy it to your Spark cluster before you can use it. Two options include copying it to a Spark node and including it in a spark-submit or spark-shell command or deploying it to the entire cluster and having it loaded automatically.

Loading the Spark connector from the command line

The quickest way to use the connector is to include it in the --jars argument when executing a spark-submit or spark-shell command. To be able to use the connector in the command line, you must first copy its assembly JAR to the Spark node on which you will run the commands. Then add the path to the assembly JAR file as part of the --jars command line argument.

For example, suppose you copied the assembly file to your current directory on a Spark node. Then you could load the connector when starting spark-shell with the command:

```
spark-shell --jars spark-vertica-connector-assembly-x.x.jar
```

You could also enable the connector when submitting a Spark job using spark-submit with the command:

```
spark-submit --jars spark-vertica-connector-assembly-x.x.jar
```

Configure Spark to automatically load the connector

You can configure your Spark cluster to automatically load the connector. Deploying the connector this way ensures that the Spark connector is available to all Spark applications on your cluster.

To have Spark automatically load the Spark connector:

1. Copy the Spark connector's assembly JAR file to the same path on all of the nodes in your Spark cluster. For example, on Linux you could copy the spark-vertica-connector-assembly- *x.x* .jar file to the /usr/local/lib directory on every Spark node. The file must be in the same location on each node.
2. In your Spark installation directories on each node in your cluster, edit the conf/spark-defaults.conf file to add or alter the following line:

   ```
   spark.jars /path_to_assembly/spark-vertica-connector-assembly-x.x.jar
   ```

   For example, if you copied the assembly JAR file to /usr/local/lib , you would add:

   ```
   spark.jars /usr/local/lib/spark-vertica-connector-assembly-x.x.jar
   ```

   Note

If the spark.jars line already exists in the configuration file, add a comma to the end of the line, then add the path to the assembly file. Do not add a space between the comma and the surrounding values.

If you have the JAR files for the Spark connector V1 in your configuration file, you do not need to remove them. Both connector versions can be loaded into Spark at the same time without a conflict.

3. Test your configuration by starting a Spark shell and entering the statement:

```
import com.vertica.spark._
```

If the statement completes successfully, then Spark was able to locate and load the Spark connector library correctly.

Prior connector versions

The Vertica Spark connector is an open source project available from GitHub. It is released on a separate schedule than the Vertica server.

Versions of Vertica prior to 11.0.2 distributed a closed-source proprietary version of the Spark connector. This legacy version is no longer supported, and is not distributed with the Vertica server install package.

The newer open source Spark connector differs from the old one in the following ways:

- The new connector uses the Spark V2 API. Using this newer API makes it more future-proof than the legacy connector, which uses an older Spark API.
- The primary class name has changed. Also, the primary class has several renamed configuration options and a few removed options. See Migrating from the legacy Vertica Spark connector for a list of these changes.
- It supports more features than the older connector, such as Kerberos authentication and S3 intermediate storage.
- It comes compiled as an assembly which contains supporting libraries such as the Vertica JDBC library.
- It is distributed separate from the Vertica server. You can directly download it from the GitHub project. It is also available from the Maven Central Repository, making it easier for you to integrate it into your Gradle, Maven, or SBT workflows.
- It is an open-source project that is not tied to the Vertica server release cycle. New features and bug fixes do not have to wait for a Vertica release. You can also contribute your own features and fixes.

If you have Spark applications that used the previous version of the Spark connector, see Migrating from the legacy Vertica Spark connector for instructions on how to update your Spark code to work with the new connector.

If you are still using a version of Spark earlier than 3.0, you must use the legacy version of the Spark connector. The new open source version is incompatible with older versions of Spark. You can get the older connector by downloading and extracting a Vertica installation package earlier than version 11.0.2.

In this section

- Migrating from the legacy Vertica Spark connector

Migrating from the legacy Vertica Spark connector

If you have existing Spark applications that used the closed-source Vertica Spark connector, you must update them to work with the newer open-source connector. The new connector offers new features, better performance, and is under ongoing development. The old connector has been removed from service.

Deployment changes between the old and connectors

The legacy connector was only distributed with the Vertica server install. The new connector is distributed via several channels giving you more ways to deploy it.

The new Spark connector is available from Maven Central. If you use Gradle, Maven, or SBT to manage your Spark applications, you may find it more convenient to deploy the Spark connector using a dependency rather than manually installing it on your Spark cluster. Integrating the connector into your Spark project as a dependency makes updating to newer versions of the connector easy—just update the required version in the dependency. See Getting the Connector from Maven Central for more information.

You can also download the precompiled connector assembly or build it from source. In this case, you must deploy the connector to your Spark cluster. The legacy connector depended on the Vertica JDBC driver and required that you separately include it. The new connector is an assembly that incorporates all of its dependencies, including the JDBC driver. You only need to deploy a single JAR file containing the Spark connector to your Spark cluster.

You can have Spark load both the legacy and new connector at the same time because the new connector's primary class name is different (see below). This renaming lets you add the new connector to your Spark configuration files without having to immediately port all of your Spark applications that use the legacy connector to the new API. You can just add the new assembly JAR file to spark-jars list in the spark-defaults.conf file.

API changes

There are several API changes from the legacy connector to the new connector require changes to your Spark application.

VerticaRDD class no longer supported

The legacy connector supported a class named VerticaRDD to load data from Vertica using the Spark resilient distributed dataset (RDD) feature. The new connector does not support this separate class. Instead, if you want to directly manipulate an RDD, access it through the DataFrame object you create using the DataSource API.

DefaultSource class renamed VerticaSource

The primary class in the legacy connector is named DataSource . In the new connector, this class has been renamed to VerticaSource . This renaming lets both connectors coexist, allowing you to gradually transition your Spark applications.

For your existing Spark application to use the new connector, you must change calls to the DataSource class to the VerticaSource class. For example, suppose your Spark application has this method call to read data from the legacy connector:

```
spark.read.format("com.vertica.spark.datasource.DefaultSource").options(opts).load()
```

Then to have it use the new connector, use this method call:

```
spark.read.format("com.vertica.spark.datasource.VerticaSource").options(opts).load()
```

Changed API options

In addition to renaming of the DataSource class to VerticaSource , some of the names for options for the primary connector class have changed. Other options are no longer supported. If you are porting a Spark application from the legacy to the new connector that uses one of the following options, you must update your code:

Legacy DataSource Option	New VerticaSource Option	Description
fileformat	none	The new connector does not support the fileformat option. The files that Vertica and Spark write to the intermediate storage location are always in parquet format.
hdfs_url	staging_fs_url	The location of the intermediate storage location that Vertica and Spark use to exchange data. Renamed to be more general, as the new connector supports storage platforms in addition to HDFS.
logging_level	none	The connector no longer supports setting a logging level. Instead, set the logging level in Spark.
numpartitions	num_partitions	The number of Spark partitions to use when reading data from Vertica.
target_table_ddl	target_table_sql	A SQL statement for Vertica to execute before loading data from Spark.
web_hdfs_url	staging_fs_url	Use the same option for a web HDFS URL as you would for an S3 or HDFS storage location.

In addition, the new connector has added options to support new features such as Kerberos authentication. For details on the connector's VerticaSource options API, see the Vertica Spark connector GitHub project .

Take advantage of new features

The new Vertica Spark connector offers new features that you may want to take advantage of.

Currently, the most notable new features are:

- Kerberos authentication. This feature lets you configure the connector for passwordless connections to Vertica. See the Kerberos documentation in the Vertica Spark connector GitHub project for details of using this feature.
- Support for using S3 for intermediate storage. This option lets you avoid having to set up a Hadoop cluster solely to host an HDFS storage location. See the S3 user guide at the connector GitHub site.

Voltage SecureData integration

Voltage SecureData is a suite of encryption technologies that let you integrate end-to-end data encryption into other applications. It uses Format-Preserving Encryption (FPE): the encrypted values have the same overall format as the unencrypted data. This feature means you do not have to change the data types of table columns that you want to encrypt. It also preserves reference integrity: the encrypted values have the same sort order as unencrypted data, and encrypted values can be cross-referenced between tables, as long as each instance of the value is encrypted with the same key.

See the Voltage web site for more about SecureData.

This section explains how you can integrate the SecureData encryption feature into Vertica.

In this section

- How Vertica and SecureData work together
- Requirements for integrating with SecureData
- Best practices for safe unicode FPE
- Verifying the Vertica server's access to the SecureData CA certificate
- Configuring access to SecureData
- Encrypting, decrypting, and hashing data
- Encapsulating encryption business logic with SQL macros
- Granting users access to the Voltage SecureData integration functions
- Automating encryption and decryption with access policies
- Querying eFPE encrypted columns
- Voltage SecureData integration function reference

How Vertica and SecureData work together

Vertica provides functions to encrypt and decrypt data using SecureData.

Voltage SecureData supplies a number of interfaces for applications to use its encryption: web-based APIs, command-line tools, and SDKs for C, C#, and Java. Vertica has developed a connector that calls the SecureData API that lets you:

- Encrypt sensitive data as it is being loaded into Vertica using SecueData's FPE feature. You can ensure data is stored in Vertica in its encrypted state (referred to as "encrypted at rest"). Authorized users can decrypt the data as needed. Unauthorized users only see the encrypted values. Decryption for authorized users can be automated using views or access policies. The data is transparently decrypted for them.
- Encrypt semi-sensitive data that is stored unencrypted in Vertica so unauthorized users only see a masked version of the data. You an also automate this on-the-fly encryption using access policies.

The encryption method you choose depends in the data you are processing. For example, regulations or contracts may require you to encrypt specific pieces of data. In these cases, use SecureData to encrypt your data as it is loaded, so it is never stored in an unencrypted format within Vertica.

In other cases, you may have semi-sensitive data that you can choose between the two options. In these cases, choose the method that requires the least number encryptions or decryptions (and therefore of calls to SecureData). If most of the queries on the data need to be masked from users who should not see the unencrypted values, then encrypt the data at rest. Alternatively, if the most of the queries will be from authorized users, with only occasional queries where the data should be masked, then store the data in an unencrypted format and use on-the-fly encryption to mask the data.

Requirements for integrating with SecureData

Before you can use SecureData with Vertica, you must verify the following:

- You are using version 6.0 or later of SecureData. The Vertica integration functions use APIs introduced in version 6.0 of SecureData.
- All of the nodes in your Vertica cluster are able to communicate with all of your SecureData appliance hosts. Any firewalls between your Vertica cluster and your SecureData appliance must allow connections on the ports that SecureData uses for communications. See "Ensuring Access to SecureData Services" in the *Voltage SecureData Appliance Installation Guide* (downloadable with the SecureData ISO) for a list of the ports reserved by SecureData.
- Your Vertica database is not using Federal information processing standard (FIPS) mode. FIPS is incompatible with the Voltage SecureData integration. When FIPS mode is active, the SecureData integration library is not installed by default. Do not manually install this library if you are using FIPS.
- You are following the best practices when using Safe Unicode FPE formats.

Best practices for safe unicode FPE

Starting with Voltage SecureData Simple API 6.0, you can use Safe Unicode Format Preserving Encryption (FPE) formats to encrypt and decrypt Unicode strings.

When encrypting Unicode with Safe Unicode FPE formats, the plaintext passed into the VoltageSecureProtect function is first encrypted with an alphabet of all Unicode code points, and then encoded with the Base32K-encoding to produce encrypted text in Unicode Normalization Form C (NFC). The encrypted text will generally consist of 3-byte Chinese and Japanese characters, as they account for a significant portion of the NFC-stable alphabet. VoltageSecureAccess reverses this process.

Unlike regular FPE, which handles ASCII strings consisting of single bytes, Safe Unicode FPE handles strings consisting of variable-length code points. This variability introduces an important complication: Safe Unicode FPE formats like UNICODE_BASE32K are not length-preserving. The encryption algorithm guarantees that the encrypted text will be larger than the original, and failing to account for this expansion can lead to truncated or otherwise improperly stored encrypted text, making decryption impossible.

Additionally, Unicode allows semantically equivalent characters to be encoded in different ways, which means that unnormalized, semantically equivalent plaintexts can have different forms when encrypted, compromising referential integrity. To prevent this, you should always normalize your plaintext strings into their NFC forms before encryption. For more information on Unicode normalization, see the Unicode Consortium's Normalization FAQ.

The encrypted text is guaranteed to be larger than the plaintext in the following ways:

- 16/15ths longer in character-length (rounded up)
- up to 4-times larger (in bytes)

For a list of predefined formats for Safe Unicode FPE, consult your copy of the Voltage SecureData Simple API documentation.

Checklist for safe unicode FPE

Before using Safe Unicode FPE:

- Normalize your plaintext to its NFC form before encryption.
- For fixed-length data types: remove any padding from the plaintext string.
- To store the encrypted text, ensure that the columns storing the encrypted text can handle:
  - strings 16/15ths times longer than your longest plaintext string
  - strings up to 4-times larger (in bytes) than your largest plaintext string

Verifying the Vertica server's access to the SecureData CA certificate

Before you can use SecureData with Vertica, you must verify that the root certificate authority (CA) and any intermediate certificate authority used to sign the SecureData Appliance's certificate is in the Vertica server's trust store (/opt/vertica/packages/voltagesecure/trustStore/). Vertica supplies many standard root certificates in this directory. If your SecureData Appliance uses a certificate signed by a standard CA authority, it is likely already in the trust store.

If your SecureData Appliance is using a certificate signed by your own internal CA authority , you must add this CA Certificate to the Vertica trust store.

If you are unsure whether your CA Certificate is in the Vertica trust store, follow the steps under Troubleshooting Certificate Problems to test whether the Vertica already has the CA certificate. If you are able to retrieve the client policy XML file from the SecureData Appliance, then your Vertica cluster has the correct CA certificate to access SecureData.

Adding the CA certificate to Vertica

You must add the CA to Vertica trust store before using the SecureData Integration if you used:

- Your own CA certificate to sign your SecureData Appliance's certificate.
- A third-party CA that is not in the Vertica trust store.

To add the CA certificate to the Vertica trust store, you need:

- The certificate authority (CA) file used to sign the SecureData Appliance's certificate. This file must be in Privacy Enhanced Mail (.pem) format. The file name does not matter, as long as it has the .pem extension.
- Access to the dbadmin account on the Vertica nodes. This access is required in order to copy the certificate file to trust store directory in the Vertica installation.

To add the necessary CA file to Vertica:

1. Login to one of the Vertica nodes as the dbadmin user.
2. Copy the .pem file to the /opt/vertica/packages/voltagesecure/trustStore/ directory. You only need to copy this file to a single node. Vertica takes care of distributing the file to the rest of the nodes in the cluster.

3. On the Linux command line, execute the following command to reinstall the SecureData integration library:

```
$ admintools -t install_package -d database_name -p 'password' --package voltagesecure --force-reinstall
```

When Vertica reinstalls the SecureData integration library, it copies the CA authority file to the all nodes in the cluster. After the file is distributed, all Vertica nodes can authenticate with the SecureData Appliance.

For example, suppose:

- Your certificate file is named my_ca.cert.pem , and you have copied it to the dbadmin home directory on node in your cluster.
- Your database is named VMart.

Then the process of installing the CA file would look like this:

```
$ cp my_ca.cert.pem /opt/vertica/packages/voltagesecure/trustStore/
$ admintools -t install_package -d VMart -p dbadminpassword --package voltagesecure --force-reinstall
Installing package voltagesecure...
...Success!
```

Troubleshooting certificate problems

You can test whether the Vertica trust store has the correct certificate by executing the following statement from the Linux command line:

```
curl --capath /vertica_catalog_directory/Libraries/\
$(vsql -A -t -c "SELECT sal_storage_id from user_libraries WHERE lib_name = 'VoltageSecureLib';")\
https://SecureData_applicance_hostname/policy/clientPolicy.xml
```

Where:

- *vertica_catalog_directory* is the absolute path to the Vertica catalog directory. See Understanding the catalog directory for more information about the catalog directory.
- *SecureData_appliance_hostname* is the host name of your Voltage SecureData Appliance.

For example, suppose you are connected to node0001 of the example VMart database. Also, your Voltage SecureData appliance's host name is voltage-pp-0000.example.com. Then you would use the following command to test your certificate installation.

```
$ curl --capath /home/dbadmin/VMart/v_vmart_node0001_catalog/Libraries/\
$(vsql -A -t -c "SELECT sal_storage_id from user_libraries WHERE lib_name = 'VoltageSecureLib';") \
https://voltage-pp-0000.example.com/policy/clientPolicy.xml


<clientPolicy version="2">

<server name="SecureDataAppliance" version="6.4.2.232000" />

<localDomains>example.com</localDomains>

<userWhitelist></userWhitelist>

<defaultDistrict value="0" />

<sendUniversalReader value="1" />

<messageFooterGlobal></messageFooterGlobal>

<parameterAggressiveDistricts>example.com</parameterAggressiveDistricts>

<localPolicyLocked value="0" />

<trustedDistricts></trustedDistricts>

<fallThroughDistrict>example.com</fallThroughDistrict>
    .   .   .
```

The <clientPolicy>... output (which is the content of the clientPolicy.xml file) indicates that the Vertica node was able to use its CA certificate to connect to the SecureData Appliance.

> **Tip**
> If you are unsure which Format Preserving Encryption (FPE) formats are defined in your SecureData Appliance, examine the output of the curl command. Look for the tags with formatName attributes which describe each of the formats.

If the CA certificate you installed on Vertica does not match the certificate installed on the SecureData Appliance, you will see an error similar to the following:

```
$ curl --capath /home/dbadmin/VMart/v_vmart_node0001_catalog/Libraries/$(vsql\
 -A -t -c "SELECT sal_storage_id from user_libraries WHERE lib_name = 'VoltageSecureLib';")\
 https://voltage-pp-0000.example.com/policy/clientPolicy.xml

curl: (60) Peer certificate cannot be authenticated with known CA certificates
More details here: http://curl.haxx.se/docs/sslcerts.html
```

In this case, verify that you have installed the correct CA certificate in Vertica, and that its file name has a .pem extension.

If you see other errors, such as "couldn't connect to host," verify that your firewall configuration allows your Vertica nodes to access your SecureData Appliance.

Configuring access to SecureData

The Vertica integration functions require several pieces of information in order to connect to and authenticate with SecureData.

SecureData global configuration settings

The SecureData integration has one required setting and two optional settings that you set globally using the VoltageSecureConfigureGlobal function. This function saves the settings to a configuration file named /voltagesecure/conf.global stored in the Vertica Distributed File System (DFS). This file system is used by Vertica to store data that can be accessed by all nodes. You must use the function to create this file before you use any of the other SecureData integration functions. All users who have access to the SecureData functions are able to access the settings in this file.

The one required setting is the URL of the SecureData policy file. This file provides the SecureData integration library with details of how the SecureData Appliance is configured. The library also uses this URL to determine the address of the SecureData Appliance.

The two optional settings are:

- allow_short_fpe : When set to True, SecureData ignores the lower length limit for encoding FPE values. Usually, SecureData does not use FPE to encrypt data shorter than a lower limit (usually, 8 bits). See the *SecureData Architecture Guide* 's section on Data Length Restrictions for more information.
- enable_file_cache : When set to True, Vertica caches the SecureData policy file and encryption keys to disk, rather than just to memory. Defaults to false.

This example sets the policy URL globally to https://voltage-pp-0000.example.com/policy/clientPolicy.xml and the network timeout to 200 seconds.

```
=> SELECT VoltageSecureConfigureGlobal(USING PARAMETERS
                                policy_url='https://voltage-pp-0000.example.com/policy/clientPolicy.xml',
                                NETWORK_TIMEOUT=200)
                                OVER ();

                      policy_url                       | allow_short_fpe | enable_file_cache | network_timeout
-------------------------------------------------------+-----------------+-------------------+-----------------
 https://voltage-pp-0000.example.com/policy/clientPolicy.xml    |                 |                   | 200
(1 row)
```

Manually refresh the client policy across the nodes:

```
=> SELECT VoltageSecureRefreshPolicy() OVER ();
                            PolicyRefresh
-------------------------------------------------------------------------------
Successfully refreshed policy on node [v_sandbox_node0001]. Policy on other nodes
will be refreshed the next time a Voltage operation is run on them.
(1 row)
```

Important

The SecureData integration only supports one configuration for the SecureData Appliance at a time.

SecureData user configuration settings

The remaining SecureData settings define SecureData user information. They are usually specific to each user accessing the integration functions. However, you can have all Vertica users share the same SecureData user configuration. These settings are:

- Your authentication credentials for SecureData. The exact information you need depends on your SecureData Appliance's configuration. There are four potential settings you that you can use:
  - username : a username that you use to identify yourself. Your user name is either defined by LDAP (when using LDAP authentication) or by the SecureData appliance when using shared secret authentication.
  - identity : the SecureData identity to use. See the *SecureData Administrator Guide* for more information about identities in SecureData. The identity usually takes the form of an email address. When your SecureData Appliance uses LDAP authentication, your LDAP account must have access to this identity.
  - shared_secret : a password set in SecureData.
  - password : the LDAP password to use to authenticate with SecureData.

You can supply both a username and identity, depending on the SecureData Appliance's configuration. Your SecureData Appliance can be configured to supply the identity based on your SecureData username.

However, you can only use a shared_secret or a password. If you set both parameters, the Vertica SecureData functions exit with an error.

You have two options for setting these configuration values: setting them in user-defined session parameters , or saving the values in a configuration file stored in the Vertica distributed file system.

Setting SecureDate user session parameters

Use the ALTER SESSION statement to set parameters for the voltagesecurelib library. This library contains all of the SecureData functions. The following example demonstrates configuring the session to access SecureData using shared secret authentication.

```
=> ALTER SESSION SET UDPARAMETER FOR voltagesecurelib identity='alice@example.com';
ALTER SESSION
=> ALTER SESSION SET UDPARAMETER FOR voltagesecurelib username='alice';
ALTER SESSION
=> ALTER SESSION SET UDPARAMETER FOR voltagesecurelib shared_secret='my_shared_secret';
ALTER SESSION
```

Once set, these parameters only last for the duration of your session.

Saving parameters in configuration files

You can use the VoltageSecureConfigure function to save SecureData user parameters to a configuration file in the DFS. You must supply a file name for your configuration file. You can choose to store your SecureData parameters in a user-specific file by supplying just a file name, such as securedata.conf . Behind the scenes, your configuration file is stored in DFS under a directory named for your username. This directory prevents your configuration file from conflicting with other users' files. You do not use this directory name when accessing your configuration file.

You can also store the configuration parameters in an absolute file named /voltagesecure/conf . All users who have access to the SecureData integration functions can use this configuration file in their function calls. You can use this file if all of the users accessing the SecureData functions share the same SecureData user settings.

Important
All users that have access to the VoltageSecureConfigure function can write to the global /voltagesecure/conf file. Users do not need to be able to write to this file in order to use it in calls to the other SecureData integration functions. You will usually choose to either grant users access to VoltageSecureConfigure in order to save their own configuration files, or you will use the global /voltagesecure/conf file and not grant users access to VoltageSecureConfigure.

Values in the session parameters override values in configuration files.

You call VoltageSecureConfigure with the parameters you want to save to the configuration file. Values that you do not set in this function call are not set in the configuration file. You must supply those values to the other SecureData integration functions using session parameters.

If you choose to save either the password or shared secret to your configuration file, you do not directly pass them to the VoltageSecureConfigure function. Instead, you set the value in the appropriate session variable, and then set either VoltageSecureConfigure's store_password or store_shared_secret parameter to true. When either of these parameters are true, VoltageSecureConfigure reads the value from the session variable

and saves it in the configuration file.

Caution

Under normal circumstances, users are not able to directly read data from files stored in DFS. However, all users who have access to UDx functions that read from the DFS could access these files from within Vertica.

In addition, these files are stored as plain text in every node's file system. Anyone with the proper file system access on the nodes can read the file's contents.

You should take both of these facts into consideration when deciding whether to store sensitive information such as passwords or shared secrets in either the shared or per-user configuration files.

Example: creating a user-specific configuration file

The following example shows how you can create a user-specific configuration file. This example does not store information such as the password or shared secret, so you must still set session parameters for these values before you can call the other SecureData integration functions.

```
=> \x
Expanded display is on.
=> SELECT VoltageSecureConfigure(USING PARAMETERS config_dfs_path='voltage.conf',
                                 username='alice', identity='alice@example.com', store_password=false
                                ) OVER ();
-[ RECORD 1 ]-----+---------------------------------------------------------------
config_dfs_path   | voltage.conf
identity          | alice@example.com
username          | alice
```

Encrypting, decrypting, and hashing data

Once you have set up authentication with the SecureData Appliance (SDA), call the VoltageSecureProtect and VoltageSecureAccess functions to encrypt/hash or decrypt your data, respectively. At a minimum, you pass these functions the value to be encrypted, hashed, or decrypted and value's FPE format as defined in the Voltage SDA. You can also capture these parameters in SQL Macros and access policies.

Connecting Vertica to the SecureData appliance

The following example demonstrates how to fetch the SDA client policy with Vertica and then how to set the session parameters for authentication to the SDA.

1. Add the SecureData CA certificate to Vertica.
2. Set the client policy with VoltageSecureConfigureGlobal.

   ```
   => SELECT VoltageSecureConfigureGlobal(USING PARAMETERS
      policy_url='https://voltage-pp-0000.example.com/policy/clientPolicy.xml')
      OVER ();
                          policy_url                        | allow_short_fpe | enable_file_cache
   -------------------------------------------------------------+-----------------+-------------------
    https://voltage-pp-0000.example.com/policy/clientPolicy.xml     |                 |
   (1 row)
   ```

3. (Optional) Retrieve the latest version of the client policy with VoltageSecureRefreshPolicy.

   ```
   => SELECT VoltageSecureRefreshPolicy() OVER ();
                               PolicyRefresh
   ------------------------------------------------------------------------------------
   Successfully refreshed policy on node [v_node0001]. Policy on other nodes
   will be refreshed the next time a Voltage operation is run on them.
   (1 row)
   ```

4. Set the authentication parameters to authenticate to the SDA as a Vertica user with ALTER SESSION or VoltageSecureConfigure. For details on this process, see Configuring access to SecureData.
   ALTER SESSION sets these parameters for the duration of your session and override parameters set in configuration files.

```
=> ALTER SESSION SET UDPARAMETER FOR voltagesecurelib identity='alice@example.com';
ALTER SESSION
=> ALTER SESSION SET UDPARAMETER FOR voltagesecurelib username='alice';
ALTER SESSION
=> ALTER SESSION SET UDPARAMETER FOR voltagesecurelib shared_secret='my_secret';
ALTER SESSION
```

VoltageSecureConfigure saves these parameters to the specified config_dfs_path . This configuration file can then be passed to other Voltage functions.

```
=> SELECT VoltageSecureConfigure(USING PARAMETERS config_dfs_path='voltage.conf',
   username='alice', identity='alice@example.com', store_password=false) OVER ();

 config_dfs_path | identity          | username
-----------------+-------------------+----------
 voltage.conf    | alice@example.com | alice

=> SELECT VoltageSecureProtect('123-45-6789' USING PARAMETERS
    format='ssn',
    config_dfs_path='voltage.conf');

 VoltageSecureProtect
----------------------
 376-69-6789
```

Encrypting and decrypting data

VoltageSecureProtect takes a format-preserving encryption (FPE) format defined in your SDA and VARCHAR data and returns format-preserved ciphertext, which can then be decrypted with VoltageSecureAccess . That is, if the plaintext follows the form of a phone number (###-###-####), the resulting ciphertext will be in the same form. Formats can be customized in various ways, like masking certain values, leaving some part of the string unencrypted, etc.

One of the many predefined formats in the SDA is ssn . The ssn format encrypts all but the last four digits of the social security number, which can be useful for certain roles like customer support, which may need the last four digits of the SSN to authenticate individuals over the phone.

```
=> SELECT VoltageSecureProtect('123-45-6789' USING PARAMETERS format='ssn');
 VoltageSecureProtect
----------------------
 376-69-6789
(1 row)

=> SELECT VoltageSecureAccess('376-69-6789' USING PARAMETERS format='ssn');
 VoltageSecureAccess
---------------------
 123-45-6789
(1 row)
```

Another predefined format is auto , which can handle a wide variety of data, including SSNs. In contrast to ssn , auto is configured in the SDA to encrypt all characters in the plaintext.

```
=> SELECT VoltageSecureProtect('123-45-6789' USING PARAMETERS format='auto');
 VoltageSecureProtect
----------------------
 820-31-5110
(1 row)
```

Note that you can only decrypt ciphertext with the same format that generated it. In this case, ciphertext created with ssn must be decrypted with ssn . Passing the format auto with a ciphertext generated with ssn , for instance, would simply return an incorrect plaintext.

Encrypting data during load

When handling sensitive data, you often want to encrypt it as you load it so the unencrypted values are never stored in your database.

The following example demonstrates loading data using COPY . Suppose you have a data filled with customer information with the following fields: id number, first name, last name, social security number, card verification number, and date of birth:

```
5345,Thane,Ross,559-32-0670,376765616314013,618,05-09-1996
5346,Talon,Wilkins,540-48-0784,4716511603424923,111,09-17-1941
5347,Daquan,Phelps,785-34-0092,342226134491834,294,05-08-1963
5348,Basia,Lopez,011-85-0705,4595818418314603,503,04-29-1940
5349,Kaseem,Hendrix,672-57-0309,4556 078 73 7944,693,03-11-1942
5350,Omar,Lott,825-45-0131,6462 0541 0799 6261,555,02-17-1956
5351,Nell,Cooke,637-50-0105,646 59756 30903 530,818,02-14-1995
5352,Illana,Middleton,831-47-0929,648 23640 86684 267,883,12-29-1949
5353,Garrett,Williamson,408-73-0207,5334 2702 1360 8370,869,11-06-1955
5354,Hanna,Ware,694-97-0394,543 38494 19219 254,586,08-08-1967
```

To encrypt the social security number and the credit card number columns, call VoltageSecureProtect to encrypt these columns in the COPY statement you use to load the data:

```
=> CREATE TABLE customers (id INTEGER, first_name VARCHAR, last_name VARCHAR,
                           ssn VARCHAR(11), cc_num VARCHAR(25), cvv VARCHAR(5), dob DATE);
CREATE TABLE
=> COPY customers (id, first_name, last_name, ssn_raw FILLER VARCHAR(11),
       cc_num_raw FILLER VARCHAR(25), cvv, dob,
       ssn AS VoltageSecureProtect(ssn_raw USING PARAMETERS format='ssn',
                                   config_dfs_path='voltage.conf'),
    cc_num AS VoltageSecureProtect(cc_num_raw USING PARAMETERS format='cc',
                                   config_dfs_path='voltage.conf'))
    FROM '/home/dbadmin/customer_data.csv' DELIMITER ',';
 Rows Loaded
-------------
         100
(1 row)

=> SELECT * FROM customers ORDER BY id ASC LIMIT 10;
  id  | first_name | last_name |     ssn     |       cc_num        | cvv |    dob
------+------------+-----------+-------------+---------------------+-----+------------
 5345 | Thane      | Ross      | 072-52-0670 | 405939553794013     | 618 | 1996-05-09
 5346 | Talon      | Wilkins   | 348-30-0784 | 5350908688294923    | 111 | 1941-09-17
 5347 | Daquan     | Phelps    | 983-53-0092 | 133383311411834     | 294 | 1963-05-08
 5348 | Basia      | Lopez     | 490-63-0705 | 7979155436134603    | 503 | 1940-04-29
 5349 | Kaseem     | Hendrix   | 268-74-0309 | 3212 314 45 7944    | 693 | 1942-03-11
 5350 | Omar       | Lott      | 872-03-0131 | 4914 1839 6801 6261 | 555 | 1956-02-17
 5351 | Nell       | Cooke     | 785-90-0105 | 332 34312 95233 530 | 818 | 1995-02-14
 5352 | Illana     | Middleton | 947-60-0929 | 219 06376 36044 267 | 883 | 1949-12-29
 5353 | Garrett    | Williamson| 333-23-0207 | 1126 1022 5922 8370 | 869 | 1955-11-06
 5354 | Hanna      | Ware      | 661-57-0394 | 106 09915 59049 254 | 586 | 1967-08-08
(10 rows)
```

Caution

Always use the same FPE format to encrypt data in a column. If you use different FPE formats in the same column (such as loading some data using ssn and other data using auto) there is no way to tell which format was used for any particular row, and properly and improperly encrypted ciphertexts will be indistinguishable.

Encrypting Non-VARCHAR data

VoltageSecureProtect function only encrypts VARCHAR values, so if you need to encrypt other data types, such as DATE or INTEGER, you must cast these values to VARCHAR in your function call and then cast them back to DATE or INTEGER when storing them.

Building off the previous example, the following encrypts the date of birth (dob) column, which holds date type DATE.

```
=> CREATE TABLE customers2 (id INTEGER, first_name VARCHAR, last_name VARCHAR, ssn VARCHAR(11),
                        cc_num VARCHAR(25), cvv VARCHAR(5), dob DATE);
CREATE TABLE
=> COPY customers2 (id, first_name, last_name, ssn, cc_num, cvv, dob_raw FILLER DATE,
                  dob AS VoltageSecureProtect(dob_raw::VARCHAR USING PARAMETERS
                                              format='birthday',
                                              config_dfs_path='voltage.conf')::DATE)
       FROM '/home/dbadmin/customer_data.csv' DELIMITER ',';
 Rows Loaded
-------------
         100
(1 row)

=> SELECT * FROM customers2 ORDER BY id ASC LIMIT 10;
  id  | first_name | last_name  |     ssn     |       cc_num        | cvv |    dob
------+------------+------------+-------------+---------------------+-----+------------
 5345 | Thane      | Ross       | 559-32-0670 | 376765616314013     | 618 | 1902-03-09
 5346 | Talon      | Wilkins    | 540-48-0784 | 4716511603424923    | 111 | 2023-07-22
 5347 | Daquan     | Phelps     | 785-34-0092 | 342226134491834     | 294 | 2091-01-18
 5348 | Basia      | Lopez      | 011-85-0705 | 4595818418314603    | 503 | 1921-08-17
 5349 | Kaseem     | Hendrix    | 672-57-0309 | 4556 078 73 7944    | 693 | 1962-08-23
 5350 | Omar       | Lott       | 825-45-0131 | 6462 0541 0799 6261 | 555 | 1930-01-12
 5351 | Nell       | Cooke      | 637-50-0105 | 646 59756 30903 530 | 818 | 2098-01-01
 5352 | Illana     | Middleton  | 831-47-0929 | 648 23640 86684 267 | 883 | 1956-09-07
 5353 | Garrett    | Williamson | 408-73-0207 | 5334 2702 1360 8370 | 869 | 2079-03-25
 5354 | Hanna      | Ware       | 694-97-0394 | 543 38494 19219 254 | 586 | 1903-07-16
(10 rows)
```

> **Note**
> To encrypt dates, you must create your own FPE format. Be sure to match your custom FPE format to standard Vertica date format of YYYY-MM-DD. The example's birthday is one such custom format.

Note that you cannot use the auto format for DATE types; ciphertext generated by auto uses the full range of numbers and letters and only preserves the number of characters and any separators, so the resulting ciphertext cannot typically be cast to a date.

```
=> SELECT VoltageSecureProtect('07-16-1969' USING PARAMETERS format='auto',
                           config_dfs_path='/voltagesecure/conf');
 VoltageSecureProtect
----------------------
 45-86-8651
(1 row)

=> SELECT VoltageSecureProtect('07-16-1969' USING PARAMETERS format='auto',
                           config_dfs_path='/voltagesecure/conf')::DATE;
ERROR 2992:  Date/time field value out of range: "45-86-8651"
HINT:  Perhaps you need a different "datestyle" setting
```

Decrypting values in queries

To decrypt ciphertext stored in tables, call VoltageSecureAccess on the encrypted column in your query.

The following example queries a table and decrypts the SSN column:

```
=> SELECT id,
        first_name,
        last_name,
        VoltageSecureAccess(ssn USING PARAMETERS format='ssn',
                        config_dfs_path='/voltagesecure/conf') AS ssn_plaintext,
        dob
     FROM customers
     WHERE dob < '1970-1-1'
     ORDER BY id ASC
     LIMIT 10;

 id  | first_name | last_name  |     ssn_plaintext     |    dob
------+------------+------------+-----------------------+------------
 5346 | Talon      | Wilkins    | 540-48-0784           | 1941-09-17
 5347 | Daquan     | Phelps     | 785-34-0092           | 1963-05-08
 5348 | Basia      | Lopez      | 011-85-0705           | 1940-04-29
 5349 | Kaseem     | Hendrix    | 672-57-0309           | 1942-03-11
 5350 | Omar       | Lott       | 825-45-0131           | 1956-02-17
 5352 | Illana     | Middleton  | 831-47-0929           | 1949-12-29
 5353 | Garrett    | Williamson | 408-73-0207           | 1955-11-06
 5354 | Hanna      | Ware       | 694-97-0394           | 1967-08-08
 5355 | Quinn      | Pruitt     | 818-91-0359           | 1965-11-14
 5356 | Clayton    | Santiago   | 102-56-0010           | 1958-02-02
(10 rows)
```

Important

The VoltageSecureAccess function has no way of determining if the values you are passing it are actually encrypted or not; the function simply takes the input and transforms it using the decryption key. If you pass it values from an unencrypted column, you will get back scrambled values. For example:

```
=> SELECT first_name,
        VoltageSecureAccess(first_name USING PARAMETERS format='auto',
                        config_dfs_path='/voltagesecure/conf')
                        AS decrypted_first_name
        FROM customers LIMIT 10;
 first_name | decrypted_first_name
------------+----------------------
 Omar       | Rftd
 Illana     | Clfkow
 Hanna      | Bodng
 Keith      | Hklnw
 Constance  | Cicgtmgtw
 Kirk       | Jwdv
 Eagan      | Hiksm
 Branden    | Ytqgngp
 Hope       | Tqzc
 Keane      | Pdcax
(10 rows)
```

Decrypting Non-VARCHAR columns

Like VoltageSecureProtect, VoltageSecureAccess can only handle VARCHAR values. When you decrypt ciphertext from a non-VARCHAR column, you must cast it to a VARCHAR before passing it to VoltageSecureAccess. If your query or client application depends on the original data type of the column, you must then cast the returned plaintext back to the column's original data type.

The following example decrypts the dob column by casting the column data from DATE to VARCHAR, passing it to VoltageSecureAccess, and then casting the plaintext back to DATE.

```
=> SELECT id, first_name, last_name,
        VoltageSecureAccess(dob::VARCHAR USING PARAMETERS format='birthday',
                        config_dfs_path='/voltagesecure/conf')::DATE AS dob
        FROM customers2 ORDER BY id LIMIT 10;
 id  | first_name | last_name |    dob
-----+------------+-----------+------------
 5345 | Thane      | Ross      | 1996-05-09
 5346 | Talon      | Wilkins   | 1941-09-17
 5347 | Daquan     | Phelps    | 1963-05-08
 5348 | Basia      | Lopez     | 1940-04-29
 5349 | Kaseem     | Hendrix   | 1942-03-11
 5350 | Omar       | Lott      | 1956-02-17
 5351 | Nell       | Cooke     | 1995-02-14
 5352 | Illana     | Middleton | 1949-12-29
 5353 | Garrett    | Williamson | 1955-11-06
 5354 | Hanna      | Ware      | 1967-08-08
(10 rows)
```

Masking decrypted values

If the ciphertext was created with a format that specifies a form of "masking," plaintexts returned by VoltageSecureAccess can be masked by using the mask=true parameter. If you don't specify this parameter, the full plaintext will be returned regardless of the format.

This format, as defined in the Voltage SDA, obscures all but the last two characters of the decrypted plaintext.

```
=> SELECT VoltageSecureAccess('g3kbx6ru19', USING PARAMETERS
                        format='maskedFormat',
                        config_dfs_path='voltage.conf');
 VoltageSecureAccess
---------------------
 1234567890

=> SELECT VoltageSecureAccess('g3kbx6ru19', USING PARAMETERS
                        format='maskedFormat',
                        config_dfs_path='voltage.conf',
                        mask=true);
 VoltageSecureAccess
---------------------
 XXXXXXXX90
```

Tweaking ciphertext

Analogous to a salt , you can pass an additional "tweak" value to add to the key used by VoltageSecureProtect to create unique ciphertexts from the same plaintext. This "tweaked" ciphertext can then only be decrypted by passing the same format and "tweak" value to VoltageSecureAccess. Support for this feature is set in the SDA.

The following example uses the custom format ssn-tweak and the value tweakvalue123 to tweak the ciphertext.

```
=> SELECT VoltageSecureProtect('681-09-2913', 'tweakvalue123' USING PARAMETERS
   format='ssn-tweak');

 VoltageSecureProtect
---------------------
 721-21-2913

=> SELECT VoltageSecureAccess('721-21-2913', 'tweakvalue123' USING PARAMETERS
   format='ssn-tweak');

 VoltageSecureProtect
---------------------
 681-09-2913
```

Tweaking an entire column

A common use case is to use the values from one column to tweak the ciphertext in another.

The following example loads the data in a similar manner to the previous section, but uses the values from the last_name column to tweak the dates of birth of each customer.

```
=> CREATE TABLE customers (id INTEGER, first_name VARCHAR, last_name VARCHAR, ssn VARCHAR(11),
   cc_num VARCHAR(25), cvv VARCHAR(5), dob DATE);
CREATE TABLE

=> COPY customers (id, first_name, last_name, ssn_raw FILLER VARCHAR(11),
         cc_num_raw FILLER VARCHAR(25), cvv,
         dob_raw FILLER DATE, ssn AS VoltageSecureProtect(ssn_raw USING PARAMETERS format='ssn',
                                          config_dfs_path='voltage.conf'),
         /* cast the dob_raw to VARCHAR and tweak with last_name, then cast back to DATE */
         dob AS VoltageSecureProtect(dob_raw::VARCHAR , last_name USING PARAMETERS format='date-tweak',
                              config_dfs_path='voltage.conf')::DATE,
         cc_num AS VoltageSecureProtect(cc_num_raw USING PARAMETERS format='cc', config_dfs_path='voltage.conf'))
         FROM 'customers.csv' DELIMITER ',';

 Rows Loaded
-------------
         100
(1 row)

=> SELECT id, first_name, last_name, dob FROM customers ORDER BY id;
  id  | first_name | last_name  |    dob
------+------------+------------+------------
 5345 | Thane      | Ross       | 2086-03-07
 5346 | Talon      | Wilkins    | 1936-06-09
 5347 | Daquan     | Phelps     | 1953-12-01
 5348 | Basia      | Lopez      | 1979-11-30
 5349 | Kaseem     | Hendrix    | 2090-01-07
 5350 | Omar       | Lott       | 1984-07-27
 5351 | Nell       | Cooke      | 2053-03-20
 5352 | Illana     | Middleton  | 1988-03-06
 5353 | Garrett    | Williamson | 1927-03-03
 5354 | Hanna      | Ware       | 1998-08-28
 5355 | James      | May        | 1941-10-24
(11 rows)
```

To decrypt the column, include the tweak column in the call to VoltageSecureAccess. The tweak column must contain the same data it did when it was used to encrypt the column.

```
=> SELECT id, first_name, last_name,
   VoltageSecureAccess(dob::VARCHAR , last_name USING PARAMETERS format='date-tweak',
   config_dfs_path='voltage.conf')::DATE AS dob
   FROM customers ORDER BY id;
 id  | first_name | last_name  |    dob
-----+------------+------------+------------
 5345 | Thane      | Ross       | 1996-05-09
 5346 | Talon      | Wilkins    | 1941-09-17
 5347 | Daquan     | Phelps     | 1963-05-08
 5348 | Basia      | Lopez      | 1940-04-29
 5349 | Kaseem     | Hendrix    | 1942-03-11
 5350 | Omar       | Lott       | 1956-02-17
 5351 | Nell       | Cooke      | 1995-02-14
 5352 | Illana     | Middleton  | 1949-12-29
 5353 | Garrett    | Williamson | 1955-11-06
 5354 | Hanna      | Ware       | 1967-08-08
 5355 | James      | May        | 1967-08-08
(11 rows)
```

Tweaking columns with stored procedures

Since tweak values become part of the encryption key, a tweak column's contents must be the same at the time of both encryption and decryption. To avoid losing access to the plaintext, when modifying a tweak column, you must:

1. decrypt the ciphertext
2. modify the tweak column
3. re-encrypt the plaintext data with the contents of the modified tweak column

You can automate this process by using stored procedures. For example, a stored procedure to update a customer's date of birth when tweaked with their last name might be:

```
=> CREATE PROCEDURE update_dob (employee_id INT, tweak_value VARCHAR) AS $$
  BEGIN
    PERFORM UPDATE customers
      SET dob = (
        WITH dob_ AS (SELECT VoltageSecureAccess(dob::VARCHAR , last_name
          USING PARAMETERS format='date-tweak', config_dfs_path='voltage.conf')::DATE AS dob
            FROM customers
            WHERE id = employee_id)
        SELECT VoltageSecureProtect(dob::VARCHAR, tweak_value
          USING PARAMETERS format='date-tweak', config_dfs_path='voltage.conf')::DATE
            FROM dob_)
      WHERE ID = employee_id;

    PERFORM UPDATE customers SET last_name = tweak_value WHERE ID = employee_id;

    PERFORM COMMIT;
  END;
$$;
```

You can then call the procedure to update the encrypted value and the tweak value:

```
=> CALL update_dob(5349, 'Henderson');
```

Hashing data

VoltageSecureProtect can also hash your data while preserving its format, which can be useful in contexts where you need to anonymize data to comply with privacy standards while retaining accessibility for client code.

You should keep in mind the following:

- hashing is a one way operation, so the original plaintext is unrecoverable
- the nature of format-preserving hash functions and plaintexts with low entropy (names, dates, SSNs, etc.) mean that collisions are much more likely than with standard hashing functions

The following example encrypts an SSN using a customFPH format, ssnHash .

```
=> SELECT VoltageSecureProtect('681-09-2913' USING PARAMETERS
                               format='ssnHash',
                               config_dfs_path='voltage.conf');
 VoltageSecureProtect
----------------------
 841-68-2913
```

Encapsulating encryption business logic with SQL macros

The VoltageSecureProtect and VoltageSecureAccess functions are fairly low-level functions and require a lot of preparation to use correctly. This includes, but is not limited to, information on the data type you're decrypting/encrypting, which Voltage format to use, casting the input to VARCHAR and then back to your desired data type, and identity management. Writing queries with this many moving parts can be tedious.

To streamline this process, you can encapsulate this information in SQL macros and decide its behavior in a more dynamic way with case expressions. This approach offers several benefits:

- You can associate a Voltage format with the data's purpose and Vertica will automatically encrypt and decrypt the value accordingly.
- You can automate the required type casting to and from your desired data type to VARCHAR.
- The THROW_ERROR function can provide a form of input validation for your macros.
- You can specify an identity for a given case expression during encryption which restricts decryption privileges to the same identity.

> **Note**
> SQL macros are incompatible with User-Defined Transform Functions, such as VoltageSecureProtectAllKeys .

To view your macros, query the USER_FUNCTIONS system table.

Encryption and decryption macro templates

In the following macros, the data_purpose parameter describes what the data is used for (e.g. temperature) and the value parameter indicates value being encrypted and its data_type (e.g. INT). The data_type must be the same across the entire macro.

The data_purpose parameter is also associated with both the voltage_format and voltage_identity parameters, which control how the data is encrypted/decrypted and which identity has access to it, respectively.

If you pass into the function a data_purpose without a corresponding case expression, the THROW_ERROR function will return a user-specified error, which itself must be casted to the function's return data_type .

Note that the standard casting operator :: will terminate the query entirely if it encounters types incompatible with the specified cast, which can be problematic when casting larger datasets. To prevent the query from terminating, you can use the ::! operator to first attempt the specified cast, but to return a NULL value for incompatible types. For more information, see Cast failures .

Encryption macro with VoltageSecureProtect

```
=> CREATE FUNCTION encryptDataType(data_purpose VARCHAR, value data_type) RETURN data_type
   AS BEGIN
   RETURN(CASE
      WHEN (data_purpose='data_purpose')
         then VoltageSecureProtect(value::VARCHAR USING PARAMETERS
            format='voltage_format',
            identity='voltage_identity')::data_type
      ELSE
         THROW_ERROR('no matching data_purpose')::data_type
      END);
   END;
```

Decryption macro with VoltageSecureAccess

```
=> CREATE FUNCTION decryptDataType(data_purpose VARCHAR, value data_type) RETURN data_type
   AS BEGIN
   RETURN(CASE
      WHEN (data_purpose='data_purpose')
         then VoltageSecureAccess(value::VARCHAR USING PARAMETERS
            format='voltage_format',
            identity='voltage_identity')::data_type
      ELSE
         THROW_ERROR('no matching data_purpose')::data_type
      END);
   END;
```

Example

The following example shows how to manage encryption and decryption for a column with dates of birth. In this case, you might want to define a separate identity for encrypting customer and employee data.

```
=> SELECT * FROM customer_dob;
  dates
-----------
 1955-11-04
 1991-12-01
 1977-07-07
(3 rows)
```

For encryption, define the following macro.

```
=> CREATE FUNCTION encryptDOB(data_purpose VARCHAR, value DATE) RETURN DATE
   AS BEGIN
   RETURN
      (CASE
         --The data_purpose parameter controls which identity is used during encryption;
         WHEN (data_purpose='customer')
         --Format, identities, and casting from DATE to VARCHAR and back to DATE are all encapsulated in the case expression;
            then VoltageSecureProtect(value::VARCHAR USING PARAMETERS
               format='birthday',
               identity='customer_data@example.com')::DATE
         WHEN (data_purpose='employee')
            then VoltageSecureProtect(value::VARCHAR USING PARAMETERS
               format='birthday',
               identity='employee_data@example.com')::DATE
         ELSE
            THROW_ERROR('Unsupported data_purpose -- You must pass "customer" when
            encrypting customer data or "employee" when encrypting employee data')::DATE
            --Because the return type of this macro is DATE, THROW_ERROR must also be casted to type DATE;
      END);
   END;
```

To encrypt the dates column in the customer_dob table:

```
=> SELECT encryptDOB('customer', dates) FROM customer_dob;
 encryptDOB
------------
 2048-08-09
 1917-03-05
 2022-01-07
```

To encrypt a value individually:

```
=> SELECT encryptDOB('customer', '1955-11-04');
 encryptDOB
------------
 2048-08-09
```

You can define a matching macro for decryption. Since encrypted data can only be decrypted with matching identities, these case expressions use the same data_purpose and identities for decryption.

For decryption, define the following macro:

```
=> CREATE FUNCTION decryptDOB(data_purpose VARCHAR, value DATE) RETURN DATE
   AS BEGIN
   RETURN
      (CASE
         --The case expressions and parameters must match the ones for encryption;
         WHEN (data_purpose='customer')
            then VoltageSecureAccess(value::VARCHAR USING PARAMETERS
               format='birthday',
               identity='customer_data@example.com')::DATE
         WHEN (data_purpose='employee')
            then VoltageSecureAccess(value::VARCHAR USING PARAMETERS
               format='birthday',
               identity='employee_data@example.com')::DATE
         ELSE
            THROW_ERROR('Unsupported data_purpose -- You must pass "customer" when
            decrypting customer data or "employee" when decrypting employee data')::DATE
      END);
   END;
```

To decrypt an encrypted column in a nested call:

```
=> SELECT decryptDOB('customer', encryptDOB('customer', dates)) FROM customer_dob;
 decryptDOB
------------
 1955-11-04
 1991-12-01
 1977-07-07
(3 rows)
```

To decrypt a value individually:

```
=> SELECT decryptDOB('customer', '2048-08-09');
 decryptDOB
------------
 1955-11-04
```

To drop these macros:

```
=> DROP FUNCTION encryptDOB(VARCHAR, DATE);
DROP FUNCTION
=> DROP FUNCTION decryptDOB(VARCHAR, DATE);
DROP FUNCTION
```

Granting users access to the Voltage SecureData integration functions

By default, Vertica users do not have access to the Voltage SecureData Integration Functions. Users attempting to call the integration functions without the correct privileges will receive the following error:

```
=> SELECT id,
        first_name,
        last_name,
        VoltageSecureAccess(ssn USING PARAMETERS format='ssn',
                        config_dfs_path='/voltagesecure/conf') AS ssn,
        dob
        FROM customers
        WHERE dob < '1970-1-1'
        ORDER BY id ASC
        LIMIT 10;
ERROR 6482:  Failed to parse Access Policies for table "customers" [Function
public.VoltageSecureProtect(varchar) does not exist, or permission is denied for
public.VoltageSecureProtect(varchar)]
```

Users must have EXECUTE access to the integration functions in order to use them. These functions are part of the PUBLIC schema. The functions in the Voltage SecureData integration library are:

Function Signature	Function Type
VoltageSecureAccess (VARCHAR)	Function
VoltageSecureConfigure ()	Transform Function
VoltageSecureProtect (VARCHAR)	Function
VoltageSecureProtectAllKeys (VARCHAR)	Transform Function

The following example demonstrates granting the user named Alice access to the VoltageSecureAccess function to be able to decrypt data.

```
=> \c vmart dbadmin
You are now connected to database "vmart" as user "dbadmin".
=> GRANT EXECUTE ON FUNCTION public.VoltageSecureProtect(VARCHAR) TO alice;
GRANT PRIVILEGE
=> \c vmart alice
You are now connected to database "vmart" as user "alice".
=> SELECT id, first_name, last_name,
        VoltageSecureAccess(ssn USING PARAMETERS format='ssn',
                        config_dfs_path='/voltagesecure/conf')
                        AS ssn,
        dob
        FROM customers
        WHERE dob < '1970-1-1'
        ORDER BY id ASC
        LIMIT 10;
 id  | first_name | last_name |     ssn     |    dob
------+------------+-----------+-------------+------------
 5345 | Thane      | Ross      | 559-32-0670 | 1902-03-09
 5348 | Basia      | Lopez     | 011-85-0705 | 1921-08-17
 5349 | Kaseem     | Hendrix   | 672-57-0309 | 1962-08-23
 5350 | Omar       | Lott      | 825-45-0131 | 1930-01-12
 5352 | Illana     | Middleton | 831-47-0929 | 1956-09-07
 5354 | Hanna      | Ware      | 694-97-0394 | 1903-07-16
 5358 | Mallory    | Vaughn    | 870-53-0272 | 1961-03-09
 5363 | Kirk       | Robinson  | 155-08-0085 | 1964-06-28
 5366 | Branden    | Coffey    | 709-38-0423 | 1923-06-11
 5367 | Raven      | Keith     | 250-31-0269 | 1918-07-31
(10 rows)
```

See GRANT (user defined extension) for a detailed explanation of granting access to UDxs to users.

Creating roles for SecureData users

Instead of granting access to each function to individual users, you can create roles that you grant access to the functions. Then you can grant users who need to access the SecureData functions access to these roles.

Consider creating at least two roles: one for access to the VoltageSecureConfigure function and another for access to the other functions. In most cases, not all users need to access VoltageSecureConfigure, especially if you choose to create a single, global configuration file. See Configuring access to SecureData for more information about using VoltageSecureConfigure.

The following example:

- Creates two roles: secure_data_users, who are granted access to the protect and access functions, and secure_data_admins, who are granted access to the SecureDataConfigure function.
- Grants the secure_data_user role to a user named Alice
- Sets the new role as her default role.
- Switches to Alice
- Calls several of the SecureData functions.

```
=> \c vmart dbadmin
You are now connected to database "vmart" as user "dbadmin".
=> CREATE ROLE secure_data_users;
CREATE ROLE
=> GRANT EXECUTE ON FUNCTION public.VoltageSecureAccess(varchar)
   TO secure_data_users;
GRANT PRIVILEGE
=> GRANT EXECUTE ON FUNCTION public.VoltageSecureProtect(varchar)
   TO secure_data_users;
GRANT PRIVILEGE
=> GRANT EXECUTE ON TRANSFORM FUNCTION
        public.VoltageSecureProtectAllKeys(varchar)
   TO secure_data_users;
GRANT PRIVILEGE
=> CREATE ROLE secure_data_admins;
CREATE ROLE
=> GRANT EXECUTE ON TRANSFORM FUNCTION public.VoltageSecureConfigure()
   TO secure_data_admins;
GRANT PRIVILEGE
=> GRANT secure_data_users TO ALICE;
GRANT ROLE
=> ALTER USER alice DEFAULT ROLE secure_data_users;
ALTER USER
=> \c vmart alice
You are now connected to database "vmart" as user "alice".
=> SET ROLE secure_data_users;
SET
=> SELECT VoltageSecureProtect('123-45-6789'
                           USING PARAMETERS format='ssn',
                           config_dfs_path='/voltagesecure/conf');
 VoltageSecureProtect
----------------------
 376-69-6789
(1 row)
=> SELECT VoltageSecureAccess('376-69-6789'
                          USING PARAMETERS format='ssn',
                          config_dfs_path='/voltagesecure/conf');
 VoltageSecureAccess
----------------------
 123-45-6789
(1 row)

=> SELECT VoltageSecureConfigure(USING PARAMETERS config_dfs_path='voltage.conf',
                            username='alice', identity='alice@example.com',
                            ) OVER ();
ERROR 3457:  Function VoltageSecureConfigure() does not exist, or permission
is denied for VoltageSecureConfigure()
HINT:  No function matches the given name and argument types. You may need to
add explicit type casts
```

Note that Alice can use the global configuration file, despite not being able to access the VoltageSecureConfigure function.

> **Note**
> In the previous example, the global configuration file includes the shared secret or password. See Configuring access to SecureData for a discussion of the security implications of saving sensitive information to a SecureData configuration file.

Automating encryption and decryption with access policies

You can automate encryption and decryption by creating access policies . These policies let you show users with specific roles unencrypted values while users without those roles see encrypted ones. Alternatively, you can create an access policy that masks unencrypted data stored in the database when queried by users who lack a specific role.

Users do not explicitly call the SecureData integration functions when you create access policies to automatically encrypt and decrypt data. However, they still must have access to the SecureData functions and have any necessary session variables set. See Configuring access to SecureData and Granting users access to the Voltage SecureData integration functions for the necessary configuration for the access policies to work. If a user who does not have access to the SecureData integration functions queries a table with an access policy that calls these functions, the query generates an error.

Automatically decrypting table columns for privileged users

To decrypt an encrypted column automatically, create an access policy for the encrypted column that calls VoltageSecureAccess if the user has a role enabled. If the user does not have the role enabled, just return the encrypted value.

The following example:

1. Creates a role named see_ssn and grants it to the user named Alice, as well as granting Alice access to the customers table.
2. Creates an access policy on the customers table's ssn column that decrypts the column's value if the user has the see_ssn role enabled.
3. Switches to the user named Alice.
4. Queries the customers table, without the see_ssn role enabled.
5. Enables the see_ssn role and queries the table again.

```
=> CREATE ROLE see_ssn;
CREATE ROLE

=> GRANT see_ssn TO alice;
GRANT ROLE

=> GRANT ALL ON TABLE customers TO alice;
GRANT PRIVILEGE

=> CREATE ACCESS POLICY ON customers FOR COLUMN ssn
    CASE
        WHEN enabled_role('see_ssn') THEN VoltageSecureAccess(ssn USING PARAMETERS format='ssn',
                                                config_dfs_path='/voltagesecure/conf')
        ELSE ssn
    END ENABLE;
CREATE ACCESS POLICY

=> \c vmart alice;
Password:
You are now connected to database "vmart" as user "alice".

=> SELECT first_name, last_name, ssn FROM customers WHERE id < 5355 ORDER BY id ASC;
 first_name | last_name |     ssn
------------+-----------+-------------
 Gil        | Reeves    | 997-92-0657
 Robert     | Moran     | 715-02-0455
 Hall       | Rice      | 938-83-0659
 Micah      | Trevino   | 495-57-0860
(4 rows)

=> SET ROLE see_ssn;
SET
=> SELECT first_name, last_name, ssn FROM customers WHERE id < 5355 ORDER BY id ASC;
 first_name | last_name |     ssn
------------+-----------+-------------
 Gil        | Reeves    | 232-28-0657
 Robert     | Moran     | 725-79-0455
 Hall       | Rice      | 285-90-0659
 Micah      | Trevino   | 853-60-0860
(4 rows)
```

In the above example, you could combine the see_ssn role with a role that grants users access to the SecureData integration functions. Users who do not have the see_ssn role do not need to have access to the SecureData functions in order to see the encrypted values.

> **Note**
> The above example assumes that the shared configuration file (/voltagesecure/conf) contains all of the credentials necessary to authenticate with the SecureData Appliance, including the password or shared secret. This configuration may not be secure enough to meet your requirements. See the caution in VoltageSecureConfigure for more information. In production use, consider having each privileged user set their identity, username, and password or shared secret in session variables. Unprivileged users do not need to set these values.

Automatically encrypting columns for unprivileged users

You can also create access policies that encrypt values. Use this technique to prevent some users from seeing values in columns that aren't sensitive enough to require encryption in the database, but should not be seen by all users. This technique is the opposite of the encryption-on-the-fly: users that have a specific role enabled get the value from the table; if they do not have the role, the access policy encrypts the value.

The following example:

1. Creates a role named see_dob and assigns it to the user Alice.
2. Creates an access policy on the dob column of the customers table. It returns the value in the column if the user has the see_dob role active and encrypts the value if not.
3. Switches to the user named Alice.
4. Queries the customers table including the dob column.
5. Sets the see_dob role and queries customers again.

```
=> CREATE ROLE see_dob;
CREATE ROLE

=> GRANT see_dob TO alice;
GRANT ROLE

=> CREATE ACCESS POLICY ON customers FOR COLUMN dob
    CASE
       WHEN enabled_role('see_dob') THEN dob
       ELSE VoltageSecureProtect(dob::varchar USING PARAMETERS format='birthday',
                                 config_dfs_path='/voltagesecure/conf')::date
    END ENABLE;
CREATE ACCESS POLICY

=> \c vmart alice
Password:
You are now connected to database "vmart" as user "alice".

=> SELECT first_name, last_name, dob FROM customers ORDER BY id ASC LIMIT 10;
 first_name | last_name  |    dob
------------+------------+------------
 Gil        | Reeves     | 2048-08-09
 Robert     | Moran      | 1917-03-05
 Hall       | Rice       | 2022-01-07
 Micah      | Trevino    | 2018-06-01
 Kuame      | Stephenson | 2053-02-13
 Hedda      | Cooper     | 2002-03-12
 MacKenzie  | Burks      | 2061-10-30
 Anne       | Marquez    | 2078-08-02
 Dominic    | Avery      | 1940-08-10
 Alfreda    | Mcdaniel   | 1904-04-27
(10 rows)

=> SET ROLE see_dob;
SET

=> SELECT first_name, last_name, dob FROM customers ORDER BY id ASC LIMIT 10;
 first_name | last_name  |    dob
------------+------------+------------
 Gil        | Reeves     | 1955-11-04
 Robert     | Moran      | 1991-12-01
 Hall       | Rice       | 1977-07-07
 Micah      | Trevino    | 1980-12-05
 Kuame      | Stephenson | 1979-09-12
 Hedda      | Cooper     | 1987-05-02
 MacKenzie  | Burks      | 1982-11-07
 Anne       | Marquez    | 1949-07-09
 Dominic    | Avery      | 1976-12-02
 Alfreda    | Mcdaniel   | 1975-02-08
(10 rows)
```

> **Note**
> The values you pass to VoltageSecureProtect and VoltageSecureAccess must be VARCHARs. In the previous example, the dob column's data type is DATE, so its value has to be cast to VARCHAR when passed to VoltageSecureProtect. The encrypted value has to be cast back to a DATE value because its output needs to match the table schema.

In the previous example, users who do not have the see_dob role enabled must have access to the SecureData functions in order to see the masked values. If they do not have access to the SecureData functions, querying the customers table results in an error message. The following example creates a new user named Bob and grants him access to the customers table, without any access to the SecureData functions.

```
=> CREATE USER bob;
CREATE USER
=> GRANT ALL ON TABLE customers TO bob;
GRANT PRIVILEGE
=> \c vmart bob
You are now connected to database "vmart" as user "bob".
=> SELECT * FROM customers LIMIT 10;
ERROR 6482:  Failed to parse Access Policies for table "customers"
[Function public.VoltageSecureProtect(varchar) does not exist, or permission
is denied for public.VoltageSecureProtect(varchar)]
```

Querying eFPE encrypted columns

You can use a feature in Voltage SecureData called Embedded Format Preserving Encryption to encrypt your data. You choose to use eFPE when you define the format in your SecureData Appliance. This format is slightly different than the standard FPE format. It uses key rotation, and embeds identification information in the encrypted value. Due to these factors, calls to VoltageSecureProtect using an eFPE format may not result in the same encrypted value, depending on key rotation schedules and the identity of the caller. This feature makes querying columns encrypted using an eFPE format more challenging.

When you want to search a standard FPE encrypted column for a value, you can just encrypt the plain text with VoltageSecureProtect and use the output in your query. For example, suppose you want to search for the customers table for an entry with the social security number 559-32-0670. Then you could use the following query:

```
=> SELECT id, first_name, last_name FROM customers
     WHERE ssn = VoltageSecureProtect('559-32-0670' USING PARAMETERS
                                format='ssn',
                                config_dfs_path='voltage.conf');

 id  | first_name | last_name
-----+------------+-----------
 5345 | Thane      | Ross
(1 row)
```

Querying a column that uses eFPE format encryption is not as simple, as you do not know which embedded key was used to encrypt the data. You could just use the VoltageSecureAccess function to decrypt the entire contents of the table column and search the result for the value you need. However, this is inefficient, as the SecureData function has to be called for every row of data in the table.

A better solution is to use the VoltageSecureProtectAllKeys function. This function is similar to VoltageSecureProtect. However, instead of returning a single encrypted value, it returns a table containing the value encrypted with each of the keys defined for the eFPE format.

```
=> SELECT VoltageSecureProtectAllKeys('376765616314013' USING PARAMETERS
                                format='cc_num',
                                config_dfs_path='/voltagesecure/conf')
        OVER ();

     data       |    protected
----------------+-----------------
 376765616314013 | XMVMRU9RJVU4013
 376765616314013 | X5FD4KO1UEE4013
 376765616314013 | M7ZXTIQVCPB4013
 376765616314013 | UBOSC9K3EXZ4013
 376765616314013 | ZJ1C50C9L9R4013
(5 rows)
```

In the previous example, the cc_num column is encrypted using an eFPE format. The output from VoltageSecureProtectAllKeys shows that this eFPE format has 5 keys defined. The content of the protected column contains the same value encrypted with each of the keys.

To use this function in a query for a value in an eFPE column, use a JOIN on the table you are searching and the result table from VoltageSecureProtectAllKeys. The following example demonstrates querying the customers table to find all rows that have a cc_num value that matches the unencrypted credit card value 376765616314013.

```
=> SELECT id, first_name, last_name FROM customers3 u
        JOIN (SELECT VoltageSecureProtectAllKeys('376765616314013' USING PARAMETERS
                                        format='cc_num',
                                        config_dfs_path='/voltagesecure/conf')
        OVER ()) pak
        ON u.cc_num = pak.protected;

 id  | first_name | last_name
------+------------+-----------
 5345 | Thane      | Ross

(1 row)
```

Voltage SecureData integration function reference

The functions in this section are part of the Vertica voltagesecure library for integrating with Voltage SecureData. These functions are automatically installed when you install or upgrade Vertica. However, you must re-install the voltagesecure library to distribute the CA certificate to all nodes in the Vertica cluster. See Verifying the Vertica server's access to the SecureData CA certificate for instructions.

In this section

- VoltageSecureAccess
- VoltageSecureConfigure
- VoltageSecureConfigureGlobal
- VoltageSecureProtect
- VoltageSecureProtectAllKeys
- VoltageSecureRefreshPolicy

VoltageSecureAccess

Calls SecureData to decrypt ciphertexts encrypted with VoltageSecureProtect .

Syntax

```
VoltageSecureAccess('ciphertext' [, 'tweak'] USING PARAMETERS
                format='format_name'
                [, mask=is_masked]
                [, config_dfs_path='config_file']
                [, identity=sd_identity]);
```

Parameters

ciphertext	A VARCHAR value that was encrypted using SecureData. You must cast other data types (for example DATE values) to VARCHAR when calling this function.
tweak	VARCHAR value analogous to a salt that allows equivalent * plaintext *s to produce different ciphertexts. The same tweak value must for encryption and decryption of a given plaintext. When encrypting or hashing an entire column, you can pass another column for a set of *tweak* values. **Caution** Never use two columns as tweak values for each other or else the original plaintext for both columns will be unrecoverable.
format_name	A string specifying the original FPE format used to generate the ciphertext. Note that SecureData has no way to tell if the value passed to it was actually encrypted or not, or what FPE format was used.

is_masked	A boolean, whether to mask the value when decrypting the *ciphertext* . Masking is defined on a per-format basis on the SecureData Appliance. Note that since masking is optional, you must specify whether to decrypt with masking enabled. If you omit the masking parameter, the plaintext will be unmasked by default:
config_file	String containing the file name of the configuration file to use when authenticating with the SecureData appliance. You must create this file using VoltageSecureConfigure. If you do not supply this parameter, you must set session parameters to configure access to SecureData. See Configuring access to SecureData . Any values set in session parameters override the values in this file.
sd_identity	A string containing the identity to use when decrypting the data. Because SecureData uses the identity to determine encryption keys, this identity much match the identity used to encrypt the data. If supplied, this value overrides any identity value set in the configuration file or session parameter.

Examples

The following example decrypts a Social Security Number (SSN) originally encrypted with a predefined format.

```
=> SELECT VoltageSecureAccess('376-69-6789' USING PARAMETERS format='ssn');

 VoltageSecureAccess
---------------------
 123-45-6789
(1 row)
```

This example demonstrates decrypting an encrypted column within a query.

```
=> SELECT id,
        first_name,
        last_name,
        VoltageSecureAccess(ssn USING PARAMETERS format='ssn',
                        config_dfs_path='/voltagesecure/conf') AS ssn,
        dob
    FROM customers
    WHERE dob < '1970-1-1'
    ORDER BY id ASC
    LIMIT 10;

 id  | first_name | last_name |     ssn     |    dob
------+------------+-----------+-------------+------------
 5346 | Talon      | Wilkins   | 540-48-0784 | 1941-09-17
 5347 | Daquan     | Phelps    | 785-34-0092 | 1963-05-08
 5348 | Basia      | Lopez     | 011-85-0705 | 1940-04-29
 5349 | Kaseem     | Hendrix   | 672-57-0309 | 1942-03-11
 5350 | Omar       | Lott      | 825-45-0131 | 1956-02-17
 5352 | Illana     | Middleton | 831-47-0929 | 1949-12-29
 5353 | Garrett    | Williamson | 408-73-0207 | 1955-11-06
 5354 | Hanna      | Ware      | 694-97-0394 | 1967-08-08
 5355 | Quinn      | Pruitt    | 818-91-0359 | 1965-11-14
 5356 | Clayton    | Santiago  | 102-56-0010 | 1958-02-02
(10 rows)
```

The following example decrypts Unicode using a predefined format. For a full list of predefined formats, consult the Voltage SecureData documentation.

```
=> SELECT VoltageSecureAccess('607-Òdìçç-ぶてぴねら' using parameters format='PREDEFINED::JU_AUTO_TYPE');

 VoltageSecureAccess
----------------------
 123-Hello-こんにちは
```

Decrypt a SSN ciphertext with the original FPE format and tweak value:

```
=> SELECT VoltageSecureAccess('721-21-2913', 'tweakvalue123' USING PARAMETERS
                              format='ssn-tweak',
                              config_dfs_path='voltage.conf');
 VoltageSecureProtect
----------------------
 681-09-2913
```

Decrypt a ciphertext that was encrypted with a masking format. This format obscures all but the last two characters of the decrypted plaintext.

```
=> SELECT VoltageSecureAccess('g3kbx6ru19', USING PARAMETERS
                              format='maskedFormat',
                              config_dfs_path='voltage.conf');
 VoltageSecureAccess
----------------------
 1234567890

=> SELECT VoltageSecureAccess('g3kbx6ru19', USING PARAMETERS
                              format='maskedFormat',
                              config_dfs_path='voltage.conf',
                              mask=true);
 VoltageSecureAccess
----------------------
 XXXXXXXX90
```

See also

- VoltageSecureConfigure
- VoltageSecureProtect
- VoltageSecureProtectAllKeys
- Encrypting, decrypting, and hashing data
- Best practices for safe unicode FPE

VoltageSecureConfigure

Saves SecureData user access configuration parameters to a file in the Vertica Distributed File System (DFS). You then pass the file's name to the other SecureData integration functions. This function can store the configuration file in the user's own DFS directory or in a globally-accessible file named /voltagesecure/conf .

Syntax

```
VoltageSecureConfigure(USING PARAMETERS config_dfs_path='filename'
                       [, identity=sd_identity]
                       [, store_password=Boolean]
                       [, store_shared_secret=Boolean]
                       [, username=sd_user]
                ) OVER ();
```

Paremeters

config_dfs_path =' *filename* '	Required. A string containing the path for the configuration file in DFS. This is either: • A file name (without any path information). The function automatically stores the file in a DFS directory named for the user. Creating this directory prevents different user's files from overwriting one another. • The absolute path /voltagesecure/conf . All users can use this file in calls to the other functions in the SecureData library. This path is the only absolute one that VoltageSecureConfigure allows for this parameter.
identity= *sd_boolean*	A string containing identity to use with the SecureData Appliance. This is usually in the form of an email address. When SecureData uses LDAP authentication, it uses this value to authenticate the user.
store_password= *Boolean*	A Boolean value. When set to true, Vertica stores the LDAP password stored in the password session parameter in the configuration file. Defaults to false.
store_shared_secret= *Boolean*	A Boolean value. When set to true, Vertica stores the shared secret set in the shared_secret session parameter in the configuration file. Defaults to false.
username= *sd_user*	A string containing the user name for authenticating with SecureData.

Notes

- Any SecureData session variables that are set override values from the configuration file. See Configuring access to SecureData for more information.
- The SecureData integration only supports one configuration for the SecureData Appliance at a time.
- Under normal circumstances, users are not able to directly read data from files stored in DFS. However, all users who have access to UDx functions that read from the DFS could access these files from within Vertica.
  In addition, these files are stored as plain text in every node's file system. Anyone with the proper file system access on the nodes can read the file's contents.
  You should take both of these facts into consideration when deciding whether to store sensitive information such as passwords or shared secrets in either the shared or per-user configuration files.

Example

The following example demonstrates saving configuration information to a configuration file named voltage.conf in the user's own Vertica DFS directory.

```
=> \x
Expanded display is on.
=> SELECT VoltageSecureConfigure(USING PARAMETERS config_dfs_path='voltage.conf',
                                 username='alice', identity='alice@example.com', store_password=false
                                ) OVER ();
-[ RECORD 1 ]-----+---------------------------------------------------------------
config_dfs_path   | voltage.conf
identity          | alice@example.com
username          | alice
```

VoltageSecureConfigureGlobal

Saves global SecureData access configuration parameters for all users to a file in the Vertica Distributed File System (DFS). This function stores the configuration file file named /voltagesecure/conf.global in the distributed file system (DFS). You must use this function to configure at least the SecureData policy URL before you can use any of the other Voltage SecureData integration functions.

To refresh the client policy, see VoltageSecureRefreshPolicy .

Syntax

```
VoltageSecureConfigureGlobal(USING PARAMETERS policy_url=url
                             [, allow_short_fpe=Boolean]
                             [, allow_file_cache=Boolean]
                             [, network_timeout=Integer]
                            ) OVER ();
```

Parameters

policy_url= *url*	A string containing the URL of the SecureData policy file. The Vertica SecureData library uses the contents of this file, such as the formats that the SecureData Appliance supports. It also uses the URL of this file to determine the location of the SecureData Appliance.
allow_short_fpe= *Boolean*	A Boolean value. When set to True, SecureData ignores the lower length limit for encoding FPE values. Usually, SecureData does not use FPE to encrypt data shorter than a lower limit (usually, 8 bits). See the *SecureData Architecture Guide* 's section on Data Length Restrictions for more information.
enable_file_cache= *Boolean*	A Boolean value. When set to True, Vertica caches the SecureData policy file and encryption keys to disk, rather than just to memory. Defaults to false.
network_timeout= *Integer*	An Integer value. Configures the network timeout in seconds. Defaults to its maximum value of 300 seconds.

Example

To set the policy URL to https://voltage-pp-0000.example.com/policy/clientPolicy.xml and set the network timeout to 200 seconds:

```
=> SELECT VoltageSecureConfigureGlobal(USING PARAMETERS
                                policy_url='https://voltage-pp-0000.example.com/policy/clientPolicy.xml',
                                NETWORK_TIMEOUT=200)
                                OVER ();

                        policy_url                        | allow_short_fpe | enable_file_cache | network_timeout
----------------------------------------------------------+-----------------+-------------------+-----------------
 https://voltage-pp-0000.example.com/policy/clientPolicy.xml     |                 |                   | 200
(1 row)
```

To view the current policy:

```
=> SELECT VoltageSecureConfigureGlobal() OVER();
                        policy_url                        | allow_short_fpe | enable_file_cache | network_timeout
----------------------------------------------------------+-----------------+-------------------+-----------------
 https://voltage-pp-0000.example.com/policy/clientPolicy.xml     |                 |                   | 200
(1 row)
```

Manually refresh the client policy across the nodes:

```
=> SELECT VoltageSecureRefreshPolicy() OVER ();
                              PolicyRefresh
-------------------------------------------------------------------------------
Successfully refreshed policy on node [v_sandbox_node0001]. Policy on other nodes
will be refreshed the next time a Voltage operation is run on them.
(1 row)
```

VoltageSecureProtect

Calls SecureData to encrypt or hash a value while preserving the structure of the original plaintext.

Syntax

```
VoltageSecureProtect('plaintext' [, 'tweak'] USING PARAMETERS
                format='format_name'
                [, config_dfs_path='config_file']
                [, identity=sd_identity]);
```

Parameters

plaintext	VARCHAR value to encrypt. You must cast other data types (for example DATE values) to VARCHAR when calling this function. NULL values return NULL.

tweak	VARCHAR value analogous to a salt that allows equivalent * plaintext *s to produce different ciphertexts. The same tweak value must for encryption and decryption of a given plaintext. When encrypting or hashing an entire column, you can pass another column for a set of *tweak* values. **Caution** Never use two columns as tweak values for each other or else the original plaintext for both columns will be unrecoverable.
format_name '	String specifying a format-preserving encryption (FPE) or format-preserving hash (FPH) format to encrypt or hash the *plaintext* . To encrypt your data, pass an FPE format defined in your SecureData Appliance. To hash your data, pass an FPH format defined in your SecureData Appliance (version 6.6+). Note that hashing operations are one-way and cannot be reversed with VoltageSecureAccess . **Caution** **Always use the same FPE format to encrypt data in a column.** If you use different FPE formats in the same column (such as loading some data using ssn and other data using auto) there is no way to tell which format was used for any particular row, and properly and improperly encrypted ciphertexts will be indistinguishable.
config_file	String containing the file name of the configuration file to use when authenticating with the SecureData appliance. You must create this file using VoltageSecureConfigure. If you do not supply this parameter, you must set session parameters to configure access to SecureData. See Configuring access to SecureData . Any values set in session parameters override the values in this file.
sd_identity	String containing the identity to use when authenticating with SecureData. SecureData uses this value as a basis for the encryption key. This value usually takes the form of an email address. If supplied, it overrides any values set in the configuration file or session parameters.

Examples

Encrypt a social security number (SSN) value using both the ssn and auto FPE formats (this example assumes that all of the necessary SecureData authentication information has been set in session variables):

```
=> SELECT VoltageSecureProtect('123-45-6789' USING PARAMETERS format='ssn');
 VoltageSecureProtect
----------------------
 376-69-6789
(1 row)

=> SELECT VoltageSecureProtect('123-45-6789' USING PARAMETERS format='auto');
 VoltageSecureProtect
----------------------
 820-31-5110
(1 row)
```

Encrypt two table columns in a COPY statement, authenticating to the SecureData Appliance with the user's private configuration file saved in DFS:

```
=> COPY customers (id, first_name, last_name, ssn_raw FILLER VARCHAR(11),
                cc_num_raw FILLER VARCHAR(25), cvv, dob,
                ssn AS VoltageSecureProtect(ssn_raw USING PARAMETERS
                                    format='ssn',
                                    config_dfs_path='voltage.conf'),
                cc_num AS VoltageSecureProtect(cc_num_raw USING PARAMETERS
                                      format='cc',
                                      config_dfs_path='voltage.conf'))
       FROM '/home/dbadmin/customer_data.csv' DELIMITER ',';
 Rows Loaded
-------------
         100
(1 row)
```

Query for a particular value in an encrypted column:

```
=> SELECT id, first_name, last_name FROM customers
      WHERE ssn = VoltageSecureProtect('559-32-0670' USING PARAMETERS
                                    format='ssn',
                                    config_dfs_path='voltage.conf');

  id  | first_name | last_name
------+------------+-----------
 5345 | Thane      | Ross
(1 row)
```

Encrypting NULL values returns NULL:

```
=> CREATE TABLE nulltable(n VARCHAR (20));
=> INSERT INTO nulltable VALUES (NULL);

=> SELECT VoltageSecureProtect(n USING PARAMETERS format='auto') FROM nulltable;
VoltageSecureProtect
----------------------

(1 row)
```

Encrypt a Unicode string using a predefined format. For a full list of predefined formats, consult the Voltage SecureData documentation.

```
=> SELECT VoltageSecureProtect('123-Hello-こんにちは' USING PARAMETERS format='PREDEFINED::JU_AUTO_TYPE');
VoltageSecureProtect
----------------------
607-Òdìçç-ぶてぴねら
```

Encrypt a SSN with a tweak value:

```
=> SELECT VoltageSecureProtect('681-09-2913', 'tweakvalue123' USING PARAMETERS
   format='ssn-tweak');

 VoltageSecureProtect
----------------------
 721-21-2913

=> SELECT VoltageSecureAccess('721-21-2913', 'tweakvalue123' USING PARAMETERS
   format='ssn-tweak');

 VoltageSecureProtect
----------------------
 681-09-2913
```

Hash a SSN with a FPH format and a tweak value:

```
=> SELECT VoltageSecureProtect('681-09-2913', 'tweakvalue123' USING PARAMETERS
                               format='ssnHash',
                               config_dfs_path='voltage.conf');
VoltageSecureProtect
----------------------
841-68-2913
```

See also

- VoltageSecureAccess
- VoltageSecureProtectAllKeys
- Encrypting, decrypting, and hashing data
- Best practices for safe unicode FPE

VoltageSecureProtectAllKeys

This function helps you locate values in a column encrypted using an Embedded Format Preserving Encryption (eFPE) format. These formats use key rotation, so the encrypted value you get back for a piece of plain text changes over time. You pass this function an unencrypted value. It returns a table consisting of two columns: the unencrypted value and the value encrypted with each of the keys defined for the eFPE. The number of rows in the table are determined by the number of keys the eFPE format contains. Usually, you use the output of this function in a join to locate a matching encrypted value in a table.

Syntax

```
VoltageSecureProtectAllKeys(value USING PARAMETERS format='eFPE_format'
                            [, config_dfs_path=config_file]
                            [, identity=sd_identity] )
```

Parameters

value	VARCHAR containing the value to encrypt. You must cast other data types (for example DATE values) to VARCHAR when calling this function.
format= *eFPE_format*	String containing the name of an eFPE format defined by SecureData. This format must be an eFPE format defined by your SecureData Appliance, or the function returns an error. This format must also match the format of value. VoltageSecureProtectAllKeys returns an error if *value* 's format does not match the one you specify in *eFPE_format* .
config_dfs_path= *config_file*	String containing the file name of the configuration file to use when authenticating with the SecureData appliance. You must create this file using VoltageSecureConfigure. If you do not supply this parameter, you must set session parameters to configure access to SecureData. See Configuring access to SecureData . Any values set in session parameters override the values in this file.
identity= *sd_identity*	String containing the identity to use when authenticating with SecureData. SecureData uses this value as a basis for the encryption key. This value usually takes the form of an email address. If supplied, it overrides any values set in the configuration file or session parameters.

Examples

The following example demonstrates a simple call to VoltageSecureProtectAllKeys.

```
=> SELECT VoltageSecureProtectAllKeys('376765616314013' USING PARAMETERS
                                format='cc_num',
                                config_dfs_path='/voltagesecure/conf')
        OVER ();

     data       |    protected
----------------+-----------------
 376765616314013 | XMVMRU9RJVU4013
 376765616314013 | X5FD4KO1UEE4013
 376765616314013 | M7ZXTIQVCPB4013
 376765616314013 | UBOSC9K3EXZ4013
 376765616314013 | ZJ1C50C9L9R4013
(5 rows)
```

In this example, the cc_num eFPE format has five keys defined for it, so the return value is a table containing five rows.

The following example shows a more common use: querying a table column that is encrypted using an eFPE format.

```
=> SELECT id, first_name, last_name FROM customers3 u
        JOIN (SELECT VoltageSecureProtectAllKeys('376765616314013' USING PARAMETERS
                                        format='cc_num',
                                        config_dfs_path='/voltagesecure/conf')
        OVER ()) pak
        ON u.cc_num = pak.protected;

  id  | first_name | last_name
------+------------+-----------
 5345 | Thane      | Ross

(1 row)
```

In the previous example, the customers3 table is joined to the output from VoltageSecureProtectAllKeys. Any rows in the customers3 table where the encryted cc_num column value matches values from the protected column of VoltageSecureProtectAllKeys matches appear in the output.

This function returns an error if you use it on a non-eFPE format:

```
=> SELECT first_name, last_name, ssn FROM customers u
       JOIN (
          SELECT VoltageSecureProtectAllKeys('232-28-0657' USING PARAMETERS format='ssn',
                                        config_dfs_path='/voltagesecure/conf')
          OVER ()
       )
       pak ON u.ssn = pak.protected;
ERROR 5861:  Error calling processPartition() in User Function VoltageSecureProtectAllKeys
at [ProtectAllKeys.cpp:21], error code: 1711, message: Error getting key numbers:
eFPE format required
```

See also

- Configuring access to SecureData
- VoltageSecureAccess
- VoltageSecureProtect
- VoltageSecureProtectAllKeys

VoltageSecureRefreshPolicy

Immediately refreshes the client policy on the initiator node. Policies on non-initiator nodes are refreshed the next time a Voltage function is called on them.

Syntax

```
VoltageSecureRefreshPolicy()
```

Parameters

None

Example

Manually refresh the client policy across the nodes:

```
=> SELECT VoltageSecureRefreshPolicy() OVER ();
                              PolicyRefresh
-------------------------------------------------------------------------------------
Successfully refreshed policy on node [v_sandbox_node0001]. Policy on other nodes
will be refreshed the next time a Voltage operation is run on them.
(1 row)
```

See also

- VoltageSecureConfigure
- VoltageSecureProtect
- VoltageSecureProtectAllKeys
- Encrypting, decrypting, and hashing data

SQL reference

Vertica offers a robust set of SQL elements that allow you to manage and analyze massive volumes of data quickly and reliably, including:

- Language elements such as keywords, operators, expressions, predicates, and hints
- Data types including complex types
- Functions including Vertica-specific functions that take advantage of Vertica's unique column-store architecture
- SQL statements that let you write robust queries to quickly return large volumes of data

In this section

- System limits
- Language elements
- Data types
- Configuration parameters
- File systems and object stores
- Functions
- Statements
- Vertica system tables

System limits

This section describes system limits on the size and number of objects in a Vertica database. In most cases, computer memory and disk drive are the limiting factors.

Item	**Maximum**
Nodes	128 (without Vertica assistance)
Database size	Dependent on maximum disk configuration, approximately: *numFiles * platformFileSize*
Table size	Smaller of: • 2^{64} rows per node • 2^{63} bytes per column
Row size	$(2^{31}) -1$ Row size is approximately the sum of its maximum column sizes. For example, a VARCHAR(80) has a maximum size of 80 bytes.
Key size	Dependent on row size

Tables/projections per database	Dependent on physical RAM, as the catalog must fit in memory.
Concurrent connections per node	Dependent on physical RAM (or threads per process), typically 1024 **Default:** 50
Concurrent connections per cluster	Dependent on physical RAM of a single node (or threads per process), typically 1024
Columns per table/view	9800
Arguments per function call	9800
Rows per load	2^{63}
ROS containers per projection	1024 See Minimizing partitions .
Length of fixed-length column	65000 bytes
Length of variable-length column	32 MB
Length of basic names	128 bytes. Basic names include table names, column names, etc.
Query length	None
Depth of nesting subqueries	None in FROM , WHERE , and HAVING clauses

Language elements

The following topics provide detailed descriptions of the language elements and conventions of Vertica SQL.

In this section

- Keywords
- Identifiers
- Literals
- Operators
- Expressions
- Lambda functions
- Predicates
- Hints
- Window clauses

Keywords

Keywords are words that have a specific meaning in the SQL language. Every SQL statement contains one or more keywords. Although SQL is not case-sensitive with respect to keywords, they are generally shown in uppercase letters throughout this documentation for readability purposes.

> **Note**
> If you use a keyword as the name of an identifier or an alias in your SQL statements, you may have to qualify the keyword with AS or double-quotes. Vertica requires AS or double-quotes for certain reserved and non-reserved words to prevent confusion with expression syntax, or where the use of a word would be ambiguous.

Reserved words and keywords

Many keywords are also reserved words.

Vertica recommends that you not use reserved words as names for objects, or as identifiers. Including reserved words can make your SQL statements confusing. Reserved words that are used as names for objects or identifiers must be enclosed in double-quotes.

> **Note**
> All reserved words are also keywords, but Vertica can add reserved words that are not keywords. A reserved word can simply be a word that is reserved for future use.

Non-reserved keywords

Non-reserved keywords have a special meaning in some contexts, but can be used as identifiers in others. You can use non-reserved keywords as aliases—for example, SOURCE :

```
=> SELECT my_node AS SOURCE FROM nodes;
```

> **Note**
> Vertica uses several non-reserved keywords in directed queries to specify special join types. You can use these keywords as table aliases only if they are double-quoted; otherwise, double-quotes can be omitted:
>
> - ANTI
> - NULLAWARE
> - SEMI
> - SEMIALL
> - UNI

Viewing the list of reserved and non-reserved keywords

To view the current list of Vertica reserved and non-reserved words, query system table KEYWORDS . Vertica lists keywords alphabetically and identifies them as reserved (R) or non-reserved (N).

For example, the following query gets all reserved keywords that begin with B:

```
=> SELECT * FROM keywords WHERE reserved = 'R' AND keyword ilike 'B%';
 keyword | reserved
---------+----------
 BETWEEN | R
 BIGINT  | R
 BINARY  | R
 BIT     | R
 BOOLEAN | R
 BOTH    | R
(6 rows)
```

Identifiers

Identifiers (names) of objects such as schema, table, projection, column names, and so on, can be up to 128 bytes in length.

Unquoted identifiers

Unquoted SQL identifiers must begin with one of the following:

- Non-Unicode letters: A–Z or a-z
- Underscore (_)

Subsequent characters in an identifier can be any combination of the following:

- Non-Unicode letters: A–Z or a-z
- Underscore (_)
- Digits(0–9)
- Unicode letters (letters with diacriticals or not in the Latin alphabet), unsupported for model names
- Dollar sign ($), unsupported for model names

> **Caution**
> The SQL standard does not support dollar sign in identifiers, so usage can cause application portability problems.

Quoted identifiers

> **Note**
> Quoted identifiers are not supported for model names

Identifiers enclosed in double quote (") characters can contain any character. If you want to include a double quote, you need a pair of them; for example """" . You can use names that would otherwise be invalid—for example, names that include only numeric characters ("123") or contain space characters, punctuation marks, and SQL or Vertica-reserved keywords. For example:

```
CREATE SEQUENCE "my sequence!";
```

Double quotes are required for non-alphanumerics and SQL keywords such as "1time", "Next week" and "Select".

Case sensitivity

Identifiers are not case-sensitive. Thus, identifiers "ABC" , "ABc" , and "aBc" are synonymous, as are ABC , ABc , and aBc.

Non-ASCII characters

Vertica accepts non-ASCII UTF-8 Unicode characters for table names, column names, and other identifiers, extending the cases where upper/lower case distinctions are ignored (case-folded) to all alphabets, including Latin, Cyrillic, and Greek.

For example, the following CREATE TABLE statement uses the ß (German eszett) in the table name:

```
=> CREATE TABLE straße(x int, y int);
  CREATE TABLE
```

Identifiers are stored as created

SQL identifiers, such as table and column names, are not converted to lowercase. They are stored as created, and references to them are resolved using case-insensitive compares. For example, the following statement creates table ALLCAPS .

```
=> CREATE TABLE ALLCAPS(c1 varchar(30));
=> INSERT INTO ALLCAPS values('upper case');
```

The following statements are variations of the same query:

```
=> SELECT * FROM ALLCAPS;
=> SELECT * FROM allcaps;
=> SELECT * FROM "allcaps";
```

The three queries all return the same result:

```
    c1
------------
 upper case
(1 row)
```

Note that Vertica returns an error if you try to create table AllCaps :

```
=> CREATE TABLE AllCaps(c1 varchar(30));
  ROLLBACK:  table "AllCaps" already exists
```

See QUOTE_IDENT for additional information.

Literals

Literals are numbers or strings used in SQL as constants. Literals are included in the select-list, along with expressions and built-in functions and can also be constants.

Vertica provides support for number-type literals (integers and numerics), string literals, VARBINARY string literals, and date/time literals. The various

string literal formats are discussed in this section.

In this section

- Number-type literals
- String literals
- Date/time literals

Number-type literals

Vertica supports three types of numbers: integers, numerics, and floats.

- Integers are whole numbers less than 2^63 and must be digits.
- Numerics are whole numbers larger than 2^63 or that include a decimal point with a precision and a scale. Numerics can contain exponents. Numbers that begin with 0x are hexadecimal numerics.

Numeric-type values can also be generated using casts from character strings. This is a more general syntax. See the Examples section below, as well as Data type coercion operators (CAST) .

Syntax

```
digits
digits.[digits] | [digits].digits
digits e[+-]digits | [digits].digits e[+-]digits | digits.[digits] e[+-]digits
```

Parameters

digits

One or more numeric characters, 0 through 9

e

Exponent marker

Notes

- At least one digit must follow the exponent marker (e), if e is present.
- There cannot be any spaces or other characters embedded in the constant.
- Leading plus (+) or minus (–) signs are not considered part of the constant; they are unary operators applied to the constant.
- In most cases a numeric-type constant is automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it as described in Data type coercion operators (CAST) .
- Floating point literals are not supported. If you specifically need to specify a float, you can cast as described in Data type coercion operators (CAST) .
- Vertica follows the IEEE specification for floating point, including NaN (not a number) and Infinity (Inf).
- A NaN is not greater than and at the same time not less than anything, even itself. In other words, comparisons always return false whenever a NaN is involved.
- Dividing INTEGERS (x / y) yields a NUMERIC result. You can use the // operator to truncate the result to a whole number.

Examples

The following are examples of number-type literals:

```
42
3.5
4.
.001
5e2
1.925e-3
```

Scientific notation :

```
=> SELECT NUMERIC '1e10';
 ?column?
-------------
 10000000000
(1 row)
```

BINARY scaling :

```
=> SELECT NUMERIC '1p10';
 ?column?
----------
    1024
(1 row)
=> SELECT FLOAT 'Infinity';
  ?column?
----------
 Infinity
(1 row)
```

The following examples illustrated using the / and // operators to divide integers:

```
=> SELECT 40/25;
 ?column?
----------------------
 1.600000000000000000
(1 row)
=> SELECT 40//25;
 ?column?
----------
        1
(1 row)
```

See also

Data type coercion

String literals

String literals are string values surrounded by single or double quotes. Double-quoted strings are subject to the backslash, but single-quoted strings do not require a backslash, except for \' and \\ .

You can embed single quotes and backslashes into single-quoted strings.

To include other backslash (escape) sequences, such as \t (tab), you must use the double-quoted form.

Precede single-quoted strings with a space between the string and its preceding word, since single quotes are allowed in identifiers.

See also

- SET STANDARD_CONFORMING_STRINGS
- SET ESCAPE_STRING_WARNING
- Internationalization parameters
- Implement locales for international data sets

In this section

- Character string literals
- Dollar-quoted string literals
- Unicode string literals
- VARBINARY string literals
- Extended string literals

Character string literals

Character string literals are a sequence of characters from a predefined character set, enclosed by single quotes.

Syntax

```
'character-seq'
```

Parameters

character-seq

Arbitrary sequence of characters

Embedded single quotes

If a character string literal includes a single quote, it must be doubled. For example:

```
=> SELECT 'Chester''s gorilla';
  ?column?
--------------------
Chester's gorilla
(1 row)
```

Standard-conforming strings and escape characters

Vertica uses standard-conforming strings as specified in the SQL standard, so backslashes are treated as string literals and not escape characters.

> **Note**
> Earlier versions of Vertica did not use standard conforming strings, and backslashes were always considered escape sequences. To revert to this older behavior, set the configuration parameter StandardConformingStrings to 0. You can also use the EscapeStringWarning parameter to locate back slashes which have been incorporated into string literals, in order to remove them.

Examples

```
=> SELECT 'This is a string';
  ?column?
-------------------
 This is a string
(1 row)
=> SELECT 'This \is a string';
   WARNING:  nonstandard use of escape in a string literal at character 8
   HINT:  Use the escape string syntax for escapes, e.g., E'\r\n'.
    ?column?
-------------------
 This is a string
(1 row)
vmartdb=> SELECT E'This \is a string';
    ?column?
-------------------
 This is a string
=> SELECT E'This is a \n new line';
      ?column?
----------------------
 This is a
 new line
(1 row)
=> SELECT 'String''s characters';
     ?column?
--------------------
 String's characters
(1 row)
```

See also

- SET STANDARD_CONFORMING_STRINGS
- SET ESCAPE_STRING_WARNING
- Implement locales for international data sets

Dollar-quoted string literals

Dollar-quoted string literals are rarely used, but are provided here for your convenience.

The standard syntax for specifying string literals can be difficult to understand. To allow more readable queries in such situations, Vertica SQL provides ***dollar quoting*** . Dollar quoting is not part of the SQL standard, but it is often a more convenient way to write complicated string literals than the standard-compliant single quote syntax.

Syntax

```
$$characters$$
```

Parameters

characters

Arbitrary sequence of characters bounded by paired dollar signs ($$)

Dollar-quoted string content is treated as a literal. Single quote, backslash, and dollar sign characters have no special meaning within a dollar-quoted string.

Notes

A dollar-quoted string that follows a keyword or identifier must be separated from the preceding word by whitespace; otherwise, the dollar-quoting delimiter is taken as part of the preceding identifier.

Examples

```
=> SELECT $$Fred's\n car$$;
    ?column?
-------------------
 Fred's\n car
(1 row)

=> SELECT 'SELECT 'fact';';
ERROR:  syntax error at or near "';'" at character 21
LINE 1: SELECT 'SELECT 'fact';';

=> SELECT 'SELECT $$fact';$$;
   ?column?
---------------
 SELECT $$fact
(1 row)

=> SELECT 'SELECT ''fact'';';
    ?column?
----------------
 SELECT 'fact';
(1 row)
```

Unicode string literals

Syntax

```
U&'characters' [ UESCAPE '<Unicode escape character>' ]
```

Parameters

characters

Arbitrary sequence of UTF-8 characters bounded by single quotes (')

Unicode escape character

A single character from the source language character set other than a hexit, plus sign (+), quote ('), double quote ("), or white space

Using standard conforming strings

With StandardConformingStrings enabled, Vertica supports SQL standard Unicode character string literals (the character set is UTF-8 only).

Before you enter a Unicode character string literal, enable standard conforming strings in one of the following ways.

- To enable for all sessions, update the configuration parameter StandardConformingStrings .
- To treat backslashes as escape characters for the current session, use the SET STANDARD_CONFORMING_STRINGS statement.

See also Extended String Literals .

Examples

To enter a Unicode character in hexadecimal, such as the Russian phrase for "thank you, use the following syntax:

```
=> SET STANDARD_CONFORMING_STRINGS TO ON;
=> SELECT U&'\0441\043F\0430\0441\0438\0431\043E' as 'thank you';
 thank you
-----------
 спасибо
(1 row)
```

To enter the German word mude (where u is really u-umlaut) in hexadecimal:

```
=> SELECT U&'m\00fcde';
?column?
----------
müde
(1 row)
=> SELECT 'ü';
?column?
----------
ü
(1 row)
```

To enter the LINEAR B IDEOGRAM B240 WHEELED CHARIOT in hexadecimal:

```
=> SELECT E'\xF0\x90\x83\x8C';
?column?
----------
(wheeled chariot character)
(1 row)
```

> **Note**
> Not all fonts support the wheeled chariot character.

See also

- SET STANDARD_CONFORMING_STRINGS
- SET ESCAPE_STRING_WARNING
- Internationalization parameters
- Implement locales for international data sets

VARBINARY string literals

VARBINARY string literals allow you to specify hexadecimal or binary digits in a string literal.

Syntax

```
X''
B''
```

Parameters

X or x

Specifies hexadecimal digits. The < *hexadecimal digits* > string must be enclosed in single quotes (').

B or b

Specifies binary digits. The <*binary digits*> string must be enclosed in single quotes (').

Examples

```
=> SELECT X'abcd';
 ?column?
----------
 \253\315
(1 row)

=> SELECT B'101100';
 ?column?
----------
 ,
(1 row)
```

Extended string literals

Syntax

```
E'characters'
```

Parameters

characters

Arbitrary sequence of characters bounded by single quotes (') You can use C-style backslash sequence in extended string literals, which are an extension to the SQL standard. You specify an extended string literal by writing the letter E as a prefix (before the opening single quote); for example:

```
E'extended character string\n'
```

Within an extended string, the backslash character (\ ) starts a C-style backslash sequence, in which the combination of backslash and following character or numbers represent a special byte value, as shown in the following list. Any other character following a backslash is taken literally; for example, to include a backslash character, write two backslashes (\\ ).

- \\ is a backslash
- \b is a backspace
- \f is a form feed
- \n is a newline
- \r is a carriage return
- \t is a tab
- \x##, where ## is a 1 or 2-digit hexadecimal number; for example \x07 is a tab
- \### , where ### is a 1, 2, or 3-digit octal number representing a byte with the corresponding code.

When an extended string literal is concatenated across lines, write only E before the first opening quote:

```
=> SELECT E'first part o'
   'f a long line';
       ?column?
--------------------------
 first part of a long line
(1 row)
```

Two adjacent single quotes are used as one single quote:

```
=> SELECT 'Aren''t string literals fun?';
         ?column?
-----------------------------
 Aren't string literals fun?
(1 row)
```

Standard conforming strings and escape characters

When interpreting commands, such as those entered in vsql or in queries passed via JDBC or ODBC, Vertica uses standard conforming strings as specified in the SQL standard. In standard conforming strings, backslashes are treated as string literals (ordinary characters), not escape characters.

> **Note**
> Text read in from files or streams (such as the data inserted using the COPY statement) are not treated as literal strings. The COPY command defines its own escape characters for the data it reads. See the COPY statement documentation for details.

The following options are available, but Vertica recommends that you migrate your application to use standard conforming strings at your earliest convenience, after warnings have been addressed.

- To treat back slashes as escape characters, set configuration parameter StandardConformingStrings to 0.
- To enable standard conforming strings permanently, set the StandardConformingStrings parameter to '1', as described below .
- To enable standard conforming strings per session, use SET STANDARD_CONFORMING_STRING TO ON , which treats backslashes as escape characters for the current session only.

Identifying strings that are not standard conforming

The following procedure can be used to identify nonstandard conforming strings in your application so that you can convert them into standard conforming strings:

1. Be sure the StandardConformingStrings parameter is off, as described in Internationalization parameters .

```
=> ALTER DATABASE DEFAULT SET StandardConformingStrings = 0;
```

> **Note**
> Vertica recommends that you migrate your application to use standard conforming strings .

2. If necessary, turn on the EscapeStringWarning parameter.

```
=> ALTER DATABASE DEFAULT SET EscapeStringWarning = 1;
```

Vertica now returns a warning each time it encounters an escape string within a string literal. For example, Vertica interprets the \n in the following example as a new line:

```
=> SELECT 'a\nb';
  WARNING:  nonstandard use of escape in a string literal at character 8
  HINT:  Use the escape string syntax for escapes, e.g., E'\r\n'.
 ?column?
----------
 a
b
(1 row)
```

When StandardConformingStrings is ON , the string is interpreted as four characters: a \ n b .

Modify each string that Vertica flags by extending it as in the following example:

```
E'a\nb'
```

Or if the string has quoted single quotes, double them; for example, 'one'' double'.

3. Turn on the StandardConformingStrings parameter for all sessions:

```
=> ALTER DATABASE DEFAULT SET StandardConformingStrings = 1;
```

Doubled single quotes

This section discusses vsql inputs that are not passed on to the server. Vertica recognizes two consecutive single quotes within a string literal as one single quote character. For example, the following inputs, 'You''re here!' ignored the second consecutive quote and returns the following:

```
=> SELECT 'You''re here!';
   ?column?
--------------
 You're here!at
(1 row)
```

This is the SQL standard representation and is preferred over the form, 'You\'re here!' , because backslashes are not parsed as before. You need to escape the backslash:

```
=> SELECT (E'You\'re here!');
   ?column?
--------------
 You're here!
(1 row)
```

This behavior change introduces a potential incompatibility in the use of the vsql meta-command \set , which automatically concatenates its arguments. For example:

```
\set file  '\''  'pwd'  '/file.txt'  '\''\echo :file
```

vsql takes the four arguments and outputs the following:

```
'/home/vertica/file.txt'
```

Vertica parses the adjacent single quotes as follows:

```
\set file  '\''pwd''/file.txt''\''\echo :file
'/home/vertica/file.txt''
```

Note the extra single quote at the end. This is due to the pair of adjacent single quotes together with the backslash-quoted single quote.

The extra quote can be resolved either as in the first example above, or by combining the literals as follows:

```
\set file '\''`pwd`'/file.txt''\''\echo :file
'/home/vertica/file.txt'
```

In either case the backslash-quoted single quotes should be changed to doubled single quotes as follows:

```
\set file '''' `pwd` '/file.txt'''
```

Additional examples

```
=> SELECT 'This \is a string';
    ?column?
------------------
 This \is a string
(1 row)

=> SELECT E'This \is a string';
    ?column?
------------------
 This is a string

=> SELECT E'This is a \n new line';
      ?column?
----------------------
 This is a
 new line
(1 row)

=> SELECT 'String''s characters';
     ?column?
--------------------
 String's characters
(1 row)
```

Date/time literals

Date or time literal input must be enclosed in single quotes. Input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional POSTGRES, and others.

Vertica handles date/time input more flexibly than the SQL standard requires. The exact parsing rules of date/time input and for the recognized text fields including months, days of the week, and time zones are described in Date/time expressions .

In this section

- Time zone values
- Day of the week names
- Month names
- Interval literal

Time zone values

Vertica attempts to be compatible with the SQL standard definitions for time zones. However, the SQL standard has an odd mix of date and time types and capabilities. Obvious problems are:

- Although the DATE type does not have an associated time zone, the TIME type can. Time zones in the real world have little meaning unless associated with a date as well as a time, since the offset can vary through the year with daylight-saving time boundaries.
- Vertica assumes your local time zone for any data type containing only date or time.
- The default time zone is specified as a constant numeric offset from UTC. It is therefore not possible to adapt to daylight-saving time when doing date/time arithmetic across DST boundaries.

To address these difficulties, OpenText recommends using Date/Time types that contain both date and time when you use time zones. OpenText recommends that you do *not* use the type TIME WITH TIME ZONE, even though it is supported it for legacy applications and for compliance with the SQL standard.

Time zones and time-zone conventions are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900's, but continue to be prone to arbitrary changes, particularly with respect to daylight-savings rules.

Vertica currently supports daylight-savings rules over the time period 1902 through 2038, corresponding to the full range of conventional UNIX system time. Times outside that range are taken to be in "standard time" for the selected time zone, no matter what part of the year in which they occur.

Example	Description
PST	Pacific Standard Time
-8:00	ISO-8601 offset for PST
-800	ISO-8601 offset for PST
-8	ISO-8601 offset for PST
zulu	Military abbreviation for UTC
z	Short form of zulu

Day of the week names

The following tokens are recognized as names of days of the week:

Day	Abbreviations
SUNDAY	SUN
MONDAY	MON
TUESDAY	TUE, TUES
WEDNESDAY	WED, WEDS
THURSDAY	THU, THUR, THURS
FRIDAY	FRI
SATURDAY	SAT

Month names

The following tokens are recognized as names of months:

Month	Abbreviations

JANUARY	JAN
FEBRUARY	FEB
MARCH	MAR
APRIL	APR
MAY	MAY
JUNE	JUN
JULY	JUL
AUGUST	AUG
SEPTEMBER	SEP, SEPT
OCTOBER	OCT
NOVEMBER	NOV
DECEMBER	DEC

Interval literal

A literal that represents a time span.

Syntax

```
[ @ ] [-] { quantity subtype-unit }[...] [ AGO ]
```

Parameters

@

Ignored

- (minus)

Specifies a negative interval value.

quantity

Integer numeric constant

[subtype-unit](/sql-reference/language-elements/literals/datetime-literals/interval-literal/interval-subtype-units.html)

See Interval subtype units for valid values. Subtype units must be specified for year-month intervals; they are optional for day-time intervals.

AGO

Specifies a negative interval value. AGO and - (minus) are synonymous.

Notes

- The amounts of different units are implicitly added up with appropriate sign accounting.
- The boundaries of an interval constant are:
  - 9223372036854775807 usec to -9223372036854775807 usec
  - 296533 years 3 mons 21 days 04:00:54.775807 to -296533 years -3 mons -21 days -04:00:54.775807
- The range of an interval constant is +/– $2^{63} - 1$ microseconds.
- In Vertica, interval fields are additive and accept large floating-point numbers.

Examples

See Specifying interval input .

In this section

- Interval subtype units
- Interval qualifier

Interval subtype units

The following tables lists subtype units that you can specify in an interval literal, divided into major categories:

- Year-month subtype units
- Day-time subtype units

Year-month subtype units

Subtypes	Units	Notes
Millennium	mil , millennium , millennia , mils , millenniums	
Century	c , cent , century , centuries	
Decade	dec , decs , decade , decades	
Year	a	Julian year: 365.25 days
	ka	Julian kilo-year: 365250 days
	y , yr , yrs , year , years	Calendar year: 365 days
Quarter	q , qtr , qtrs , quarter , quarters	
Month	m , mon , mons , months , month	Vertica can interpret m as minute or month, depending on context. See Processing m Input below.
Week	w , wk , week , wks , weeks	

Day-time subtype units

Subtypes	Units	Notes
Day	d , day , days	
Hour	h , hr , hrs , hour , hours	
Minute	m , min , mins , minute , minutes	Vertica can interpret input unit m as minute or month, depending on context. See Processing m Input below.
Second	s , sec , secs , second , seconds	
Millisecond	ms , msec , msecs , msecond , mseconds , millisecond , milliseconds	
Microsecond	us , usec , usecs , usecond , useconds , microseconds , microsecond	

Processing m input

Vertica uses context to interpret the input unit m as months or minutes. For example, the following command creates a one-column table with an interval value:

```
=> CREATE TABLE int_test(i INTERVAL YEAR TO MONTH);
```

Given the following INSERT statement, Vertica interprets the interval literal 1y 6m as 1 year 6 months:

```
=> INSERT INTO int_test VALUES('1y 6m');
 OUTPUT
--------
      1
(1 row)
=> COMMIT;
COMMIT
=> SET INTERVALSTYLE TO UNITS;
SET
=> SELECT * FROM int_test;
        i
-----------------
 1 year 6 months
(1 row)
```

> **Tip**
> The SET INTERVALSTYLE statement changes interval output to include subtype unit identifiers. For details, see Setting interval unit display .

The following ALTER TABLE statement adds a DAY TO MINUTE interval column to table int_test :

```
=>  ALTER TABLE int_test ADD COLUMN x INTERVAL DAY TO MINUTE;
ALTER TABLE
```

The next INSERT statement sets the first and second columns to 3y 20m and 1y 6m, respectively. In this case, Vertica interprets the m input literals in two ways:

- For column i, Vertica interprets the m input as months, and displays 4 years 8 months.
- For column x, Vertica interprets the m input as minutes. Because the interval is defined as DAY TO MINUTE, it converts the inserted input value 1y 6m to 365 days 6 minutes:

```
=> INSERT INTO int_test VALUES ('3y 20m', '1y 6m');
 OUTPUT
--------
      1
(1 row)

=> SELECT * FROM int_test;
        i         |       x
------------------+-----------------
 1 year 6 months  |
 4 years 8 months | 365 days 6 mins
(2 rows)
```

Interval qualifier

Specifies how to interpret and format an interval literal for output, and optionally sets precision. Interval qualifiers are composed of one or two units:

```
unit [ TO unit ] [ (p) ]
```

where:

- *unit* specifies a day-time or year-month subtype .
- *p* specifies precision, an integer between 0 and 6.

> **Note**
> Precision only applies to SECOND units, specifying the number of decimal digits to show after the seconds value decimal point. The default precision for SECOND is 6.
>
> When SECOND is the second unit of a qualifier—for example, DAY TO SECOND or MINUTE TO SECOND—it has a precision of 2 places before the decimal point.

For example:

```
=> SELECT INTERVAL '6 122.538987' MINUTE TO SECOND (5);
  ?column?
-------------
 08:02.53899
(1 row)
```

For details, see Specifying interval precision .

If an interval omits an interval qualifier, the default is DAY TO SECOND(6) .

Interval qualifiers are divided into two categories:

- day-time
- year-month

Day-time interval qualifiers

Qualifier	Description
DAY	Unconstrained
DAY TO HOUR	Span of days and hours
DAY TO MINUTE	Span of days and minutes
DAY TO SECOND [(*p*)]	Span of days, hours, minutes, seconds, and fractions of a second.
HOUR	Hours within days
HOUR TO MINUTE	Span of hours and minutes
HOUR TO SECOND [(*p*)]	Span of hours and seconds
MINUTE	Minutes within hours
MINUTE TO SECOND [(*p*)]	Span of minutes and seconds
SECOND [(*p*)]	Seconds within minutes

Year-month interval qualifiers

YEAR
: Unconstrained

MONTH
: Months within year

YEAR TO MONTH
: Span of years and months

Note
Vertica also supports INTERVALYM , which is an alias for INTERVAL YEAR TO MONTH . Thus, the following two statements are equivalent:

```
=> SELECT INTERVALYM '1 2';
 ?column?
----------
 1-2
(1 row)
=> SELECT INTERVAL '1 2' YEAR TO MONTH;
?column?
----------
1-2
(1 row)
```

Examples

See Controlling interval format .

Operators

Operators are logical, mathematical, and equality symbols used in SQL to evaluate, compare, or calculate values.

In this section

- Bitwise operators
- Logical operators
- Comparison operators
- Data type coercion operators (CAST)
- Date/time operators
- Mathematical operators
- NULL operators
- String concatenation operators

Bitwise operators

Bitwise operators perform bit manipulations on INTEGER and BINARY/VARBINARY data types:

Operator	Description	Example	Result
&	AND	12 & 4	4
|	OR	32 | 3	35
#	XOR	17 # 5	20
~	NOT	~1	-2
<< †	Bitwise shift left	1 << 4	16
>> †	Bitwise shift right	8 >> 2	2

† Invalid for BINARY/VARBINARY data types

String argument handling

String arguments must be explicitly cast as BINARY or VARBINARY data types for all bitwise operators. For example:

```
=> SELECT 'xyz'::VARBINARY & 'zyx'::VARBINARY AS AND;
 AND
-----
 xyx
(1 row)

=> SELECT 'xyz'::VARBINARY | 'zyx'::VARBINARY AS OR;
 OR
-----
 zyz
(1 row)
```

Bitwise operators treats all string arguments as equal in length. If the arguments have different lengths, the operator function right-pads the smaller string with one or more zero bytes to equal the length of the larger string.

For example, the following statement ANDs unequal strings xyz and zy . Vertica right-pads string zy with one zero byte. The last character in the result is represented accordingly, as \000 :

```
=> SELECT 'xyz'::VARBINARY & 'zy'::VARBINARY AS AND;
  AND
--------
 xy\000
(1 row)
```

Logical operators

Vertica supports the logical operators AND, OR, and NOT:

- AND evaluates to true when both of the conditions joined by AND are true.
- OR evaluates to true when either condition is true.
- NOT negates the result of any Boolean expression.

AND and OR are commutative—that is, you can switch left and right operands without affecting the result. However, the order of evaluation of sub-expressions is not defined. To force evaluation order, use a CASE construct.

> **Caution**
> Do not confuse Boolean operators with the Boolean predicate or Boolean data type, which can have only two values: true and false.

Logic

SQL uses a three-valued Boolean logic where NULL represents "unknown":

- true AND NULL = NULL
- true OR NULL = true
- false AND NULL = false
- false OR NULL = NULL
- NULL AND NULL = NULL
- NULL OR NULL = NULL
- NOT NULL = NULL

Comparison operators

Comparison operators are available for all data types where comparison makes sense. All comparison operators are binary operators that return values of true, false, or NULL (unknown).

Operator	Description	Binary function
<	less than	binary_lt

>	greater than	binary_gt
<=	less than or equal to	binary_le
>=	greater than or equal to	binary_ge
= , <=>	equal **Note** Do not use the negation operator (!) with the <=> operator. Instead, use != to test for inequality. ! , except as part of != , is the factorial operator.	binary_eq
!= , <>	not equal (unsupported for correlated subqueries)	binary_ne

NULL handling

Comparison operators return NULL (unknown) if either or both operands are null. One exception applies: <=> returns true if both operands are NULL, and false if one operand is NULL.

Collections

When comparing collections, null collections are ordered last. Otherwise, collections are compared element by element until there is a mismatch, and then they are ordered based on the non-matching elements. If all elements are equal up to the length of the shorter one, then the shorter one is ordered first.

Data type coercion operators (CAST)

Data type coercion (casting) passes an expression value to an input conversion routine for a specified data type, resulting in a constant of the indicated type. In Vertica, data type coercion can be invoked by an explicit cast request that uses one of the following constructs:

Syntax

```
SELECT CAST ( expression AS data-type )
SELECT expression::data-type
SELECT data-type 'string'
```

Parameters

expression	An expression of any type
data-type	An SQL data type that Vertica supports to convert *expression* .

Truncation

If a binary value is cast (implicitly or explicitly) to a binary type with a smaller length, the value is silently truncated. For example:

```
=> SELECT 'abcd'::BINARY(2);
 ?column?
----------
 ab
(1 row)
```

Similarly, if a character value is cast (implicitly or explicitly) to a character value with a smaller length, the value is silently truncated. For example:

```
=> SELECT 'abcd'::CHAR(3);
 ?column?
----------
 abc
(1 row)
```

Binary casting and resizing

Vertica supports only casts and resize operations as follows:

- BINARY to and from VARBINARY
- VARBINARY to and from LONG VARBINARY
- BINARY to and from LONG VARBINARY

On binary data that contains a value with fewer bytes than the target column, values are right-extended with the zero byte '\0' to the full width of the column. Trailing zeros on variable-length binary values are not right-extended:

```
=> SELECT 'ab'::BINARY(4), 'ab'::VARBINARY(4), 'ab'::LONG VARBINARY(4);
  ?column?  | ?column? | ?column?
------------+----------+----------
 ab\000\000 | ab       | ab
(1 row)
```

Automatic coercion

The explicit type cast can be omitted if there is no ambiguity as to the type the constant must be. For example, when a constant is assigned directly to a column, it is automatically coerced to the column's data type.

Examples

```
=> SELECT CAST((2 + 2) AS VARCHAR);
 ?column?
----------
 4
(1 row)

=> SELECT (2 + 2)::VARCHAR;
 ?column?
----------
 4
(1 row)

=> SELECT INTEGER '123';
 ?column?
----------
      123
(1 row)

=> SELECT (2 + 2)::LONG VARCHAR
 ?column?
----------
 4
(1 row)

=> SELECT '2.2' + 2;
   ERROR:  invalid input syntax for integer: "2.2"

=> SELECT FLOAT '2.2' + 2;
 ?column?
----------
      4.2
(1 row)
```

See also

- Data Type Conversions
- Data Type Coercion Chart
- CAST Failures

In this section

- Cast failures

Cast failures

When you invoke data type coercion (casting) by an explicit cast and the cast fails, the result returns either an error or NULL. Cast failures commonly occur when you try to cast conflicting conversions, such as coercing a VARCHAR expression that contains letters to an integer.

When a cast fails, the result returned depends on the data type.

Data type	Cast failure default
date , time	NULL
literals	error
all other types	error

Enabling strict time casts

You can enable all cast failures to result in an error, including those for date/time data types. Doing so lets you see the reason why some or all of the cast failed. To return an error instead of NULL, set the configuration parameter EnableStrictTimeCasts to 1:

```
ALTER SESSION SET EnableStrictTimeCasts=1;
```

By default, EnableStrictTimeCasts is set to 0. Thus, the following attempt to cast a VARCHAR to a TIME data type returns NULL:

```
==> SELECT current_value from configuration_parameters WHERE parameter_name ilike '%EnableStrictTimeCasts%';
 current_value
---------------
 0
(1 row)

=> CREATE TABLE mytable (a VARCHAR);
CREATE TABLE
=> INSERT INTO mytable VALUES('one');
OUTPUT
--------
1
(1 row)

=> INSERT INTO mytable VALUES('1');
OUTPUT
--------
1
(1 row)

=> COMMIT;
COMMIT
=> SELECT a::time FROM mytable;
 a
---


(2 rows)
```

If EnableStrictTimeCasts is enabled, the cast failure returns an error:

```
=> ALTER SESSION SET EnableStrictTimeCasts=1;
ALTER SESSION
=> SELECT a::time FROM mytable;
ERROR 3679:  Invalid input syntax for time: "1"
```

Returning all cast failures as NULL

To explicitly cast an expression to a requested data type, use the following construct:

```
SELECT expression::data-type
```

Using this command to cast any values to a conflicting data type returns the following error:

```
=> SELECT 'one'::time;
ERROR 3679:  Invalid input syntax for time: "one"
```

Vertica also supports the use of the coercion operator ::! , which is useful when you want to return:

- NULL instead of an error for any non-date/time data types
- NULL instead of an error after setting EnableStrictTimeCasts

Returning all cast failures as NULL allows those expressions that succeed during the cast to appear in the result. Those expressions that fail during the cast, however, have a NULL value.

The following example queries mytable using the coercion operator ::! . The query returns NULL where column a contains the string one , and returns 1 where the column contains 1 . Query results are identical no matter how EnableStrictTimeCasts is set:

```
=> SELECT current_value FROM configuration_parameters WHERE parameter_name ilike '%EnableStrictTimeCasts%';
 current_value
---------------
 0
(1 row)

=> SELECT a::!int FROM mytable;
a
---

1
(2 rows)

ALTER SESSION SET EnableStrictTimeCasts=1;
ALTER SESSION
=> SELECT a::!int FROM mytable;
 a
---

 1
(2 rows)
```

You can use ::! for casts of arrays and sets. The cast resolves each element individually, producing NULL for elements that cannot be cast.

Note that this functionality only applies to table data. It does not work on inline constant casts and in expressions automatically reduced to such. For example,

```
SELECT constant ::! FLOAT from (select 'some string' as constant) a;
```

results in ERROR 2826: Could not convert "some string" to a float8 . However, the following returns cast failures as NULL as described:

```
SELECT string_field ::! float FROM (SELECT 'a string' as string_field UNION ALL SELECT 'another string' ) a;
```

Date/time operators

Vertica supports usage of arithmetic operators on DATE/TIME operands:

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)

Examples

The operators described below that take TIME or TIMESTAMP input have two variants:

- Operators that take TIME WITH TIME ZONE or TIMESTAMP WITH TIME ZONE input.

- Operators that take TIME WITHOUT TIME ZONE or TIMESTAMP WITHOUT TIME ZONE input.

For brevity, these variants are not shown separately.

The + and * operators come in commutative pairs—for example, both DATE + INTEGER and INTEGER + DATE . Only one of each pair is shown.

Example	Result Type	Result
DATE '2001-09-28' + INTEGER '7'	DATE	'2001-10-05'
DATE '2001-09-28' + INTERVAL '1 HOUR'	TIMESTAMP	'2001-09-28 01:00:00'
DATE '2001-09-28' + TIME '03:00'	TIMESTAMP	'2001-09-28 03:00:00'
INTERVAL '1 DAY' + INTERVAL '1 HOUR'	INTERVAL	'1 DAY 01:00:00'
TIMESTAMP '2001-09-28 01:00' + INTERVAL '23 HOURS'	TIMESTAMP	'2001-09-29 00:00:00'
TIME '01:00' + INTERVAL '3 HOURS'	TIME	'04:00:00'
- INTERVAL '23 HOURS'	INTERVAL	'-23:00:00'
DATE '2001-10-01' – DATE '2001-09-28'	INTEGER	'3'
DATE '2001-10-01' – INTEGER '7'	DATE	'2001-09-24'
DATE '2001-09-28' – INTERVAL '1 HOUR'	TIMESTAMP	'2001-09-27 23:00:00'
TIME '05:00' – TIME '03:00'	INTERVAL	'02:00:00'
TIME '05:00' INTERVAL '2 HOURS'	TIME	'03:00:00'
TIMESTAMP '2001-09-28 23:00' – INTERVAL '23 HOURS'	TIMESTAMP	'2001-09-28 00:00:00'
INTERVAL '1 DAY' – INTERVAL '1 HOUR'	INTERVAL	'1 DAY -01:00:00'
TIMESTAMP '2001-09-29 03:00' – TIMESTAMP '2001-09-27 12:00'	INTERVAL	'1 DAY 15:00:00'
900 * INTERVAL '1 SECOND'	INTERVAL	'00:15:00'
21 * INTERVAL '1 DAY'	INTERVAL	'21 DAYS'

`DOUBLE PRECISION '3.5'` `* INTERVAL '1 HOUR'`	INTERVAL	'03:30:00'
`INTERVAL '1 HOUR' /` `DOUBLE PRECISION '1.5'`	INTERVAL	'00:40:00'

Mathematical operators

Mathematical operators are provided for many data types.

Operator	Description	Example	Result
!	Factorial	5 !	120
+	Addition	2 + 3	5
–	Subtraction	2 – 3	–1
*	Multiplication	2 * 3	6
/	Division (integer division produces NUMERIC results).	4 / 2	2.00...
//	With integer division, returns an INTEGER rather than a NUMERIC.	117.32 // 2.5	46
%	Modulo (remainder). For details, see MOD .	5 % 4	1
^	Exponentiation	2.0 ^ 3.0	8
|/	Square root	|/ 25.0	5
||/	Cube root	||/ 27.0	3
!!	Factorial (prefix operator)	!! 5	120
@	Absolute value	@ -5.0	5

Factorial operator support

Vertica supports use of factorial operators on positive and negative floating point (DOUBLE PRECISION) numbers and integers. For example:

```
=> SELECT 4.98!;
   ?column?
------------------
 115.978600750905
(1 row)
```

Factorial is defined in terms of the gamma function, where (-1) = Infinity and the other negative integers are undefined. For example:

```
(–4)! = NaN
–(4!) = –24
```

Factorial is defined as follows for all complex numbers *z* :

```
z! = gamma(z+1)
```

For details, see Abramowitz and Stegun: Handbook of Mathematical Functions .

NULL operators

To check whether a value is or is not NULL, use the following equivalent constructs:

Standard:

```
[expression IS NULL | expression IS NOT NULL]
```

Non-standard:

```
[expression ISNULL | expression NOTNULL]
```

Do not write *expression* = NULL : NULL represents an unknown value, and two unknown values are not necessarily equal. This behavior conforms to the SQL standard.

> **Note**
> Some applications might expect that *expression* = NULL returns true if *expression* evaluates to NULL. In this case, modify the application to comply with the SQL standard.

String concatenation operators

To concatenate two strings on a single line, use the concatenation operator (two consecutive vertical bars).

Syntax

```
string || string
```

Parameters

string	Expression of type CHAR or VARCHAR

Notes

- || is used to concatenate expressions and constants. The expressions are cast to VARCHAR if possible, otherwise to VARBINARY , and must both be one or the other.
- Two consecutive strings within a single SQL statement on separate lines are automatically concatenated

Examples

The following example is a single string written on two lines:

```
=> SELECT E'xx'-> '\\';
 ?column?
----------
 xx\
(1 row)
```

The following examples show two strings concatenated:

```
=> SELECT E'xx' ||-> '\\';
 ?column?
----------
 xx\\
(1 row)

=> SELECT 'auto' || 'mobile';
 ?column?
----------
 automobile
(1 row)

=> SELECT 'auto'-> 'mobile';
 ?column?
----------
 automobile
(1 row)

=> SELECT 1 || 2;
 ?column?
----------
 12
(1 row)


=> SELECT '1' || '2';
 ?column?
----------
 12
(1 row)
=> SELECT '1'-> '2';
 ?column?
----------
 12
(1 row)
```

Expressions

SQL expressions are the components of a query that compare a value or values against other values. They can also perform calculations. An expression found inside a SQL statement is usually in the form of a conditional statement.

Some functions also use Lambda functions .

Operator precedence

The following table shows operator precedence in decreasing (high to low) order.

When an expression includes more than one operator, specify the order of operation using parentheses, rather than relying on operator precedence.

Operator/Element	Associativity	Description
.	left	table/column name separator
::	left	typecast
[ ]	left	array element selection
-	right	unary minus
^	left	exponentiation

* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
IS		IS TRUE, IS FALSE, IS UNKNOWN, IS NULL
IN		set membership
BETWEEN		range containment
OVERLAPS		time interval overlap
LIKE		string pattern matching
< >		less than, greater than
=	right	equality, assignment
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

Expression evaluation rules

The order of evaluation of subexpressions is not defined. In particular, the inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order. To force evaluation in a specific order, use a CASE construct. For example, this is an untrustworthy way of trying to avoid division by zero in a WHERE clause:

```
=> SELECT x, y WHERE x <> 0 AND y/x > 1.5; --- unsafe
```

But this is safe:

```
=> SELECT x, y
  WHERE
    CASE
      WHEN x <> 0 THEN y/x > 1.5
      ELSE false
    END;
```

A CASE construct used in this fashion defeats optimization attempts, so use it only when necessary. (In this particular example, it would be best to avoid the issue by writing y > 1.5*x instead.)

Limits to SQL expressions

Expressions are limited by the available stack. Vertica requires at least 100KB of free stack. If this limit is exceeded then the error "The query contains an expression that is too complex to analyze" might be thrown. Adding physical memory and/or increasing the value of ulimit -s can increase the available stack and prevent the error.

Analytic expressions have a maximum recursion depth of 2000. If this limit is exceeded then the error "The query contains an expression that is too complex to analyze" might be thrown. This limit cannot be increased.

In this section

- Aggregate expressions
- CASE expressions
- Column references
- Comments
- Date/time expressions
- NULL value

Aggregate expressions

An aggregate expression applies an aggregate function across the rows or groups of rows selected by a query.

An aggregate expression only can appear in the select list or HAVING clause of a SELECT statement. It is invalid in other clauses such as WHERE , because those clauses are evaluated before the results of aggregates are formed.

Syntax

An aggregate expression has the following format:

```
aggregate-function ( [ * ] [ ALL | DISTINCT ] expression )
```

Parameters

aggregate-function	A Vertica function that aggregates data over groups of rows from a query result set.
ALL | DISTINCT	Specifies which input rows to process: • ALL (default): Invokes *aggregate-function* across all input rows where *expression* evaluates to a non-null value. • DISTINCT : Invokes *aggregate-function* across all input rows where *expression* evaluates to a unique non-null value.
expression	A value expression that does not itself contain an aggregate expression.

Examples

The AVG aggregate function returns the average income from the customer_dimension table:

```
=> SELECT AVG(annual_income) FROM customer_dimension;
 AVG
--------------
 2104270.6485
(1 row)
```

The following example shows how to use the COUNT aggregate function with the DISTINCT keyword to return all distinct values of evaluating the expression x+y for all inventory_fact records.

```
=> SELECT COUNT (DISTINCT date_key + product_key) FROM inventory_fact;
COUNT
-------
21560
(1 row)
```

CASE expressions

The CASE expression is a generic conditional expression that can be used wherever an expression is valid. It is similar to CASE and IF/THEN/ELSE statements in other languages.

Syntax (form 1)

```
CASE
  WHEN condition THEN result
  [ WHEN condition THEN result ]
  ...
  [ ELSE result ]
END
```

Parameters

* condition	An expression that returns a Boolean (true/false) result. If the result is false, subsequent WHEN clauses are evaluated in the same way.
* result	Specifies the value to return when the associated *condition* is true.
ELSE *result*	If no *condition* is true then the value of the CASE expression is the result in the ELSE clause. If the ELSE clause is omitted and no condition matches, the result is NULL.

Syntax (form 2)

```
CASE expression
 WHEN value THEN result
 [ WHEN value THEN result ]
 ...
 [ ELSE result ]
END
```

Parameters

* expression	An expression that is evaluated and compared to all the value specifications in WHEN clauses until one is found that is equal.
* value	Specifies a value to compare to the expression .
* result	Specifies the value to return when the expression is equal to the specified value .
ELSE *result*	Specifies the value to return when the expression is not equal to any value ; if no ELSE clause is specified, the value returned is null.

Notes

The data types of all result expressions must be convertible to a single output type.

Examples

The following examples show two uses of the CASE statement.

```
=> SELECT * FROM test;
 a
---
 1
 2
 3
=> SELECT a,
    CASE WHEN a=1 THEN 'one'
         WHEN a=2 THEN 'two'
         ELSE 'other'
    END
  FROM test;
 a | case
---+-------
 1 | one
 2 | two
 3 | other
=> SELECT a,
    CASE a WHEN 1 THEN 'one'
           WHEN 2 THEN 'two'
           ELSE 'other'
    END
  FROM test;
 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

Special example

A CASE expression does not evaluate subexpressions that are not needed to determine the result. You can use this behavior to avoid division-by-zero errors:

```
=> SELECT x FROM T1 WHERE
    CASE WHEN x <> 0 THEN y/x > 1.5
    ELSE false
   END;
```

Column references

Syntax

```
[[[database.]schema.]table-name.]column-name
```

Parameters

schema	Database and schema . The default schema is public . If you specify a database, it must be the current database.
table-name	One of the following: • Name of a table • Table alias defined in the query's FROM clause
column-name	A column name that is unique among all queried tables.

Restrictions

A column reference cannot contain any spaces.

Comments

A comment is an arbitrary sequence of characters beginning with two consecutive hyphen characters and extending to the end of the line. For example:

```
-- This is a standard SQL comment
```

A comment is removed from the input stream before further syntax analysis and is effectively replaced by white space.

Alternatively, C-style block comments can be used where the comment begins with /* and extends to the matching occurrence of */ .

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

These block comments nest, as specified in the SQL standard. Unlike C, you can comment out larger blocks of code that might contain existing block comments.

Date/time expressions

Vertica uses an internal heuristic parser for all date/time input support. Dates and times are input as strings, and are broken up into distinct fields with a preliminary determination of what kind of information might be in the field. Each field is interpreted and either assigned a numeric value, ignored, or rejected. The parser contains internal lookup tables for all textual fields, including months, days of the week, and time zones.

Vertica parses date/time type inputs as follows:

1. Break the input string into tokens and categorize each token as a string, time, time zone, or number.
2. Numeric token contains:
   - colon (:) — Parse as a time string, include all subsequent digits and colons.
   - dash (-), slash (/), or two or more dots (.) — Parse as a date string which might have a text month.
   - Numeric only — Parse as a single field or an ISO 8601 concatenated date (19990113 for January 13, 1999) or time (141516 for 14:15:16).
3. Token starts with a plus (+) or minus (–): Parse as a time zone or a special field.
4. Token is a text string: match up with possible strings.
   - Perform a binary-search table lookup for the token as either a special string (for example, today), day (for example, Thursday), month (for example, January), or noise word (for example, at, on).
   - Set field values and bit mask for fields. For example, set year, month, day for today, and additionally hour, minute, second for now.
   - If not found, do a similar binary-search table lookup to match the token with a time zone.
   - If still not found, throw an error.
5. Token is a number or number field:

- If eight or six digits, and if no other date fields were previously read, interpret as a "concatenated date" (19990118 or 990118). The interpretation is YYYYMMDD or YYMMDD .
- If token is three digits and a year was already read, interpret as day of year.
- If four or six digits and a year was already read, interpret as a time (HHMM or HHMMSS).
- If three or more digits and no date fields were found yet, interpret as a year (this forces yy-mm-dd ordering of the remaining date fields).
- Otherwise the date field ordering is assumed to follow the DateStyle setting: mm-dd-yy, dd-mm-yy, or yy-mm-dd. Throw an error if a month or day field is found to be out of range.

6. If BC is specified: negate the year and add one for internal storage. (In the Vertica implementation, 1 BC = year zero.)
7. If BC is not specified, and year field is two digits in length: adjust the year to four digits. If field is less than 70, add 2000, otherwise add 1900.

> **Tip**
> Gregorian years AD 1–99 can be entered as 4 digits with leading zeros— for example, 0099 = AD 99.

Month day year ordering

For some formats, ordering of month, day, and year in date input is ambiguous and there is support for specifying the expected ordering of these fields.

Special date/time values

Vertica supports several special date/time values for convenience, as shown below. All of these values need to be written in single quotes when used as constants in SQL statements.

The values INFINITY and -INFINITY are specially represented inside the system and are displayed the same way. The others are simply notational shorthands that are converted to ordinary date/time values when read. (In particular, NOW and related strings are converted to a specific time value as soon as they are read.)

String	Valid Data Types	Description
epoch	DATE , TIMESTAMP	1970-01-01 00:00:00+00 (UNIX SYSTEM TIME ZERO)
INFINITY	TIMESTAMP	Later than all other time stamps
-INFINITY	TIMESTAMP	Earlier than all other time stamps
NOW	DATE , TIME , TIMESTAMP	Current transaction's start time **Note:** NOW is not the same as the NOW function.
TODAY	DATE , TIMESTAMP	Midnight today
TOMORROW	DATE , TIMESTAMP	Midnight tomorrow
YESTERDAY	DATE , TIMESTAMP	Midnight yesterday
ALLBALLS	TIME	00:00:00.00 UTC

The following SQL-compatible functions can also be used to obtain the current time value for the corresponding data type:

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- LOCALTIME
- LOCALTIMESTAMP

The latter four accept an optional precision specification. (See Date/time functions .) However, these functions are SQL functions and are not recognized as data input strings.

NULL value

NULL is a reserved keyword used to indicate that a data value is unknown. It is the ASCII abbreviation for NULL characters (\0).

Usage in expressions

Vertica does not treat an empty string as a NULL value. An expression must specify NULL to indicate that a column value is unknown.

The following considerations apply to using NULL in expressions:

- NULL is not greater than, less than, equal to, or not equal to any other expression. Use the Boolean to determine whether an expression value is NULL.
- You can write queries with expressions that contain the <=> operator for NULL=NULL joins. See Equi-joins and non equi-joins .
- Vertica accepts NULL characters ('\0') in constant strings and does not remove null characters from VARCHAR fields on input or output.

Projection ordering of NULL data

Vertica sorts NULL values in projection columns as follows:

Column data type	NULL values placed at...
NUMERIC INTEGER DATE TIME TIMESTAMP INTERVAL	Beginning of sorted column (NULLS FIRST)
FLOAT STRING BOOLEAN	End of sorted column (NULLS LAST)

See also

NULL-handling functions

Lambda functions

Some SQL functions have arguments that are lambda functions. A lambda function is an unnamed inline function that is evaluated by the containing SQL function and returns a value.

Syntax

Lambda with one argument:

```
argument -> expression
```

Lambda with more than one argument:

```
(argument, ...) -> expression
```

Arguments

argument	Name to use for an input value for the expression. The name cannot be a reserved keyword, the name of an argument to a parent or nested lambda, or a column name or alias.
expression	Expression that uses the input arguments and returns a result to the containing SQL function. See the documentation of individual SQL functions for restrictions on return values. For example, some functions require a Boolean result.

Examples

The ARRAY_FIND function returns the first index that matches the element being searched for. Instead of a literal element, you can write a lambda function that returns a Boolean. The lambda function is applied to each element in the array until a match is found or all elements have been tested. In the following example, each person in the table has an array of email addresses, and the function locates fake addresses:

```
=> CREATE TABLE people (id INT, name VARCHAR, email ARRAY[VARCHAR,5]);

=> SELECT name, ARRAY_FIND(email, e -> REGEXP_LIKE(e,'example.com','i'))
  AS 'example.com'
  FROM people;
    name      | example.com
--------------+-------------
 Alice Adams  |           1
 Bob Adams    |           1
 Carol Collins |          0
 Dave Jones   |           0
(4 rows)
```

The argument e represents the individual element, and the body of the lambda expression is the regular-expression comparison. The input table has

four rows; in each row, the lambda function is called once per array element.

In the following example, a schedules table includes an array of events, where each event is a ROW with several fields:

```
=> CREATE TABLE schedules
    (guest VARCHAR,
    events ARRAY[ROW(e_date DATE, e_name VARCHAR, price NUMERIC(8,2))]);
```

You can use the CONTAINS function with a lambda expression to find people who have more than one event on the same day. The second argument, idx , is the index of the current element:

```
=> SELECT guest FROM schedules
WHERE CONTAINS(events, (e, idx) ->
                (idx < ARRAY_LENGTH(events) - 1)
                AND (e.e_date = events[idx + 1].e_date));
  guest
-------------
 Alice Adams
(1 row)
```

Predicates

Predicates are truth-tests. If the predicate test is true, it returns a value. Each predicate is evaluated per row, so that when the predicate is part of an entire table SELECT statement, the statement can return multiple results.

Predicates consist of a set of parameters and arguments. For example, in the following WHERE clause:

```
WHERE name = 'Smith'
```

- name = 'Smith' is the predicate
- 'Smith' is an expression

In this section

- ANY and ALL
- BETWEEN
- Boolean
- EXISTS
- IN
- INTERPOLATE
- LIKE
- NULL

ANY and ALL

ANY and ALL are logical operators that let you make comparisons on subqueries that return one or more rows. Both operators must be preceded by a comparison operator and followed by a subquery:

```
expression comparison-operator { ANY | ALL } (subquery)
```

- ANY returns true if the comparison between *expression* and any value returned by *subquery* evaluates to true.
- ALL returns true only if the comparison between *expression* and all values returned by *subquery* evaluates to true.

Equivalent operators

You can use the following operators instead of ANY or ALL:

This operator...	**Is equivalent to:**
SOME	ANY
IN	= ANY
NOT IN	<> ALL

NULL handling

Vertica supports multicolumn <> ALL subqueries where the columns are not marked NOT NULL . If any column contains a NULL value, Vertica returns a run-time error.

Vertica does not support ANY subqueries that are nested in another expression if any column values are NULL.

Examples

Examples below use the following tables and data:

```
=> SELECT * FROM t1 ORDER BY c1;
 c1 | c2
----+-----
  1 | cab
  1 | abc
  2 | fed
  2 | def
  3 | ihg
  3 | ghi
  4 | jkl
  5 | mno
(8 rows)
```

```
=> SELECT * FROM t2 ORDER BY c1;
 c1 | c2
----+-----
  1 | abc
  2 | fed
  3 | jkl
  3 | stu
  3 | zzz
(5 rows)
```

ANY subqueries

Subqueries that use the ANY keyword return true when any value retrieved in the subquery matches the value of the left-hand expression.

ANY subquery within an expression:

```
=> SELECT c1, c2 FROM t1 WHERE COALESCE((t1.c1 > ANY (SELECT c1 FROM t2)));
 c1 | c2
----+-----
  2 | fed
  2 | def
  3 | ihg
  3 | ghi
  4 | jkl
  5 | mno
(6 rows)
```

ANY noncorrelated subqueries without aggregates:

```
=> SELECT c1 FROM t1 WHERE c1 = ANY (SELECT c1 FROM t2) ORDER BY c1;
 c1
----
  1
  1
  2
  2
  3
  3
(6 rows)
```

ANY noncorrelated subqueries with aggregates:

```
=> SELECT c1, c2 FROM t1 WHERE c1 <> ANY (SELECT MAX(c1) FROM t2) ORDER BY c1;
 c1 | c2
----+-----
  1 | cab
  1 | abc
  2 | fed
  2 | def
  4 | jkl
  5 | mno
(6 rows)

=> SELECT c1 FROM t1 GROUP BY c1 HAVING c1 <> ANY (SELECT MAX(c1) FROM t2) ORDER BY c1;
 c1
----
  1
  2
  4
  5
(4 rows)
```

ANY noncorrelated subqueries with aggregates and a GROUP BY clause:

```
=> SELECT c1, c2 FROM t1 WHERE c1 <> ANY (SELECT MAX(c1) FROM t2 GROUP BY c2) ORDER BY c1;
 c1 | c2
----+-----
  1 | cab
  1 | abc
  2 | fed
  2 | def
  3 | ihg
  3 | ghi
  4 | jkl
  5 | mno
(8 rows)
```

ANY noncorrelated subqueries with a GROUP BY clause:

```
=> SELECT c1, c2 FROM t1 WHERE c1 <=> ANY (SELECT c1 FROM t2 GROUP BY c1) ORDER BY c1;
 c1 | c2
----+-----
  1 | cab
  1 | abc
  2 | fed
  2 | def
  3 | ihg
  3 | ghi
(6 rows)
```

ANY correlated subqueries with no aggregates or GROUP BY clause:

```
=> SELECT c1, c2 FROM t1 WHERE c1 >= ANY (SELECT c1 FROM t2 WHERE t2.c2 = t1.c2) ORDER BY c1;
 c1 | c2
----+-----
  1 | abc
  2 | fed
  4 | jkl
(3 rows)
```

ALL subqueries

A subquery that uses the ALL keyword returns true when all values retrieved by the subquery match the left-hand expression, otherwise it returns false.

ALL noncorrelated subqueries without aggregates:

```
=> SELECT c1, c2 FROM t1 WHERE c1 >= ALL (SELECT c1 FROM t2) ORDER BY c1;
 c1 | c2
----+-----
  3 | ihg
  3 | ghi
  4 | jkl
  5 | mno
(4 rows)
```

ALL noncorrelated subqueries with aggregates:

```
=> SELECT c1, c2 FROM t1 WHERE c1 = ALL (SELECT MAX(c1) FROM t2) ORDER BY c1;
 c1 | c2
----+-----
  3 | ihg
  3 | ghi
(2 rows)

=> SELECT c1 FROM t1 GROUP BY c1 HAVING c1 <> ALL (SELECT MAX(c1) FROM t2) ORDER BY c1;
 c1
----
  1
  2
  4
  5
(4 rows)
```

ALL noncorrelated subqueries with aggregates and a GROUP BY clause:

```
=> SELECT c1, c2 FROM t1 WHERE c1 <= ALL (SELECT MAX(c1) FROM t2 GROUP BY c2) ORDER BY c1;
 c1 | c2
----+-----
  1 | cab
  1 | abc
(2 rows)
```

ALL noncorrelated subqueries with a GROUP BY clause:

```
=> SELECT c1, c2 FROM t1 WHERE c1 <> ALL (SELECT c1 FROM t2 GROUP BY c1) ORDER BY c1;
 c1 | c2
----+-----
  4 | jkl
  5 | mno
(2 rows)
```

BETWEEN

Checks whether an expression is within the range of two other expressions, inclusive. All expressions must be of the same or compatible data types.

Syntax

```
WHERE a BETWEEN x AND y
```

Equivalent predicates

The following BETWEEN predicates can be rewritten in conventional SQL with logical operators AND and OR.

This BETWEEN predicate...	**Is equivalent to...**
WHERE *a* BETWEEN *x* AND *y*	WHERE *a* >= *x* AND *a* <= *y*

WHERE *a* NOT BETWEEN *x* AND *y*	WHERE *a* < *x* OR *a* > *y*

Examples

The BETWEEN predicate can be especially useful for querying date ranges, as shown in the following examples:

```
=> SELECT NOW()::DATE;
   NOW
------------
 2022-12-15
(1 row)

=> CREATE TABLE t1 (a INT, b varchar(12), c DATE);
CREATE TABLE
=> INSERT INTO t1 VALUES
   (0,'today',NOW()),
   (1,'today+1',NOW()+1),
   (2,'today+2',NOW()+2),
   (3,'today+3',NOW()+3),
   (4,'today+4',NOW()+4),
   (5,'today+5',NOW()+5),
   (6,'today+6',NOW()+6);
 OUTPUT
--------
      7
(1 row)

=> COMMIT;
COMMIT
=> SELECT * FROM t1;
 a |    b    |     c
---+---------+------------
 0 | today   | 2022-12-15
 1 | today+1 | 2022-12-16
 2 | today+2 | 2022-12-17
 3 | today+3 | 2022-12-18
 4 | today+4 | 2022-12-19
 5 | today+5 | 2022-12-20
 6 | today+6 | 2022-12-21
(7 rows)

=> SELECT * FROM t1 WHERE c BETWEEN '2022-12-17' AND '2022-12-20';
 a |    b    |     c
---+---------+------------
 2 | today+2 | 2022-12-17
 3 | today+3 | 2022-12-18
 4 | today+4 | 2022-12-19
 5 | today+5 | 2022-12-20
(4 rows)
```

Use the NOW and INTERVAL keywords to query a date range:

```
=> SELECT * FROM t1 WHERE c BETWEEN NOW()::DATE AND NOW()::DATE + INTERVAL '2 days';
 a |    b    |     c
---+---------+------------
 0 | today   | 2022-12-15
 1 | today+1 | 2022-12-16
 2 | today+2 | 2022-12-17
(3 rows)
```

Boolean

Retrieves rows where the value of an expression is true, false, or unknown (NULL).

Syntax

```
expression IS [NOT] TRUE
expression IS [NOT] FALSE
expression IS [NOT] UNKNOWN
```

Notes

- NULL input is treated as the value UNKNOWN .
- IS UNKNOWN and IS NOT UNKNOWN are effectively the same as the NULL predicate , except that the input expression does not have to be a single column value. To check a single column value for NULL, use the NULL predicate.
- Do not confuse the Boolean predicate with Boolean operators or the Boolean data type, which can have only two values: true and false.

EXISTS

EXISTS and NOT EXISTS predicates compare an expression against a subquery:

- EXISTS returns true if the subquery returns one or more rows.
- NOT EXISTS returns true if the subquery returns no rows.

Syntax

```
expression [ NOT ] EXISTS ( subquery )
```

Usage

EXISTS results only depend on whether any or no records are returned, and not on the contents of those records. Because the subquery output is usually of no interest, EXISTS tests are commonly written in one of the following ways:

```
EXISTS (SELECT 1 WHERE...)
EXISTS (SELECT * WHERE...)
```

In the first case, the subquery returns 1 for every record found by the subquery. For example, the following query retrieves a list of all customers whose store purchases were greater than 550 dollars:

```
=> SELECT customer_key, customer_name, customer_state
   FROM public.customer_dimension WHERE EXISTS
     (SELECT 1 FROM store.store_sales_fact
      WHERE customer_key = public.customer_dimension.customer_key
      AND sales_dollar_amount > 550)
   AND customer_state = 'MA' ORDER BY customer_key;
 customer_key |     customer_name      | customer_state
--------------+------------------------+----------------
            2 | Anna G. Li             | CA
            4 | Daniel I. Fortin       | TX
            7 | David H. Greenwood     | MA
            8 | Wendy S. Young         | IL
            9 | Theodore X. Brown      | MA
      ...
        49902 | Amy Q. Pavlov          | MA
        49922 | Doug C. Carcetti       | MA
        49930 | Theodore G. McNulty    | MA
        49979 | Ben Z. Miller          | MA
(1058 rows)
```

EXISTS versus IN

Whether you use EXISTS or IN subqueries depends on which predicates you select in outer and inner query blocks. For example, the following query gets a list of all the orders placed by all stores on January 2, 2007 for vendors with records in the vendor table:

```
=> SELECT store_key, order_number, date_ordered
   FROM store.store_orders_fact WHERE EXISTS
     (SELECT 1 FROM public.vendor_dimension vd JOIN store.store_orders_fact ord ON vd.vendor_key = ord.vendor_key)
   AND date_ordered = '2007-01-02';
 store_key | order_number | date_ordered
-----------+--------------+--------------
       114 |       271071 | 2007-01-02
        19 |       290888 | 2007-01-02
       132 |        58942 | 2007-01-02
       232 |         9286 | 2007-01-02
       126 |       224474 | 2007-01-02
       196 |        63482 | 2007-01-02
  ...
       196 |        83327 | 2007-01-02
       138 |       278373 | 2007-01-02
       179 |       293586 | 2007-01-02
       155 |       213413 | 2007-01-02
(506 rows)
```

The above query looks for existence of the vendor and date ordered. To return a particular value, rather than simple existence, the query looks for orders placed by the vendor who got the best deal on January 2, 2007:

```
=> SELECT store_key, order_number, date_ordered, vendor_name
   FROM store.store_orders_fact ord JOIN public.vendor_dimension vd ON ord.vendor_key = vd.vendor_key
   WHERE vd.deal_size IN (SELECT MAX(deal_size) FROM public.vendor_dimension) AND date_ordered = '2007-01-02';
 store_key | order_number | date_ordered |     vendor_name
-----------+--------------+--------------+---------------------
        50 |        99234 | 2007-01-02   | Everything Wholesale
        81 |       200802 | 2007-01-02   | Everything Wholesale
       115 |        13793 | 2007-01-02   | Everything Wholesale
       204 |        41842 | 2007-01-02   | Everything Wholesale
       133 |       169025 | 2007-01-02   | Everything Wholesale
       163 |       208580 | 2007-01-02   | Everything Wholesale
        29 |       154972 | 2007-01-02   | Everything Wholesale
       145 |       236790 | 2007-01-02   | Everything Wholesale
       249 |        54838 | 2007-01-02   | Everything Wholesale
         7 |       161536 | 2007-01-02   | Everything Wholesale
(10 rows)
```

See also

IN

IN

Checks whether a single value is found (or not found) within a set of values.

Syntax

```
(column-list) [ NOT ] IN ( values-list )
```

Parameters

column-list	One or more comma-delimited columns in the queried tables.
values-list	Comma-delimited list of constant values to find in the *column-list* columns. Each *values-list* value maps to a *column-list* column according to their order in *values-list* and *column-list* , respectively. Column/value pairs must have compatible data types. You can specify multiple sets of values as follows: ((*values-list*), (*values-list*)[,...])

Null handling

Vertica supports multicolumn NOT IN subqueries where the columns are not marked NOT NULL . If one of the columns is found to contain a NULL value during query execution, Vertica returns a run-time error.

Similarly, IN subqueries nested within another expression are not supported if any column values are NULL. For example, if in the following statement column x from either table contains a NULL value, Vertica returns a run-time error:

```
=> SELECT * FROM t1 WHERE (x IN (SELECT x FROM t2)) IS FALSE;
   ERROR: NULL value found in a column used by a subquery
```

EXISTS versus IN

Whether you use EXISTS or IN subqueries depends on which predicates you select in outer and inner query blocks. For example, the following query gets a list of all the orders placed by all stores on January 2, 2007 for vendors with records in the vendor table:

```
=> SELECT store_key, order_number, date_ordered
   FROM store.store_orders_fact WHERE EXISTS
     (SELECT 1 FROM public.vendor_dimension vd JOIN store.store_orders_fact ord ON vd.vendor_key = ord.vendor_key)
   AND date_ordered = '2007-01-02';
 store_key | order_number | date_ordered
-----------+--------------+--------------
       114 |       271071 | 2007-01-02
        19 |       290888 | 2007-01-02
       132 |        58942 | 2007-01-02
       232 |         9286 | 2007-01-02
       126 |       224474 | 2007-01-02
       196 |        63482 | 2007-01-02
  ...
       196 |        83327 | 2007-01-02
       138 |       278373 | 2007-01-02
       179 |       293586 | 2007-01-02
       155 |       213413 | 2007-01-02
(506 rows)
```

The above query looks for existence of the vendor and date ordered. To return a particular value, rather than simple existence, the query looks for orders placed by the vendor who got the best deal on January 2, 2007:

```
=> SELECT store_key, order_number, date_ordered, vendor_name
   FROM store.store_orders_fact ord JOIN public.vendor_dimension vd ON ord.vendor_key = vd.vendor_key
   WHERE vd.deal_size IN (SELECT MAX(deal_size) FROM public.vendor_dimension) AND date_ordered = '2007-01-02';
 store_key | order_number | date_ordered |     vendor_name
-----------+--------------+--------------+---------------------
        50 |        99234 | 2007-01-02   | Everything Wholesale
        81 |       200802 | 2007-01-02   | Everything Wholesale
       115 |        13793 | 2007-01-02   | Everything Wholesale
       204 |        41842 | 2007-01-02   | Everything Wholesale
       133 |       169025 | 2007-01-02   | Everything Wholesale
       163 |       208580 | 2007-01-02   | Everything Wholesale
        29 |       154972 | 2007-01-02   | Everything Wholesale
       145 |       236790 | 2007-01-02   | Everything Wholesale
       249 |        54838 | 2007-01-02   | Everything Wholesale
         7 |       161536 | 2007-01-02   | Everything Wholesale
(10 rows)
```

Examples

The following SELECT statement queries all data in table t11 .

```
=> SELECT * FROM t11 ORDER BY pk;
 pk | col1 | col2 | SKIP_ME_FLAG
----+------+------+--------------
  1 |    2 |    3 | t
  2 |    3 |    4 | t
  3 |    4 |    5 | f
  4 |    5 |    6 | f
  5 |    6 |    7 | t
  6 |      |    8 | f
  7 |    8 |      | t
(7 rows)
```

The following query specifies an IN predicate, to find all rows in t11 where columns col1 and col2 contain values of (2,3) or (6,7) :

```
=> SELECT * FROM t11 WHERE (col1, col2) IN ((2,3), (6,7)) ORDER BY pk;
 pk | col1 | col2 | SKIP_ME_FLAG
----+------+------+--------------
  1 |    2 |    3 | t
  5 |    6 |    7 | t
(2 rows)
```

The following query uses the VMart schema to illustrate the use of outer expressions referring to different inner expressions:

```
=> SELECT product_description, product_price FROM product_dimension
   WHERE (product_dimension.product_key, product_dimension.product_key) IN
     (SELECT store.store_orders_fact.order_number,
       store.store_orders_fact.quantity_ordered
      FROM store.store_orders_fact);
    product_description     | product_price
----------------------------+---------------
 Brand #73 wheechair        |           454
 Brand #72 box of candy     |           326
 Brand #71 vanilla ice cream |          270
(3 rows)
```

INTERPOLATE

Joins two event series using some ordered attribute. Event series joins let you compare values from two series directly, rather than having to normalize the series to the same measurement interval.

An event series join is an extension of a regular outer join . The difference between expressing a regular outer join and an event series join is the INTERPOLATE predicate, which is used in the ON clause (see Examples below). Instead of padding the non-preserved side with null values when there is no match, the event series join pads the non-preserved side with the previous/next values from the table.

Interpolated values come from the table that contains the null, not from the other table.Vertica does not guarantee that the output contains no null values. If there is no previous/next value for a mismatched row, that row is padded with nulls.

Syntax

```
expression1 INTERPOLATE { PREVIOUS | NEXT } VALUE expression2
```

Arguments

expression1 , *expression2*	A column reference from one of the tables specified in the FROM clause . The columns can be of any data type. Because event series are time-based, the type is typically DATE/TIME or TIMESTAMP .

{ PREVIOUS | NEXT } VALUE	Pads the non-preserved side with the previous/next values when there is no match. If previous is called on the first row (or next on the last row), will pad with null values. Input rows are sorted in ascending logical order of the join column. **Note** An ORDER BY clause, if used, does not determine the input order but only determines query output order.

Notes

- Data is logically partitioned on the table in which it resides, based on other ON clause equality predicates.
- Event series join requires that the joined tables are both sorted on columns in equality predicates, in any order, followed by the INTERPOLATED column. If data is already sorted in this order, then an explicit sort is avoided, which can improve query performance. For example, given the following tables:

```
ask: exchange, stock, ts, pricebid: exchange,
stock, ts, price
```

  In the query that follows:
  - ask is sorted on exchange, stock (or the reverse), ts
  - bid is sorted on exchange, stock (or the reverse), ts

```
SELECT ask.price - bid.price, ask.ts, ask.stock, ask.exchange
FROM ask FULL OUTER JOIN bid
  ON ask.stock = bid.stock AND ask.exchange =
  bid.exchange AND ask.ts INTERPOLATE PREVIOUS
  VALUE bid.ts;
```

Restrictions

- Only one INTERPOLATE expression is allowed per join.
- INTERPOLATE expressions are used only with ANSI SQL-99 syntax (the ON clause), which is already true for full outer joins.
- INTERPOLATE can be used with equality predicates only.
- The AND operator is supported but not the OR and NOT operators.
- Expressions and implicit or explicit casts are not supported, but subqueries are allowed.

Semantics

When you write an event series join in place of normal join, values are evaluated as follows (using the schema in the examples below):

- t is the outer, preserved table.
- t1 is the inner, non-preserved table.
- For each row in outer table t , the ON clause predicates are evaluated for each combination of each row in the inner table t1 .
- If the ON clause predicates evaluate to true for any combination of rows, those combination rows are produced at the output.
- If the ON clause is false for all combinations, a single output row is produced with the values of the row from t along with the columns of t1 chosen from the row in t1 with the greatest t1.y value such that t1.y < t.x ; If no such row is found, pad with nulls.

> **Note**
> t LEFT OUTER JOIN t1 is equivalent to t1 RIGHT OUTER JOIN t .

In the case of a full outer join, all values from both tables are preserved.

Examples

The examples that follow use this simple schema.

```
CREATE TABLE t(x TIME);
CREATE TABLE t1(y TIME);
INSERT INTO t VALUES('12:40:23');
INSERT INTO t VALUES('13:40:25');
INSERT INTO t VALUES('13:45:00');
INSERT INTO t VALUES('14:49:55');
INSERT INTO t1 VALUES('12:40:23');
INSERT INTO t1 VALUES('14:00:00');
COMMIT;
```

Normal full outer join

```
=> SELECT * FROM t FULL OUTER JOIN t1 ON t.x = t1.y;
```

Notice the null rows from the non-preserved table:

```
    x     |    y
----------+----------
 12:40:23 | 12:40:23
 13:40:25 |
 13:45:00 |
 14:49:55 |
          | 14:00:00
(5 rows)
```

Full outer join with interpolation

```
=> SELECT * FROM t FULL OUTER JOIN t1 ON t.x INTERPOLATE PREVIOUS VALUE t1.y;
```

In this case, the rows with no entry point are padded with values from the previous row.

```
    x     |    y
----------+----------
 12:40:23 | 12:40:23
 13:40:25 | 12:40:23
 13:45:00 | 12:40:23
 14:49:55 | 12:40:23
 13:40:25 | 14:00:00
(5 rows)
```

Likewise, interpolate next is also supported:

```
=> SELECT * FROM t FULL OUTER JOIN t1 ON t.x INTERPOLATE NEXT VALUE t1.y;
```

In this case, the rows with no entry point are padded with values from the next row.

```
    x     |    y
----------+----------
 12:40:23 | 12:40:23
 13:40:25 | 14:00:00
 13:45:00 | 14:00:00
 14:49:55 |
 14:49:55 | 14:00:00
(5 rows)
```

Normal left outer join

```
=> SELECT * FROM t LEFT OUTER JOIN t1 ON t.x = t1.y;
```

Again, there are nulls in the non-preserved table

```
   x    |   y
----------+----------
 12:40:23 | 12:40:23
 13:40:25 |
 13:45:00 |
 14:49:55 |
(4 rows)
```

Left outer join with interpolation

```
=> SELECT * FROM t LEFT OUTER JOIN t1 ON t.x INTERPOLATE PREVIOUS VALUE t1.y;
```

Nulls have been padded with interpolated values.

```
   x    |   y
----------+----------
 12:40:23 | 12:40:23
 13:40:25 | 12:40:23
 13:45:00 | 12:40:23
 14:49:55 | 14:00:00
(4 rows)
```

Likewise, interpolate next is also supported:

```
=> SELECT * FROM t LEFT OUTER JOIN t1 ON t.x INTERPOLATE NEXT VALUE t1.y;
```

Nulls have been padded with interpolated values here as well.

```
   x    |   y
----------+----------
 12:40:23 | 12:40:23
 13:40:25 | 14:00:00
 13:45:00 | 14:00:00
 14:49:55 |
 (4 rows)
```

Inner joins

For inner joins, there is no difference between a regular inner join and an event series inner join. Since null values are eliminated from the result set, there is nothing to interpolate.

A regular inner join returns only the single matching row at 12:40:23:

```
=> SELECT * FROM t INNER JOIN t1 ON t.x = t1.y;
   x    |   y
----------+----------
 12:40:23 | 12:40:23
(1 row)
```

An event series inner join finds the same single-matching row at 12:40:23:

```
=> SELECT * FROM t INNER JOIN t1 ON t.x INTERPOLATE PREVIOUS VALUE t1.y;
   x    |   y
----------+----------
 12:40:23 | 12:40:23
(1 row)
```

See also

Event series joins

In this section

- Join predicate

Join predicate

Specifies the columns on which records from two or more tables are joined. You can connect multiple join predicates with logical operators AND , OR , and NOT .

Syntax

```
ON column-ref = column-ref [ {AND | OR | NOT } column-ref = column-ref ]...
```

Parameters

column-ref	Specifies a column in a queried table. For best performance, do not join on LONG VARBINARY and LONG VARCHAR columns.

See also

Joins

LIKE

Retrieves rows where a string expression—typically a column—matches the specified pattern or, if qualified by ANY or ALL, set of patterns. The pattern can contain one or more wildcard characters.

Syntax

```
string-expression [ NOT ] { LIKE | ILIKE | LIKEB | ILIKEB }
  { pattern | { ANY | ALL } ( pattern,... ) } [ ESCAPE 'char' ]
```

Arguments

string-expression	String expression, typically a column, to test for instances of the specified pattern or patterns.
NOT	Returns true if the LIKE predicate returns false and vice-versa, equivalent to: NOT *string-expression like-operator*
LIKE | ILIKE | LIKEB | ILIKEB	One of the following operators: • LIKE : Complies with the SQL standard, case-sensitive, operates on UTF-8 character strings, exact behavior depends on collation parameters such as strength. LIKE is stable for character strings, but immutable for binary strings • ILIKE : Same as LIKE but case-insensitive. • LIKEB : Performs case-sensitive byte-at-a-time ASCII comparisons, immutable for character and binary strings. • ILIKEB : Same as LIKEB but case-insensitive.
pattern	A pattern to test against *string-expression* : ' *pattern* ' | { ANY | ALL } (*pattern* ,...) ANY / ALL specify a comma-delimited list of patterns, where: • ANY returns true if any pattern matches, equivalent to logical OR. • ALL returns true only if all patterns match, equivalent to logical AND. Pattern strings can contain the following wildcard characters • _ (underscore): Match any single character. • % (percent): Match any string of zero or more characters.
ESCAPE *char*	Specifies an escape character, by default backslash (\ ), used to escape reserved characters: wildcard characters (underscore and percent), and the escape character itself. This option is enforced only for non-default collations; it is currently unsupported with ANY/ALL pattern matching. **Note** Backslash is not valid for binary data types character. To embed an escape character for binary data types, use ESCAPE to specify a valid binary character.

Substitute symbols

You can substitute the following symbols for LIKE and its variants:

> **Note**
> ESCAPE usage is not valid for these symbols.

Symbol	Eqivalent to:
~~	LIKE
~#	LIKEB
~~*	ILIKE
~#*	ILIKEB
!~~	NOT LIKE
!~#	NOT LIKEB
!~~*	NOT ILIKE
!~#*	NOT ILIKEB

Pattern matching

LIKE and its variants require that the entire string expression match the specified patterns. To match a sequence of characters anywhere within a string, the pattern must start and end with a percent sign.

LIKE does not ignore trailing white space characters. If the data values to match end with an indeterminate amount of white space, append the wildcard character % to *pattern* .

Locale dependencies

In the default locale, LIKE and ILIKE handle UTF-8 character-at-a-time, locale-insensitive comparisons. ILIKE handles language-independent case-folding.

In non-default locales, LIKE and ILIKE perform locale-sensitive string comparisons, including some automatic normalization, using the same algorithm as the = operator on VARCHAR types.

ESCAPE expressions evaluate to exactly one octet—or one UTF-8 character for non-default locales.

Examples

Basic pattern matching

The following query searches for customer names that start with Ever :

```
=> SELECT DISTINCT(customer_name) FROM customer_dimension WHERE customer_name LIKE 'Ever%';
 customer_name
---------------
 Evermedia
 Evergen
 Evercom
 Evershop
 Everdata
 Evercorp
 Everstar
 Everhope
 Evertech
 Evercore
 Evercare
(11 rows)
```

LIKE ANY/ALL usage

LIKE operators support the keywords ANY and ALL, which let you specify multiple patterns to test against a string expression. For example, the following query uses case-insensitive ILIKE to find all customer names that contain the strings media or ever :

```
=> SELECT DISTINCT (customer_name) FROM customer_dimension
    WHERE customer_name ILIKE ANY ('%media%','%ever%') ORDER BY customer_name;
 customer_name
---------------
 Amerimedia
 Bettermedia
 Evercare
 Evercom
 Evercore
 Evercorp
 Everdata
 Evergen
 Everhope
 Evermedia
 Evershop
 Everstar
 Evertech
 Foodmedia
 Goldmedia
 Infomedia
 Inimedia
 Intramedia
 Metamedia
 Verimedia
 Virtamedia
(21 rows)
```

LIKE ANY usage is equivalent to specifying multiple conditions that are combined with OR:

```
=> SELECT DISTINCT (customer_name) FROM customer_dimension
    WHERE customer_name ILIKE '%media%' OR customer_name ILIKE '%ever%' ORDER BY customer_name;
```

Similarly, LIKE ALL usage is equivalent to specifying multiple conditions that are combined with AND. For example, the following query uses case-insensitive ILIKE to find all customer names that contain the strings media and ever :

```
=> SELECT DISTINCT (customer_name) FROM customer_dimension
    WHERE customer_name ILIKE ALL ('%media%','%ever%') ORDER BY customer_name;
 customer_name
---------------
 Evermedia
(1 row)
```

Pattern matching in locales

The following example illustrates pattern matching in locales.

```

=> \locale default
INFO 2567:  Canonical locale: 'en_US'
Standard collation: 'LEN_KBINARY'
English (United States)
=> CREATE TABLE src(c1 VARCHAR(100));
=> INSERT INTO src VALUES (U&'\00DF'); --The sharp s (ß)
=> INSERT INTO src VALUES ('ss');
=> COMMIT;
```

Querying the src table in the default locale returns both ss and sharp s.

```
=> SELECT * FROM src;
 c1
----
 ß
 ss
(2 rows)
```

The following query combines pattern-matching predicates to return the results from column c1 :

```
=> SELECT c1, c1 = 'ss' AS equality, c1 LIKE 'ss'
   AS LIKE, c1 ILIKE 'ss' AS ILIKE FROM src;
 c1 | equality | LIKE | ILIKE
----+----------+------+-------
 ß  | f        | f    | f
 ss | t        | t    | t
(2 rows)
```

The next query specifies unicode format for c1 :

```
=> SELECT c1, c1 = U&'\00DF' AS equality,
   c1 LIKE U&'\00DF' AS LIKE,
   c1 ILIKE U&'\00DF' AS ILIKE from src;
 c1 | equality | LIKE | ILIKE
----+----------+------+-------
 ß  | t        | t    | t
 ss | f        | f    | f
(2 rows)
```

Now change the locale to German with a strength of 1 (ignore case and accents):

```

=> \locale LDE_S1
INFO 2567:  Canonical locale: 'de'
Standard collation: 'LDE_S1'
German  Deutsch
=> SELECT c1, c1 = 'ss' AS equality,
c1 LIKE 'ss' as LIKE, c1 ILIKE 'ss' AS ILIKE from src;
 c1 | equality | LIKE | ILIKE
----+----------+------+-------
 ß  | t        | t    | t
 ss | t        | t    | t
(2 rows)
```

This example illustrates binary data types with pattern-matching predicates:

```
=> CREATE TABLE t (c BINARY(1));
CREATE TABLE
=> INSERT INTO t VALUES (HEX_TO_BINARY('0x00')), (HEX_TO_BINARY('0xFF'));
 OUTPUT
--------
      2
(1 row)

=> COMMIT;
COMMIT
=> SELECT TO_HEX(c) FROM t;
 TO_HEX
--------
 00
 ff
(2 rows)

=> SELECT * FROM t;
  c
------
 \000
 \377
(2 rows)

=> SELECT c, c = '\000', c LIKE '\000', c ILIKE '\000' from t;
  c   | ?column? | ?column? | ?column?
------+----------+----------+----------
 \000 | t        | t        | t
 \377 | f        | f        | f
(2 rows)

=> SELECT c, c = '\377', c LIKE '\377', c ILIKE '\377' FROM t;
  c   | ?column? | ?column? | ?column?
------+----------+----------+----------
 \000 | f        | f        | f
 \377 | t        | t        | t
(2 rows)
```

NULL

Tests for null values.

Syntax

value-expression IS [NOT] NULL

Arguments

value-expression	Column name, literal, or function

Examples

Column name:

```
=> SELECT date_key FROM date_dimension WHERE date_key IS NOT NULL;
 date_key
----------
        1
      366
     1462
     1097
        2
        3
        6
        7
        8
...
```

Function:

```
=> SELECT MAX(household_id) IS NULL FROM customer_dimension;
 ?column?
----------
 f
(1 row)
```

Literal:

```
=> SELECT 'a' IS NOT NULL;
 ?column?
----------
 t
(1 row)
```

Hints

Hints are directives that you embed within a query or directed query . They conform to the following syntax:

```
/*+hint-name[, hint-name]...*/
```

Hints are bracketed by comment characters /*+ and */ , which can enclose multiple comma-delimited hints. For example:

```
SELECT /*+syntactic_join,verbatim*/
```

Restrictions

When embedding hints in a query, be aware of the following restrictions:

- Do not embed spaces in the comment characters /* and */ .
- In general, spaces are allowed before and after the plus (+) character and *hint-name* ; however, some third-party tools do not support spaces embedded inside /* +.

Supported hints

Vertica supports the following hints:

General hints

Hint	Description
ALLNODES	Qualifies an EXPLAIN statement to request a query plan that assumes all nodes are active.
EARLY_MATERIALIZATION	Specifies early materialization of a table for the current query.
ENABLE_WITH_CLAUSE_MATERIALIZATION	Enables and disables WITH clause materialization for a specific query.
LABEL	Labels a query so you can identify it for profiling and debugging.
SKIP_STATISTICS	Directs the optimizer to produce a query plan that incorporates only minimal statistics.

Eon Mode hints

Hint	Description
DEPOT_FETCH	Specifies whether a query fetches data to the depot from communal storage when the depot lacks data for this query.
ECSMODE	Specifies the elastic crunch scaling (ECS) strategy for dividing shard data among its subscribers.

Join hints

Hint	Description
SYNTACTIC_JOIN	Enforces join order and enables other join hints.
DISTRIB	Sets the input operations for a distributed join to broadcast, resegment, local, or filter.
GBYTYPE	Specifies which algorithm—GROUPBY HASH or GROUPBY PIPELINED—the Vertica query optimizer should use to implement a GROUP BY clause.
JTYPE	Enforces the join type: merge or hash join.
UTYPE	Specifies how to combine UNION ALL input.

Projection hints

Hint	Description
PROJS	Specifies one or more projections to use for a queried table.
SKIP_PROJS	Specifies which projections to avoid using for a queried table.

Directed query hints

The following hints are only supported by directed queries:

Hint	Description
:c	Marks a query constant that must be included in an input query; otherwise, that input query is disqualified from using the directed query.
:v	Maps an input query constant to one or more annotated query constants.
VERBATIM	Enforces execution of an annotated query exactly as written.

In this section

- :c
- :v
- ALLNODES
- DEPOT_FETCH
- DISTRIB
- EARLY_MATERIALIZATION
- ECSMODE
- ENABLE_WITH_CLAUSE_MATERIALIZATION
- GBYTYPE
- JFMT
- JTYPE
- LABEL
- PROJS
- SKIP_PROJS
- SKIP_STATISTICS
- SYNTACTIC_JOIN
- UTYPE

- VERBATIM

:c

In a directed query, marks a query constant that must be included in an input query; otherwise, that input query is disqualified from using the directed query.

Syntax

```
/*+:c*/
```

Usage

By default, optimizer-generated directed queries set ignore constant (:v) hints on predicate constants. You can override this behavior by setting the :c hint on input query constants that must not be ignored. For example, the following statement creates a directed query that can be used only for input queries where the join predicate constant is the same as in the original input query— 8 :

```
=> CREATE DIRECTED QUERY OPTIMIZER simpleJoin_KeepPredicateConstant SELECT * FROM S JOIN T ON S.a = T.b WHERE S.a = 8 /*+:c*/;
CREATE DIRECTED QUERY
=> ACTIVATE DIRECTED QUERY simpleJoin_KeepPredicateConstant;
```

See also

Conserving Predicate Constants in Directed Queries

:v

In a directed query, marks an input query constant that the optimizer ignores when it considers whether to use the directed query for a given query. Use this hint to create a directed query that can be used for multiple variants of an input query.

Vertica also supports IGNORECONST as an alias of :v . Optimizer-generated directed queries automatically mark predicate constants in input and annotated queries with :v hints.

For details, see Ignoring constants in directed queries .

Syntax

```
/*+:v(arg)*/
/*+IGNORECONST(arg)*/
```

arg

Integer argument that is used in the directed query to pair each input query :v hint with one or more annotated query :v hints.

Examples

See Ignoring constants in directed queries .

ALLNODES

Qualifies an EXPLAIN statement to request a query plan that assumes all nodes are active. If you omit this hint, the EXPLAIN statement produces a query plan that takes into account any nodes that are currently down.

Syntax

```
EXPLAIN /*+ALLNODES*/
```

Examples

In the following example, the ALLNODES hint requests a query plan that assumes all nodes are active.

```
QUERY PLAN DESCRIPTION:
 ------------------------------

 Opt Vertica Options
 --------------------
 PLAN_ALL_NODES_ACTIVE


 EXPLAIN /*+ALLNODES*/ select * from Emp_Dimension;

 Access Path:
+-STORAGE ACCESS for Emp_Dimension [Cost: 125, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
|  Projection: public.Emp_Dimension_b0
|  Materialize: Emp_Dimension.Employee_key, Emp_Dimension.Employee_gender, Emp_Dimension.Courtesy_title, Emp_Dimension.Employee_first_name,
Emp_Dimension.Employee_middle_initial, Emp_Dimension.Employee_last_name, Emp_Dimension.Employee_age, Emp_Dimension.Employee_birthdate,
Emp_Dimension.Employee_street, Emp_Dimension.Employee_city, Emp_Dimension.Employee_state, Emp_Dimension.Employee_region,
Emp_Dimension.Employee_position
|  Execute on: All Nodes
```

DEPOT_FETCH

Eon Mode only

Specifies whether a query fetches data to the depot from communal storage when the depot lacks data for this query. This hint overrides configuration parameter DepotOperationsForQuery .

Syntax

```
SELECT /*+DEPOT_FETCH (option)*/
```

Arguments

option

Specifies behavior when the depot does not contain queried file data, one of the following:

- ALL (default): Fetch file data from communal storage, if necessary displace existing files by evicting them from the depot.
- FETCHES : Fetch file data from communal storage only if space is available; otherwise, read the queried data directly from communal storage.
- NONE : Do not fetch file data to the depot, read the queried data directly from communal storage.

Examples

```
SELECT /*+DEPOT_FETCH(All)*/ count(*) FROM bar;
SELECT /*+DEPOT_FETCH(FETCHES)*/ count(*) FROM bar;
SELECT /*+DEPOT_FETCH(NONE)*/ count(*) FROM bar;
```

DISTRIB

The DISTRIB hint specifies to the optimizer how to distribute join key data in order to implement a join. If a specified distribution is not feasible, the optimizer ignores the hint and throws a warning.

The following requirements apply:

- Queries that include the DISTRIB hint must also include the SYNTACTIC_JOIN hint. Otherwise, the optimizer ignores the DISTRIB hint and throws a warning.
- Join syntax must conform with ANSI SQL-92 join conventions.

Syntax

```
JOIN /*+DISTRIB(outer-join, inner-join)*/
```

Arguments

outer-join

inner-join

Specifies how to distribute data on the outer and inner joins:

- L (local): Inner and outer join keys are identically segmented on each node, join locally.
- R (resegment): Inner and outer join keys are not identically segmented. Resegment join-key data before implementing the join.

- B (broadcast): Inner and outer join keys are not identically segmented. Broadcast data of this join key to other nodes before implementing the join.
- F (filter): Join table is unsegmented. Filter data as needed by the other join key before implementing the join.
- A (any): Let the optimizer choose the distribution method that it considers to be most cost-effective.

Examples

In the following query, the join is qualified with a DISTRIB hint of /*+DISTRIB(L,R)*/ . This hint tells the optimizer to resegment data of join key stores.store_key before joining it to the sales.store_key data:

```
SELECT /*+SYNTACTIC_JOIN*/ sales.store_key, stores.store_name, sales.product_description, sales.sales_quantity, sales.sale_date
FROM (store.storeSales AS sales JOIN /*+DISTRIB(L,R),JTYPE(H)*/ store.store_dimension AS stores ON (sales.store_key = stores.store_key))
WHERE (sales.sale_date = '2014-12-01'::date) ORDER BY sales.store_key, sales.sale_date;
```

EARLY_MATERIALIZATION

Specifies early materialization of a table for the current query. A query can include this hint for any number of tables. Typically, the query optimizer delays materialization until late in the query execution process. This hint overrides any choices that the optimizer otherwise would make.

This hint can be useful in cases where late materialization of join inputs precludes other optimizations—for example, pushing aggregation down the joins, or using live aggregate projections. In these cases, qualifying a join input with EARLY_MATERIALIZATION can enable the optimizations.

Syntax

```
table-name [ [AS] alias ] /*+EARLY_MATERIALIZATION*/
```

ECSMODE

Eon Mode only

Sets the ECS strategy that the optimizer uses when it divides responsibility for processing shard data among subscriber nodes. This hint is applied only if the subcluster uses elastic crunch scaling (ECS).

Syntax

```
SELECT /*+ECSMODE(option)*/
```

Arguments

option

Specifies the strategy to use when dividing shard data among its subscribing nodes, one of the following:

- AUTO : The optimizer chooses the strategy to use, useful only if ECS mode is set at the session level (see Setting the ECS Strategy for the Session or Database).
- IO_OPTIMIZED : Use I/O-optimized strategy.
- COMPUTE_OPTIMIZED : Use compute-optimized strategy.
- NONE : Disable use of ECS for this query. Only participating nodes are involved in query execution; collaborating nodes are not.

Example

The following example shows the query plan for a simple single-table query that is forced to use the compute-optimized strategy:

```
=> EXPLAIN SELECT /*+ECSMode(COMPUTE_OPTIMIZED)*/ employee_last_name,
        employee_first_name,employee_age
        FROM employee_dimension
        ORDER BY employee_age DESC;

                        QUERY PLAN
--------------------------------------------------------------------------------
 ------------------------------
 QUERY PLAN DESCRIPTION:
 The execution of this query involves non-participating nodes.
 Crunch scaling strategy preserves data segmentation
 ------------------------------
. . .
```

ENABLE_WITH_CLAUSE_MATERIALIZATION

Enables materialization of all queries in the current WITH clause. Otherwise, materialization is set by configuration parameter WithClauseMaterialization, by default set to 0 (disabled). If WithClauseMaterialization is disabled, materialization is automatically cleared when the primary query of the WITH clause returns. For details, see Materialization of WITH clause .

Syntax

```
WITH /*+ENABLE_WITH_CLAUSE_MATERIALIZATION*/
```

GBYTYPE

Specifies which algorithm—GROUPBY HASH or GROUPBY PIPELINED —the Vertica query optimizer should use to implement a GROUP BY clause. If both algorithms are valid for this query, the query optimizer chooses the specified algorithm over the algorithm that the query optimizer might otherwise choose in its query plan.

> **Note**
> Vertica uses the GROUPBY PIPELINED algorithm only if the query and one of its projections comply with GROUPBY PIPELINED requirements . Otherwise, Vertica issues a warning and uses GROUPBY HASH.

For more information about both algorithms, see GROUP BY implementation options .

Syntax

```
GROUP BY /*+GBYTYPE( HASH | PIPE )*/
```

Arguments

HASH

Use the GROUPBY HASH algorithm.

PIPE

Use the GROUPBY PIPELINED algorithm.

Examples

See Controlling GROUPBY Algorithm Choice .

JFMT

Specifies how to size VARCHAR column data when joining tables on those columns, and buffer that data accordingly. The JFMT hint overrides the default behavior that is set by configuration parameter JoinDefaultTupleFormat , which can be set at database and session levels.

For more information, see Joining variable length string data .

Syntax

```
JOIN /*+JFMT(format-type)*/
```

Arguments

format-type

Specifies how to format VARCHAR column data when joining tables on those columns, and buffers the data accordingly. Set to one of the following:

- f (fixed): Use join column metadata to size column data to a fixed length, and buffer accordingly.
- v (variable): Use the actual length of join column data, so buffer size varies for each join.

For example:

```
SELECT /*+SYNTACTIC_JOIN*/ s.store_region, SUM(e.vacation_days) TotalVacationDays
  FROM public.employee_dimension e
  JOIN /*+JFMT(f)*/ store.store_dimension s ON s.store_region=e.employee_region
  GROUP BY s.store_region ORDER BY TotalVacationDays;
```

Requirements

- Queries that include the JFMT hint must also include the SYNTACTIC_JOIN hint. Otherwise, the optimizer ignores the JFMT hint and throws a warning.
- Join syntax must conform with ANSI SQL-92 join conventions.

JTYPE

Specifies the join algorithm as hash or merge.

Use the JTYPE hint to specify the algorithm the optimizer uses to join table data. If the specified algorithm is not feasible, the optimizer ignores the hint and throws a warning.

Syntax

```
JOIN /*+JTYPE(join-type)*/
```

Arguments

join-type

One of the following:

- H : Hash join
- M : Merge join, valid only if both join inputs are already sorted on the join columns, otherwise Vertica ignores it and throws a warning. The optimizer relies upon the query or DDL to verify whether input data is sorted, rather than the actual runtime order of the data.
- FM : Forced merge join. Before performing the merge, the optimizer re-sorts the join inputs. Join columns must be of the same type and precision or scale, except that string columns can have different lengths.

A value of FM is valid only for simple join conditions. For example:

```
=> SELECT /*+SYNTACTIC_JOIN*/ * FROM x JOIN /*+JTYPE(FM)*/ y ON x.c1 = y.c1;
```

Requirements

- Queries that include the JTYPE hint must also include the SYNTACTIC_JOIN hint. Otherwise, the optimizer ignores the JTYPE hint and throws a warning.
- Join syntax must conform with ANSI SQL-92 join conventions.

LABEL

Assigns a label to a statement so it can easily be identified to evaluate performance and debug problems.

LABEL hints are valid in the following statements:

- COPY
- DELETE
- EXPORT statements:
  - EXPORT TO DELIMITED
  - EXPORT TO ORC
  - EXPORT TO PARQUET
  - EXPORT TO VERTICA
- INSERT
- MERGE
- SELECT
- UPDATE
- UNION : Valid in the UNION's first SELECT statement. Vertica ignores labels in subsequent SELECT statements.

Syntax

```
statement-name /*+LABEL (label-string)*/
```

Arguments

label-string

A string that is up to 128 octets long. If enclosed with single quotes, *label-string* can contain embedded spaces.

Examples

See Labeling statements .

PROJS

Specifies one or more projections to use for a queried table.

The PROJS hint can specify multiple projections; the optimizer determines which ones are valid and uses the one that is most cost-effective for the queried table. If no hinted projection is valid, the query returns a warning and ignores projection hints.

Syntax

```
FROM `*`table-name`*` /*+PROJS( [[`*`database`*`.]`*`schema.`*`]`*`projection`*`[,...] )*/
```

Arguments

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection

The projection to use. You can specify a list of comma-delimited projections.

Examples

The employee_dimension table has two projections: segmented superprojection public.employee_dimension , which includes all table columns; and the unsegmented projection public.employee_dimension_rep , which includes a subset of the columns:

```
=> SELECT export_objects('','employee_dimension');
                                                export_objects
-------------------------------------------------------------------------------------------------------------------------

CREATE TABLE public.employee_dimension
(
    employee_key int NOT NULL,
    employee_gender varchar(8),
    courtesy_title varchar(8),
    employee_first_name varchar(64),
    employee_middle_initial varchar(8),
    employee_last_name varchar(64),
    employee_age int,
    hire_date date,
    employee_street_address varchar(256),
    employee_city varchar(64),
    employee_state char(2),
    employee_region char(32),
    job_title varchar(64),
    reports_to int,
    salaried_flag int,
    annual_salary int,
    hourly_rate float,
    vacation_days int,
    CONSTRAINT C_PRIMARY PRIMARY KEY (employee_key) DISABLED
);

CREATE PROJECTION public.employee_dimension
...
AS
 SELECT employee_dimension.employee_key,
        employee_dimension.employee_gender,
        employee_dimension.courtesy_title,
        employee_dimension.employee_first_name,
        employee_dimension.employee_middle_initial,
        employee_dimension.employee_last_name,
        employee_dimension.employee_age,
        employee_dimension.hire_date,
        employee_dimension.employee_street_address,
        employee_dimension.employee_city,
        employee_dimension.employee_state,
        employee_dimension.employee_region,
        employee_dimension.job_title,
        employee_dimension.reports_to,
        employee_dimension.salaried_flag,
        employee_dimension.annual_salary,
        employee_dimension.hourly_rate,
        employee_dimension.vacation_days
 FROM public.employee_dimension
```

```
 ORDER BY employee_dimension.employee_key
SEGMENTED BY hash(employee_dimension.employee_key) ALL NODES KSAFE 1;

CREATE PROJECTION public.employee_dimension_rep
...
AS
 SELECT employee_dimension.employee_key,
        employee_dimension.employee_gender,
        employee_dimension.employee_first_name,
        employee_dimension.employee_middle_initial,
        employee_dimension.employee_last_name,
        employee_dimension.employee_age,
        employee_dimension.employee_street_address,
        employee_dimension.employee_city,
        employee_dimension.employee_state,
        employee_dimension.employee_region
 FROM public.employee_dimension
 ORDER BY employee_dimension.employee_key
UNSEGMENTED ALL NODES;

SELECT MARK_DESIGN_KSAFE(1);

(1 row)
```

The following query selects all table columns from employee_dimension and includes the PROJS hint, which specifies both projections. public.employee_dimension_rep does not include all columns in the queried table, so the optimizer cannot use it. The segmented projection includes all table columns so the optimizer uses it, as verified by the following query plan:

```
=> EXPLAIN SELECT * FROM employee_dimension /*+PROJS('public.employee_dimension_rep', 'public.employee_dimension')*/;

QUERY PLAN DESCRIPTION:
------------------------------
EXPLAIN SELECT * FROM employee_dimension /*+PROJS('public.employee_dimension_rep', 'public.employee_dimension')*/;

 Access Path:
 +-STORAGE ACCESS for employee_dimension [Cost: 177, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
 |  Projection: public.employee_dimension_b0
```

SKIP_PROJS

Specifies which projections to avoid using for a queried table. If SKIP_PROJS excludes all available projections that are valid for the query, the optimizer issues a warning and ignores the projection hints.

Syntax

```
FROM table-name /*+SKIP_PROJS( [[database.]schema.]projection[,...] )*/
```

Arguments

[*database.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection

A projection to skip. You can specify a list of comma-delimited projections.

Examples

In this example, the EXPLAIN output shows that the optimizer uses the projection public.employee_dimension_b0 for a given query:

```
QUERY PLAN DESCRIPTION:
------------------------------

 EXPLAIN SELECT employee_last_name, employee_first_name, employee_city, job_title FROM employee_dimension;

 Access Path:
 +-STORAGE ACCESS for employee_dimension [Cost: 59, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
 |  Projection: public.employee_dimension_b0
 |  Materialize: employee_dimension.employee_first_name, employee_dimension.employee_last_name, employee_dimension.employee_city,
employee_dimension.job_title
 |  Execute on: All Nodes
```

You can use the SKIP_PROJS hint to avoid using this projection. If another projection is available that is valid for this query, the optimizer uses it instead:

```
QUERY PLAN DESCRIPTION:
------------------------------

EXPLAIN SELECT employee_last_name, employee_first_name, employee_city, job_title FROM employee_dimension
/*+SKIP_PROJS('public.employee_dimension')*/;

 Access Path:
 +-STORAGE ACCESS for employee_dimension [Cost: 156, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
 |  Projection: public.employee_dimension_super
 |  Materialize: employee_dimension.employee_first_name, employee_dimension.employee_last_name, employee_dimension.emplo
yee_city, employee_dimension.job_title
 |  Execute on: Query Initiator
```

SKIP_STATISTICS

Directs the optimizer to produce a query plan that incorporates only the minimal statistics that are collected by ANALYZE_ROW_COUNT . The optimizer ignores other statistics that would otherwise be used, that are generated by ANALYZE_STATISTICS and ANALYZE_STATISTICS_PARTITION . This hint is especially useful when used in queries on small tables, where the amount of time required to collect full statistics is often greater than actual execution time.

Syntax

```
SELECT /*+SKIP_STAT[ISTIC]S*/
```

EXPLAIN output

EXPLAIN returns the following output for a query that includes SKIP_STATISTICS (using its shortened form SKIP_STATS):

```
=> EXPLAIN SELECT /*+ SKIP_STATS*/ customer_key, customer_name, customer_gender, customer_city||', '||customer_state, customer_age
   FROM customer_dimension WHERE customer_region = 'East' AND customer_age > 60;

 QUERY PLAN DESCRIPTION:
 ------------------------------

 EXPLAIN SELECT /*+ SKIP_STATS*/ customer_key, customer_name, customer_gender, customer_city||', '||customer_state,
 customer_age FROM customer_dimension WHERE customer_region = 'East' AND customer_age > 60;

 Access Path:
 +-STORAGE ACCESS for customer_dimension [Cost: 2K, Rows: 10K (STATISTICS SKIPPED)] (PATH ID: 1)
 |  Projection: public.customer_dimension_b0
 |  Materialize: public.customer_dimension.customer_age, public.customer_dimension.customer_key, public.customer_dimensi
on.customer_name, public.customer_dimension.customer_gender, public.customer_dimension.customer_city, public.customer_di
mension.customer_state
 |  Filter: (public.customer_dimension.customer_region = 'East')
 |  Filter: (public.customer_dimension.customer_age > 60)
 |  Execute on: All Nodes
...
```

SYNTACTIC_JOIN

Enforces join order and enables other join hints.

In order to achieve optimal performance, the optimizer often overrides a query's specified join order. By including the SYNTACTIC_JOIN hint, you can ensure that the optimizer enforces the query's join order exactly as specified. One requirement applies: the join syntax must conform with ANSI SQL-92 conventions.

The SYNTACTIC_JOIN hint must immediately follow SELECT . If the annotated query includes another hint that must also follow SELECT , such as VERBATIM , combine the two hints together. For example:

```
SELECT /*+ syntactic_join,verbatim*/
```

Syntax

```
SELECT /*+SYN[TACTIC]_JOIN*/
```

Examples

In the following examples, the optimizer produces different plans for two queries that differ only by including or excluding the SYNTACTIC_JOIN hint.

Excludes SYNTACTIC_JOIN:

```
EXPLAIN SELECT sales.store_key, stores.store_name, products.product_description, sales.sales_quantity, sales.sale_date
FROM (store.store_sales sales JOIN products ON sales.product_key=products.product_key)
JOIN store.store_dimension stores ON sales.store_key=stores.store_key
WHERE sales.sale_date='2014-12-01' order by sales.store_key, sales.sale_date;

 Access Path:
 +-SORT [Cost: 14K, Rows: 100K (NO STATISTICS)] (PATH ID: 1)
 |  Order: sales.store_key ASC, sales.sale_date ASC
 |  Execute on: All Nodes
 | +---> JOIN HASH [Cost: 11K, Rows: 100K (NO STATISTICS)] (PATH ID: 2) Outer (RESEGMENT)(LOCAL ROUND ROBIN) Inner (RESEGMENT)
 | |       Join Cond: (sales.product_key = products.product_key)
 | |       Materialize at Input: sales.store_key, sales.product_key, sales.sale_date, sales.sales_quantity
 | |       Execute on: All Nodes
 | | +-- Outer -> JOIN HASH [Cost: 1K, Rows: 100K (NO STATISTICS)] (PATH ID: 3)
 | | |       Join Cond: (sales.store_key = stores.store_key)
 | | |       Execute on: All Nodes
 | | | +-- Outer -> STORAGE ACCESS for sales [Cost: 1K, Rows: 100K (NO STATISTICS)] (PATH ID: 4)
 | | | |       Projection: store.store_sales_b0
 | | | |       Materialize: sales.store_key
 | | | |       Filter: (sales.sale_date = '2014-12-01'::date)
 | | | |       Execute on: All Nodes
 | | | |       Runtime Filter: (SIP1(HashJoin): sales.store_key)
 | | | +-- Inner -> STORAGE ACCESS for stores [Cost: 34, Rows: 250] (PATH ID: 5)
 | | | |       Projection: store.store_dimension_DBD_10_rep_VMartDesign_node0001
 | | | |       Materialize: stores.store_key, stores.store_name
 | | | |       Execute on: All Nodes
 | | +-- Inner -> STORAGE ACCESS for products [Cost: 3K, Rows: 60K (NO STATISTICS)] (PATH ID: 6)
 | | |       Projection: public.products_b0
 | | |       Materialize: products.product_key, products.product_description
 | | |       Execute on: All Nodes
```

Includes SYNTACTIC_JOIN:

```
EXPLAIN SELECT /*+SYNTACTIC_JOIN*/ sales.store_key, stores.store_name, products.product_description, sales.sales_quantity, sales.sale_date
FROM (store.store_sales sales JOIN products ON sales.product_key=products.product_key)
JOIN store.store_dimension stores ON sales.store_key=stores.store_key
WHERE sales.sale_date='2014-12-01' order by sales.store_key, sales.sale_date;

 Access Path:
 +-SORT [Cost: 11K, Rows: 100K (NO STATISTICS)] (PATH ID: 1)
 |  Order: sales.store_key ASC, sales.sale_date ASC
 |  Execute on: All Nodes
 | +---> JOIN HASH [Cost: 8K, Rows: 100K (NO STATISTICS)] (PATH ID: 2)
 | |      Join Cond: (sales.store_key = stores.store_key)
 | |      Execute on: All Nodes
 | | +-- Outer -> JOIN HASH [Cost: 7K, Rows: 100K (NO STATISTICS)] (PATH ID: 3) Outer (BROADCAST)(LOCAL ROUND ROBIN)
 | | |      Join Cond: (sales.product_key = products.product_key)
 | | |      Execute on: All Nodes
 | | |      Runtime Filter: (SIP1(HashJoin): sales.store_key)
 | | | +-- Outer -> STORAGE ACCESS for sales [Cost: 2K, Rows: 100K (NO STATISTICS)] (PATH ID: 4)
 | | | |      Projection: store.store_sales_b0
 | | | |      Materialize: sales.sale_date, sales.store_key, sales.product_key, sales.sales_quantity
 | | | |      Filter: (sales.sale_date = '2014-12-01'::date)
 | | | |      Execute on: All Nodes
 | | | +-- Inner -> STORAGE ACCESS for products [Cost: 3K, Rows: 60K (NO STATISTICS)] (PATH ID: 5)
 | | | |      Projection: public.products_b0
 | | | |      Materialize: products.product_key, products.product_description
 | | | |      Execute on: All Nodes
 | | +-- Inner -> STORAGE ACCESS for stores [Cost: 34, Rows: 250] (PATH ID: 6)
 | | |      Projection: store.store_dimension_DBD_10_rep_VMartDesign_node0001
 | | |      Materialize: stores.store_key, stores.store_name
 | | |      Execute on: All Nodes
```

UTYPE

Specifies how to combine UNION ALL input.

Syntax

```
UNION ALL /*+UTYPE(union-type)*/
```

Arguments

union-type

One of the following values:

- U : Concatenates UNION ALL input (default).
- M : Merges UNION ALL input in the same sort order as the source query results. This option requires all input from the source queries to use the same sort order; otherwise, Vertica throws a warning and concatenates the UNION ALL input.

> **Note**
> The optimizer relies upon the query or DDL to verify whether input data is sorted, rather than the actual runtime order of the data.

Requirements

Queries that include the UTYPE hint must also include the SYNTACTIC_JOIN hint. Otherwise, the optimizer ignores the UTYPE hint and throws a warning.

VERBATIM

Enforces execution of an annotated query exactly as written.

VERBATIM directs the optimizer to create a query plan that incorporates all hints in a annotated query. Furthermore, it directs the optimizer not to apply its own plan development processing on query plan components that pertain to those hints.

Usage of this hint varies between optimizer-generated and custom directed queries, as described below.

Syntax

```
SELECT /*+ VERBATIM*/
```

Requirements

The VERBATIM hint must immediately follow SELECT. If the annotated query includes another hint that must also follow SELECT, such as SYNTACTIC_JOIN, combine the two hints together. For example:

```
SELECT /*+ syntactic_join,verbatim*/
```

Optimizer-generated directed queries

The optimizer always includes the VERBATIM hint in the annotated queries that it generates for directed queries . For example, given the following CREATE DIRECTED QUERY OPTIMIZER statement:

```
=> CREATE DIRECTED QUERY OPTIMIZER getStoreSales SELECT sales.store_key, stores.store_name, sales.product_description, sales.sales_quantity, sales.sale_date FROM store.storesales sales JOIN store.store_dimension stores ON sales.store_key=stores.store_key WHERE sales.sale_date='2014-12-01' /*+IGNORECONST(1)*/ AND stores.store_name='Store1' /*+IGNORECONST(2)*/ ORDER BY sales.store_key, sales.sale_date;
CREATE DIRECTED QUERY
```

The optimizer generates an annotated query that includes the VERBATIM hint:

```
=> SELECT query_name, annotated_query FROM V_CATALOG.DIRECTED_QUERIES WHERE query_name = 'getStoreSales';
-[ RECORD 1 ]---+------
query_name      | getStoreSales
annotated_query | SELECT /*+ syntactic_join,verbatim*/ sales.store_key AS store_key, stores.store_name AS store_name, sales.product_description AS product_description, sales.sales_quantity AS sales_quantity, sales.sale_date AS sale_date
FROM (store.storeSales AS sales/*+projs('store.storeSales')*/ JOIN /*+Distrib(L,L),JType(H)*/ store.store_dimension AS stores/*+projs('store.store_dimension_DBD_10_rep_VMartDesign')*/  ON (sales.store_key = stores.store_key))
WHERE (sales.sale_date = '2014-12-01'::date /*+IgnoreConst(1)*/) AND (stores.store_name = 'Store1'::varchar(6) /*+IgnoreConst(2)*/)
ORDER BY 1 ASC, 5 ASC
```

When the optimizer uses this directed query, it produces a query plan that is equivalent to the query plan that it used when it created the directed query:

```
=> ACTIVATE DIRECTED QUERY getStoreSales;
ACTIVATE DIRECTED QUERY

=> EXPLAIN SELECT sales.store_key, stores.store_name, sales.product_description, sales.sales_quantity, sales.sale_date FROM store.storesales sales
JOIN store.store_dimension stores ON sales.store_key=stores.store_key WHERE sales.sale_date='2014-12-04' AND stores.store_name='Store14' ORDER
BY sales.store_key, sales.sale_date;

 QUERY PLAN DESCRIPTION:
 ------------------------------

 EXPLAIN SELECT sales.store_key, stores.store_name, sales.product_description, sales.sales_quantity, sales.sale_date FROM store.storesales sales JOIN
store.store_dimension stores ON sales.store_key=stores.store_key WHERE sales.sale_date='2014-12-04' AND stores.store_name='Store14' ORDER BY
sales.store_key, sales.sale_date;

 The following active directed query(query name: getStoreSales) is being executed:
 SELECT /*+syntactic_join,verbatim*/ sales.store_key, stores.store_name, sales.product_description, sales.sales_quantity, sales.sale_date
FROM (store.storeSales sales/*+projs('store.storeSales')*/ JOIN /*+Distrib('L', 'L'), JType('H')*/store.store_dimension stores
/*+projs('store.store_dimension_DBD_10_rep_VMartDesign')*/ ON ((sales.store_key = stores.store_key))) WHERE ((sales.sale_date = '2014-12-04'::date)
AND (stores.store_name = 'Store14'::varchar(7))) ORDER BY sales.store_key, sales.sale_date

 Access Path:
 +-JOIN HASH [Cost: 463, Rows: 622 (NO STATISTICS)] (PATH ID: 2)
 |  Join Cond: (sales.store_key = stores.store_key)
 |  Materialize at Output: sales.sale_date, sales.sales_quantity, sales.product_description
 |  Execute on: All Nodes
 | +-- Outer -> STORAGE ACCESS for sales [Cost: 150, Rows: 155K (NO STATISTICS)] (PATH ID: 3)
 | |      Projection: store.storeSales_b0
 | |      Materialize: sales.store_key
 | |      Filter: (sales.sale_date = '2014-12-04'::date)
 | |      Execute on: All Nodes
 | |      Runtime Filter: (SIP1(HashJoin): sales.store_key)
 | +-- Inner -> STORAGE ACCESS for stores [Cost: 35, Rows: 2] (PATH ID: 4)
 | |      Projection: store.store_dimension_DBD_10_rep_VMartDesign_node0001
 | |      Materialize: stores.store_name, stores.store_key
 | |      Filter: (stores.store_name = 'Store14')
 | |      Execute on: All Nodes
```

Custom directed queries

The VERBATIM hint is included in a custom directed query only if you explicitly include it in the annotated query that you write for that directed query. When the optimizer uses that directed query, it respects the VERBATIM hint and creates a query plan accordingly.

If you omit the VERBATIM hint when you create a custom directed query, the hint is not stored with the annotated query. When the optimizer uses that directed query, it applies its own plan development processing on the annotated query before it generates a query plan. This query plan might not be equivalent to the query plan that the optimizer would have generated for the Vertica version in which the directed query was created.

Window clauses

When used with an analytic function , window clauses specify how to partition and sort function input, as well as how to frame input with respect to the current row. When used with a single-phase transform function, the PARTITION ROW window clause supports single-row partitions for single-phase transform functions, rather than analytic functions.

In this section

- Window partition clause
- Window order clause
- Window frame clause
- Window name clause

Window partition clause

When specified, a window partition clause divides the rows of the function input based on user-provided expressions. If no expression is provided, the partition clause can improve query performance by using parallelism. If you omit specifying a window partition clause, all input rows are treated as a single partition.

Window partitioning is similar to the GROUP BY clause. However, PARTITION BEST and PARTITION NODES may only be used with analytic functions and return only one result per input row, while PARTITION ROW and PARTITION LEFT JOIN can be used for single-phase transform functions and return multiple values per input row.

When used with analytic functions, results are computed per partition and start over again (reset) at the beginning of each subsequent partition.

Syntax

```
{ PARTITION BY expression[,...] | PARTITION BEST | PARTITION NODES | PARTITION ROW }
```

Arguments

PARTITION BY *expression*	Expression on which to sort the partition, where *expression* can be a column, constant, or an arbitrary expression formed on columns. Use PARTITION BY for functions with specific partitioning requirements.
PARTITION BEST	Use parallelism to improve performance for multi-threaded queries across multiple nodes. OVER(PARTITION BEST) provides the best performance on multi-threaded queries across multiple nodes. The following considerations apply to using PARTITION BEST : • Use PARTITION BEST for analytic functions that have no partitioning requirements and are thread safe—for example, a one-to-many transform. • Do not use PARTITION BEST on user-defined transform functions (UDTFs) that are not thread-safe. Doing so can produce an error or incorrect results. If a UDTF is not thread safe, use PARTITION NODES .
PARTITION NODES	Use parallelism to improve performance for single-threaded queries across multiple nodes. OVER(PARTITION NODES) provides the best performance on single-threaded queries across multiple nodes.
PARTITION ROW	Use to feed input partitions of exactly one row. If used, any arbitrary expression may be used in the query target list alongside the UDTF. May not be used for analytic functions or multi-phase transform functions. Note that only one PARTITION ROW transform function is allowed in the target list for each level of the query.

Examples

See Window partitioning .

Window order clause

Specifies how to sort rows that are supplied to an analytic function. If the OVER clause also includes a window partition clause , rows are sorted within each partition.

The window order clause only specifies order within a window result set. The query can have its own ORDER BY clause outside the OVER clause. This has precedence over the window order clause and orders the final result set.

A window order clause also creates a default window frame if none is explicitly specified.

Syntax

```
ORDER BY { expression [ ASC | DESC [ NULLS { FIRST | LAST | AUTO } ] ]
 }[,...]
```

Arguments

expression	A column, constant, or arbitrary expression formed on columns on which to sort input rows.
ASC | DESC	Specifies the ordering sequence as ascending (default) or descending.

NULLS {FIRST | LAST | AUTO}	Specifies whether to position null values first or last. Default positioning depends on whether the sort order is ascending or descending: • Ascending default: NULLS LAST • Descending default: NULLS FIRST If you specify NULLS AUTO , Vertica chooses the positioning that is most efficient for this query, either NULLS FIRST or NULLS LAST . If you omit all sort qualifiers, Vertica uses ASC NULLS LAST . For more information, see: • NULL sort order • Runtime sorting of NULL values in analytic functions

Examples

See Window ordering .

Window frame clause

Specifies a window frame, which comprises a set of rows relative to the row that is currently being evaluated by an analytic function. After the function processes that row and its window, Vertica advances the current row and adjusts the window boundaries accordingly. If the OVER clause also specifies a partition , Vertica also checks that window boundaries do not cross partition boundaries. This process repeats until the function evaluates the last row of the last partition.

Syntax

```
{ ROWS | RANGE } { BETWEEN start-point AND end-point } | start-point
```

***start-point* / *end-point* :**

```
{ UNBOUNDED {PRECEDING | FOLLOWING}
 | CURRENT ROW
 | constant-value  {PRECEDING | FOLLOWING}}
```

Arguments

ROWS | RANGE	Specifies whether Vertica determines window frame dimensions as physical or logical offsets from the current row. See ROWS versus RANGE below for details.
BETWEEN *start-point* AND *end-point*	Specifies the window's first and last rows, where *start-point* and *end-point* can be one of the following (discussed in detail below): • UNBOUNDED {PRECEDING | FOLLOWING} • CURRENT ROW • *constant-value* {PRECEDING | FOLLOWING} *start-point* must resolve to a row or value that is less than or equal to *end-point* .
UNBOUNDED PRECEDING	Specifies that the window frame extends to the current partition's first row.
start-point	If ROWS or RANGE specifies only a start point, Vertica uses the current row as the end point and creates the window frame accordingly. In this case, *start-point* must resolve to a row that is less than or equal to the current row.
UNBOUNDED FOLLOWING	Specifies that the window frame extends to the current partition's last row.
CURRENT ROW	Specifies the current row or value as the window's start or end point.

constant-value {PRECEDING | FOLLOWING}	Specifies a constant value or expression that evaluates to a constant value. The value specifies a physical or logical offset from the current row, depending on whether you specify ROWS or RANGE . Other dependencies also pertain, depending whether you specify ROWS and RANGE . See ROWS versus RANGE below for details.

Requirements

In order to specify a window frame, the OVER must also specify a window order (ORDER BY) clause . If the OVER clause omits specifying a window frame, the function creates a default window that extends from the current row to the first row in the current partition. This is equivalent to the following clause:

```
RANGE UNBOUNDED PRECEDING AND CURRENT ROW
```

ROWS versus RANGE

The window frame's offset from the current row can be physical or logical:

- ROWS specifies the window's *start-point* and *end-point* as a number of rows relative to the current row. If *start-point* and *end-point* are expressed as constant values, the value must evaluate to a positive integer.
- RANGE specifies the window as a logical offset such as time. The range value must match the window order (ORDER BY) clause data type: NUMERIC, DATE/TIME, FLOAT or INTEGER.

Use of ROWS or RANGE imposes specific requirements on setting the window's start and end points as constant values. When setting constant values for ROWS , the constant must evaluate to a positive INTEGER.

When setting constant values for RANGE , the following requirements apply:

- The constant must evaluate to a positive numeric value or INTERVAL literal.
- If the constant evaluates to a NUMERIC value, the ORDER BY column type must be a NUMERIC data type.
- If the constant evaluates to an INTERVAL DAY TO SECOND subtype, the ORDER BY column type must be one of the following: TIMESTAMP, TIME, DATE, or INTERVAL DAY TO SECOND.
- If the constant evaluates to an INTERVAL YEAR TO MONTH, the ORDER BY column type must be one of the following: TIMESTAMP, DATE, or INTERVAL YEAR TO MONTH.
- The window order clause can specify only one expression.

Examples

See Window framing .

Window name clause

Defines a named window that specifies window partition and order clauses for an analytic function. This window is specified in the function's OVER clause. Named windows can be useful when you write queries that invoke multiple analytic functions with similar OVER clauses—for example, they use the same partition (PARTITION BY) clauses.

Syntax

```
WINDOW window-name AS ( window-partition-clause [window-order-clause] )
```

Arguments

WINDOW *window-name*	Specifies the window name. All window names must be unique within the same query.
window-partition-clause [*window-order-clause*]	Clauses to invoke when an OVER clause references this window. If the window definition omits a window order clause , the OVER clause can specify its own order clause.

Requirements

- A WINDOW clause cannot include a window frame clause .
- Each WINDOW clause within the same query must have a unique name.
- A WINDOW clause can reference another window that is already named. For example, the following query names window w1 before w2 . Thus, the WINDOW clause that defines w2 can reference w1 :

```
=> SELECT RANK() OVER(w1 ORDER BY sal DESC), RANK() OVER w2
  FROM EMP WINDOW w1 AS (PARTITION BY deptno), w2 AS (w1 ORDER BY sal);
```

Examples

See Named windows .

See also

Analytic functions

Data types

The following table summarizes the internal data types that Vertica supports. It also shows the default placement of null values in projections. The Size column lists uncompressed bytes.

Data Type	Size / bytes	Description	NULL Sorting
Binary			
BINARY	1 to 65,000	Fixed-length binary string	NULLS LAST
VARBINARY (synonyms: BYTEA , RAW)	1 to 65,000	Variable-length binary string	NULLS LAST
LONG VARBINARY	1 to 32,000,000	Long variable-length binary string	NULLS LAST
Boolean			
BOOLEAN	1	True or False or NULL	NULLS LAST
Character / Long			
CHAR	1 to 65,000	Fixed-length character string	NULLS LAST
VARCHAR	1 to 65,000	Variable-length character string	NULLS LAST
LONG VARCHAR	1 to 32,000,000	Long variable-length character string	NULLS LAST
Date/Time			
DATE	8	A month, day, and year	NULLS FIRST
TIME	8	A time of day without timezone	NULLS FIRST
TIME WITH TIMEZONE	8	A time of day with timezone	NULLS FIRST
TIMESTAMP (synonyms: DATETIME , SMALLDATETIME)	8	A date and time without timezone	NULLS FIRST
TIMESTAMP WITH TIMEZONE	8	A date and time with timezone	NULLS FIRST
INTERVAL	8	The difference between two points in time	NULLS FIRST
INTERVAL DAY TO SECOND	8	An interval measured in days and seconds	NULLS FIRST
INTERVAL YEAR TO MONTH	8	An interval measured in years and months	NULLS FIRST

Approximate Numeric			
DOUBLE PRECISION	8	Signed 64-bit IEEE floating point number, requiring 8 bytes of storage	NULLS LAST
FLOAT	8	Signed 64-bit IEEE floating point number, requiring 8 bytes of storage	NULLS LAST
FLOAT(n)	8	Signed 64-bit IEEE floating point number, requiring 8 bytes of storage	NULLS LAST
FLOAT8	8	Signed 64-bit IEEE floating point number, requiring 8 bytes of storage	NULLS LAST
REAL	8	Signed 64-bit IEEE floating point number, requiring 8 bytes of storage	NULLS LAST
Exact Numeric			
INTEGER	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
INT	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
BIGINT	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
INT8	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
SMALLINT	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
TINYINT	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
DECIMAL	8+	8 bytes for the first 18 digits of precision, plus 8 bytes for each additional 19 digits	NULLS FIRST
NUMERIC	8+	8 bytes for the first 18 digits of precision, plus 8 bytes for each additional 19 digits	NULLS FIRST
NUMBER	8+	8 bytes for the first 18 digits of precision, plus 8 bytes for each additional 19 digits	NULLS FIRST
MONEY	8+	8 bytes for the first 18 digits of precision, plus 8 bytes for each additional 19 digits	NULLS FIRST
Spatial			
GEOMETRY	1 to 10,000,000	Coordinates expressed as (*x* , *y*) pairs, defined in the Cartesian plane.	NULLS LAST
GEOGRAPHY	1 to 10,000,000	Coordinates expressed in longitude/latitude angular values, measured in degrees	NULLS LAST
UUID			
UUID	16	Stores universally unique identifiers (UUIDs).	NULLS FIRST
Complex			

ARRAY	1 to 32,000,000	Collection of values of a primitive or complex type.	Native array: same as the element type Non-native array: cannot be used to order projections
ROW	1 to 32,000,000	Structure of property-value pairs.	Cannot be used to order projections
SET	1 to 32,000,000	Collection of unique values of a primitive type.	Same as the primitive type

In this section

- Binary data types (BINARY and VARBINARY)
- Boolean data type
- Character data types (CHAR and VARCHAR)
- Date/time data types
- Long data types
- Numeric data types
- Spatial data types
- UUID data type
- Data type coercion
- Data type coercion chart
- Complex types
- Data type mappings between Vertica and Oracle

Binary data types (BINARY and VARBINARY)

Store raw-byte data, such as IP addresses, up to 65000 bytes. The BINARY and BINARY VARYING (VARBINARY) data types are collectively referred to as *binary string types* and the values of binary string types are referred to as *binary strings* . A binary string is a sequence of octets or bytes.

BYTEA and RAW are synonyms for VARBINARY .

Syntax

```
BINARY ( length )
{ VARBINARY | BINARY VARYING | BYTEA | RAW } ( max-length )
```

Arguments

length* , *max-length

The length of the string or column width, in bytes (octets).

BINARY and VARBINARY data types

BINARY and VARBINARY data types have the following attributes:

- BINARY : A fixed-width string of *length* bytes, where the number of bytes is declared as an optional specifier to the type. If *length* is omitted, the default is 1. Where necessary, values are right-extended to the full width of the column with the zero byte. For example:

```
=> SELECT TO_HEX('ab'::BINARY(4));
 to_hex
----------
 61620000
```

- VARBINARY : A variable-width string up to a length of *max-length* bytes, where the maximum number of bytes is declared as an optional specifier to the type. The default is the default attribute size, which is 80, and the maximum length is 65000 bytes. VARBINARY values are not extended to the full width of the column. For example:

```
=> SELECT TO_HEX('ab'::VARBINARY(4));
 to_hex
--------
 6162
```

Input formats

You can use several formats when working with binary values. The hexadecimal format is generally the most straightforward and is emphasized in Vertica documentation.

Binary values can also be represented in octal format by prefixing the value with a backslash '\' .

> **Note**
> If you use vsql , you must use the escape character (\ ) when you insert another backslash on input; for example, input '\141' as '\\141' .

You can also input values represented by printable characters. For example, the hexadecimal value '0x61' can also be represented by the symbol a .

See Data load .

On input, strings are translated from:

- Hexadecimal representation to a binary value using the function HEX_TO_BINARY .
- Bitstring representation to a binary value using the function BITSTRING_TO_BINARY .

Both functions take a VARCHAR argument and return a VARBINARY value.

Output formats

Like the input format, the output format is a hybrid of octal codes and printable ASCII characters. A byte in the range of printable ASCII characters (the range [0x20, 0x7e]) is represented by the corresponding ASCII character, with the exception of the backslash ('\'), which is escaped as '\\' . All other byte values are represented by their corresponding octal values. For example, the bytes {97,92,98,99}, which in ASCII are {a,\,b,c} , are translated to text as 'a\\bc' .

Binary operators and functions

The binary operators & , ~ , | , and # have special behavior for binary data types, as described in Bitwise operators .

The following aggregate functions are supported for binary data types:

- BIT_AND
- BIT_OR
- BIT_XOR
- MAX
- MIN

BIT_AND , BIT_OR , and BIT_XOR are bit-wise operations that are applied to each non-null value in a group, while MAX and MIN are byte-wise comparisons of binary values.

Like their binary operator counterparts, if the values in a group vary in length, the aggregate functions treat the values as though they are all equal in length by extending shorter values with zero bytes to the full width of the column. For example, given a group containing the values 'ff', null, and 'f' , a binary aggregate ignores the null value and treats the value 'f' as 'f0' . Also, like their binary operator counterparts, these aggregate functions operate on VARBINARY types explicitly and operate on BINARY types implicitly through casts. See Data type coercion operators (CAST) .

Binary versus character data types

The BINARY and VARBINARY binary types are similar to the CHAR and VARCHAR character data types , respectively. They differ as follows:

- Binary data types contain byte strings (a sequence of octets or bytes).
- Character data types contain character strings (text).
- The lengths of binary data types are measured in bytes, while character data types are measured in characters.

Examples

The following example shows HEX_TO_BINARY and TO_HEX usage.

Table t and its projection are created with binary columns:

```
=> CREATE TABLE t (c BINARY(1));
=> CREATE PROJECTION t_p (c) AS SELECT c FROM t;
```

Insert minimum byte and maximum byte values:

```
=> INSERT INTO t values(HEX_TO_BINARY('0x00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFF'));
```

Binary values can then be formatted in hex on output using the TO_HEX function:

```
=> SELECT TO_HEX(c) FROM t;
 to_hex
--------
 00
 ff
(2 rows)
```

The BIT_AND , BIT_OR , and BIT_XOR functions are interesting when operating on a group of values. For example, create a sample table and projections with binary columns:

The example that follows uses table t with a single column of VARBINARY data type:

```
=> CREATE TABLE t ( c VARBINARY(2) );
=> INSERT INTO t values(HEX_TO_BINARY('0xFF00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFFFF'));
=> INSERT INTO t values(HEX_TO_BINARY('0xF00F'));
```

Query table t to see column c output:

```
=> SELECT TO_HEX(c) FROM t;
 TO_HEX
--------
 ff00
 ffff
 f00f
(3 rows)
```

Now issue the bitwise AND operation. Because these are aggregate functions, an implicit GROUP BY operation is performed on results using (ff00&(ffff)&f00f) :

```
=> SELECT TO_HEX(BIT_AND(c)) FROM t;
 TO_HEX
--------
f000
(1 row)
```

Issue the bitwise OR operation on (ff00|(ffff)|f00f) :

```
=> SELECT TO_HEX(BIT_OR(c)) FROM t;
 TO_HEX
--------
ffff
(1 row)
```

Issue the bitwise XOR operation on (ff00#(ffff)#f00f) :

```
=> SELECT TO_HEX(BIT_XOR(c)) FROM t;
 TO_HEX
--------
f0f0
(1 row)
```

Boolean data type

Vertica provides the standard SQL type BOOLEAN, which has two states: true and false. The third state in SQL boolean logic is unknown, which is represented by the NULL value.

Syntax

```
BOOLEAN
```

Parameters

Valid literal data values for input are:

TRUE	't'	'true'	'y'	'yes'	'1'	1
FALSE	'f'	'false'	'n'	'no'	'0'	0

Notes

- Do not confuse the BOOLEAN data type with Logical operators or the Boolean .
- The keywords TRUE and FALSE are preferred and are SQL-compliant.
- A Boolean value of NULL appears last (largest) in ascending order.
- All other values must be enclosed in single quotes.
- Boolean values are output using the letters t and f.

See also

- NULL value
- Data type coercion chart

Character data types (CHAR and VARCHAR)

Stores strings of letters, numbers, and symbols. The CHARACTER (CHAR) and CHARACTER VARYING (VARCHAR) data types are collectively referred to as *character string types* , and the values of character string types are known as *character strings* .

Character data can be stored as fixed-length or variable-length strings. Fixed-length strings are right-extended with spaces on output; variable-length strings are not extended.

String literals in SQL statements must be enclosed in single quotes.

Syntax

```
{ CHAR | CHARACTER } [ (octet-length) ]
{ VARCHAR | CHARACTER VARYING ] } [ (octet-length) ]
```

Arguments

octet-length

Length of the string or column width, declared in bytes (octets).

This argument is optional.

CHAR versus VARCHAR data types

The following differences apply to CHAR and VARCHAR data:

- CHAR is conceptually a fixed-length, blank-padded string. Trailing blanks (spaces) are removed on input and are restored on output. The default length is 1, and the maximum length is 65000 octets (bytes).
- VARCHAR is a variable-length character data type. The default length is 80, and the maximum length is 65000 octets. For string values longer than 65000, use Long data types . Values can include trailing spaces.

Normally, you use VARCHAR for all of string data. Use CHAR when you need fixed-width string output. For example, you can use CHAR columns for data to be transferred to a legacy system that requires fixed-width strings.

Setting maximum length

When you define character columns, specify the maximum size of any string to be stored in a column. For example, to store strings up to 24 octets in length, use one of the following definitions:

```
CHAR(24)    --- fixed-length
VARCHAR(24) --- variable-length
```

The maximum length parameter for VARCHAR and CHAR data types refers to the number of octets that can be stored in that field, not the number of characters (Unicode code points). When using multibyte UTF-8 characters, the fields must be sized to accommodate from 1 to 4 octets per character, depending on the data. If the data loaded into a VARCHAR or CHAR column exceeds the specified maximum size for that column, data is truncated on UTF-8 character boundaries to fit within the specified size. See COPY .

> **Note**
> Remember to include the extra octets required for multibyte characters in the column-width declaration, keeping in mind the 65000 octet column-width limit.

Due to compression in Vertica, the cost of overestimating the length of these fields is incurred primarily at load time and during sorts.

NULL versus NUL

NULL and NUL differ as follows:

- NUL represents a character whose ASCII/Unicode code is 0, sometimes qualified "ASCII NUL".
- NULL means no value, and is true of a field (column) or constant, not of a character.

CHAR , LONG VARCHAR , and VARCHAR string data types accept ASCII NUL values.

In ascending sorts, NULL appears last (largest).

For additional information about NULL ordering, see NULL sort order .

The following example casts the input string containing NUL values to VARCHAR :

```
=> SELECT 'vert\0ica'::CHARACTER VARYING AS VARCHAR;
 VARCHAR
---------
 vert\0ica
(1 row)
```

The result contains 9 characters:

```
=> SELECT LENGTH('vert\0ica'::CHARACTER VARYING);
 length
--------
      9
(1 row)
```

If you use an extended string literal , the length is 8 characters:

```
=> SELECT E'vert\0ica'::CHARACTER VARYING AS VARCHAR;
 VARCHAR
---------
 vertica
(1 row)
=> SELECT LENGTH(E'vert\0ica'::CHARACTER VARYING);
 LENGTH
--------
      8
(1 row)
```

Date/time data types

Vertica supports the full set of SQL date and time data types.

The following rules apply to all date/time data types:

- All have a size of 8 bytes.
- A date/time value of NULL is smallest relative to all other date/time values,.
- Vertica uses Julian dates for all date/time calculations, which can correctly predict and calculate any date more recent than 4713 BC to far into the future, based on the assumption that the average length of the year is 365.2425 days.
- All the date/time data types accept the special literal value NOW to specify the current date and time. For example:

  ```
  => SELECT TIMESTAMP 'NOW';
          ?column?
  ----------------------------
   2020-09-23 08:23:50.42325
  (1 row)
  ```

- By default, Vertica rounds with a maximum precision of six decimal places. You can substitute an integer between 0 and 6 for p to specify your preferred level of precision .

The following table lists specific attributes of date/time data types:

Name	Description	Low Value	High Value	Resolution
DATE	Dates only (no time of day)	~ 25e+15 BC	~ 25e+15 AD	1 day
TIME [(p)]	Time of day only (no date)	00:00:00.00	23:59:60.999999	1 µs
TIMETZ [(p)]	Time of day only, with time zone	00:00:00.00+14	23:59:59.999999-14	1 µs
TIMESTAMP [(p)]	Both date and time, without time zone	290279-12-22 19:59:05.224194 BC	294277-01-09 04:00:54.775806 AD	1 µs
TIMESTAMPTZ [(p)]*	Both date and time, with time zone	290279-12-22 19:59:05.224194 BC UTC	294277-01-09 04:00:54.775806 AD UTC	1 µs
INTERVAL DAY TO SECOND [(p)]	Time intervals	-106751991 days 04:00:54.775807	+-106751991 days 04:00:54.775807	1 µs
INTERVAL YEAR TO MONTH	Time intervals	~ -768e15 yrs	~ 768e15 yrs	1 month

Time zone abbreviations for input

Vertica recognizes the files in /opt/vertica/share/timezonesets as date/time input values and defines the default list of strings accepted in the AT TIME ZONE *zone* parameter. The names are not necessarily used for date/time output—output is driven by the official time zone abbreviations associated with the currently selected time zone parameter setting.

In this section

- DATE
- DATETIME
- INTERVAL
- SMALLDATETIME
- TIME/TIMETZ
- TIME AT TIME ZONE
- TIMESTAMP/TIMESTAMPTZ
- TIMESTAMP AT TIME ZONE

DATE

Consists of a month, day, and year. The following limits apply:

- Lowest value: ~ 25e+15 BC
- Higest value: ~ 25e+15 AD
- Resolution: 1 DAY

See SET DATESTYLE for information about ordering.

> **Note**
> '0000-00-00' is not valid. If you try to insert that value into a DATE or TIMESTAMP field, an error occurs. If you copy '0000-00-00' into a DATE or TIMESTAMP field, Vertica converts the value to 0001-01-01 00:00:00 BC.

Syntax

```
DATE
```

Examples

Example	Description
January 8, 1999	Unambiguous in any datestyle input mode

1999-01-08	ISO 8601; January 8 in any mode (recommended format)
1/8/1999	January 8 in MDY mode; August 1 in DMY mode
1/18/1999	January 18 in MDY mode; rejected in other modes
01/02/03	January 2, 2003 in MDY mode February 1, 2003 in DMY mode February 3, 2001 in YMD mode
1999-Jan-08	January 8 in any mode
Jan-08-1999	January 8 in any mode
08-Jan-1999	January 8 in any mode
99-Jan-08	January 8 in YMD mode, else error
08-Jan-99	January 8, except error in YMD mode
Jan-08-99	January 8, except error in YMD mode
19990108	ISO 8601; January 8, 1999 in any mode
990108	ISO 8601; January 8, 1999 in any mode
1999.008	Year and day of year
J2451187	Julian day
January 8, 99 BC	Year 99 before the Common Era

DATETIME

DATETIME is an alias for TIMESTAMP/TIMESTAMPTZ .

INTERVAL

Measures the difference between two points in time. Intervals can be positive or negative. The INTERVAL data type is SQL:2008 compliant, and supports interval qualifiers that are divided into two major subtypes:

- Year-month : Span of years and months
- Day-time : Span of days, hours, minutes, seconds, and fractional seconds

Intervals are represented internally as some number of microseconds and printed as up to 60 seconds, 60 minutes, 24 hours, 30 days, 12 months, and as many years as necessary. You can control the output format of interval units with SET INTERVALSTYLE and SET DATESTYLE .

Syntax

```
INTERVAL 'interval-literal' [ interval-qualifier ] [ (p) ]
```

Parameters

interval-literal

A character string that expresses an interval, conforming to this format:

```
[-] { quantity subtype-unit }[...] [ AGO ]
```

For details, see Interval literal .

interval-qualifier

Optionally specifies how to interpret and format an interval literal for output, and, optionally, sets precision. If omitted, the default is DAY TO SECOND(6) . For details, see Interval qualifier .

p

Specifies precision of the seconds field, where *p* is an integer between 0 - 6. For details, see Specifying interval precision .

Default: 6

Limits

Name	Low Value	High Value	Resolution
INTERVAL DAY TO SECOND [(*p*)]	-106751991 days 04:00:54.775807	+/-106751991 days 04:00:54.775807	1 microsecond
INTERVAL YEAR TO MONTH	~/ -768e15 yrs	~ 768e15 yrs	1 month

In this section

- Setting interval unit display
- Specifying interval input
- Controlling interval format
- Specifying interval precision
- Fractional seconds in interval units
- Processing signed intervals
- Casting with intervals
- Operations with intervals

Setting interval unit display

SET INTERVALSTYLE and SET DATESTYLE control the output format of interval units.

Important
DATESTYLE settings supersede INTERVALSTYLE. If DATESTYLE is set to SQL, interval unit display always conforms to the SQL:2008 standard, which omits interval unit display. If DATESTYLE is set to ISO, you can use SET INTERVALSTYLE to omit or display interval unit display, as described below.

Omitting interval units

To omit interval units from the output, set INTERVALSTYLE to PLAIN . This is the default setting, which conforms with the SQL:2008 standard:

```
=> SET INTERVALSTYLE TO PLAIN;
SET
=> SELECT INTERVAL '3 2';
?column?
----------
 3 02:00
```

When INTERVALSTYLE is set to PLAIN , units are omitted from the output, even if the query specifies input units:

```
=> SELECT INTERVAL '3 days 2 hours';
 ?column?
----------
 3 02:00
```

If DATESTYLE is set to SQL , Vertica conforms with SQL:2008 standard and always omits interval units from output:

```
=> SET DATESTYLE TO SQL;
SET
=> SET INTERVALSTYLE TO UNITS;
SET
=> SELECT INTERVAL '3 2';
 ?column?
----------
 3 02:00
```

Displaying interval units

To enable display of interval units, DATESTYLE must be set to ISO. You can then display interval units by setting INTERVALSTYLE to UNITS :

```
=> SET DATESTYLE TO ISO;
SET
=> SET INTERVALSTYLE TO UNITS;
SET
=> SELECT INTERVAL '3 2';
 ?column?
----------------
 3 days 2 hours
```

Checking INTERVALSTYLE and DATESTYLE settings

Use SHOW statements to check INTERVALSTYLE and DATESTYLE settings:

```
=> SHOW INTERVALSTYLE;
    name      | setting
--------------+---------
 intervalstyle | units
=> SHOW DATESTYLE;
  name    | setting
----------+----------
 datestyle | ISO, MDY
```

Specifying interval input

Interval values are expressed through interval literals . An interval literal is composed of one or more interval fields, where each field represents a span of days and time, or years and months, as follows:

```
[-] { quantity subtype-unit }[...] [AGO]
```

Using subtype units

Subtype units are optional for day-time intervals; they must be specified for year-month intervals.

For example, the first statement below implicitly specifies days and time; the second statement explicitly identifies day and time units. Both statements return the same result:

```
=> SET INTERVALSTYLE TO UNITS;
=> SELECT INTERVAL '1 12:59:10:05';
     ?column?
--------------------
 1 day 12:59:10.005
(1 row)

=> SELECT INTERVAL '1 day 12 hours 59 min 10 sec 5 milliseconds';
     ?column?
--------------------
 1 day 12:59:10.005
(1 row)
```

The following two statements add 28 days and 4 weeks to the current date, respectively. The intervals in both cases are equal and the statements return the same result. However, in the first statement, the interval literal omits the subtype (implicitly days); in the second statement, the interval literal must include the subtype unit weeks :

```
=> SELECT CURRENT_DATE;
 ?column?
------------
 2016-08-15
(1 row)

=> SELECT CURRENT_DATE + INTERVAL '28';
     ?column?
---------------------
 2016-09-12 00:00:00
(1 row)

dbadmin=> SELECT CURRENT_DATE + INTERVAL '4 weeks';
     ?column?
---------------------
 2016-09-12 00:00:00
(1 row)
```

An interval literal can include day-time and year-month fields. For example, the following statement adds an interval of 4 years, 4 weeks, 4 days and 14 hours to the current date. The years and weeks fields must include subtype units; the days and hours fields omit them:

```
> SELECT CURRENT_DATE + INTERVAL '4 years 4 weeks 4 14';
     ?column?
---------------------
 2020-09-15 14:00:00
(1 row)
```

Omitting subtype units

You can specify quantities of days, hours, minutes, and seconds without specifying units. Vertica recognizes colons in interval literals as part of the timestamp:

```
=> SELECT INTERVAL '1 4 5 6';
 ?column?
------------
 1 day 04:05:06
=> SELECT INTERVAL '1 4:5:6';
 ?column?
------------
 1 day 04:05:06
=> SELECT INTERVAL '1 day 4 hour 5 min 6 sec';
 ?column?
------------
 1 day 04:05:06
```

If Vertica cannot determine the units, it applies the quantity to any missing units based on the interval qualifier. In the next two examples, Vertica uses the default interval qualifier (DAY TO SECOND(6)) and assigns the trailing 1 to days, since it has already processed hours, minutes, and seconds in the output:

```
=> SELECT INTERVAL '4:5:6 1';
 ?column?
------------
 1 day 04:05:06
=> SELECT INTERVAL '1 4:5:6';
 ?column?
------------
 1 day 04:05:06
```

In the next two examples, Vertica recognizes 4:5 as hours:minutes . The remaining values in the interval literal are assigned to the missing units: 1 is assigned to days and 2 is assigned to seconds.

```
SELECT INTERVAL '4:5 1 2';
 ?column?
------------
 1 day 04:05:02
=> SELECT INTERVAL '1 4:5 2';
 ?column?
------------
 1 day 04:05:02
```

Specifying the interval qualifier can change how Vertica interprets 4:5 :

```
=> SELECT INTERVAL '4:5' MINUTE TO SECOND;
 ?column?
------------
 00:04:05
```

Controlling interval format

Interval qualifiers specify a range of options that Vertica uses to interpret and format an interval literal . The interval qualifier can also specify precision. Each interval qualifier is composed of one or two units:

```
unit[p] [ TO unit[p] ]
```

where:

- *unit* specifies a day-time or year-month subtype .
- *p* specifies precision, an integer between 0 and 6. In general, precision only applies to SECOND units. The default precision for SECOND is 6. For details, see Specifying interval precision .

If an interval omits an interval qualifier, Vertica uses the default DAY TO SECOND(6) .

Interval qualifier categories

Interval qualifiers belong to one of the following categories:

- Year-month : Span of years and months
- Day-time : Span of days, hours, minutes, seconds, and fractional seconds

> **Note**
> All examples below assume that INTERVALSTYLE is set to plain.

Year-Month

Vertica supports two year-month subtypes: YEAR and MONTH .

In the following example, YEAR TO MONTH qualifies the interval literal 1 2 to indicate a span of 1 year and two months:

```
=> SELECT interval '1 2' YEAR TO MONTH;
 ?column?
----------
 1-2
(1 row)
```

If you omit the qualifier, Vertica uses the default interval qualifier DAY TO SECOND and returns a different result:

```
=> SELECT interval '1 2';
 ?column?
----------
 1 02:00
(1 row)
```

The following example uses the interval qualifier YEAR . In this case, Vertica extracts only the year from the interval literal 1y 10m :

```
=> SELECT INTERVAL '1y 10m' YEAR;
 ?column?
----------
 1
(1 row)
```

In the next example, the interval qualifier MONTH converts the same interval literal to months:

```
=> SELECT INTERVAL '1y 10m' MONTH;
 ?column?
----------
 22
(1 row)
```

Day-time

Vertica supports four day-time subtypes: DAY , HOUR , MINUTE , and SECOND .

In the following example, the interval qualifier DAY TO SECOND(4) qualifies the interval literal 1h 3m 6s 5msecs 57us . The qualifier also sets precision on seconds to 4:

```
=> SELECT INTERVAL '1h 3m 6s 5msecs 57us' DAY TO SECOND(4);
   ?column?
---------------
 01:03:06.0051
(1 row)
```

If no interval qualifier is specified, Vertica uses the default subtype DAY TO SECOND(6) , regardless of how you specify the interval literal. For example, as an extension to SQL:2008, both of the following commands return 910 days :

```
=> SELECT INTERVAL '2-6';
    ?column?
-----------------
 910
=> SELECT INTERVAL '2 years 6 months';
 ?column?
-----------------
 910
```

An interval qualifier can extract other values from the input parameters. For example, the following command extracts the HOUR value from the interval literal 3 days 2 hours :

```
=> SELECT INTERVAL '3 days 2 hours' HOUR;
 ?column?
----------
 74
```

The primary day/time (DAY TO SECOND) and year/month (YEAR TO MONTH) subtype ranges can be restricted to more specific range of types by an interval qualifier. For example, HOUR TO MINUTE is a limited form of day/time interval, which can be used to express time zone offsets.

```
=> SELECT INTERVAL '1 3' HOUR to MINUTE;
 ?column?
---------------
 01:03
```

hh:mm:ss and hh:mm formats are used only when at least two of the fields specified in the interval qualifier are non-zero and there are no more than 23 hours or 59 minutes:

```
=> SELECT INTERVAL '2 days 12 hours 15 mins' DAY TO MINUTE;
 ?column?
--------------
 2 12:15
=> SELECT INTERVAL '15 mins 20 sec' MINUTE TO SECOND;
 ?column?
----------
 15:20
=> SELECT INTERVAL '1 hour 15 mins 20 sec' MINUTE TO SECOND;
 ?column?
-----------------
 75:20
```

Specifying interval precision

In general, interval precision only applies to seconds. If no precision is explicitly specified, Vertica rounds precision to a maximum of six decimal places. For example:

```
=> SELECT INTERVAL '2 hours 4 minutes 3.709384766 seconds' DAY TO SECOND;
    ?column?
-----------------
 02:04:03.709385
(1 row)
```

Vertica lets you specify interval precision in two ways:

- After the INTERVAL keyword
- After the SECOND unit of an interval qualifier, one of the following:
  - DAY TO SECOND
  - HOUR TO SECOND
  - MINUTE TO SECOND
  - SECOND

For example, the following statements use both methods to set precision, and return identical results:

```
=> SELECT INTERVAL(4) '2 hours 4 minutes 3.709384766 seconds' DAY TO SECOND;
   ?column?
---------------
 02:04:03.7094
(1 row)

=> SELECT INTERVAL '2 hours 4 minutes 3.709384766 seconds' DAY TO SECOND(4);
   ?column?
---------------
 02:04:03.7094
(1 row)
```

If the same statement specifies precision more than once, Vertica uses the lesser precision. For example, the following statement specifies precision twice: the INTERVAL keyword specifies precision of 1, while the interval qualifier SECOND specifies precision of 2. Vertica uses the lesser precision of 1:

```
=> SELECT INTERVAL(1) '1.2467' SECOND(2);
 ?column?
----------
 1.2 secs
```

Setting precision on interval table columns

If you create a table with an interval column, the following restrictions apply to the column definition:

- You can set precision on the INTERVAL keyword only if you omit specifying an interval qualifier. If you try to set precision on the INTERVAL keyword and include an interval qualifier, Vertica returns an error.
- You can set precision only on the last unit of an interval qualifier. For example:

```
CREATE TABLE public.testint2
(
   i INTERVAL HOUR TO SECOND(3)
);
```

If you specify precision on another unit, Vertica discards it when it saves the table definition.

Fractional seconds in interval units

Vertica supports intervals in milliseconds (hh:mm:ss:ms), where 01:02:03:25 represents 1 hour, 2 minutes, 3 seconds, and 025 milliseconds. Milliseconds are converted to fractional seconds as in the following example, which returns 1 day, 2 hours, 3 minutes, 4 seconds, and 25.5 milliseconds:

```
=> SELECT INTERVAL '1 02:03:04:25.5';
 ?column?
------------
1 day 02:03:04.0255
```

Vertica allows fractional minutes. The fractional minutes are rounded into seconds:

```
=> SELECT INTERVAL '10.5 minutes';
 ?column?
------------
 00:10:30
=> select interval '10.659 minutes';
 ?column?
-------------
 00:10:39.54
=> select interval '10.3333333333333 minutes';
 ?column?
----------
 00:10:20
```

Considerations

- An INTERVAL can include only the subset of units that you need; however, year/month intervals represent calendar years and months with no fixed number of days, so year/month interval values cannot include days, hours, minutes. When year/month values are specified for day/time intervals, the intervals extension assumes 30 days per month and 365 days per year. Since the length of a given month or year varies, day/time intervals are never output as months or years, only as days, hours, minutes, and so on.
- Day/time and year/month intervals are logically independent and cannot be combined with or compared to each other. In the following example, an interval-literal that contains DAYS cannot be combined with the YEAR TO MONTH type:

  ```
  => SELECT INTERVAL '1 2 3' YEAR TO MONTH;
  ERROR 3679:  Invalid input syntax for interval year to month: "1 2 3"
  ```

- Vertica accepts intervals up to 2^63 – 1 microseconds or months (about 18 digits).
- INTERVAL YEAR TO MONTH can be used in an analytic RANGE window when the ORDER BY column type is TIMESTAMP/TIMESTAMP WITH TIMEZONE , or DATE . Using TIME/TIME WITH TIMEZONE are not supported.
- You can use INTERVAL DAY TO SECOND when the ORDER BY column type is TIMESTAMP/TIMESTAMP WITH TIMEZONE , DATE , and TIME/TIME WITH TIMEZONE .

Examples

Examples in this section assume that INTERVALSTYLE is set to PLAIN, so results omit subtype units . Interval values that omit an interval qualifier use the default to DAY TO SECOND(6).

```
=> SELECT INTERVAL '00:2500:00';
 ?column?
----------
 1 17:40
(1 row)

=> SELECT INTERVAL '2500' MINUTE TO SECOND;
 ?column?
----------
 2500
```

```
(1 row)

=> SELECT INTERVAL '2500' MINUTE;
 ?column?
----------
 2500
(1 row)

=> SELECT INTERVAL '28 days 3 hours' HOUR TO SECOND;
 ?column?
----------
 675:00
(1 row)

=> SELECT INTERVAL(3) '28 days 3 hours';
 ?column?
----------
 28 03:00
(1 row)

=> SELECT INTERVAL(3) '28 days 3 hours 1.234567';
    ?column?
-----------------
 28 03:01:14.074
(1 row)

=> SELECT INTERVAL(3) '28 days 3 hours 1.234567 sec';
    ?column?
-----------------
 28 03:00:01.235
(1 row)

=> SELECT INTERVAL(3) '28 days 3.3 hours' HOUR TO SECOND;
 ?column?
----------
 675:18
(1 row)

=> SELECT INTERVAL(3) '28 days 3.35 hours' HOUR TO SECOND;
 ?column?
----------
 675:21
(1 row)

=> SELECT INTERVAL(3) '28 days 3.37 hours' HOUR TO SECOND;
 ?column?
-----------
 675:22:12
(1 row)

=> SELECT INTERVAL '1.234567 days' HOUR TO SECOND;
   ?column?
---------------
 29:37:46.5888
(1 row)

=> SELECT INTERVAL '1.23456789 days' HOUR TO SECOND;
    ?column?
-----------------
 29:37:46.665696
(1 row)
```

```
=> SELECT INTERVAL(3) '1.23456789 days' HOUR TO SECOND;
   ?column?
--------------
 29:37:46.666
(1 row)

=> SELECT INTERVAL(3) '1.23456789 days' HOUR TO SECOND(2);
  ?column?
-------------
 29:37:46.67
(1 row)

=> SELECT INTERVAL(3) '01:00:01.234567' as "one hour+";
  one hour+
--------------
 01:00:01.235
(1 row)

=> SELECT INTERVAL(3) '01:00:01.234567' = INTERVAL(3) '01:00:01.234567';
 ?column?
----------
 t
(1 row)

=> SELECT INTERVAL(3) '01:00:01.234567' = INTERVAL '01:00:01.234567';
 ?column?
----------
 f
(1 row)

=> SELECT INTERVAL(3) '01:00:01.234567' = INTERVAL '01:00:01.234567'
HOUR TO SECOND(3);
 ?column?
----------
 t
(1 row)

=> SELECT INTERVAL(3) '01:00:01.234567' = INTERVAL '01:00:01.234567'
MINUTE TO SECOND(3);
 ?column?
----------
 t
(1 row)

=> SELECT INTERVAL '255 1.1111' MINUTE TO SECOND(3);
  ?column?
------------
 255:01.111
(1 row)

=> SELECT INTERVAL '@ - 5 ago';
 ?column?
----------
 5
(1 row)

=> SELECT INTERVAL '@ - 5 minutes ago';
 ?column?
----------
 00:05
```

```
(1 row)

=> SELECT INTERVAL '@ 5 minutes ago';
 ?column?
----------
 -00:05
(1 row)

=> SELECT INTERVAL '@ ago -5 minutes';
 ?column?
----------
 00:05
(1 row)

=> SELECT DATE_PART('month', INTERVAL '2-3' YEAR TO MONTH);
 DATE_PART
-----------
         3
(1 row)

=> SELECT FLOOR((TIMESTAMP '2005-01-17 10:00'
  - TIMESTAMP '2005-01-01')
/ INTERVAL '7');
 FLOOR
-------
     2
(1 row)
```

Processing signed intervals

In the SQL:2008 standard, a minus sign before an interval-literal or as the first character of the interval-literal negates the entire literal, not just the first component. In Vertica, a leading minus sign negates the entire interval, not just the first component. The following commands both return the same value:

```
=> SELECT INTERVAL '-1 month - 1 second';
 ?column?
----------
 -29 days 23:59:59

=> SELECT INTERVAL -'1 month - 1 second';
 ?column?
----------
 -29 days 23:59:59
```

Use one of the following commands instead to return the intended result:

```
=> SELECT INTERVAL -'1 month 1 second';
 ?column?
----------
 -30 days 1 sec
=> SELECT INTERVAL -'30 00:00:01';
 ?column?
----------
 -30 days 1 sec
```

Two negatives together return a positive:

```
=> SELECT INTERVAL -'-1 month - 1 second';
 ?column?
----------
 29 days 23:59:59
=> SELECT INTERVAL -'-1 month 1 second';
 ?column?
----------
 30 days 1 sec
```

You can use the year-month syntax with no spaces. Vertica allows the input of negative months but requires two negatives when paired with years.

```
=> SELECT INTERVAL '3-3' YEAR TO MONTH;
 ?column?
----------
 3 years 3 months
=> SELECT INTERVAL '3--3' YEAR TO MONTH;
 ?column?
----------
 2 years 9 months
```

When the interval-literal looks like a year/month type, but the type is day/second, or vice versa, Vertica reads the interval-literal from left to right, where number-number is years-months, and number <space> <signed number> is whatever the units specify. Vertica processes the following command as (–) 1 year 1 month = (–) 365 + 30 = –395 days:

```
=> SELECT INTERVAL '-1-1' DAY TO HOUR;
 ?column?
----------
 -395 days
```

If you insert a space in the interval-literal, Vertica processes it based on the subtype DAY TO HOUR : (–) 1 day – 1 hour = (–) 24 – 1 = –23 hours:

```
=> SELECT INTERVAL '-1 -1' DAY TO HOUR;
 ?column?
----------
 -23 hours
```

Two negatives together returns a positive, so Vertica processes the following command as (–) 1 year – 1 month = (–) 365 – 30 = –335 days:

```
=> SELECT INTERVAL '-1--1' DAY TO HOUR;
  ?column?
----------
 -335 days
```

If you omit the value after the hyphen, Vertica assumes 0 months and processes the following command as 1 year 0 month –1 day = 365 + 0 – 1 = –364 days:

```
=> SELECT INTERVAL '1- -1' DAY TO HOUR;
  ?column?
----------
 364 days
```

Casting with intervals

You can use CAST to convert strings to intervals, and vice versa.

String to interval

You cast a string to an interval as follows:

```
CAST( [ INTERVAL[(p)] ] [-] ] interval-literal AS INTERVAL[(p)] interval-qualifier )
```

For example:

```
=> SELECT CAST('3700 sec' AS INTERVAL);
 ?column?
----------
 01:01:40
```

You can cast intervals within day-time or the year-month subtypes but not between them:

```
=> SELECT CAST(INTERVAL '4440' MINUTE as INTERVAL);
  ?column?
----------
 3 days 2 hours
=> SELECT CAST(INTERVAL -'01:15' as INTERVAL MINUTE);
 ?column?
----------
 -75 mins
```

Interval to string

You cast an interval to a string as follows:

```
CAST( (SELECT interval ) AS VARCHAR[(n)] )
```

For example:

```
=> SELECT CONCAT(
  'Tomorrow at this time: ',
  CAST((SELECT INTERVAL '24 hours') + CURRENT_TIMESTAMP(0) AS VARCHAR));
                  CONCAT
-------------------------------------------------
 Tomorrow at this time: 2016-08-17 08:41:23-04
(1 row)
```

Operations with intervals

If you divide an interval by an interval, you get a FLOAT :

```
=> SELECT INTERVAL '28 days 3 hours' HOUR(4) / INTERVAL '27 days 3 hours' HOUR(4);
 ?column?
------------
 1.036866359447
```

An INTERVAL divided by FLOAT returns an INTERVAL :

```
=> SELECT INTERVAL '3' MINUTE / 1.5;
 ?column?
------------
 2 mins
```

INTERVAL MODULO (remainder) INTERVAL returns an INTERVAL :

```
=> SELECT INTERVAL '28 days 3 hours' HOUR % INTERVAL '27 days 3 hours' HOUR;
  ?column?
------------
 24 hours
```

If you add INTERVAL and TIME , the result is TIME , modulo 24 hours:

```
=> SELECT INTERVAL '1' HOUR + TIME '1:30';
 ?column?
------------
 02:30:00
```

SMALLDATETIME

SMALLDATETIME is an alias for TIMESTAMP/TIMESTAMPTZ .

TIME/TIMETZ

Stores the specified time of day. TIMETZ is the same as TIME WITH TIME ZONE : both data types store the UTC offset of the specified time.

Syntax

```
TIME [ (p) ] [ { WITHOUT | WITH } TIME ZONE ] 'input-string' [ AT TIME ZONE zone ]
```

Parameters

p

Optional precision value that specifies the number of fractional digits retained in the seconds field, an integer value between 0 and 6. If you omit specifying precision, Vertica returns up to 6 fractional digits.

WITHOUT TIME ZONE

Ignore any time zone in the input string and use a value without a time zone (default).

WITH TIME ZONE

Convert the time to UTC. If the input string includes a time zone, use its UTC offset for the conversion. If the input string omits a time zone, Vertica uses the UTC offset of the time zone that is configured for your system .

input-string

See Input String below.

AT TIME ZONE *zone*

See TIME AT TIME ZONE and TIMESTAMP AT TIME ZONE .

TIME versus TIMETZ

TIMETZ and TIMESTAMPTZ are not parallel SQL constructs. TIMESTAMPTZ records a time and date in GMT, converting from the specified TIME ZONE.TIMETZ records the specified time and the specified time zone, in minutes, from GMT.

Limits

Name	Low Value	High Value	Resolution
TIME [*p*]	00:00:00.00	23:59:60.999999	1 µs
TIME [*p*] WITH TIME ZONE	00:00:00.00+14	23:59:59.999999-14	1 µs

Input string

A TIME input string can be set to any of the formats shown below:

Example	Description
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	Same as 04:05; AM does not affect value
04:05 PM	Same as 16:05
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601

04:05:06 PST	Time zone specified by name

Data type coercion

You can cast a TIME or TIMETZ interval to a TIMESTAMP . This returns the local date and time as follows:

```
=> SELECT (TIME '3:01am')::TIMESTAMP;
      ?column?
---------------------
 2012-08-30 03:01:00
(1 row)

=> SELECT (TIMETZ '3:01am')::TIMESTAMP;
     ?column?
---------------------
 2012-08-22 03:01:00
(1 row)
```

Casting the same TIME or TIMETZ interval to a TIMESTAMPTZ returns the local date and time, appended with the UTC offset—in this example, -05 :

```
=> SELECT (TIME '3:01am')::TIMESTAMPTZ;
       ?column?
------------------------
 2016-12-08 03:01:00-05
(1 row)
```

TIME AT TIME ZONE

Converts the specified TIME to the time in another time zone.

Syntax

```
TIME [WITH TIME ZONE] 'input-string' AT TIME ZONE 'zone'
```

Parameters

WITH TIME ZONE

Converts the input string to UTC, using the UTC offset for the specified time zone. If the input string omits a time zone, Vertica uses the UTC offset of the time zone that is configured for your system , and converts the input string accordingly

zone

Specifies the time zone to use in the conversion, either as a literal or interval that specifies UTC offset:

- AT TIME ZONE INTERVAL ' *utc-offset* '
- AT TIME ZONE ' *time-zone-literal* '

For details, see Specifying Time Zones below.

> **Note**
> Vertica treats literals TIME ZONE and TIMEZONE as synonyms.

Specifying time zones

You can specify time zones in two ways:

- A string literal such as America/Chicago or PST
- An interval that specifies a UTC offset—for example, INTERVAL '-08:00'

It is generally good practice to specify time zones with literals that indicate a geographic location. Vertica makes the necessary seasonal adjustments, and thereby avoids inconsistent results. For example, the following two queries are issued when daylight time is in effect. Because the local UTC offset during daylight time is -04 , both queries return the same results:

```
=> SELECT CURRENT_TIME(0) "EDT";
    EDT
-------------
 12:34:35-04
(1 row)

=> SELECT CURRENT_TIME(0) AT TIME ZONE 'America/Denver' "Mountain Time";
 Mountain Time
---------------
 10:34:35-06
(1 row)
=> SELECT CURRENT_TIME(0) AT TIME ZONE INTERVAL '-06:00' "Mountain Time";
 Mountain Time
---------------
 10:34:35-06
(1 row)
```

If you issue a use the UTC offset in a similar query when standard time is in effect, you must adjust the UTC offset accordingly—for Denver time, to -07—otherwise, Vertica returns a different (and erroneous) result:

```
=> SELECT CURRENT_TIME(0) "EST";
    EST
-------------
 14:18:22-05
(1 row)

=> SELECT CURRENT_TIME(0) AT TIME ZONE INTERVAL '-06:00' "Mountain Time";
 Mountain Time
---------------
 13:18:22-06
(1 row)
```

You can show and set the session's time zone with SHOW TIMEZONE and SET TIME ZONE, respectively:

```
=> SHOW TIMEZONE;
  name   |     setting
----------+-------------------
 timezone | America/New_York
(1 row)

=> SELECT CURRENT_TIME(0) "Eastern Daylight Time";
 Eastern Daylight Time
-----------------------
 12:18:24-04
(1 row)

=> SET TIMEZONE 'America/Los_Angeles';
SET

=> SELECT CURRENT_TIME(0) "Pacific Daylight Time";
 Pacific Daylight Time
-----------------------
 09:18:24-07
(1 row)
```

Time zone literals

To view the default list of valid literals, see the files in the following directory:

```
opt/vertica/share/timezonesets
```

For example:

```
$ cat Antarctica.txt
...
# src/timezone/tznames/Antarctica.txt
#

AWST    28800    # Australian Western Standard Time
              #     (Antarctica/Casey)
              #     (Australia/Perth)
...

NZST    43200    # New Zealand Standard Time
              #     (Antarctica/McMurdo)
              #     (Pacific/Auckland)
ROTT   -10800    # Rothera Time
              #     (Antarctica/Rothera)
SYOT    10800    # Syowa Time
              #     (Antarctica/Syowa)
VOST    21600    # Vostok time
              #     (Antarctica/Vostok)
```

Examples

The following example assumes that local time is EST (Eastern Standard Time). The query converts the specified time to MST (mountain standard time):

```
=> SELECT CURRENT_TIME(0);
  timezone
-------------
 10:10:56-05
(1 row)

=> SELECT TIME '10:10:56' AT TIME ZONE 'America/Denver' "Denver Time";
 Denver Time
-------------
 08:10:56-07
(1 row)
```

The next example adds a time zone literal to the input string—in this case, Europe/Vilnius —and converts the time to MST:

```
=> SELECT TIME '09:56:13 Europe/Vilnius' AT TIME ZONE 'America/Denver';
 Denver Time
-------------
 00:56:13-07
(1 row)
```

See also

- TIMESTAMP AT TIME ZONE
- TZ environment variable
- Using time zones with Vertica
- Sources for Time Zone and Daylight Saving Time Data

TIMESTAMP/TIMESTAMPTZ

Stores the specified date and time. TIMESTAMPTZ is the same as TIMESTAMP WITH TIME ZONE : both data types store the UTC offset of the specified time.

TIMESTAMP is an alias for DATETIME and SMALLDATETIME .

Syntax

```
TIMESTAMP [ (p) ] [ { WITHOUT | WITH } TIME ZONE ] 'input-string' [AT TIME ZONE zone ]
TIMESTAMPTZ [ (p) ] 'input-string' [ AT TIME ZONE zone ]
```

Parameters

p

Optional precision value that specifies the number of fractional digits retained in the seconds field, an integer value between 0 and 6. If you omit specifying precision, Vertica returns up to 6 fractional digits.

WITHOUT TIME ZONE
WITH TIME ZONE

Specifies whether to include a time zone with the stored value:

- WITHOUT TIME ZONE (default): Specifiesthat *input-string* does not include a time zone. If the input string contains a time zone, Vertica ignores this qualifier. Instead, it conforms to WITH TIME ZONE behavior.
- WITH TIME ZONE : Specifies to convert *input-string* to UTC, using the UTC offset for the specified time zone. If the input string omits a time zone, Vertica uses the UTC offset of the time zone that is configured for your system .

input-string

See Input String below.

AT TIME ZONE *zone*

See TIMESTAMP AT TIME ZONE .

Limits

In the following table, values are rounded. See Date/time data types for more detail.

Name	Low Value	High Value	Resolution
TIMESTAMP [(p)] [WITHOUT TIME ZONE]	290279 BC	294277 AD	1 µs
TIMESTAMP [(p)] WITH TIME ZONE	290279 BC	294277 AD	1 µs

Input string

The date/time input string concatenates a date and a time. The input string can include a time zone, specified as a literal such as America/Chicago , or as a UTC offset.

The following list represents typical date/time input variations:

- 1999-01-08 04:05:06
- 1999-01-08 04:05:06 -8:00
- January 8 04:05:06 1999 PST

> **Note**
> 0000-00-00 is invalid input. If you try to insert that value into a DATE or TIMESTAMP field, an error occurs. If you copy 0000-00-00 into a DATE or TIMESTAMP field, Vertica converts the value to 0001-01-01 00:00:00 BC .

The input string can also specify the calendar era, either AD (default) or BC . If you omit the calendar era, Vertica assumes the current calendar era (AD). The calendar era typically follows the time zone; however, the input string can include it in various locations. For example, the following queries return the same results:

```
=> SELECT TIMESTAMP WITH TIME ZONE 'March 1, 44 12:00 CET BC ' "Caesar's Time of Death EST";
 Caesar's Time of Death EST
----------------------------
 0044-03-01 06:00:00-05 BC
(1 row)

=> SELECT TIMESTAMP WITH TIME ZONE 'March 1, 44 12:00 BC CET' "Caesar's Time of Death EST";
 Caesar's Time of Death EST
----------------------------
 0044-03-01 06:00:00-05 BC
(1 row)
```

Examples

```
=> SELECT (TIMESTAMP '2014-01-17 10:00' - TIMESTAMP '2014-01-01');
 ?column?
```

```
 ?column?
----------
 16 10:00
(1 row)

=> SELECT (TIMESTAMP '2014-01-17 10:00' - TIMESTAMP '2014-01-01') / 7;
     ?column?
-------------------
 2 08:17:08.571429
(1 row)

=> SELECT TIMESTAMP '2009-05-29 15:21:00.456789'-TIMESTAMP '2009-05-28';
     ?column?
-------------------
 1 15:21:00.456789
(1 row)

=> SELECT (TIMESTAMP '2009-05-29 15:21:00.456789'-TIMESTAMP '2009-05-28')(3);
    ?column?
----------------
 1 15:21:00.457
(1 row)

=> SELECT '2017-03-18 07:00'::TIMESTAMPTZ(0) + INTERVAL '1.5 day';
        ?column?
------------------------
 2017-03-19 19:00:00-04
(1 row)

=> SELECT (TIMESTAMP '2014-01-17 10:00' - TIMESTAMP '2014-01-01') day;
 ?column?
----------
 16
(1 row)

=> SELECT cast((TIMESTAMP '2014-01-17 10:00' - TIMESTAMP '2014-01-01')
day as integer) / 7;
       ?column?
----------------------
 2.285714285714285714
(1 row)

=> SELECT floor((TIMESTAMP '2014-01-17 10:00' - TIMESTAMP '2014-01-01')
/ interval '7');
 floor
-------
     2
(1 row)

=> SELECT (TIMESTAMP '2009-05-29 15:21:00.456789'-TIMESTAMP '2009-05-28') second;
   ?column?
---------------
 141660.456789
(1 row)

=> SELECT (TIMESTAMP '2012-05-29 15:21:00.456789'-TIMESTAMP '2009-01-01') year;
 ?column?
----------
 3
(1 row)

=> SELECT (TIMESTAMP '2012-05-29 15:21:00.456789'-TIMESTAMP '2009-01-01') month;
```

```
 ?column?
----------
 40
(1 row)

=> SELECT (TIMESTAMP '2012-05-29 15:21:00.456789'-TIMESTAMP '2009-01-01')
year to month;
 ?column?
----------
 3-4
(1 row)

=> SELECT (TIMESTAMP '2012-05-29 15:21:00.456789'-TIMESTAMP '2009-01-01')
second(3);
   ?column?
---------------
 107536860.457
(1 row)

=> SELECT (TIMESTAMP '2012-05-29 15:21:00.456789'-TIMESTAMP '2009-01-01') minute;
 ?column?
----------
 1792281
(1 row)

=> SELECT (TIMESTAMP '2012-05-29 15:21:00.456789'-TIMESTAMP '2009-01-01')
minute to second(3);
    ?column?
----------------
 1792281:00.457
(1 row)

=> SELECT TIMESTAMP 'infinity';
 ?column?
----------
 infinity
(1 row)
```

TIMESTAMP AT TIME ZONE

Converts the specified TIMESTAMP or TIMESTAMPTZ (TIMESTAMP WITH TIMEZONE) to another time zone. Vertica executes AT TIME ZONE differently, depending on whether the date input is a TIMESTAMP or TIMESTAMPTZ . See TIMESTAMP versus TIMESTAMPTZ Behavior below.

Syntax

timestamp-clause AT TIME ZONE '*zone*'

Parameters

[timestamp-clause](/sql-reference/data-types/datetime-data-types/timestamptimestamptz.html)

Specifies the timestamp to convert, either TIMESTAMP or TIMESTAMPTZ .

For details, see TIMESTAMP/TIMESTAMPTZ .

AT TIME ZONE *zone*

Specifies the time zone to use in the timestamp conversion, where *zone* is a literal or interval that specifies a UTC offset:

- AT TIME ZONE INTERVAL ' *utc-offset* '
- AT TIME ZONE ' *time-zone-literal* '

For details, see Specifying Time Zones below.

> **Note**
> Vertica treats literals TIME ZONE and TIMEZONE as synonyms.

TIMESTAMP versus TIMESTAMPTZ behavior

How Vertica interprets AT TIME ZONE depends on whether the date input is a TIMESTAMP or TIMESTAMPTZ :

Date input	Action

TIMESTAMP	If the input string specifies no time zone, Vertica performs two actions:
	1. Converts the input string to the time zone of the AT TIME ZONE argument.
	2. Returns the time for the current session's time zone.
	If the input string includes a time zone, Vertica implicitly casts it to a TIMESTAMPTZ and converts it accordingly (see TIMESTAMPTZ below).
	For example, the following statement specifies a TIMESTAMP with no time zone. Vertica executes the statement as follows:
	1. Converts the input string to PDT (Pacific Daylight Time).
	2. Returns that time in the local time zone, which is three hours later:
	`=> SHOW TIMEZONE;`
	`  name  |  setting`
	`----------+------------------`
	` timezone | America/New_York`
	`(1 row)`
	
	`SELECT TIMESTAMP '2017-3-14 5:30' AT TIME ZONE 'PDT';`
	`       timezone`
	`------------------------`
	` 2017-03-14 08:30:00-04`
	`(1 row)`

TIMESTAMPTZ	Vertica converts the input string to the time zone of the AT TIME ZONE argument and returns that time.
	For example, the following statement specifies a TIMESTAMPTZ data type. The input string omits any time zone expression, so Vertica assumes the input string to be in local time zone (America/New_York) and returns the time of the AT TIME ZONE argument:
	`=> SHOW TIMEZONE;`
	`  name  |  setting`
	`----------+------------------`
	` timezone | America/New_York`
	`(1 row)`
	
	`=> SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';`
	`     timezone`
	`---------------------`
	` 2001-02-16 18:38:40`
	`(1 row)`
	The input string in the next statement explicitly specifies a time zone, so Vertica coerces the TIMESTAMP to a TIMESTAMPTZ and returns the time of the AT TIME ZONE argument:
	`=> SELECT TIMESTAMP '2001-02-16 20:38:40 America/Mexico_City' AT TIME ZONE 'Asia/Tokyo';`
	`     timezone`
	`---------------------`
	` 2001-02-17 11:38:40`
	`(1 row)`

Specifying time zones

You can specify time zones in two ways:

- A string literal such as America/Chicago or PST
- An interval that specifies a UTC offset—for example, INTERVAL '-08:00'

It is generally good practice to specify time zones with literals that indicate a geographic location. Vertica makes the necessary seasonal adjustments, and thereby avoids inconsistent results. For example, the following two queries are issued when daylight time is in effect. Because the local UTC offset during daylight time is -04 , both queries return the same results:

```
=> SELECT TIMESTAMPTZ '2017-03-16 09:56:13' AT TIME ZONE 'America/Denver' "Denver Time";
    Denver Time
---------------------
 2017-03-16 07:56:13
(1 row)

=> SELECT TIMESTAMPTZ '2017-03-16 09:56:13' AT TIME ZONE INTERVAL '-06:00' "Denver Time";
    Denver Time
---------------------
 2017-03-16 07:56:13
(1 row)
```

If you issue a use the UTC offset in a similar query when standard time is in effect, you must adjust the UTC offset accordingly—for Denver time, to -07 —otherwise, Vertica returns a different (and erroneous) result:

```
=> SELECT TIMESTAMPTZ '2017-01-16 09:56:13' AT TIME ZONE 'America/Denver' "Denver Time";
    Denver Time
---------------------
 2017-0-16 07:56:13
(1 row)

=> SELECT TIMESTAMPTZ '2017-01-16 09:56:13' AT TIME ZONE INTERVAL '-06:00' "Denver Time";
    Denver Time
---------------------
 2017-01-16 08:56:13
(1 row)
```

You can show and set the session's time zone with SHOW TIMEZONE and SET TIME ZONE , respectively:

```
=> SHOW TIMEZONE;
  name   |     setting
----------+------------------
 timezone | America/New_York
(1 row)

=> SELECT CURRENT_TIMESTAMP(0) "Eastern Daylight Time";
 Eastern Daylight Time
------------------------
 2017-03-20 12:18:24-04
(1 row)

=> SET TIMEZONE 'America/Los_Angeles';
SET

=> SELECT CURRENT_TIMESTAMP(0) "Pacific Daylight Time";
 Pacific Daylight Time
------------------------
 2017-03-20 09:18:24-07
(1 row)
```

Time zone literals

To view the default list of valid literals, see the files in the following directory:

```
opt/vertica/share/timezonesets
```

For example:

```
$ cat Antarctica.txt
...
# src/timezone/tznames/Antarctica.txt
#

AWST    28800    # Australian Western Standard Time
              #     (Antarctica/Casey)
              #     (Australia/Perth)
...

NZST    43200    # New Zealand Standard Time
              #     (Antarctica/McMurdo)
              #     (Pacific/Auckland)
ROTT   -10800    # Rothera Time
              #     (Antarctica/Rothera)
SYOT    10800    # Syowa Time
              #     (Antarctica/Syowa)
VOST    21600    # Vostok time
              #     (Antarctica/Vostok)
```

See also

- TIME AT TIME ZONE
- TZ environment variable
- Using time zones with Vertica
- Sources for Time Zone and Daylight Saving Time Data

Long data types

Store data up to 32000000 octets. Vertica supports two long data types:

- LONG VARBINARY : Variable-length raw-byte data, such as spatial data. LONG VARBINARY values are not extended to the full width of the column.
- LONG VARCHAR : Variable-length strings, such as log files and unstructured data. LONG VARCHAR values are not extended to the full width of the column.

Use LONG data types only when you need to store data greater than the maximum size of VARBINARY and VARCHAR data types (65 KB). Long data can include unstructured data, online comments or posts, or small log files.

Flex tables have a default LONG VARBINARY __raw__ column, with a NOT NULL constraint. For more information, see Flex tables .

Syntax

```
LONG VARBINARY [(max-length)]
LONG VARCHAR [(octet-length)]
```

Parameters

max-length

Length of the byte string or column width, declared in bytes (octets), up to 32000000.

Default: 1 MB

octet-length

Length of the string or column width, declared in bytes (octets), up to 32000000.

Default: 1 MB

Optimized performance

For optimal performance of LONG data types, Vertica recommends that you:

- Use the LONG data types as *storage only* containers; Vertica supports operations on the content of LONG data types, but does not support all the operations that VARCHAR and VARBINARY take.
- Use VARBINARY and VARCHAR data types, instead of their LONG counterparts, whenever possible. VARBINARY and VARCHAR data types are more flexible and have a wider range of operations.

- Do not sort, segment, or partition projections on LONG data type columns.
- Do not add constraints, such as a primary key, to any LONG VARBINARY or LONG VARCHAR columns.
- Do not join or aggregate any LONG data type columns.

Examples

The following example creates a table user_comments with a LONG VARCHAR column and inserts data into it:

```
=> CREATE TABLE user_comments
                (id INTEGER,
                username VARCHAR(200),
                time_posted TIMESTAMP,
                comment_text LONG VARCHAR(200000));
=> INSERT INTO user_comments VALUES
                (1,
                'User1',
                TIMESTAMP '2013-06-25 12:47:32.62',
                'The weather tomorrow will be cold and rainy and then
                on the day after, the sun will come and the temperature
                will rise dramatically.');
```

Numeric data types

Numeric data types are numbers stored in database columns. These data types are typically grouped by:

- **Exact** numeric types, values where the precision and scale need to be preserved. The exact numeric types are INTEGER , BIGINT , DECIMAL , NUMERIC , NUMBER , and MONEY .
- **Approximate** numeric types, values where the precision needs to be preserved and the scale can be floating. The approximate numeric types are DOUBLE PRECISION , FLOAT , and REAL .

Implicit casts from INTEGER , FLOAT , and NUMERIC to VARCHAR are not supported. If you need that functionality, write an explicit cast using one of the following forms:

```
CAST(numeric-expression AS data-type)
numeric-expression::data-type
```

For example, you can cast a float to an integer as follows:

```
=> SELECT(FLOAT '123.5')::INT;
 ?column?
----------
      124
(1 row)
```

String-to-numeric data type conversions accept formats of quoted constants for scientific notation, binary scaling, hexadecimal, and combinations of numeric-type literals:

- Scientific notation :

  ```
  => SELECT FLOAT '1e10';
    ?column?
  -------------
   10000000000
  (1 row)
  ```

- BINARY scaling :

  ```
  => SELECT NUMERIC '1p10';
   ?column?
  ----------
      1024
  (1 row)
  ```

- hexadecimal:

```
=> SELECT NUMERIC '0x0abc';
 ?column?
----------
    2748
(1 row)
```

In this section

- DOUBLE PRECISION (FLOAT)
- INTEGER
- NUMERIC
- Numeric data type overflow
- Numeric data type overflow with SUM, SUM_FLOAT, and AVG

DOUBLE PRECISION (FLOAT)

Vertica supports the numeric data type DOUBLE PRECISION , which is the IEEE-754 8-byte floating point type, along with most of the usual floating point operations.

Syntax

```
[ DOUBLE PRECISION | FLOAT | FLOAT(n) | FLOAT8 | REAL ]
```

Parameters

> **Note**
> On a machine whose floating-point arithmetic does not follow IEEE-754, these values probably do not work as expected.

Double precision is an inexact, variable-precision numeric type. In other words, some values cannot be represented exactly and are stored as approximations. Thus, input and output operations involving double precision might show slight discrepancies.

- All of the DOUBLE PRECISION data types are synonyms for 64-bit IEEE FLOAT.
- The *n* in FLOAT(n) must be between 1 and 53, inclusive, but a 53-bit fraction is always used. See the IEEE-754 standard for details.
- For exact numeric storage and calculations (money for example), use NUMERIC .
- Floating point calculations depend on the behavior of the underlying processor, operating system, and compiler.
- Comparing two floating-point values for equality might not work as expected.
- While Vertica treats decimal values as FLOAT internally, if a column is defined as FLOAT then you cannot read decimal values from ORC and Parquet files. In those formats, FLOAT and DECIMAL are different types.

Values

COPY accepts floating-point data in the following format:

- Optional leading white space
- An optional plus ("+") or minus sign ("-")
- A decimal number, a hexadecimal number, an infinity, a NAN, or a null value

Decimal Number

A decimal number consists of a non-empty sequence of decimal digits possibly containing a radix character (decimal point "."), optionally followed by a decimal exponent. A decimal exponent consists of an "E" or "e", followed by an optional plus or minus sign, followed by a non-empty sequence of decimal digits, and indicates multiplication by a power of 10.

Hexadecimal Number

A hexadecimal number consists of a "0x" or "0X" followed by a non-empty sequence of hexadecimal digits possibly containing a radix character, optionally followed by a binary exponent. A binary exponent consists of a "P" or "p", followed by an optional plus or minus sign, followed by a non-empty sequence of decimal digits, and indicates multiplication by a power of 2. At least one of radix character and binary exponent must be present.

Infinity

An infinity is either INF or INFINITY , disregarding case.

NaN (Not A Number)

A NaN is NAN (disregarding case) optionally followed by a sequence of characters enclosed in parentheses. The character string specifies the value of NAN in an implementation-dependent manner. (The Vertica internal representation of NAN is 0xfff8000000000000LL on x86 machines.)

When writing infinity or NAN values as constants in a SQL statement, enclose them in single quotes. For example:

```
=> UPDATE table SET x = 'Infinity'
```

> **Note**
> Vertica follows the IEEE definition of NaNs (IEEE 754). The SQL standards do not specify how floating point works in detail.
>
> IEEE defines NaNs as a set of floating point values where each one is not equal to anything, even to itself. A NaN is not greater than and at the same time not less than anything, even itself. In other words, comparisons always return false whenever a NaN is involved.
>
> However, for the purpose of sorting data, NaN values must be placed somewhere in the result. The value generated 'NaN' appears in the context of a floating point number matches the NaN value generated by the hardware. For example, Intel hardware generates (0xfff8000000000000LL), which is technically a Negative, Quiet, Non-signaling NaN.
>
> Vertica uses a different NaN value to represent floating point NULL (0x7fffffffffffffeLL). This is a Positive, Quiet, Non-signaling NaN and is reserved by Vertica

A NaN example follows.

```
=> SELECT CBRT('Nan'); -- cube root
 CBRT
------
 NaN
(1 row)
=> SELECT 'Nan' > 1.0;
 ?column?
----------
 f
(1 row)
```

Null Value

The load file format of a null value is user defined, as described in the COPY command. The Vertica internal representation of a null value is 0x7fffffffffffffffLL. The interactive format is controlled by the vsql printing option null. For example:

```
\pset null '(null)'
```

The default option is not to print anything.

Rules

- -0 == +0
- 1/0 = Infinity
- 0/0 == Nan
- NaN != anything (even NaN)

To search for NaN column values, use the following predicate:

```
... WHERE column != column
```

This is necessary because WHERE *column* = 'Nan' cannot be true by definition.

Sort order (ascending)

- NaN
- -Inf
- numbers
- +Inf
- NULL

Notes

- NULL appears last (largest) in ascending order.

- All overflows in floats generate +/-infinity or NaN, per the IEEE floating point standard.

INTEGER

A signed 8-byte (64-bit) data type.

Syntax

```
[ INTEGER | INT | BIGINT | INT8 | SMALLINT | TINYINT ]
```

Parameters

INT , INTEGER , INT8 , SMALLINT , TINYINT , and BIGINT are all synonyms for the same signed 64-bit integer data type. Automatic compression techniques are used to conserve disk space in cases where the full 64 bits are not required.

Notes

- The range of values is –2^63+1 to 2^63-1.
- 2^63 = 9,223,372,036,854,775,808 (19 digits).
- The value –2^63 is reserved to represent NULL.
- NULL appears first (smallest) in ascending order.
- Vertica does not have an explicit 4-byte (32-bit integer) or smaller types. Vertica's encoding and compression automatically eliminate the storage overhead of values that fit in less than 64 bits.

Restrictions

- The JDBC type INTEGER is 4 bytes and is not supported by Vertica. Use BIGINT instead.
- Vertica does not support the SQL/JDBC types NUMERIC , SMALLINT , or TINYINT .
- Vertica does not check for overflow (positive or negative) except in the aggregate function SUM () . If you encounter overflow when using SUM , use SUM_FLOAT () , which converts to floating point.

See also

Data Type Coercion Chart

NUMERIC

Numeric data types store fixed-point numeric data. For example, a value of $123.45 can be stored in a NUMERIC(5,2) field. Note that the first number, the precision, specifies the *total* number of digits.

Syntax

```
numeric-type [ ( precision[, scale] ) ]
```

Parameters

numeric-type

One of the following:

- NUMERIC
- DECIMAL
- NUMBER
- MONEY

precision

An unsigned integer that specifies the total number of significant digits that the data type stores, where *precision* is ≤ 1024. If omitted, the default precision depends on numeric type that you specify. If you assign a value that exceeds *precision* , Vertica returns an error.

If a data type's precision is ≤ 18, performance is equivalent to an INTEGER data type, regardless of scale. When possible, Vertica recommends using a precision ≤ 18.

scale

An unsigned integer that specifies the maximum number of digits to the right of the decimal point to store. *scale* must be ≤ *precision* . If omitted, the default scale depends on numeric type that you specify. If you assign a value with more decimal digits than *scale* , the scale is rounded to *scale* digits.

When using ALTER to modify the data type of a numeric column, *scale* cannot be changed.

Default precision and scale

NUMERIC , DECIMAL , NUMBER , and MONEY differ in their default precision and scale values:

Type	Precision	Scale
NUMERIC	37	15
DECIMAL	37	15
NUMBER	38	0
MONEY	18	4

Supported encoding

Vertica supports the following encoding for numeric data types :

- Precision ≤ 18: AUTO , BLOCK_DICT , BLOCKDICT_COMP , COMMONDELTA_COMP , DELTAVAL , GCDDELTA , and RLE
- Precision > 18: AUTO , BLOCK_DICT , BLOCKDICT_COMP , RLE

For details, see Encoding types .

Numeric versus integer and floating data types

Numeric data types are *exact* data types that store values of a specified precision and scale, expressed with a number of digits before and after a decimal point. This contrasts with the Vertica integer and floating data types:

- DOUBLE PRECISION (FLOAT) supports ~15 digits, variable exponent, and represents numeric values approximately. It can be less precise than NUMERIC data types.
- INTEGER supports ~18 digits, whole numbers only.

The NUMERIC data type is preferred for non-integer constants, because it is always exact. For example:

```
=> SELECT 1.1 + 2.2 = 3.3;
?column?
----------
 t
(1 row)

=>  SELECT 1.1::float + 2.2::float = 3.3::float;
 ?column?
----------
 f
(1 row)
```

Numeric operations

Supported numeric operations include the following:

- Basic math : + , - , * , /
- Aggregation : SUM , MIN , MAX , COUNT
- Comparison : < , <= , = , <=> , <> , > , >=

NUMERIC divide operates directly on numeric values, without converting to floating point. The result has at least 18 decimal places and is rounded.

NUMERIC mod (including %) operates directly on numeric values, without converting to floating point. The result has the same scale as the numerator and never needs rounding.

Some complex operations used with numeric data types result in an implicit cast to FLOAT. When using SQRT, STDDEV, transcendental functions such as LOG, and TO_CHAR/TO_NUMBER formatting, the result is always FLOAT.

Examples

The following series of commands creates a table that contains a numeric data type and then performs some mathematical operations on the data:

```
=> CREATE TABLE num1 (id INTEGER, amount NUMERIC(8,2));
```

Insert some values into the table:

```
=> INSERT INTO num1 VALUES (1, 123456.78);
```

Query the table:

```
=> SELECT * FROM num1;
 id  |  amount
-----+-----------
   1 | 123456.78
(1 row)
```

The following example returns the NUMERIC column, amount, from table num1:

```
=> SELECT amount FROM num1;
  amount
-----------
 123456.78
(1 row)
```

The following syntax adds one (1) to the amount:

```
=> SELECT amount+1 AS 'amount' FROM num1;
  amount
-----------
 123457.78
(1 row)
```

The following syntax multiplies the amount column by 2:

```
=> SELECT amount*2 AS 'amount' FROM num1;
  amount
-----------
 246913.56
(1 row)
```

The following syntax returns a negative number for the amount column:

```
=> SELECT -amount FROM num1;
?column?
------------
-123456.78
(1 row)
```

The following syntax returns the absolute value of the amount argument:

```
=> SELECT ABS(amount) FROM num1;
ABS
-----------
123456.78
(1 row)
```

The following syntax casts the NUMERIC amount as a FLOAT data type:

```
=> SELECT amount::float FROM num1;
  amount
-----------
 123456.78
(1 row)
```

See also

Mathematical functions

Numeric data type overflow

Vertica does not check for overflow (positive or negative) except in the aggregate function SUM () . If you encounter overflow when using SUM , use SUM_FLOAT () which converts to floating point.

For a detailed discussion of how Vertica handles overflow when you use the functions SUM, SUM_FLOAT, and AVG with numeric data types, see Numeric data type overflow with SUM, SUM_FLOAT, and AVG . The discussion includes directives for turning off silent numeric overflow and setting precision for numeric data types. Dividing by zero returns an error:

```
=> SELECT 0/0;
ERROR 3117:  Division by zero

=> SELECT 0.0/0;
ERROR 3117:  Division by zero

=> SELECT 0 // 0;
ERROR 3117:  Division by zero

=> SELECT 200.0/0;
ERROR 3117:  Division by zero

=> SELECT 116.43 // 0;
ERROR 3117:  Division by zero
```

Dividing zero as a FLOAT by zero returns NaN:

```
=> SELECT 0.0::float/0;
 ?column?
----------
      NaN
=> SELECT 0.0::float//0;
 ?column?
----------
      NaN
```

Dividing a non-zero FLOAT by zero returns Infinity:

```
=> SELECT 2.0::float/0;
 ?column?
----------
Infinity
=> SELECT 200.0::float//0;
?column?
----------
Infinity
```

Add, subtract, and multiply operations ignore overflow. Sum and average operations use 128-bit arithmetic internally. SUM () reports an error if the final result overflows, suggesting the use of SUM_FLOAT (INT) , which converts the 128-bit sum to a FLOAT . For example:

```
=> CREATE TEMP TABLE t (i INT);
=> INSERT INTO t VALUES (1<<62);
=> INSERT INTO t VALUES (1<<62);
=> INSERT INTO t VALUES (1<<62);
=> INSERT INTO t VALUES (1<<62);
=> INSERT INTO t VALUES (1<<62);
=> SELECT SUM(i) FROM t;
   ERROR: sum() overflowed
   HINT: try sum_float() instead
=> SELECT SUM_FLOAT(i) FROM t;
     SUM_FLOAT
---------------------
 2.30584300921369e+19
```

Numeric data type overflow with SUM, SUM_FLOAT, and AVG

When you use the SUM, SUM_FLOAT, and AVG functions (aggregate and analytic) to query a numeric column, overflow can occur. How Vertica responds to that overflow depends on the settings of two configuration parameters:

- AllowNumericOverflow (Boolean, default 1) allows numeric overflow. Vertica does not implicitly extend precision of numeric data types.
- NumericSumExtraPrecisionDigits (integer, default 6) determines whether to return an overflow error if a result exceeds the specified precision. This parameter is ignored if AllowNumericOverflow is set to 1 (true).

Vertica also allows numeric overflow when you use SUM or SUM_FLOAT to query pre-aggregated data. See Impact on Pre-Aggregated Data Projections below.

Default overflow handling

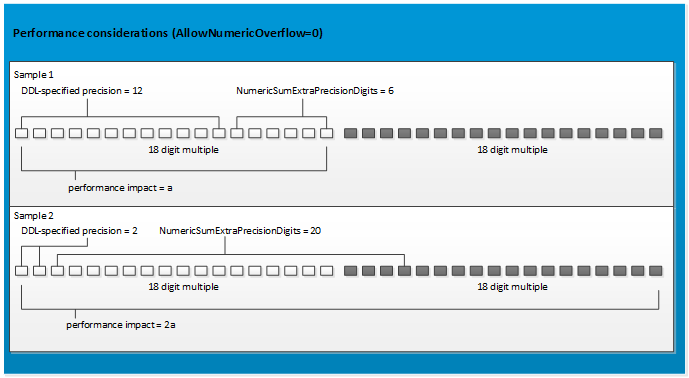
With numeric columns, Vertica internally works with multiples of 18 digits. If specified precision is less than 18—for example, x(12,0) —Vertica allows overflow up to and including the first multiple of 18. In some situations, if you sum a column, you can exceed the number of digits Vertica internally reserves for the result. In this case, Vertica allows silent overflow.

Turning off silent numeric overflow

You can turn off silent numeric overflow by setting AllowNumericOverflow to 0. In this case, Vertica checks the value of configuration parameter NumericSumExtraPrecisionDigits. By default, this parameter is set to 6, which means that Vertica internally adds extra digit places beyond a column's DDL-specified precision. Adding extra precision digits enables Vertica to consistently return results that overflow the column's precision. However, crossing into the second multiple of 18 internally can adversely impact performance.

For example, if AllowNumericOverflow is set to 0 :

- Column x is defined as x(12,0) and NumericSumExtraPrecisionDigits is set to 6: Vertica internally stays within the first multiple of 18 digits and no additional performance impact occurs (*a*).
- Column x is defined as x(2,0) and NumericSumExtraPrecisionDigits is set to 20: Vertica internally crosses a threshold into the second multiple of 18. In this case, performance is significantly affected (*2a*). Performance beyond the second multiple of 18 continues to be *2a* .



Tip

Vertica recommends that you turn off silent numeric overflow and set the parameter NumericSumExtraPrecisionDigits if you expect query results to exceed the precision specified in the DDL of numeric columns. Be aware of the following considerations:

- If you turn off AllowNumericOverflow and exceed the number of extra precision digits set by NumericSumExtraPrecisionDigits, Vertica returns an error.
- Be careful to set NumericSumExtraPrecisionDigits only as high as necessary to return the SUM of numeric columns.

Impact on pre-aggregated data projections

Vertica supports silent numeric overflow for queries that use SUM or SUM_FLOAT on projections with pre-aggregated data such as live aggregate or Top-K projections. To turn off silent numeric overflow for these queries:

1. Set AllowNumericOverflow to 0.
2. Set NumericSumExtraPrecisionDigits to the desired number of implicit digits. Alternatively, use the default setting of 6.
3. Drop and re-create the affected projections.

If you turn off silent numeric overflow, be aware that overflow can sometimes cause rollback or errors:

- Overflow occurs during load operations, such as COPY, MERGE, or INSERT:
  Vertica aggregates data before loading the projection with data. If overflow occurs while data is aggregated, , Vertica rolls back the load operation.
- Overflow occurs after load, while Vertica sums existing data.
  Vertica computes the sum of existing data separately from the computation that it does during data load. If the projection selects a column with SUM or SUM_FLOAT and overflow occurs, Vertica produces an error message. This response resembles the way Vertica produces an error for a query that uses SUM or SUM_FLOAT.
- Overflow occurs during mergeout .
  Vertica logs a message during mergeout if overflow occurs while Vertica computes a final sum during the mergeout operation. If an error occurs, Vertica marks the projection as out of date and disqualifies it from further mergeout operations.

Spatial data types

Vertica supports two spatial data types. These data types store two- and three-dimensional spatial objects in a table column:

- GEOMETRY : Spatial object with coordinates expressed as (*x* , *y*) pairs, defined in the Cartesian plane. All calculations use Cartesian coordinates.
- GEOGRAPHY : Spatial object defined as on the surface of a perfect sphere, or a spatial object in the WGS84 coordinate system. Coordinates are expressed in longitude/latitude angular values, measured in degrees. All calculations are in meters. For perfect sphere calculations, the sphere has a radius of 6371 kilometers, which approximates the shape of the earth.

> **Note**
> Some spatial programs use an ellipsoid to model the earth, resulting in slightly different data.

The maximum size of a GEOMETRY or GEOGRAPHY data type is 10,000,000 bytes (10 MB). You cannot use either data type as a table's primary key.

Syntax

```
GEOMETRY [ (length) ]
GEOGRAPHY [ (length) ]
```

Parameters

length

The maximum amount of spatial data that a GEOMETRY or GEOGRAPHY column can store, up to 10 MB.

Default: 1 MB

UUID data type

Stores universally unique identifiers (UUIDs). UUIDs are 16-byte (128-bit) numbers used to uniquely identify records. To generate UUIDs, Vertica provides the function UUID_GENERATE , which returns UUIDs based on high-quality randomness from /dev/urandom .

Syntax

```
UUID
```

UUID input and output formats

UUIDs support input of case-insensitive string literal formats, as specified by RFC 4122 . In general, a UUID is written as a sequence of hexadecimal digits, in several groups optionally separated by hyphens, for a total of 32 digits representing 128 bits.

The following input formats are valid:

```
6bbf0744-74b4-46b9-bb05-53905d4538e7
{6bbf0744-74b4-46b9-bb05-53905d4538e7}
6BBF074474B446B9BB0553905D4538E7
6BBf-0744-74B4-46B9-BB05-5390-5D45-38E7
```

On output, Vertica always uses the following format:

```
xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

For example, the following table defines column cust_id as a UUID:

```
=> CREATE TABLE public.Customers
(
   cust_id uuid,
   lname varchar(36),
   fname varchar(24)
);
```

The following input for cust_id uses several valid formats:

```
=>  COPY Customers FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {cede66b7-3d29-4da6-b700-871fc0ac57be}|Kearney|Thomas
>> 34462732ed5649838f3be735b0c32d50|Pham|Duc
>> 9fb0-1de0-1d63-4d09-9415-90e0-b4e9-3b9a|Steinberg|Jeremy
>> \.
```

On querying this table, Vertica formats all cust_id data in the same way:

```
=> SELECT cust_id, fname, lname FROM Customers;
               cust_id               | fname  |   lname
-------------------------------------+--------+-----------
 9fb01de0-1d63-4d09-9415-90e0b4e93b9a | Jeremy | Steinberg
 34462732-ed56-4983-8f3b-e735b0c32d50 | Duc    | Pham
 cede66b7-3d29-4da6-b700-871fc0ac57be | Thomas | Kearney
(3 rows)
```

Generating UUIDs

You can use the Vertica function UUID_GENERATE to automatically generate UUIDs that uniquely identify table records. For example:

```
=> INSERT INTO Customers SELECT UUID_GENERATE(),'Rostova','Natasha';
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
=> SELECT cust_id, fname, lname FROM Customers;
               cust_id               |  fname  |   lname
-------------------------------------+---------+-----------
 9fb01de0-1d63-4d09-9415-90e0b4e93b9a | Jeremy  | Steinberg
 34462732-ed56-4983-8f3b-e735b0c32d50 | Duc     | Pham
 cede66b7-3d29-4da6-b700-871fc0ac57be | Thomas  | Kearney
 9aad6757-fe1b-473a-a109-b89b7b358c69 | Natasha | Rostova
(4 rows)
```

NULL input and output

The following string is reserved as NULL for UUID columns:

```
00000000-0000-0000-0000-000000000000
```

Vertica always renders NULL as blank.

The following COPY statements insert NULL values into the UUID column, explicitly and implicitly:

```
=>  COPY Customers FROM STDIN NULL AS 'null';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> null|Doe|Jane
>> 00000000-0000-0000-0000-000000000000|Man|Nowhere
>> \.
=>  COPY Customers FROM STDIN;
>> |Doe|John
>> \.
```

In all cases, Vertica renders NULL as blank:

```
=> SELECT cust_id, fname, lname FROM Customers WHERE cust_id IS NULL;
 cust_id |  fname  | lname
---------+---------+-------
         | Nowhere | Man
         | Jane    | Doe
         | John    | Doe
(3 rows)
```

Usage restrictions

UUID data types only support relational operators and functions that are also supported by CHAR and VARCHAR data types—for example, MIN , MAX , and COUNT . UUID data types do not support mathematical operators or functions, such as SUM and AVG .

Data type coercion

Vertica supports two types of data type casting:

- *Implicit casting* : The expression automatically converts the data from one type to another.
- *Explicit casting* : A SQL statement specifies the target data type for the conversion.

Implicit casting

The ANSI SQL-92 standard supports implicit casting among similar data types:

- Number types
- CHAR, VARCHAR, LONG VARCHAR
- BINARY, VARBINARY, LONG VARBINARY

Vertica supports two types of nonstandard implicit casts of scalar types:

- From CHAR to FLOAT, to match the one from VARCHAR to FLOAT. The following example converts the CHAR '3' to a FLOAT so it can add the number 4.33 to the FLOAT result of the second expression:

  ```
  => SELECT '3'::CHAR + 4.33::FLOAT;
   ?column?
  ----------
    7.33
  (1 row)
  ```

- Between DATE and TIMESTAMP. The following example DATE to a TIMESTAMP and calculates the time 6 hours, 6 minutes, and 6 seconds back from 12:00 AM:

  ```
  => SELECT DATE('now') - INTERVAL '6:6:6';
       ?column?
  ---------------------
   2013-07-30 17:53:54
  (1 row)
  ```

When there is no ambiguity about the data type of an expression value, it is implicitly coerced to match the expected data type. In the following statement, the quoted string constant ' 2 ' is implicitly coerced into an INTEGER value so that it can be the operand of an arithmetic operator (addition):

```
=> SELECT 2 + '2';
 ?column?
----------
        4
(1 row)
```

A concatenate operation explicitly takes arguments of any data type. In the following example, the concatenate operation implicitly coerces the arithmetic expression 2 + 2 and the INTEGER constant 2 to VARCHAR values so that they can be concatenated.

```
=> SELECT 2 + 2 || 2;
 ?column?
----------
 42
(1 row)
```

Another example is to first get today's date:

```
=> SELECT DATE 'now';
 ?column?
------------
 2013-07-31
(1 row)
```

The following command converts DATE to a TIMESTAMP and adds a day and a half to the results by using INTERVAL:

```
=> SELECT DATE 'now' + INTERVAL '1 12:00:00';
     ?column?
---------------------
 2013-07-31 12:00:00
(1 row)
```

Most implicit casts stay within their relational family and go in one direction, from less detailed to more detailed. For example:

- DATE to TIMESTAMP/TZ
- INTEGER to NUMERIC to FLOAT
- CHAR to FLOAT
- CHAR to VARCHAR
- CHAR and/or VARCHAR to FLOAT
- CHAR to LONG VARCHAR
- VARCHAR to LONG VARCHAR
- BINARY to VARBINARY
- BINARY to LONG VARBINARY
- VARBINARY to LONG VARBINARY

More specifically, data type coercion works in this manner in Vertica:

Conversion	**Notes**
INT8 > FLOAT8	Implicit, can lose significance
FLOAT8 > INT8	Explicit, rounds
VARCHAR <-> CHAR	Implicit, adjusts trailing spaces
VARBINARY <-> BINARY	Implicit, adjusts trailing NULs
VARCHAR > LONG VARCHAR	Implicit, adjusts trailing spaces
VARBINARY > LONG VARBINARY	Implicit, adjusts trailing NULs

No other types cast to or from LONGVARBINARY, VARBINARY, or BINARY. In the following list, <any> means one these types: INT8, FLOAT8, DATE, TIME, TIMETZ, TIMESTAMP, TIMESTAMPTZ, INTERVAL:

- <any> -> VARCHAR—implicit
- VARCHAR -> <any>—explicit, except that VARCHAR->FLOAT is implicit
- <any> <-> CHAR—explicit
- DATE -> TIMESTAMP/TZ—implicit
- TIMESTAMP/TZ -> DATE—explicit, loses time-of-day
- TIME -> TIMETZ—implicit, adds local timezone
- TIMETZ -> TIME—explicit, loses timezone
- TIME -> INTERVAL—implicit, day to second with days=0
- INTERVAL -> TIME—explicit, truncates non-time parts
- TIMESTAMP <-> TIMESTAMPTZ—implicit, adjusts to local timezone
- TIMESTAMP/TZ -> TIME—explicit, truncates non-time parts
- TIMESTAMPTZ -> TIMETZ—explicit
- VARBINARY -> LONG VARBINARY—implicit
- LONG VARBINARY -> VARBINARY—explicit
- VARCHAR -> LONG VARCHAR—implicit
- LONG VARCHAR -> VARCHAR—explicit

Important

Implicit casts from INTEGER, FLOAT, and NUMERIC to VARCHAR are not supported. If you need that functionality, write an explicit cast:

```
CAST(x AS data-type-name)
```

or

```
x::data-type-name
```

The following example casts a FLOAT to an INTEGER:

```
=> SELECT(FLOAT '123.5')::INT;
 ?column?
----------
      124
(1 row)
```

String-to-numeric data type conversions accept formats of quoted constants for scientific notation, binary scaling, hexadecimal, and combinations of numeric-type literals:

- Scientific notation :

```
=> SELECT FLOAT '1e10';
  ?column?
-------------
 10000000000
(1 row)
```

- BINARY scaling :

```
=> SELECT NUMERIC '1p10';
 ?column?
----------
     1024
(1 row)
```

- hexadecimal:

```
=> SELECT NUMERIC '0x0abc';
 ?column?
----------
     2748
(1 row)
```

Complex types

Collections (arrays and sets) can be cast implicitly and explicitly. Casting a collection casts each element of the collection. You can, for example, cast an ARRAY[VARCHAR] to an ARRAY[INT] or a SET[DATE] to SET[TIMESTAMPTZ]. You can cast between arrays and sets.

When casting to a bounded native array, inputs that are too long are truncated. When casting to a non-native array (an array containing complex data types including other arrays), if the new bounds are too small for the data the cast fails

Rows (structs) can be cast implicitly and explicitly. Casting a ROW casts each field value. You can specify new field names in the cast or specify only the field types to use the existing field names.

Casting can increase the storage needed for a column. For example, if you cast an array of INT to an array of VARCHAR(50), each element takes more space and thus the array takes more space. If the difference is extreme or the array has many elements, this could mean that the array no longer fits within the space allotted for the column. In this case the operation reports an error and fails.

Examples

The following example casts three strings as NUMERICs:

```
=> SELECT NUMERIC '12.3e3', '12.3p10'::NUMERIC, CAST('0x12.3p-10e3' AS NUMERIC);
 ?column? | ?column? |     ?column?
----------+----------+------------------
    12300 |  12595.2 | 17.76123046875000
(1 row)
```

This example casts a VARBINARY string into a LONG VARBINARY data type:

```
=> SELECT B'101111000'::LONG VARBINARY;
 ?column?
----------
 \001x
(1 row)
```

The following example concatenates a CHAR with a LONG VARCHAR, resulting in a LONG VARCHAR:

```
=> \set s "`cat longfile.txt`"
=> SELECT length ('a' || :s ::LONG VARCHAR);
 length
----------
 65002
(1 row)
```

The following example casts a combination of NUMERIC and INTEGER data into a NUMERIC result:

```
=> SELECT (18. + 3./16)/1024*1000;
                ?column?
-----------------------------------------
 17.761230468750000000000000000000000000
(1 row)
```

> **Note**
> In SQL expressions, pure numbers between (–2^63–1) and (2^63–1) are INTEGERs. Numbers with decimal points are NUMERIC.

See also

- Data type coercion chart
- Data type coercion operators (CAST)

Data type coercion chart

Conversion types

The following table defines all possible type conversions that Vertica supports. The data types in the first column of the table are the inputs to convert, and the remaining columns indicate the result for the different conversion types.

Source Data Type	Implicit	Explicit	Assignment	Assignment without numeric meaning	Conversion without explicit casting
BOOLEAN			INTEGER, LONG VARCHAR, VARCHAR, CHAR		
INTEGER	BOOLEAN, NUMERIC, FLOAT		INTERVAL DAY/SECOND, INTERVAL YEAR/MONTH	LONG VARCHAR, VARCHAR, CHAR	
NUMERIC	FLOAT		INTEGER	LONG VARCHAR, VARCHAR, CHAR	NUMERIC

FLOAT			INTEGER, NUMERIC	LONG VARCHAR, VARCHAR, CHAR	
LONG VARCHAR	FLOAT, CHAR	BOOLEAN, INTEGER, NUMERIC, VARCHAR, TIMESTAMP, TIMESTAMPTZ, DATE, TIME, TIMETZ, INTERVAL DAY/SECOND, INTERVAL YEAR/MONTH, LONG VARBINARY			LONG VARCHAR
VARCHAR	CHAR, FLOAT, LONG VARCHAR	BOOLEAN, INTEGER, NUMERIC, TIMESTAMP, TIMESTAMPTZ, DATE, TIME, TIMETZ, UUID, BINARY, VARBINARY, INTERVAL DAY/SECOND, INTERVAL YEAR/MONTH			VARCHAR
CHAR	FLOAT, LONG VARCHAR, VARCHAR	BOOLEAN, INTEGER, NUMERIC, TIMESTAMP, TIMESTAMPTZ, DATE, TIME, TIMETZ, UUID (CHAR length ≥ 36), BINARY, VARBINARY, INTERVAL DAY/SECOND, INTERVAL YEAR/MONTH			CHAR
TIMESTAMP	TIMESTAMPTZ		LONG CHAR, VARCHAR, CHAR, DATE, TIME		TIMESTAMP
TIMESTAMPTZ	TIMESTAMP		LONG CHAR, VARCHAR, CHAR, DATE, TIME, TIMETZ		TIMESTAMPTZ
DATE	TIMESTAMP		LONG CHAR, VARCHAR, CHAR, TIMESTAMPTZ		
TIME	TIMETZ	TIMESTAMP, TIMESTAMPTZ, INTERVAL DAY/SECOND	LONG CHAR, VARCHAR, CHAR		TIME
TIMETZ		TIMESTAMP, TIMESTAMPTZ	LONG CHAR, VARCHAR, CHAR, TIME		TIMETZ
INTERVAL DAY/SECOND		TIME	INTEGER, LONG CHAR, VARCHAR, CHAR		INTERVAL DAY/SECOND
INTERVAL YEAR/MONTH			INTEGER, LONG CHAR, VARCHAR, CHAR		INTERVAL YEAR/MONTH
LONG VARBINARY		VARBINARY			LONG VARBINARY
VARBINARY	LONG VARBINARY, BINARY				VARBINARY
BINARY	VARBINARY				BINARY

UUID		CHAR(36), VARCHAR			UUID

Implicit and explicit conversion

Vertica supports data type conversion of values without explicit casting, such as NUMERIC(10,6) -> NUMERIC(18,4) .Implicit data type conversion occurs automatically when converting values of different, but compatible, types to the target column's data type. For example, when adding values, (INTEGER + NUMERIC), the result is implicitly cast to a NUMERIC type to accommodate the prominent type in the statement. Depending on the input data types, different precision and scale can occur.

An explicit type conversion must occur when the source data cannot be cast implicitly to the target column's data type.

Assignment conversion

In data assignment conversion, coercion implicitly occurs when values are assigned to database columns in an INSERT or UPDATE...SET statement. For example, in a statement that includes INSERT...VALUES('2.5') , where the target column data type is NUMERIC(18,5) , a cast from VARCHAR to the column data type is inferred.

In an assignment without numeric meaning, the value is subject to CHAR/VARCHAR/LONG VARCHAR comparisons.

See also

- Data type coercion
- Data type coercion operators (CAST)

Complex types

Complex types such as structures (also known as rows), arrays, and maps are composed of primitive types and sometimes other complex types. Complex types can be used in the following ways:

- Arrays and rows (in any combination) can be used as column data types in both native and external tables.
- Sets of primitive element types can be used as column data types in native and external tables.
- Arrays and rows, but not combinations of them, can be created as literals, for example to use in query expressions.

The MAP type is a legacy type. To represent maps, use ARRAY[ROW] .

If a flex table has a real column that uses a complex type, the values from that column are not included in the __raw__ column. For more information, see Loading Data into Flex Table Real Columns .

In this section

- ARRAY
- MAP
- ROW
- SET

ARRAY

Represents array data. There are two types of arrays in Vertica:

- Native array: a one-dimensional array of a primitive type. Native arrays are tracked in the TYPES system table and used in native tables.
- Non-native array: all other supported arrays, including arrays that contain other arrays (multi-dimensional arrays) or structs (ROW s). Non-native arrays have some usage restrictions. Non-native arrays are tracked in the COMPLEX_TYPES system table.

Both types of arrays operate in the same way, but they have different OIDs.

Arrays can be bounded, meaning they specify a maximum element count, or unbounded. Unbounded arrays have a maximum binary size, which can be set explicitly or defaulted. See Limits on Element Count and Collection Size .

Selected parsers support using COPY to load arrays. See the documentation of individual parsers for more information.

Syntax

In column definitions :

```
ARRAY[data_type, max_elements] |
ARRAY[data_type](max_size) |
ARRAY[data_type]
```

In literals :

```
ARRAY[value[, ...] ]
```

Restrictions

- Native arrays support only data of primitive types, for example, int, UUID, and so on.
- Array dimensionality is enforced. A column cannot contain arrays of varying dimensions. For example, a column that contains a three-dimensional array can only contain other three-dimensional arrays; it cannot simultaneously include a one-dimensional array. However, the arrays in a column can vary in size, where one array can contain four elements while another contains ten.
- Array bounds, if specified, are enforced for all operations that load or alter data. Unbounded arrays may have as many elements as will fit in the allotted binary size.
- An array has a maximum binary size. If this size is not set when the array is defined, a default value is used.
- Arrays do not support LONG types (like LONG VARBINARY or LONG VARCHAR) or user-defined types (like Geometry).

Syntax for column definition

Arrays used in column definitions can be either bounded or unbounded. Bounded arrays must specify a maximum number of elements. Unbounded arrays can specify a maximum binary size (in bytes) for the array, or the value of DefaultArrayBinarySize is used. You can specify a bound or a binary size but not both. For more information about these values, see Limits on Element Count and Collection Size .

Type	Syntax	Semantics
Bounded array	ARRAY[*data_type* , *max_elements*] Example: ARRAY[VARCHAR(50),100]	Can contain no more than *max_elements* elements. Attempting to add more is an error. Has a binary size of the size of the data type multiplied by the maximum number of elements (possibly rounded up).
Unbounded array with maximum binary size	ARRAY[*data_type*](*max_size*) Example: ARRAY[VARCHAR(50)](32000)	Can contain as many elements as fit in *max_size* . Ignores the value of DefaultArrayBinarySize.
Unbounded array with default binary size	ARRAY[*data_type*] Example: ARRAY[VARCHAR(50)]	Can contain as many elements as fit in the default binary size. Equivalent to: ARRAY[*data_type*](DefaultArrayBinarySize)

The following example defines a table for customers using an unbounded array:

```
=> CREATE TABLE customers (id INT, name VARCHAR, email ARRAY[VARCHAR(50)]);
```

The following example uses a bounded array for customer email addresses and an unbounded array for order history:

```
=> CREATE TABLE customers (id INT, name VARCHAR, email ARRAY[VARCHAR(50),5], orders ARRAY[INT]);
```

The following example uses an array that has ROW elements:

```
=> CREATE TABLE orders(
 orderid INT,
 accountid INT,
 shipments ARRAY[
   ROW(
     shipid INT,
     address ROW(
       street VARCHAR,
       city VARCHAR,
       zip INT
       ),
     shipdate DATE
   )
 ]
);
```

To declare a multi-dimensional array, use nesting. For example, ARRAY[ARRAY[int]] specifies a two-dimensional array.

Syntax for direct construction (literals)

Use the ARRAY keyword to construct an array value. The following example creates an array of integer values.

```
=> SELECT ARRAY[1,2,3];
 array
-------
 [1,2,3]
(1 row)
```

You can nest an array inside another array, as in the following example.

```
=> SELECT ARRAY[ARRAY[1],ARRAY[2]];
   array
-----------
 [[1],[2]]
(1 row)
```

If an array of arrays contains no null elements and no function calls, you can abbreviate the syntax:

```
=> SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 [[1,2],[3,4]]
(1 row)

---not valid:
=> SELECT ARRAY[[1,2],null,[3,4]];

ERROR 4856:  Syntax error at or near "null" at character 20
LINE 1: SELECT ARRAY[[1,2],null,[3,4]];
                           ^
```

Array literals can contain elements of all scalar types, ROW, and ARRAY. ROW elements must all have the same set of fields:

```
=> SELECT ARRAY[ROW(1,2),ROW(1,3)];
                array
-----------------------------------
 [{"f0":1,"f1":2},{"f0":1,"f1":3}]
(1 row)

=> SELECT ARRAY[ROW(1,2),ROW(1,3,'abc')];
ERROR 3429:  For 'ARRAY', types ROW(int,int) and ROW(int,int,unknown) are inconsistent
```

Because the elements are known at the time you directly construct an array, these arrays are implicitly bounded.

You can use ARRAY literals in comparisons, as in the following example:

```
=> SELECT id.name, id.num, GPA FROM students
   WHERE major = ARRAY[ROW('Science','Physics')];
 name  | num | GPA
-------+-----+-----
 bob   | 121 | 3.3
 carol | 123 | 3.4
(2 rows)
```

Output format

Queries of array columns return JSON format, with the values shown in comma-separated lists in brackets. The following example shows a query that includes array columns.

```
=> SELECT cust_custkey,cust_custstaddress,cust_custcity,cust_custstate from cust;
cust_custkey |              cust_custstaddress                  |              cust_custcity              | cust_custstate
-------------+-------  ---------------------------------------------+---------------------------------------------+----------------
      342176 | ["668 SW New Lane","518 Main Ave","7040 Campfire Dr"] | ["Winchester","New Hyde Park","Massapequa"] | ["VA","NY","NY"]
      342799 | ["2400 Hearst Avenue","3 Cypress Street"]            | ["Berkeley","San Antonio"]                  | ["CA","TX"]
      342845 | ["336 Boylston Street","180 Clarkhill Rd"]           | ["Boston","Amherst"]                        | ["MA","MA"]
      342321 | ["95 Fawn Drive"]                                    | ["Allen Park"]                              | ["MI"]
      342989 | ["5 Thompson St"]                                    | ["Massillon"]                               | ["OH"]
(5 rows)
```

Note that JSON format escapes some characters that would not be escaped in native VARCHARs. For example, if you insert "c:\users\data" into an array, the JSON output for that value is "c:\\users\\data" .

Element access

Arrays are 0-indexed. The first element's ordinal position is 0, second is 1, and so on.

You can access (dereference) elements from an array by index:

```
=> SELECT (ARRAY['a','b','c','d','e'])[1];
array
-------
b
(1 row)
```

To specify a range, use the format *start* : *end* . The end of the range is non-inclusive.

```
=> SELECT(ARRAY['a','b','c','d','e','f','g'])[1:4];
array
---------
["b","c","d"]
(1 row)
```

To dereference an element from a multi-dimensional array, put each index in brackets:

```
=> SELECT(ARRAY[ARRAY[1,2],ARRAY[3,4]])[0][0];
 array
-------
 1
(1 row)
```

Out-of-bound index references return NULL.

Limits on element count and collection size

When declaring a collection type for a table column, you can limit either the number of elements or the total binary size of the collection. During query processing, Vertica always reserves the maximum memory needed for the column, based on either the element count or the binary size. If this size is much larger than your data actually requires, setting one of these limits can improve query performance by reducing the amount of memory that must be reserved for the column.

You can change the bounds of a collection, including changing between bounded and unbounded collections, by casting. See Casting .

A bounded collection specifies a maximum element count. A value in a bounded collection column may contain fewer elements, but it may not contain more. Any attempt to insert more elements into a bounded collection than the declared maximum is an error. A bounded collection has a binary size that is the product of the data-type size and the maximum number of elements, possibly rounded up.

An unbounded collection specifies a binary size in bytes, explicitly or implicitly. It may contain as many elements as can fit in that binary size.

If a nested array specifies bounds for all dimensions, Vertica sets a single bound that is the product of the bounds. In the following example, the inner and outer arrays each have a bound of 10, but only a total element count of 100 is enforced.

```
ARRAY[ARRAY[INT,10],10]
```

If a nested array specifies a bound for only the outer collection, it is treated as the total bound. The previous example is equivalent to the following:

```
ARRAY[ARRAY[INT],100]
```

You must either specify bounds for all nested collections or specify a bound only for the outer one. For any other distribution of bounds, Vertica treats the collection as unbounded.

Instead of specifying a bound, you can specify a maximum binary size for an unbounded collection. The binary size acts as an absolute limit, regardless of how many elements the collection contains. Collections that do not specify a maximum binary size use the value of DefaultArrayBinarySize . This size is set at the time the collection is defined and is not affected by later changes to the value of DefaultArrayBinarySize.

You cannot set a maximum binary size for a bounded collection, only an unbounded one.

You can change the bounds or the binary size of an array column using ALTER TABLE as in the following example:

```
=> ALTER TABLE cust ALTER COLUMN orders SET DATA TYPE ARRAY[INTEGER](100);
```

If the change reduces the size of the collection and would result in data loss, the change fails.

Comparisons

All collections support equality (=), inequality (<>), and null-safe equality (<=>). 1D collections also support comparison operators (< , <= , > , >=) between collections of the same type (arrays or sets). Comparisons follow these rules:

- A null collection is ordered last.
- Non-null collections are compared element by element, using the ordering rules of the element's data type. The relative order of the first pair of non-equal elements determines the order of the two collections.
- If all elements in both collections are equal up to the length of the shorter collection, the shorter collection is ordered before the longer one.
- If all elements in both collections are equal and the collections are of equal length, the collections are equal.

Null-handling

Null semantics for collections are consistent with normal columns in most regards. See NULL sort order for more information on null-handling.

The null-safe equality operator (<=>) behaves differently from equality (=) when the collection is null rather than empty. Comparing a collection to NULL strictly returns null:

```
=> SELECT ARRAY[1,3] = NULL;
?column?
----------

(1 row)

=> SELECT ARRAY[1,3] <=> NULL;
 ?column?
----------
 f
(1 row)
```

In the following example, the grants column in the table is null for employee 99:

```
=> SELECT grants = NULL FROM employees WHERE id=99;
 ?column?
----------

(1 row)

=> SELECT grants <=> NULL FROM employees WHERE id=99;
 ?column?
----------
 t
(1 row)
```

Empty collections are not null and behave as expected:

```
=> SELECT ARRAY[]::ARRAY[INT] = ARRAY[]::ARRAY[INT];
 ?column?
----------
 t
(1 row)
```

Collections are compared element by element. If a comparison depends on a null element, the result is unknown (null), not false. For example, ARRAY[1,2,null]=ARRAY[1,2,null] and ARRAY[1,2,null]=ARRAY[1,2,3] both return null, but ARRAY[1,2,null]=ARRAY[1,4,null] returns false because the second elements do not match.

Casting

Casting an array casts each element of the array. You can therefore cast between data types following the same rules as for casts of scalar values.

You can cast both literal arrays and array columns explicitly:

```
=> SELECT ARRAY['1','2','3']::ARRAY[INT];
  array
---------
[1,2,3]
(1 row)
```

You can change the bound of an array or set by casting. When casting to a bounded native array, inputs that are too long are truncated. When casting to a non-native array (an array containing complex data types including other arrays), if the new bounds are too small for the data the cast fails:

```
=> SELECT ARRAY[1,2,3]::ARRAY[VARCHAR,2];
   array
-----------
 ["1","2"]
(1 row)

=> SELECT ARRAY[ARRAY[1,2,3],ARRAY[4,5,6]]::ARRAY[ARRAY[VARCHAR,2],2];
ERROR 9227:  Output array isn't big enough
DETAIL:  Type limit is 4 elements, but value has 6 elements
```

If you cast to a bounded multi-dimensional array, you must specify the bounds at all levels:

```
=> SELECT ARRAY[ARRAY[1,2,3],ARRAY[4,5,6]]::ARRAY[ARRAY[VARCHAR,5],10];
             array
-------------------------------
 [["1","2","3"],["4","5","6"]]
(1 row)

=> SELECT ARRAY[ARRAY[1,2,3],ARRAY[4,5,6]]::ARRAY[ARRAY[VARCHAR,2]];
WARNING 9753:  Collection type bound will not be used
DETAIL:  A bound was provided for an inner dimension, but not for an outer dimension
             array
-------------------------------
 [["1","2","3"],["4","5","6"]]
(1 row)
```

Assignment casts and implicit casts work the same way as for scalars:

```
=> CREATE TABLE transactions (tid INT, prod_ids ARRAY[VARCHAR,100], quantities ARRAY[INT,100]);
CREATE TABLE

=> INSERT INTO transactions VALUES (12345, ARRAY['p1265', 'p4515'], ARRAY[15,2]);
 OUTPUT
--------
      1
(1 row)

=> CREATE TABLE txreport (prod_ids ARRAY[VARCHAR(12),100], quants ARRAY[VARCHAR(32),100]);
CREATE TABLE

=> INSERT INTO txreport SELECT prod_ids, quantities FROM transactions;
 OUTPUT
--------
      1
(1 row)

=> SELECT * FROM txreport;
    prod_ids      |   quants
------------------+------------
 ["p1265","p4515"] | ["15","2"]
(1 row)
```

You can perform explicit casts, but not implicit casts, between the ARRAY and SET types (native arrays only). If the collection is unbounded and the data type does not change, the binary size is preserved. For example, if you cast an ARRAY[INT] to a SET[INT], the set has the same binary size as the array.

If you cast from one element type to another, the resulting collection uses the default binary size. If this would cause the data not to fit, the cast fails.

You cannot cast from an array to an array with a different dimensionality, for example from a two-dimensional array to a one-dimensional array.

Functions and operators

See Collection functions for a comprehensive list of functions that can be used to manipulate arrays and sets.

Collections can be used in the following ways:

- As the grouping column in a GROUP BY clause .
- For native arrays only, as the sort key in an ORDER BY clause in a query, in an OVER clause (see Window partitioning), or in a CREATE PROJECTION statement.
- As the sort key in the PARTITION BY part of an OVER clause.
- As a JOIN key (see Joined-table).
- In CASE expressions .

Collections cannot be used in the following ways:

- As part of an IN or NOT IN expression.
- As partition columns when creating tables.
- With ANALYZE_STATISTICS or TopK projections.
- Non-native arrays only: ORDER BY, PARTITION BY, DEFAULT, SET USING, or constraints.

MAP

Represents map data in external tables in the Parquet, ORC, and Avro formats only. A MAP must use only primitive types and may not contain other complex types. You can use the MAP type in a table definition to consume columns in the data, but you cannot query those columns.

A superior alternative to MAP is ARRAY [ROW]. An array of rows can use all supported complex types and can be queried. This is the representation that INFER_TABLE_DDL suggests. For Avro data, the ROW must have fields named key and value .

Within a single table you must define all map columns using the same approach, MAP or ARRAY[ROW].

Syntax

In column definitions:

```
MAP<key,value>
```

Map input format for column definition

In a column definition in an external table, a MAP consists of a key-value pair, specified as types. The table in the following example defines a map of product IDs to names.

```
=> CREATE EXTERNAL TABLE store (storeID INT, inventory MAP<INT,VARCHAR(100)>)
   AS COPY FROM '...' PARQUET;
```

ROW

Represents structured data (structs). A ROW can contain fields of any primitive or complex type supported by Vertica.

Syntax

In column definitions :

```
ROW([field] type[, ...])
```

If the field name is omitted, Vertica generates names starting with "f0".

In literals :

```
ROW(value [AS field] [, ...]) [AS name(field[, ...])]
```

Syntax for column definition

In a column definition, a ROW consists of one or more comma-separated pairs of field names and types. In the following example, the Parquet data file contains a struct for the address, which is read as a ROW in an external table:

```
=> CREATE EXTERNAL TABLE customers (name VARCHAR,
   address ROW(street VARCHAR, city VARCHAR, zipcode INT))
   AS COPY FROM '...' PARQUET;
```

ROWs can be nested; a field can have a type of ROW:

```
=> CREATE TABLE employees(
   employeeID INT,
   personal ROW(
     name VARCHAR,
     address ROW(street VARCHAR, city VARCHAR, zipcode INT),
     taxID INT),
   department VARCHAR);
```

ROWs can contain arrays:

```
=> CREATE TABLE customers(
 name VARCHAR,
 contact ROW(
   street VARCHAR,
   city VARCHAR,
   zipcode INT,
   email ARRAY[VARCHAR]
 ),
 accountid INT );
```

When loading data, the primitive types in the table definition must match those in the data. The ROW structure must also match; a ROW must contain all and only the fields in the struct in the data.

Restrictions on ROW columns

ROW columns have several restrictions:

- Maximum nesting depth is 100.
- Vertica tables support up to 9800 columns and fields. The ROW itself is not counted, only its fields.
- ROW columns cannot use any constraints (such as NOT NULL) or defaults.
- ROW fields cannot be auto_increment or setof.

- ROW definition must include at least one field.
- Row is a reserved keyword within a ROW definition, but is permitted as the name of a table or column.
- Tables containing ROW columns cannot also contain IDENTITY, default, SET USING, or named sequence columns.

Syntax for direct construction (literals)

In a literal, such as a value in a comparison operation, a ROW consists of one or more values. If you omit field names in the ROW expression, Vertica generates them automatically. If you do not coerce types, Vertica infers the types from the data values.

```
=> SELECT ROW('Amy',2,false);
                row
-----------------------------------------------
 {"f0":"Amy","f1":2,"f2":false}
(1 row)
```

You can use an AS clause to name the ROW and its fields:

```
=> SELECT ROW('Amy',2,false) AS student(name, id, current);
               student
-----------------------------------------------
 {"name":"Amy","id":2,"current":false}
(1 row)
```

You can also name individual fields using AS. This query produces the same output as the previous one:

```
=> SELECT ROW('Amy' AS name, 2 AS id, false AS current) AS student;
```

You do not need to name all fields.

In an array of ROW elements, if you use AS to name fields and the names differ among the elements, Vertica uses the right-most names for all elements:

```
=> SELECT ARRAY[ROW('Amy' AS name, 2 AS id),ROW('Fred' AS first_name, 4 AS id)];
                          array
-----------------------------------------------------------------
 [{"first_name":"Amy","id":2},{"first_name":"Fred","id":4}]
(1 row)
```

You can coerce types explicitly:

```
=> SELECT ROW('Amy',2.5::int,false::varchar);
               row
-------------------------------------------
 {"f0":"Amy","f1":3,"f2":"f"}
(1 row)
```

Escape single quotes in literal inputs using single quotes, as in the following example:

```
=> SELECT ROW('Howard''s house',2,false);
                    row
-----------------------------------------------------
 {"f0":"Howard's house","f1":2,"f2":false}
(1 row)
```

You can use fields of all scalar types, ROW, and ARRAY, as in the following example:

```
=> SELECT id.name, major, GPA FROM students
   WHERE id = ROW('alice',119, ARRAY['alice@example.com','ap16@cs.example.edu']);
 name  |               major               | GPA
-------+-----------------------------------+-----
 alice | [{"school":"Science","dept":"CS"}] | 3.8
(1 row)
```

Output format

ROW values are output in JSON format as in the following example.

```
=> CREATE EXTERNAL TABLE customers (name VARCHAR,
    address ROW(street VARCHAR, city VARCHAR, zipcode INT))
    AS COPY FROM '...' PARQUET;

=> SELECT address FROM customers WHERE address.city ='Pasadena';
                          address
-------------------------------------------------------------------
 {"street":"100 Main St Apt 4B","city":"Pasadena","zipcode":91001}
 {"street":"100 Main St Apt 4A","city":"Pasadena","zipcode":91001}
 {"street":"23 Fifth Ave Apt 8C","city":"Pasadena","zipcode":91001}
 {"street":"15 Raymond Dr","city":"Pasadena","zipcode":91003}
(4 rows)
```

The following table specifies the mappings from Vertica data types to JSON data types.

Vertica Type	**JSON Type**
Integer	Integer
Float	Numeric
Numeric	Numeric
Boolean	Boolean
All others	String

Comparisons

ROW supports equality (=), inequality (<>), and null-safe equality (<=>) between inputs that have the same set of fields. ROWs that contain only primitive types, including nested ROWs of primitive types, also support comparison operators (< , <= , > , >=).

Two ROWs are equal if and only if all fields are equal. Vertica compares fields in order until an inequality is found or all fields have been compared. The evaluation of the first non-equal field determines which ROW is greater:

```
=> SELECT ROW(1, 'joe') > ROW(2, 'bob');
?column?
----------
f
(1 row)
```

Comparisons between ROWs with different schemas fail:

```
=> SELECT ROW(1, 'joe') > ROW(2, 'bob', 123);
ERROR 5162:  Unequal number of entries in row expressions
```

If the result of a comparison depends on a null field, the result is null:

```
=>  select row(1, null, 3) = row(1, 2, 3);
 ?column?
----------

(1 row)
```

Null-handling

If a struct exists but a field value is null, Vertica assigns NULL as its value in the ROW. A struct where all fields are null is treated as a ROW with null fields. If the struct itself is null, Vertica reads the ROW as NULL.

Casting

Casting a ROW casts each field. You can therefore cast between data types following the same rules as for casts of scalar values.

The following example casts the contact ROW in the customers table, changing the zipcode field from INT to VARCHAR and adding a bound to the array:

```
=> SELECT contact::ROW(VARCHAR,VARCHAR,VARCHAR,ARRAY[VARCHAR,20]) FROM customers;
                                          contact

--------------------------------------------------------------------------------
-----------------------------------------
 {"street":"911 San Marcos St","city":"Austin","zipcode":"73344","email":["missy@mit.edu","mcooper@cern.gov"]}
 {"street":"100 Main St Apt 4B","city":"Pasadena","zipcode":"91001","email":["shelly@meemaw.name","cooper@caltech.edu"]}
 {"street":"100 Main St Apt 4A","city":"Pasadena","zipcode":"91001","email":["hofstadter@caltech.edu"]}
 {"street":"23 Fifth Ave Apt 8C","city":"Pasadena","zipcode":"91001","email":[]}
 {"street":null,"city":"Pasadena","zipcode":"91001","email":["raj@available.com"]}

(6 rows)
```

You can specify new field names to change them in the output:

```
=> SELECT contact::ROW(str VARCHAR, city VARCHAR, zip VARCHAR, email ARRAY[VARCHAR,
20]) FROM customers;
                                          contact

--------------------------------------------------------------------------------
----------------------------------
 {"str":"911 San Marcos St","city":"Austin","zip":"73344","email":["missy@mit.edu","mcooper@cern.gov"]}
 {"str":"100 Main St Apt 4B","city":"Pasadena","zip":"91001","email":["shelly@meemaw.name","cooper@caltech.edu"]}
 {"str":"100 Main St Apt 4A","city":"Pasadena","zip":"91001","email":["hofstadter@caltech.edu"]}
 {"str":"23 Fifth Ave Apt 8C","city":"Pasadena","zip":"91001","email":[]}
 {"str":null,"city":"Pasadena","zip":"91001","email":["raj@available.com"]}
(6 rows)
```

Supported operators and predicates

ROW values may be used in queries in the following ways:

- INNER and OUTER JOIN
- Comparisons , IN, BETWEEN (non-nullable filters only)
- IS NULL, IS NOT NULL
- CASE
- GROUP BY, ORDER BY
- SELECT DISTINCT
- Arguments to user-defined scalar, transform, and analytic functions

The following operators and predicates are not supported for ROW values:

- Math operators
- Type coercion of whole rows (coercion of field values is supported)
- BITWISE, LIKE
- MLA (ROLLUP, CUBE, GROUPING SETS)
- Aggregate functions including MAX, MIN, and SUM
- Set operators including UNION, UNION ALL, MINUS, and INTERSECT

COUNT is not supported for ROWs returned from user-defined scalar functions, but is supported for ROW columns and literals.

In comparison operations (including implicit comparisons like ORDER BY), a ROW literal is treated as the sequence of its field values. For example, the following two statements are equivalent:

```
GROUP BY ROW(zipcode, city)
GROUP BY zipcode, city
```

Using rows in views and subqueries

You can use ROW columns to construct views and in subqueries. Consider employee and customer tables with the following definitions:

```
=> CREATE EXTERNAL TABLE customers(name VARCHAR,
      address ROW(street VARCHAR, city VARCHAR, zipcode INT), accountID INT)
   AS COPY FROM '...' PARQUET;

=> CREATE EXTERNAL TABLE employees(employeeID INT,
      personal ROW(name VARCHAR,
      address ROW(street VARCHAR, city VARCHAR, zipcode INT),
      taxID INT), department VARCHAR)
   AS COPY FROM '...' PARQUET;
```

The following example creates a view and queries it.

```
=> CREATE VIEW neighbors (num_neighbors, area(city, zipcode))
AS SELECT count(*), ROW(address.city, address.zipcode)
FROM customers GROUP BY address.city, address.zipcode;
CREATE VIEW

=> SELECT employees.personal.name, neighbors.area FROM neighbors, employees
WHERE employees.personal.address.zipcode=neighbors.area.zipcode AND neighbors.nu
m_neighbors > 1;
      name       |              area
-----------------+----------------------------------
 Sheldon Cooper     | {"city":"Pasadena","zipcode":91001}
 Leonard Hofstadter | {"city":"Pasadena","zipcode":91001}
(2 rows)
```

SET

Represents a collection of unordered, unique elements. Sets may contain only primitive types. In sets, unlike in arrays, element position is not meaningful.

Sets do not support LONG types (like LONG VARBINARY or LONG VARCHAR) or user-defined types (like Geometry).

If you populate a set from an array, Vertica sorts the values and removes duplicate elements. If you do not care about element position and plan to run queries that check for the presence of specific elements (find, contains), using a set could improve query performance.

Sets can be bounded, meaning they specify a maximum element count, or unbounded. Unbounded sets have a maximum binary size, which can be set explicitly or defaulted. See Limits on Element Count and Collection Size .

Syntax

In column definitions :

```
SET[data_type, max_elements] |
SET[data_type](max_size) |
SET[data_type]
```

In literals :

```
SET[value[, ...] ]
```

Restrictions

- Sets support only data of primitive (scalar) types.
- Bounds, if specified, are enforced for all operations that load or alter data. Unbounded sets may have as many elements as will fit in the allotted binary size.
- A set has a maximum binary size. If this size is not set when the set is defined, a default value is used.

Syntax for column definition

Sets used in column definitions can be either bounded or unbounded. Bounded sets must specify a maximum number of elements. Unbounded sets can specify a maximum binary size for the set, or the value of DefaultArrayBinarySize is used. You can specify a bound or a binary size but not both. For more information about these values, see Limits on Element Count and Collection Size .

Type	Syntax	Semantics

Bounded set	SET[*data_type* , *max_elements*] Example: SET[VARCHAR(50),100]	Can contain no more than *max_elements* elements. Attempting to add more is an error. Has a binary size of the size of the data type multiplied by the maximum number of elements (possibly rounded up).
Unbounded set with maximum size	SET[*data_type*](*max_size*) Example: SET[VARCHAR(50)](32000)	Can contain as many elements as fit in *max_size* . Ignores the value of DefaultArrayBinarySize.
Unbounded set	SET[*data_type*] Example: SET[VARCHAR(50)]	Can contain as many elements as fit in the default binary size. Equivalent to: SET[*data_type*](DefaultArrayBinarySize)

The following example defines a table with an unbounded set colum.

```
=> CREATE TABLE users
(
user_id INTEGER,
display_name VARCHAR,
email_addrs SET[VARCHAR]
);
```

When you load array data into a column defined as a set, the array data is automatically converted to a set.

Syntax for direct construction (literals)

Use the SET keyword to construct a set value. Literal set values are contained in brackets. For example, to create a set of INT, you would do the following:

```
=> SELECT SET[1,2,3];
 set
-------
 [1,2,3]
(1 row)
```

You can explicitly convert an array to a set by casting, as in the following example:

```
=> SELECT ARRAY[1, 5, 2, 6, 3, 0, 6, 4]::SET[INT];
    set
-----------------
[0,1,2,3,4,5,6]
(1 row)
```

Notice that duplicate elements have been removed and the elements have been sorted.

Because the elements are known at the time you directly construct a set, these sets are implicitly bounded.

Output format

Sets are shown in a JSON-like format, with comma-separated elements contained in brackets (like arrays). In the following example, the email_addrs column is a set.

```
=> SELECT custkey,email_addrs FROM customers LIMIT 4;
 custkey |                          email_addrs
---------+-------------------------------------------------------------------------
 342176  | ["joe.smith@example.com"]
 342799  | ["bob@example,com","robert.jones@example.com"]
 342845  | ["br92@cs.example.edu"]
 342321  | ["789123@example-isp.com","sjohnson@eng.example.com","sara@johnson.example.name"]
```

Limits on element count and collection size

When declaring a collection type for a table column, you can limit either the number of elements or the total binary size of the collection. During query processing, Vertica always reserves the maximum memory needed for the column, based on either the element count or the binary size. If this size is much larger than your data actually requires, setting one of these limits can improve query performance by reducing the amount of memory that must be reserved for the column.

You can change the bounds of a collection, including changing between bounded and unbounded collections, by casting. See Casting .

A bounded collection specifies a maximum element count. A value in a bounded collection column may contain fewer elements, but it may not contain more. Any attempt to insert more elements into a bounded collection than the declared maximum is an error. A bounded collection has a binary size that is the product of the data-type size and the maximum number of elements, possibly rounded up.

An unbounded collection specifies a binary size in bytes, explicitly or implicitly. It may contain as many elements as can fit in that binary size.

Instead of specifying a bound, you can specify a maximum binary size for an unbounded collection. The binary size acts as an absolute limit, regardless of how many elements the collection contains. Collections that do not specify a maximum binary size use the value of DefaultArrayBinarySize . This size is set at the time the collection is defined and is not affected by later changes to the value of DefaultArrayBinarySize.

You cannot set a maximum binary size for a bounded collection, only an unbounded one.

Comparisons

All collections support equality (=), inequality (<>), and null-safe equality (<=>). 1D collections also support comparison operators (< , <= , > , >=) between collections of the same type (arrays or sets). Comparisons follow these rules:

- A null collection is ordered last.
- Non-null collections are compared element by element, using the ordering rules of the element's data type. The relative order of the first pair of non-equal elements determines the order of the two collections.
- If all elements in both collections are equal up to the length of the shorter collection, the shorter collection is ordered before the longer one.
- If all elements in both collections are equal and the collections are of equal length, the collections are equal.

Null handling

Null semantics for collections are consistent with normal columns in most regards. See NULL sort order for more information on null-handling.

The null-safe equality operator (<=>) behaves differently from equality (=) when the collection is null rather than empty. Comparing a collection to NULL strictly returns null:

```
=> SELECT ARRAY[1,3] = NULL;
?column?
----------

(1 row)

=> SELECT ARRAY[1,3] <=> NULL;
 ?column?
----------
 f
(1 row)
```

In the following example, the grants column in the table is null for employee 99:

```
=> SELECT grants = NULL FROM employees WHERE id=99;
 ?column?
----------

(1 row)

=> SELECT grants <=> NULL FROM employees WHERE id=99;
 ?column?
----------
 t
(1 row)
```

Empty collections are not null and behave as expected:

```
=> SELECT ARRAY[]::ARRAY[INT] = ARRAY[]::ARRAY[INT];
 ?column?
----------
 t
(1 row)
```

Collections are compared element by element. If a comparison depends on a null element, the result is unknown (null), not false. For example, ARRAY[1,2,null]=ARRAY[1,2,null] and ARRAY[1,2,null]=ARRAY[1,2,3] both return null, but ARRAY[1,2,null]=ARRAY[1,4,null] returns false because the second elements do not match.

Casting

Casting a set casts each element of the set. You can therefore cast between data types following the same rules as for casts of scalar values.

You can cast both literal sets and set columns explicitly:

```
=> SELECT SET['1','2','3']::SET[INT];
   set
---------
[1,2,3]
(1 row)

=> CREATE TABLE transactions (tid INT, prod_ids SET[VARCHAR], quantities SET[VARCHAR(32)]);

=> INSERT INTO transactions VALUES (12345, SET['p1265', 'p4515'], SET['15','2']);

=> SELECT quantities :: SET[INT] FROM transactions;
 quantities
------------
   [15,2]
(1 row)
```

Assignment casts and implicit casts work the same way as for scalars.

You can perform explicit casts, but not implicit casts, between ARRAY and SET types. If the collection is unbounded and the data type does not change, the binary size is preserved. For example, if you cast an ARRAY[INT] to a SET[INT], the set has the same binary size as the array.

When casting an array to a set, Vertica first casts each element and then sorts the set and removes duplicates. If two source values are cast to the same target value, one of them will be removed. For example, if you cast an array of FLOAT to a set of INT, two values in the array might be rounded to the same integer and then be treated as duplicates. This also happens if the array contains more than one value that is cast to NULL.

If you cast from one element type to another, the resulting collection uses the default binary size. If this would cause the data not to fit, the cast fails.

Functions and operators

See Collection functions for a comprehensive list of functions that can be used to manipulate arrays and sets.

Collections can be used in the following ways:

- As the grouping column in a GROUP BY clause .
- For native arrays only, as the sort key in an ORDER BY clause in a query, in an OVER clause (see Window partitioning), or in a CREATE PROJECTION statement.

- As the sort key in the PARTITION BY part of an OVER clause.
- As a JOIN key (see Joined-table).
- In CASE expressions .

Collections cannot be used in the following ways:

- As part of an IN or NOT IN expression.
- As partition columns when creating tables.
- With ANALYZE_STATISTICS or TopK projections.
- Non-native arrays only: ORDER BY, PARTITION BY, DEFAULT, SET USING, or constraints.

Data type mappings between Vertica and Oracle

Oracle uses proprietary data types for all main data types, such as VARCHAR, INTEGER, FLOAT, DATE. Before migrating a database from Oracle to Vertica, first convert the schema to minimize errors and time spent fixing erroneous data issues.

The following table compares the behavior of Oracle data types to Vertica data types.

Oracle	Vertica	Notes
NUMBER (no explicit precision)	INTEGER	In Oracle, the NUMBER data type with no explicit precision stores each number *N* as an integer *M* , together with a scale S. The scale can range from -84 to 127, while the precision of *M* is limited to 38 digits. Thus: $N = M * 10^{S}$ When precision is specified, precision/scale applies to all entries in the column. If omitted, the scale defaults to 0. For the common case—Oracle NUMBER with no explicit precision used to store only integer values—the Vertica INTEGER data type is the most appropriate and the fastest equivalent data type. However, INTEGER is limited to a little less than 19 digits, with a scale of 0: [-9223372036854775807, +9223372036854775807] .
	NUMERIC	If an Oracle column contains integer values outside of the range [-9223372036854775807, +9223372036854775807] , then use the Vertica data type NUMERIC(*p* ,0) where *p* is the maximum number of digits required to represent values of the source data. If the data is exact with fractional places—for example dollar amounts—Vertica recommends NUMERIC(*p* , *s*) where *p* is the precision (total number of digits) and *s* is the maximum scale (number of decimal places). Vertica conforms to standard SQL, which requires that $p \geq s$ and $s \geq 0$. Vertica's NUMERIC data type is most effective for *p* =18, and increasingly expensive for *p* =37, 58, 67, etc., where $p \leq 1024$. **Tip** Vertica recommends against using the data type NUMERIC(38, *s*) as a default "failsafe" mapping to guarantee no loss of precision. NUMERIC(18, *s*) is better, and INTEGER or FLOAT better yet, if one of these data types will do the job.
	FLOAT	Even though no explicit scale is specified for an Oracle NUMBER column, Oracle allows non-integer values, each with its own scale. If the data stored in the column is approximate, Vertica recommends using the Vertica data type FLOAT, which is standard IEEE floating point, like ORACLE BINARY_DOUBLE.
NUMBER(*P* ,0) $P \leq 18$	INTEGER	For Oracle NUMBER data types with 0 scale and a precision less than or equal to 18, use the Vertica INTEGER data type.
NUMBER(*P* ,0) $P > 18$	NUMERIC (*p* ,0)	In the rare case where a Oracle column specifies precision greater than 18, use the Vertica data type NUMERIC(*p* , 0), where $p = P$.

NUMBER(*P* , *S*) All cases other than above	NUMERIC (*p* , *s*) FLOAT	• When *P* ≥ *S* and *S* ≥ 0, use *p* = *P* and *s* = *S* , unless the data allows reducing *P* or using FLOAT as discussed above. • If *S* > *P* , use *p* = *S* , *s* = *S* . • If *S* < 0, use *p* = *P* – S, *s* = 0.
NUMERIC(*P* , *S*)		Rarely used in Oracle, see notes for Oracle NUMBER .
DECIMAL(*P* , *S*)		Synonym for Oracle NUMERIC.
BINARY_FLOAT	FLOAT	Same as FLOAT(53) or DOUBLE PRECISION
BINARY_DOUBLE	FLOAT	Same as FLOAT(53) or DOUBLE PRECISION
RAW	VARBINARY	Maximum sizes compared: • Oracle RAW data type: 2000 bytes • Vertica VARBINARY: 65000 bytes
LONG RAW	LONG VARBINARY	Maximum sizes compared: • Oracle's LONG RAW is 2GB • Vertica LONG VARBINARY is 32M bytes/octets (~30MB) **Caution** Be careful to avoid truncation when migrating Oracle LONG RAW data to Vertica.
CHAR(*n*)	CHAR (*n*)	Maximum sizes compared: • Oracle CHAR: 2000 bytes • Vertica CHAR : 65000 bytes
NCHAR(*n*)	CHAR (*n**3)	Vertica supports national characters with CHAR(*n*) as variable-length UTF8-encoded UNICODE character string. UTF-8 represents ASCII in 1 byte, most European characters in 2 bytes, and most oriental and Middle Eastern characters in 3 bytes.
VARCHAR2(*n*)	VARCHAR (*n*)	Maximum sizes compared: • Oracle VARCHAR2: 4000 bytes • Vertica VARCHAR: 65000 bytes Important The Oracle VARCHAR2 and Vertica VARCHAR data types are semantically different: • VARCHAR exhibits standard SQL behavior • VARCHAR2 is inconsistent with standard SQL behavior in that it treats an empty string as NULL value, and uses non-padded comparison if one operand is VARCHAR2.
NVARCHAR2(*n*)	VARCHAR (*n**3)	See notes for NCHAR .
DATE	TIMESTAMP DATE	Oracle's DATE is different from the SQL standard DATE data type implemented by Vertica. Oracle's DATE includes the time (no fractional seconds), while Vertica DATE data types include only date as per the SQL standard.

TIMESTAMP	TIMESTAMP	TIMESTAMP defaults to six places—that is, to microseconds.
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITH TIME ZONE	TIME ZONE defaults to the currently SET or system time zone.
INTERVAL YEAR TO MONTH	INTERVAL YEAR TO MONTH	As per the SQL standard, you can qualify Vertica INTERVAL data types with the YEAR TO MONTH subtype.
INTERVAL DAY TO SECOND	INTERVAL DAY TO SECOND	The default subtype for Vertica INTERVAL data types is DAY TO SECOND.
CLOB	LONG VARCHAR	You can store a CLOB (character large object) or BLOB (binary large object) value in a table or in an external location. The maximum size of a CLOB or BLOB is 128 TB. You can store Vertica LONG data types only in LONG VARCHAR and LONG VARBINARY columns. The maximum size of LONG data types is 32M bytes.
BLOB	LONG VARBINARY	
LONG	LONG VARCHAR	Oracle recommends using CLOB and BLOB data types instead of LONG and LONG RAW data types. An Oracle table can contain only one LONG column, The maximum size of a LONG or LONG RAW data type is 2 GB.
LONG RAW	LONG VARBINARY	

Configuration parameters

Vertica supports a wide variety of configuration parameters that affect many facets of database behavior. These parameters can be set with the appropriate ALTER statements at one or more levels, listed here in descending order of precedence:

1. User (ALTER USER)
2. Session (ALTER SESSION)
3. Node (ALTER NODE)
4. Database (ALTER DATABASE)

You can query system table CONFIGURATION_PARAMETERS to obtain the current settings for all user-accessible parameters. For example, the following query obtains settings for partitioning parameters: their current and default values, which levels they can be set at, and whether changes require a database restart to take effect:

```
=> SELECT parameter_name, current_value, default_value, allowed_levels, change_requires_restart
     FROM configuration_parameters  WHERE parameter_name ILIKE '%partitioncount%';
   parameter_name    | current_value | default_value | allowed_levels | change_requires_restart
---------------------+---------------+---------------+----------------+-------------------------
 MaxPartitionCount   | 1024          | 1024          | NODE, DATABASE | f
 ActivePartitionCount | 1             | 1             | NODE, DATABASE | f
(2 rows)
```

In this section

- General parameters
- Azure parameters
- Constraints parameters
- Database Designer parameters
- Eon Mode parameters
- Epoch management parameters
- Google Cloud Storage parameters
- Hadoop parameters
- Internationalization parameters
- Kafka user-defined session parameters
- Kerberos parameters
- Machine learning parameters

- Memory management parameters
- Monitoring parameters
- Numeric precision parameters
- Profiling parameters
- Projection parameters
- S3 parameters
- SNS parameters
- Security parameters
- Stored procedure parameters
- Text search parameters
- Tuple mover parameters

General parameters

The following parameters configure basic database operations. Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

ApplyEventsDuringSALCheck

Boolean, specifies whether Vertica uses catalog events to filter out dropped corrupt partitions during node startup. Dropping corrupt partitions can speed node recovery.

When disabled (0), Vertica reports corrupt partitions, but takes no action. Leaving corrupt partitions in place can reset the current projection checkpoint epoch to the epoch before the corruption occurred.
This parameter has no effect on unpartitioned tables.

Default: 0

ApportionedFileMinimumPortionSizeKB

Specifies the minimum portion size (in kilobytes) for use with apportioned file loads. Vertica apportions a file load across multiple nodes only if:

- The load can be divided into portions at least equaling this value.
- EnableApportionedFileLoad and EnableApportionLoad are set to 1 (enabled).

See also EnableApportionLoad and EnableApportionedFileLoad.
Default: 1024

BlockedSocketGracePeriod

Sets how long a session socket remains blocked while awaiting client input or output for a given query. See Handling session socket blocking .

Default: None (Socket blocking can continue indefinitely.)

CatalogCheckpointPercent

Specifies the threshold at which a checkpoint is created for the database catalog.

By default, this parameter is set to 50 (percent), so when transaction logs reach 50% of the size of the last checkpoint, Vertica adds a checkpoint. Each checkpoint demarcates all changes to the catalog since the last checkpoint.
Default: 50 (percent)

ClusterSequenceCacheMode

Boolean, specifies whether the initiator node requests cache for other nodes in a cluster, and then sends cache to other nodes along with the execution plan, one of the following.

- 1 (enabled): Initiator node requests cache.
- 0: (disabled): All nodes request their own cache.

See Distributing sequences .
Default: 1 (enabled)

CompressCatalogOnDisk

Whether to compress the size of the catalog on disk, one of the following:

- 0: Do not compress.
- 1: Compress checkpoints, but not logs.
- 2: Compress checkpoints and logs.

This parameter is most effective if the catalog disk partition is small (<50 GB) and the metadata is large (hundreds of tables, partitions, or nodes).
Default: 1

CompressNetworkData

Boolean, specifies whether to compress all data sent over the internal network when enabled (set to 1). This compression speeds up network traffic at the expense of added CPU load. If the network is throttling database performance, enable compression to correct the issue.

Default: 0

CopyFaultTolerantExpressions

Boolean, indicates whether to report record rejections during transformations and proceed (true) or abort COPY operations if a transformation fails.

Default: 0 (false)

CopyFromVerticaWithIdentity

Allows COPY FROM VERTICA and EXPORT TO VERTICA to load values into IDENTITY columns. The destination IDENTITY column is not incremented automatically. To disable the default behavior, set this parameter to 0 (zero).

Default: 1

DatabaseHeartbeatInterval

Determines the interval (in seconds) at which each node performs a health check and communicates a heartbeat. If a node does not receive a message within five times of the specified interval, the node is evicted from the cluster. Setting the interval to 0 disables the feature.

See Automatic eviction of unhealthy nodes .
Default: 120

DefaultArrayBinarySize

The maximum binary size, in bytes, for an unbounded collection, if a maximum size is not specified at creation time.

Default: 65000

DefaultResourcePoolForUsers

Resource pool that is assigned to the profile of a new user, whether created in Vertica or LDAP. This pool is also assigned to users when their assigned resource pool is dropped.

You can set DefaultResourcePoolForUsers only to a global resource pool; attempts to set it to a subcluster resource pool return with an error.

> **Note**
> If an LDAP user is merged with an existing Vertica user, the resource pool setting on the existing user remains unchanged.

For details, see User resource allocation .

Default: GENERAL

DefaultTempTableLocal

Boolean, specifies whether CREATE TEMPORARY TABLE creates a local or global temporary table, one of the following:

- 0: Create global temporary table.
- 1: Create local temporary table.

For details, see Creating temporary tables .
Default: 0

DivideZeroByZeroThrowsError

Boolean, specifies whether to return an error if a division by zero operation is requested:

- 0: Return 0.
- 1: Returns an error.

Default: 1

EnableApportionedChunkingInDefaultLoadParser

Boolean, specifies whether to enable the built-in parser for delimited files to take advantage of both apportioned load and cooperative parse for potentially better performance.

Default: 1 (enable)

EnableApportionedFileLoad

Boolean, specifies whether to enable automatic apportioning across nodes of file loads using COPY FROM VERTICA . Vertica attempts to apportion the load if:

- This parameter and EnableApportionLoad are both enabled.
- The parser supports apportioning.
- The load is divisible into portion sizes of at least the value of ApportionedFileMinimumPortionSizeKB.

Setting this parameter does not guarantee that loads will be apportioned, but disabling it guarantees that they will not be.
See **Distributing a load .**

Default: 1 (enable)

EnableApportionLoad

Boolean, specifies whether to enable automatic apportioning across nodes of data loads using COPY...WITH SOURCE . Vertica attempts to apportion the load if:

- This parameter is enabled.
- The source and parser both support apportioning.

Setting this parameter does not guarantee that loads will be apportioned, but disabling it guarantees that they will not be.
For details, see Distributing a load .

Default: 1 (enable)

EnableBetterFlexTypeGuessing

Boolean, specifies whether to enable more accurate type guessing when assigning data types to non-string keys in a flex table __raw__ column with COMPUTE_FLEXTABLE_KEYS or COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW . If this parameter is disabled (0), Vertica uses a limited set of Vertica data type assignments.

Default: 1 (enable)

EnableCooperativeParse

Boolean, specifies whether to implement multi-threaded parsing capabilities on a node. You can use this parameter for both delimited and fixed-width loads.

Default: 1 (enable)

EnableForceOuter

Boolean, specifies whether Vertica uses a table's force_oute r value to implement a join. For more information, see Controlling join inputs .

Default: 0 (forced join inputs disabled)

EnableMetadataMemoryTracking

Boolean, specifies whether to enable Vertica to track memory used by database metadata in the METADATA resource pool .

Default: 1 (enable)

EnableResourcePoolCPUAffinity

Boolean, specifies whether Vertica aligns queries to the resource pool of the processing CPU. When disabled (0), queries run on any CPU, regardless of the CPU_AFFINITY_SET of the resource pool.

Default: 1

EnableStrictTimeCasts

Specifies whether all cast failures result in an error.

Default: 0 (disable)

EnableUniquenessOptimization

Boolean, specifies whether to enable query optimization that is based on guaranteed uniqueness of column values. Columns that can be guaranteed to include unique values include:

- Columns that are defined with IDENTITY constraints
- Primary key columns where key constraints are enforced
- Columns that are constrained to unique values , either individually or as a set

Default: 1 (enable)

EnableWithClauseMaterialization

Superseded by WithClauseMaterialization .

ExternalTablesExceptionsLimit

Determines the maximum number of COPY exceptions and rejections allowed when a SELECT statement references an external table. Set to -1 to remove any exceptions limit. See Querying external tables .

Default: 100

FailoverToStandbyAfter

Specifies the length of time that an active standby node waits before taking the place of a failed node.

This parameter is set to an interval literal .

Default: None

FencedUDxMemoryLimitMB

Sets the maximum amount of memory, in megabytes (MB), that a fenced-mode UDF can use. If a UDF attempts to allocate more memory than this limit, that attempt triggers an exception. For more information, see Fenced and unfenced modes .

Default: -1 (no limit)

FlexTableDataTypeGuessMultiplier

Specifies a multiplier that the COMPUTE_FLEXTABLE_KEYS and COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW functions use when assigning a data type and column width for the flex keys table. Both functions assign each key a data type, and multiply the longest key value by this factor to estimate column width. This value is not used to calculate the width of any real columns in a flex table.

Default: 2.0

FlexTableRawSize

Specifies the default column width for the __raw__ column of new flex tables, a value between 1 and 32000000, inclusive.

Default: 130000

ForceUDxFencedMode

When enabled (1), forces all UDxs that support fenced mode to run in fenced mode even if their definition specified NOT FENCED.

Default: 0

HTTPServerPortOffset

Controls the offset for the HTTPS port. The default HTTPS port is 8443, the sum of the client port (5433) and default HTTPServerPortOffset (3010).

> **Caution**
> This parameter should not be changed without support guidance.

Default: 3010

IcebergPathMapping

For external tables using Iceberg data, a map of URI prefixes used by Iceberg to prefixes accessible to Vertica. The value is a JSON object:

```
'{"source-prefix":"new-prefix"[,...]}'
```

Specify prefixes only (up through port number), not complete paths.

IdleTimeoutInitializingConnectionsMs

The length of time (in milliseconds) that the server waits before timing out, during each step in connection initialization. After connection initialization, the session is created.

Default : 10000

JavaBinaryForUDx

Sets the full path to the Java executable that Vertica uses to run Java UDxs. See Installing Java on Vertica hosts .

JoinDefaultTupleFormat

Specifies how to size VARCHAR column data when joining tables on those columns, and buffers accordingly, one of the following:

- fixed : Use join column metadata to size column data to a fixed length, and buffer accordingly.
- variable : Use the actual length of join column data, so buffer size varies for each join.

Default: fixed

KeepAliveIdleTime

Length (in seconds) of the idle period before the first TCP keepalive probe is sent to ensure that the client is still connected. If set to 0, Vertica uses the kernel's tcp_keepalive_time parameter setting.

Default: 0

KeepAliveProbeCount

Number of consecutive keepalive probes that must go unacknowledged by the client before the client connection is considered lost and closed. If set to 0, Vertica uses the kernel's tcp_keepalive_probes parameter setting.

Default: 0

KeepAliveProbeInterval

Time interval (in seconds) between keepalive probes. If set to 0, Vertica uses the kernel's tcp_keepalive_intvl parameter setting.

Default: 0

LockTimeout

Specifies in seconds how long a table waits to acquire a lock.

Default: 300

LoadSourceStatisticsLimit

Specifies the maximum number of sources per load operation that are profiled in the LOAD_SOURCES system table. Set it to 0 to disable profiling.

Default: 256

MaxBundleableROSSizeKB

Specifies the minimum size, in kilobytes, of an independent ROS file. Vertica bundles storage container ROS files below this size into a single file. Bundling improves the performance of any file-intensive operations, including backups, restores, and mergeouts.

If you set this parameter to a value of 0, Vertica bundles .fdb and .pidx files without bundling other storage container files.
Default: 1024

MaxClientSessions

Determines the maximum number of client sessions that can run on a single node of the database. The default value allows for five additional administrative logins. These logins prevent DBAs from being locked out of the system if non-dbadmin users reach the login limit.

> **Tip**
> Setting this parameter to 0 prevents new client sessions from being opened while you are shutting down the database. Restore the parameter to its original setting after you restart the database. For details, see Managing Sessions .

Default: 50 user logins and 5 additional administrative logins

ObjectStoreGlobStrategy

For partitioned external tables in object stores, the strategy to use for expanding globs before pruning partitions:

- Flat : COPY fetches a list of all full object names with a given prefix, which can incur latency if partitions are numerous or deeply nested.
- Hierarchical : COPY fetches object names one partition layer at a time, allowing earlier pruning but possibly requiring more calls to the object store when queries are not selective or there are not many partittion directory levels.

For details, see Partitions on Object Stores .
Default: Flat

ParquetMetadataCacheSizeMB

Size of the cache used for metadata when reading Parquet data. The cache uses local TEMP storage.

Default: 4096

PatternMatchingUseJit

Boolean, specifies whether to enables just-in-time compilation (to machine code) of regular expression pattern matching functions used in queries. Enabling this parameter can usually improve pattern matching performance on large tables. The Perl Compatible Regular Expressions (PCRE) pattern-match library evaluates regular expressions. Restart the database for this parameter to take effect.

See also Regular expression functions .
Default: 1 (enable)

PatternMatchStackAllocator

Boolean, specifies whether to override the stack memory allocator for the pattern-match library. The Perl Compatible Regular Expressions (PCRE) pattern-match library evaluates regular expressions. Restart the database for this parameter to take effect.

See also Regular expression functions .
Default: 1 (enable override)

TerraceRoutingFactor

Specifies whether to enable or disable terrace routing on any Enterprise Mode large cluster that implements rack-based fault groups.

- To enable, set as follows:

  $$\text{TerraceRoutingFactor} < \frac{(numRackNodes-1) + (numRacks-1)}{(numRacks * numRackNodes) - 1}$$

  where: * *numRackNodes* : Number of nodes in a rack * *numRacks* : Number of racks in the cluster
- To disable, set to a value so large that terrace routing cannot be enabled for the largest clusters—for example, 1000.

For details, see Terrace routing .

Default: 2

TransactionIsolationLevel

Changes the isolation level for the database. After modification, Vertica uses the new transaction level for every new session. Existing sessions and their transactions continue to use the original isolation level.

See also Change transaction isolation levels .

Default: READ COMMITTED

TransactionMode

Specifies whether transactions are in read/write or read-only modes. Read/write is the default. Existing sessions and their transactions continue to use the original isolation level.

Default: READ WRITE

UDxFencedBlockTimeout

Specifies the number of seconds to wait for output before aborting a UDx running in Fenced and unfenced modes . If the server aborts a UDx for this reason, it produces an error message similar to "ERROR 3399: Failure in UDx RPC call: timed out in receiving a UDx message". If you see this error frequently, you can increase this limit. UDxs running in fenced mode do not run in the server process, so increasing this value does not impede server performance.

Default: 60

UseLocalTzForParquetTimestampConversion

Boolean, specifies whether to do timezone conversion when reading Parquet files. Hive version 1.2.1 introduced an option to localize timezones when writing Parquet files. Previously it wrote them in UTC and Vertica adjusted the value when reading the files.

Set to 0 if Hive already adjusted the timezones.

Default: 1 (enable conversion)

UseServerIdentityOverUserIdentity

Boolean, specifies whether to ignore user-supplied credentials for non-Linux file systems and always use a USER storage location to govern access to data. See Creating a Storage Location for USER Access .

Default: 0 (disable)

WithClauseMaterialization

Boolean, specifies whether to enable materialization of WITH clause results. When materialization is enabled (1), Vertica evaluates each WITH clause once and stores results in a temporary table.

For WITH queries with complex types, temp relations are disabled.

> **Note**
> You can enable materialization of a given WITH clause with the hint ENABLE_WITH_CLAUSE_MATERIALIZATION .

Default: 0 (disable)

WithClauseRecursionLimit

Specifies the maximum number of times a WITH RECURSIVE clause iterates over the content of its own result set before it exits. For details, see WITH clause recursion .

Important

Be careful to set WithClauseRecursionLimit only as high as needed to traverse the deepest hierarchies. Vertica sets no limit on this parameter; however, a high value can incur considerable overhead that adversely affects performance and exhausts system resources.

If a high recursion count is required, then consider enabling materialization. For details, see WITH RECURSIVE Materialization .

Default: 8

Azure parameters

Use the following parameters to configure reading from Azure blob storage. For more information about reading data from Azure, see Azure Blob Storage object store .

For external tables using highly partitioned data in an object store, see the ObjectStoreGlobStrategy configuration parameter and Partitions on Object Stores .

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

AzureStorageCredentials

Collection of JSON objects, each of which specifies connection credentials for one endpoint. This parameter takes precedence over Azure managed identities.

The collection must contain at least one object and may contain more. Each object must specify at least one of accountName or blobEndpoint , and at least one of accountKey or sharedAccessSignature .

- accountName : If not specified, uses the label of blobEndpoint .
- blobEndpoint : Host name with optional port (host:port). If not specified, uses *account* .blob.core.windows.net .
- accountKey : Access key for the account or endpoint.
- sharedAccessSignature : Access token for finer-grained access control, if being used by the Azure endpoint.

AzureStorageEndpointConfig

Collection of JSON objects, each of which specifies configuration elements for one endpoint. Each object must specify at least one of accountName or blobEndpoint .

- accountName : If not specified, uses the label of blobEndpoint .
- blobEndpoint : Host name with optional port (host:port). If not specified, uses *account* .blob.core.windows.net .
- protocol : HTTPS (default) or HTTP.
- isMultiAccountEndpoint : true if the endpoint supports multiple accounts, false otherwise (default is false). To use multiple-account access, you must include the account name in the URI. If a URI path contains an account, this value is assumed to be true unless explicitly set to false.

Constraints parameters

The following configuration parameters control how Vertica evaluates and enforces constraints. All parameters are set at the database level through ALTER DATABASE .

Three of these parameters—EnableNewCheckConstraintsByDefault, EnableNewPrimaryKeysByDefault, and EnableNewUniqueKeysByDefault—can be used to enforce CHECK, PRIMARY KEY, and UNIQUE constraints, respectively. For details, see Constraint enforcement .

EnableNewCheckConstraintsByDefault

Boolean parameter, set to 0 or 1:

- 0: Disable enforcement of new CHECK constraints except where the table DDL explicitly enables them.
- 1 (default): Enforce new CHECK constraints except where the table DDL explicitly disables them.

EnableNewPrimaryKeysByDefault

Boolean parameter, set to 0 or 1:

- 0 (default): Disable enforcement of new PRIMARY KEY constraints except where the table DDL explicitly enables them.
- 1: Enforce new PRIMARY KEY constraints except where the table DDL explicitly disables them.

> **Note**
> Vertica recommends enforcing constraints PRIMARY KEY and UNIQUE together.

EnableNewUniqueKeysByDefault

Boolean parameter, set to 0 or 1:

- 0 (default): Disable enforcement of new UNIQUE constraints except where the table DDL explicitly enables them.
- 1: Enforce new UNIQUE constraints except where the table DDL explicitly disables them.

MaxConstraintChecksPerQuery

Sets the maximum number of constraints that ANALYZE_CONSTRAINTS can handle with a single query:

- -1 (default): No maximum set, ANALYZE_CONSTRAINTS uses a single query to evaluate all constraints within the specified scope.

- Integer > 0: The maximum number of constraints per query. If the number of constraints to evaluate exceeds this value, ANALYZE_CONSTRAINTS handles it with multiple queries.

For details, see Distributing Constraint Analysis .

Database Designer parameters

The following table describes the parameters for configuring the Vertica Database Designer.

DBDCorrelationSampleRowCount

Minimum number of table rows at which Database Designer discovers and records correlated columns.

Default: 4000

DBDLogInternalDesignProcess

Enables or disables Database Designer logging.

Default: 0 (False)

DBDUseOnlyDesignerResourcePool

Enables use of the DBD pool by the Vertica Database Designer.

When set to false, design processing is mostly contained by the user's resource pool, but might spill over into some system resource pools for less-intensive tasks
Default: 0 (False)

Eon Mode parameters

The following parameters configure how the database operates when running in Eon Mode. Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

BackgroundDepotWarming

Specifies background depot warming behavior:

- 1: The depot loads objects while it is warming, and continues to do so in the background after the node becomes active and starts executing queries.
- 0: Node activation is deferred until the depot fetches and loads all queued objects

For details, see Depot Warming .
Default: 1

CatalogSyncInterval

Specifies in minutes how often the transaction log sync service syncs metadata to communal storage. If you change this setting, Vertica restarts the interval count.

Default: 5

DelayForDeletes

Specifies in hours how long to wait before deleting a file from communal storage. Vertica first deletes a file from the depot. After the specified time interval, the delete also occurs in communal storage.

Default: 0. Deletes the file from communal storage as soon as it is not in use by shard subscribers.

DepotOperationsForQuery

Specifies behavior when the depot does not contain queried file data, one of the following:

- ALL (default): Fetch file data from communal storage, if necessary displace existing files by evicting them from the depot.
- FETCHES : Fetch file data from communal storage only if space is available; otherwise, read the queried data directly from communal storage.
- NONE : Do not fetch file data to the depot, read the queried data directly from communal storage.

You can also specify query-level behavior with the hint DEPOT_FETCH .

ECSMode

String parameter that sets the strategy Vertica uses when dividing the data in a shard among subscribing nodes during an ECS-enabled query, one of the following:

- AUTO : Optimizer automatically determines the strategy to use.
- COMPUTE_OPTIMIZED : Force use of the compute-optimized strategy.
- IO_OPTIMIZED : Force use of the I/O-optimized strategy.

For details, see Manually choosing an ECS strategy .
Default: AUTO

ElasticKSafety
Boolean parameter that controls whether Vertica adjusts shard subscriptions due to the loss of a primary node :

- 1: When a primary node is lost, Vertica subscribes other primary nodes to the down node's shard subscriptions. This action helps reduce the chances of a database into going read-only mode due to loss of shard coverage.
- 0 : Vertica does not change shard subscriptions in reaction to the loss of a primary node.

Default: 1
For details, see Maintaining Shard Coverage .

EnableDepotWarmingFromPeers
Boolean parameter, specifies whether Vertica warms a node depot while the node is starting up and not ready to process queries:

- 1: Warm the depot while a node comes up.
- 0: Warm the depot only after the node is up.

For details, see Depot Warming .
Default: 0

FileDeletionServiceInterval
Specifies in seconds the interval between each execution of the reaper cleaner service task.

Default: 60 seconds

MaxDepotSizePercent
An integer value that specifies the maximum size of the depot as a percentage of disk size,

Default: 80

PreFetchPinnedObjectsToDepotAtStartup
If enabled (set to 1), a warming depot fetches objects that are pinned on its subcluster. For details, see Depot Warming .

Default: 0

ReaperCleanUpTimeoutAtShutdown
Specifies in seconds how long Vertica waits for the reaper to delete files from communal storage before shutting down. If set to a negative value, Vertica shuts down without waiting for the reaper.

> **Note**
> The reaper is a service task that deletes disk files.

Default: 300

StorageMergeMaxTempCacheMB
The size of temp space allocated per query to the StorageMerge operator for caching the data of S3 storage containers.

> **Note**
> The actual temp space that is allocated is the lesser of:
>
> - StorageMergeMaxTempCacheMB
> - A user's TEMPSPACECAP setting
> - The session TEMPSPACECAP setting

For details, see Local caching of storage containers .

UseCommunalStorageForBatchDepotWarming
Boolean parameter, specifies whether where a node retrieves data when warming its depot:

- 1: Retrieve data from communal storage.
- 0: Retrieve data from a peer.

> **Note**

The actual temp space that is allocated is the lesser of two settings:

Default: 1
Important
This parameter is for internal use only. Do not change it unless directed to do so by Vertica support.

UseDepotForReads
Boolean parameter, specifies whether Vertica accesses the depot to answer queries, or accesses only communal storage:
- 1: Vertica first searches the depot for the queried data; if not there, Vertica fetches the data from communal storage for this and future queries.
- 0: Vertica bypasses the depot and always obtains queried data from communal storage.

Note
Enable depot reads to improve query performance and support K-safety.

Default: 1

UseDepotForWrites
Boolean parameter, specifies whether Vertica writes loaded data to the depot and then uploads files to communal storage:
- 1: Write loaded data to the depot, upload files to communal storage.
- 0: Bypass the depot and always write directly to communal storage.

Default: 1

UsePeerToPeerDataTransfer
Boolean parameter, specifies whether Vertica pushes loaded data to other shard subscribers:
- 1: Send loaded data to all shard subscribers.
- 0: Do not push data to other shard subscribers.

Note
Setting to 1 helps improve performance when a node is down.

Default: 0
Important
This parameter is for internal use only. Do not change it unless directed to do so by Vertica support.

Epoch management parameters

The following table describes the epoch management parameters for configuring Vertica.

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

AdvanceAHMInterval
Determines how frequently (in seconds) Vertica checks the history retention status.

AdvanceAHMInterval cannot be set to a value that is less than the EpochMapInterval.
Default: 180 (seconds)

AHMBackupManagement
Blocks the advancement of the Ancient History Mark (AHM). When this parameter is enabled, the AHM epoch cannot be later than the epoch of your latest full backup. If you advance the AHM to purge and delete data, do not enable this parameter.

Caution
Do not enable this parameter before taking full backups, as it would prevent the AHM from advancing.

Default: 0

EpochMapInterval

Determines the granularity of mapping between epochs and time available to historical queries . When a historical queries AT TIME T request is issued, Vertica maps it to an epoch within a granularity of EpochMapInterval seconds. It similarly affects the time reported for Last Good Epoch during Failure recovery . Note that it does not affect internal precision of epochs themselves.

> **Tip**
> Decreasing this interval increases the number of epochs saved on disk. Therefore, consider reducing the HistoryRetentionTime parameter to limit the number of history epochs that Vertica retains.

Default: 180 (seconds)

HistoryRetentionTime

Determines how long deleted data is saved (in seconds) as an historical reference. When the specified time since the deletion has passed, you can purge the data. Use the -1 setting if you prefer to use HistoryRetentionEpochs to determine which deleted data can be purged.

> **Note**
> The default setting of 0 effectively prevents the use of the Administration tools 'Roll Back Database to Last Good Epoch' option because the AHM remains close to the current epoch and a rollback is not permitted to an epoch prior to the AHM.

> **Tip**
> If you rely on the Roll Back option to remove recently loaded data, consider setting a day-wide window to remove loaded data. For example:
>
> ```
> ALTER DATABASE DEFAULT SET HistoryRetentionTime = 86400;
> ```

Default: 0 (Data saved when nodes are down.)

HistoryRetentionEpochs

Specifies the number of historical epochs to save, and therefore, the amount of deleted data.

Unless you have a reason to limit the number of epochs, Vertica recommends that you specify the time over which deleted data is saved. If you specify both History parameters, HistoryRetentionTime takes precedence. Setting both parameters to -1, preserves all historical data.

See Setting a purge policy .

Default: -1 (disabled)

Google Cloud Storage parameters

Use the following parameters to configure reading from Google Cloud Storage (GCS) using COPY FROM. For more information about reading data from S3, see Specifying where to load data from .

For external tables using highly partitioned data in an object store, see the ObjectStoreGlobStrategy configuration parameter and Partitions on Object Stores .

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

GCSAuth

An ID and secret key to authenticate to GCS. You can set parameters globally and for the current session with ALTER DATABASE...SET PARAMETER and ALTER SESSION...SET PARAMETER , respectively. For extra security, do not store credentials in the database; instead, set it for the current session with ALTER SESSION. For example:

```
=> ALTER SESSION SET GCSAuth='ID:secret';
```

If you use a shared credential, set it in the database with ALTER DATABASE.

GCSEnableHttps

Specifies whether to use the HTTPS protocol when connecting to GCS, can be set only at the database level with ALTER DATABASE...SET PARAMETER .

Default: 1 (enabled)

GCSEndpoint

The connection endpoint address.

Default: storage.googleapis.com

Hadoop parameters

The following table describes general parameters for configuring integration with Apache Hadoop. See Apache Hadoop integration for more information.

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

EnableHDFSBlockInfoCache

Boolean, whether to distribute block location metadata collected during planning on the initiator to all database nodes for execution. Distributing this metadata reduces name node accesses, and thus load, but can degrade database performance somewhat in deployments where the name node isn't contended. This performance effect is because the data must be serialized and distributed. Enable distribution if protecting the name node is more important than query performance; usually this applies to large HDFS clusters where name node contention is already an issue.

Default: 0 (disabled)

HadoopConfDir

Directory path containing the XML configuration files copied from Hadoop. The same path must be valid on every Vertica node. You can use the VERIFY_HADOOP_CONF_DIR meta-function to test that the value is set correctly. Setting this parameter is required to read data from HDFS.

For all Vertica users, the files are accessed by the Linux user under which the Vertica server process runs.
When you set this parameter, previously-cached configuration information is flushed.

You can set this parameter at the session level. Doing so overrides the database value; it does not append to it. For example:

```
=> ALTER SESSION SET HadoopConfDir='/test/conf:/hadoop/hcat/conf';
```

To append, get the current value and include it on the new path after your additions. Setting this parameter at the session level does not change how the files are accessed.

Default: obtained from environment if possible

HadoopFSAuthentication

How (or whether) to use Kerberos authentication with HDFS. By default, if KerberosKeytabFile is set, Vertica uses that credential for both Vertica and HDFS. Usually this is the desired behavior. However, if you are using a Kerberized Vertica cluster with a non-Kerberized HDFS cluster, set this parameter to "none" to indicate that Vertica should not use the Vertica Kerberos credential to access HDFS.

Default: "keytab" if KerberosKeytabFile is set, otherwise "none"

HadoopFSBlockSizeBytes

Block size to write to HDFS. Larger files are divided into blocks of this size.

Default: 64MB

HadoopFSNNOperationRetryTimeout

Number of seconds a metadata operation (such as list directory) waits for a response before failing. Accepts float values for millisecond precision.

Default: 6 seconds

HadoopFSReadRetryTimeout

Number of seconds a read operation waits before failing. Accepts float values for millisecond precision. If you are confident that your file system will fail more quickly, you can improve performance by lowering this value.

Default: 180 seconds

HadoopFSReplication

Number of replicas HDFS makes. This is independent of the replication that Vertica does to provide K-safety. Do not change this setting unless directed otherwise by Vertica support.

Default: 3

HadoopFSRetryWaitInterval

Initial number of seconds to wait before retrying read, write, and metadata operations. Accepts float values for millisecond precision. The retry

interval increases exponentially with every retry.

Default: 3 seconds

HadoopFSTokenRefreshFrequency

How often, in seconds, to refresh the Hadoop tokens used to hold Kerberos tickets (see Token expiration).

Default: 0 (refresh when token expires)

HadoopFSWriteRetryTimeout

Number of seconds a write operation waits before failing. Accepts float values for millisecond precision. If you are confident that your file system will fail more quickly, you can improve performance by lowering this value.

Default: 180 seconds

HadoopImpersonationConfig

Session parameter specifying the delegation token or Hadoop user for HDFS access. See HadoopImpersonationConfig format for information about the value of this parameter and Proxy users and delegation tokens for more general context.

HDFSUseWebHDFS

Boolean, whether URLs in the hdfs scheme use WebHDFS instead of LibHDFS++ to access data.

> **Deprecated**
> This parameter is deprecated because LibHDFS++ is deprecated. In the future, Vertica will use WebHDFS for all hdfs URLs.

Default: 1 (enabled)

WebhdfsClientCertConf

mTLS configurations for accessing one or more WebHDFS servers. The value is a JSON string; each member has the following properties:

- nameservice : WebHDFS name service
- authority : *host* : *port*
- certName : name of a certificate defined by CREATE CERTIFICATE

nameservice and authority are mutually exclusive.

For example:

```
=> ALTER SESSION SET WebhdfsClientCertConf =
   '[{"authority" : "my.authority.com:50070", "certName" : "myCert"},
    {"nameservice" : "prod", "certName" : "prodCert"}]';
```

HCatalog Connector parameters

The following table describes the parameters for configuring the HCatalog Connector. See Using the HCatalog Connector for more information.

> **Note**
> You can override HCatalog configuration parameters when you create an HCatalog schema with CREATE HCATALOG SCHEMA .

EnableHCatImpersonation

Boolean, whether the HCatalog Connector uses (impersonates) the current Vertica user when accessing Hive. If impersonation is enabled, the HCatalog Connector uses the Kerberos credentials of the logged-in Vertica user to access Hive data. Disable impersonation if you are using an authorization service to manage access without also granting users access to the underlying files. For more information, see Configuring security .

Default: 1 (enabled)

HCatalogConnectorUseHiveServer2

Boolean, whether Vertica internally uses HiveServer2 instead of WebHCat to get metadata from Hive.

Default: 1 (enabled)

HCatalogConnectorUseLibHDFSPP

Boolean, whether the HCatalog Connector should use the hdfs scheme instead of webhdfs with the HCatalog Connector.

> **Deprecated**

> Vertica uses the hdfs scheme by default. To use webhdfs , set the HDFSUseWebHDFS parameter.

Default: 1 (enabled)

HCatConnectionTimeout
The number of seconds the HCatalog Connector waits for a successful connection to the HiveServer2 (or WebHCat) server before returning a timeout error.

Default: 0 (Wait indefinitely)

HCatSlowTransferLimit
Lowest transfer speed (in bytes per second) that the HCatalog Connector allows when retrieving data from the HiveServer2 (or WebHCat) server. In some cases, the data transfer rate from the server to Vertica is below this threshold. In such cases, after the number of seconds specified in the HCatSlowTransferTime parameter pass, the HCatalog Connector cancels the query and closes the connection.

Default: 65536

HCatSlowTransferTime
Number of seconds the HCatalog Connector waits before testing whether the data transfer from the server is too slow. See the HCatSlowTransferLimit parameter.

Default: 60

Internationalization parameters

The following table describes internationalization parameters for configuring Vertica.

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

DefaultIntervalStyle
Sets the default interval style to use. If set to 0 (default), the interval is in PLAIN style (the SQL standard), no interval units on output. If set to 1, the interval is in UNITS on output. This parameter does not take effect until the database is restarted.

Default: 0

DefaultSessionLocale
Sets the default session startup locale for the database. This parameter does not take effect until the database is restarted.

Default: en_US@collation=binary

EscapeStringWarning
Issues a warning when backslashes are used in a string literal. This can help locate backslashes that are being treated as escape characters so they can be fixed to follow the SQL standard-conforming string syntax instead.

Default: 1

StandardConformingStrings
Determines whether character string literals treat backslashes () as string literals or escape characters. When set to -1, backslashes are treated as string literals; when set to 0, backslashes are treated as escape characters.

> **Tip**
> To treat backslashes as escape characters, use the Extended string syntax
>
> ```
> (E'...');
> ```

Default: -1

Kafka user-defined session parameters

Set Vertica user-defined session parameters to configure Kafka SSL when not using a scheduler, using ALTER SESSION SET UDPARAMETER . The kafka - prefixed parameters configure SSL authentication for Kafka. For details, see TLS/SSL encryption with Kafka .

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

kafka_SSL_CA
The contents of the certificate authority certificate. For example:

```
=> ALTER SESSION SET UDPARAMETER kafka_SSL_CA='MIIBOQIBAAJBAIOL';
```

Default: none

kafka_SSL_Certificate

The contents of the SSL certificate. For example:

```
=> ALTER SESSION SET UDPARAMETER kafka_SSL_Certificate='XrM07O4dV/nJ5g';
```

This parameter is optional when the Kafka server's parameter ssl.client.auth is set to none or requested .

Default: none

kafka_SSL_PrivateKey_secret

The private key used to encrypt the session. Vertica does not log this information. For example:

```
=> ALTER SESSION SET UDPARAMETER kafka_SSL_PrivateKey_secret='A60iThKtezaCk7F';
```

This parameter is optional when the Kafka server's parameter ssl.client.auth is set to none or requested .

Default: none

kafka_SSL_PrivateKeyPassword_secret

The password used to create the private key. Vertica does not log this information.

For example:

```
ALTER SESSION SET UDPARAMETER kafka_SSL_PrivateKeyPassword_secret='secret';
```

This parameter is optional when the Kafka server's parameter ssl.client.auth is set to none or requested .

Default: none

kafka_Enable_SSL

Enables SSL authentication for Vertica-Kafka integration. For example:

```
=> ALTER SESSION SET UDPARAMETER kafka_Enable_SSL=1;
```

Default: 0

MaxSessionUDParameterSize

Sets the maximum length of a value in a user-defined session parameter. For example:

```
=> ALTER SESSION SET MaxSessionUDParameterSize = 2000
```

Default: 1000

Related topics

User-defined session parameters

Kerberos parameters

The following parameters let you configure the Vertica principal for Kerberos authentication and specify the location of the Kerberos keytab file.

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

KerberosEnableKeytabPermissionCheck

Whether the Vertica server verifies permissions on the keytab file. By default, the Vertica server verifies these permissions.

In a hybrid Kubernetes cluster , set this parameter to 0 so that there is no permissions check. Otherwise, Kerberos authentication fails because the keytab file is stored in a Secret, and the VerticaDB operator cannot verify permissions on a Secret.

Default : 1

KerberosHostname

Instance or host name portion of the Vertica Kerberos principal. For example:

```
vertica/host@EXAMPLE.COM
```

If you omit the optional KerberosHostname parameter, Vertica uses the return value from the function gethostname() . Assuming each cluster node has a different host name, those nodes will each have a different principal, which you must manage in that node's keytab file.

KerberosKeytabFile

Location of the keytab file that contains credentials for the Vertica Kerberos principal. By default, this file is located in /etc . For example:

```
KerberosKeytabFile=/etc/krb5.keytab
```

> **Note**
> - The principal must take the form KerberosServiceName/KerberosHostName@KerberosRealm
> - The keytab file must be readable by the file owner who is running the process (typically the Linux dbadmin user assigned file permissions 0600).

KerberosRealm

Realm portion of the Vertica Kerberos principal. A realm is the authentication administrative domain and is usually formed in uppercase letters. For example:

```
vertica/hostEXAMPLE.COM
```

KerberosServiceName

Service name portion of the Vertica Kerberos principal. By default, this parameter is vertica . For example:

```
vertica/host@EXAMPLE.COM
```

Default: vertica

KerberosTicketDuration

Lifetime of the ticket retrieved from performing a kinit. The default is 0 (zero) which disables this parameter.

If you omit setting this parameter, the lifetime is determined by the default Kerberos configuration.

Machine learning parameters

You use machine learning parameters to configure various aspects of machine learning functionality in Vertica.

MaxModelSizeKB

Sets the maximum size of models that can be imported. The sum of the size of files specified in the metadata.json file determines the model size. The unit of this parameter is KBytes. The native Vertica model (category=VERTICA_MODELS) is exempted from this limit. If you can export the model from Vertica, and the model is not altered while outside Vertica, you can import it into Vertica again.

The MaxModelSizeKB parameter can be set only by a superuser and only at the database level. It is visible only to a superuser. Its default value is 4GB, and its valid range is between 1KB and 64GB (inclusive).
Examples:

To set this parameter to 3KB:

```
=> ALTER DATABASE DEFAULT SET MaxModelSizeKB = 3;
```

To set this parameter to 64GB (the maximum allowed):

```
=> ALTER DATABASE DEFAULT SET MaxModelSizeKB = 67108864;
```

To reset this parameter to the default value:

```
=> ALTER DATABASE DEFAULT CLEAR MaxModelSizeKB;
```

Default: 4GB

Memory management parameters

The following table describes parameters for managing Vertica memory usage.

> **Caution**
> Modify these parameters only under guidance from Vertica Support.

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

MemoryPollerIntervalSec

Specifies in seconds how often the Vertica memory poller checks whether Vertica memory usage is below the thresholds of several configuration parameters (see below):

- MemoryPollerMallocBloatThreshold
- MemoryPollerReportThreshold

- MemoryPollerTrimThreshold

Important
To disable polling of all thresholds, set this parameter to 0. Doing so effectively disables automatic memory usage reporting and trimming .

Default: 2

MemoryPollerMallocBloatThreshold
Threshold of glibc memory bloat .

The memory poller calls glibc function malloc_info() to obtain the amount of free memory in malloc. It then compares MemoryPollerMallocBloatThreshold —by default, set to 0.3—with the following expression:

```
free-memory-in-malloc / RSS
```

If this expression evaluates to a value higher than MemoryPollerMallocBloatThreshold , the memory poller calls glibc function malloc_trim() . This function reclaims free memory from malloc and returns it to the operating system. Details on calls to malloc_trim() are written to system table MEMORY_EVENTS .

To disable polling of this threshold, set the parameter to 0.

Default: 0.3

MemoryPollerReportThreshold
Threshold of memory usage that determines whether the Vertica memory poller writes a report .

The memory poller compares MemoryPollerReportThreshold with the following expression:

```
RSS / available-memory
```

When this expression evaluates to a value higher than MemoryPollerReportThreshold —by default, set to 0.93, then the memory poller writes a report to MemoryReport.log , in the Vertica working directory. This report includes information about Vertica memory pools, how much memory is consumed by individual queries and session, and so on. The memory poller also logs the report as an event in system table MEMORY_EVENTS , where it sets EVENT_TYPE to MEMORY_REPORT .

To disable polling of this threshold, set the parameter to 0.

Default: 0.93

MemoryPollerTrimThreshold
Threshold for the memory poller to start checking whether to trim glibc-allocated memory .

The memory poller compares MemoryPollerTrimThreshold —by default, set to 0.83— with the following expression:

```
RSS / available-memory
```

If this expression evaluates to a value higher than MemoryPollerTrimThreshold , then the memory poller starts checking the next threshold—set in MemoryPollerMallocBloatThreshold —for glibc memory bloat.

To disable polling of this threshold, set the parameter to 0. Doing so also disables polling of MemoryPollerMallocBloatThreshold .

Default: 0.83

Monitoring parameters

The following table describes parameters that control options for monitoring the Vertica database.

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

EnableDataCollector
Enables and disables the Data Collector, which is the Workload Analyzer 's internal diagnostics utility. Affects all sessions on all nodes. Use 0 to turn off data collection.

Default: 1 (enabled)

SnmpTrapDestinationsList
Defines where Vertica sends traps for SNMP. See Configuring reporting for SNMP . For example:

```
=> ALTER DATABASE DEFAULT SET SnmpTrapDestinationsList = 'localhost 162 public';
```

Default: none

SnmpTrapsEnabled

Enables event trapping for SNMP. See Configuring reporting for SNMP .

Default: 0

SnmpTrapEvents

Define which events Vertica traps through SNMP. See Configuring reporting for SNMP . For example:

```
ALTER DATABASE DEFAULT SET SnmpTrapEvents = 'Low Disk Space, Recovery Failure';
```

Default: Low Disk Space, Read Only File System, Loss of K Safety, Current Fault Tolerance at Critical Level, Too Many ROS Containers, Node State Change, Recovery Failure, Stale Checkpoint, and CRC Mismatch.

SyslogEnabled

Enables event trapping for syslog. See Configuring reporting for syslog .

Default: 0

SyslogEvents

Defines events that generate a syslog entry. See Configuring reporting for syslog . For example:

```
ALTER DATABASE DEFAULT SET SyslogEvents = 'Low Disk Space, Recovery Failure';
```

Default: none

SyslogFacility

Defines which SyslogFacility Vertica uses. See Configuring reporting for syslog .

Default: user

Numeric precision parameters

The following configuration parameters let you configure numeric precision for numeric data types. For more about using these parameters, see Numeric data type overflow with SUM, SUM_FLOAT, and AVG .

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

AllowNumericOverflow

Boolean, set to one of the following:

- 1 (true): Allows silent numeric overflow. Vertica does not implicitly extend precision of numeric data types. Vertica ignores the value of NumericSumExtraPrecisionDigits.
- 0 (false): Vertica produces an overflow error, if a result exceeds the precision set by NumericSumExtraPrecisionDigits.

Default: 1 (true)

NumericSumExtraPrecisionDigits

An integer between 0 and 20, inclusive. Vertica produces an overflow error if a result exceeds the specified precision. This parameter setting only applies if AllowNumericOverflow is set to 0 (false).

Default: 6 (places beyond the DDL-specified precision)

Profiling parameters

The following table describes the profiling parameters for configuring Vertica. See Profiling database performance for more information on profiling queries.

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

GlobalEEProfiling

Enables profiling for query execution runs in all sessions on all nodes.

Default: 0

GlobalQueryProfiling

Enables query profiling for all sessions on all nodes.

Default: 0

GlobalSessionProfiling

Enables session profiling for all sessions on all nodes.

Default: 0

SaveDCEEProfileThresholdUS

Sets in microseconds the query duration threshold for saving profiling information to system tables QUERY_CONSUMPTION and EXECUTION_ENGINE_PROFILES . You can set this parameter to a maximum value of 2147483647 (2^{31} -1, or ~35.79 minutes).

Default: 60000000 (60 seconds)

Projection parameters

The following configuration parameters help you manage projections.

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

AnalyzeRowCountInterval

Specifies how often Vertica checks the number of projection rows and whether the threshold set by ARCCommitPercentage has been crossed.

For more information, see Collecting database statistics .

Default: 86400 seconds (24 hours)

ARCCommitPercentage

Sets the threshold percentage of difference between the last-recorded aggregate projection row count and current row count for a given table. When the difference exceeds this threshold, Vertica updates the catalog with the current row count.

Default: 3 (percent)

ContainersPerProjectionLimit

Specifies how many ROS containers Vertica creates per projection before ROS pushback occurs.

> **Caution**
> Increasing this parameter's value can cause serious degradation of database performance. Vertica strongly recommends that you not modify this parameter without first consulting with Customer Support professionals.

Default: 1024

MaxAutoSegColumns

Specifies the number of columns (0 –1024) to use in an auto-projection 's hash segmentation clause. Set to 0 to use all columns.

Default: 8

MaxAutoSortColumns

Specifies the number of columns (0 –1024) to use in an auto-projection 's sort expression. Set to 0 to use all columns.

Default: 8

RebalanceQueryStorageContainers

By default, prior to performing a rebalance, Vertica performs a system table query to compute the size of all projections involved in the rebalance task. This query enables Vertica to optimize the rebalance to most efficiently utilize available disk space. This query can, however, significantly increase the time required to perform the rebalance.

By disabling the system table query, you can reduce the time required to perform the rebalance. If your nodes are low on disk space, disabling the query increases the chance that a node runs out of disk space. In that situation, the rebalance fails.

Default: 1 (enable)

RewriteQueryForLargeDim

If enabled (1), Vertica rewrites a SET USING or DEFAULT USING query during a REFRESH_COLUMNS operation by reversing the inner and outer join between the target and source tables. Doing so can optimize refresh performance in cases where the source data is in a table that is larger than the target table.

Important

Enable this parameter only if the SET USING source data is in a table that is larger than the target table. If the source data is in a table smaller than the target table, then enabling RewriteQueryForLargeDim can adversely affect refresh performance.

Default: 0

SegmentAutoProjection

Determines whether auto-projections are segmented if the table definition omits a segmentation clause. You can set this parameter at

database and session scopes.

Default: 1 (create segmented auto projections)

S3 parameters

Use the following parameters to configure reading from S3 file systems and on-premises storage with S3-compatible APIs, using COPY . For more information about reading data from S3, see S3 Object Store .

For external tables using highly partitioned data in an object store, see the ObjectStoreGlobStrategy configuration parameter and Partitions on Object Stores .

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

AWSAuth

ID and secret key for authentication. For extra security, do not store credentials in the database; use ALTER SESSION...SET PARAMETER to set this value for the current session only. If you use a shared credential, you can set it in the database with ALTER DATABASE...SET PARAMETER . For example:

```
=> ALTER SESSION SET AWSAuth='ID:secret';
```

In AWS, these arguments are named AccessKeyID and SecretAccessKey.
To use admintools create_db or revive_db for Eon Mode on-premises, create a configuration file called auth_params.conf with these settings:

```
AWSAuth = key:secret
 AWSEndpoint = IP:port
```

AWSCAFile

File name of the TLS server certificate bundle to use. Setting this parameter overrides the Vertica default CA bundle path specified in the SystemCABundlePath parameter.
If set, this parameter overrides the Vertica default CA bundle path specified in the SystemCABundlePath parameter.

```
=> ALTER DATABASE DEFAULT SET AWSCAFile = '/etc/ssl/ca-bundle.pem';
```

Default: system-dependent

AWSCAPath

Path Vertica uses to look up TLS server certificates. The file name of the TLS server certificate bundle to use.
If set, this parameter overrides the Vertica default CA bundle path specified in the SystemCABundlePath parameter.

```
=> ALTER DATABASE DEFAULT SET AWSCAPath = '/etc/ssl/';
```

Default: system-dependent

AWSEnableHttps

Boolean, specifies whether to use the HTTPS protocol when connecting to S3, can be set only at the database level with ALTER DATABASE . If you choose not to use TLS, this parameter must be set to 0.
Default : 1 (enabled)

AWSEndpoint

Endpoint to use when interpreting S3 URLs, set as follows.
Important
Do not include http(s):// for AWS endpoints.

- AWS: *hostname_or_ip* : *port_number* .
- AWS with a FIPS-compliant S3 Endpoint : *S3_hostname* and enable virtual addressing:
  Important
  Do not include http(s)://

  ```
  AWSEndpoint = s3-fips.dualstack.us-east-1.amazonaws.com
  S3EnableVirtualAddressing = 1
  ```

- On-premises/Pure: IP address of the Pure Storage server. If using admintools create_db or revive_db , create configuration file auth_params.conf and include these settings:

```
awsauth = key:secret
    awsendpoint = IP:port
```

- When AWSEndpoint is not set, the default behavior is to use virtual-hosted request URLs.

Default: s3.amazonaws.com

AWSLogLevel

Log level, one of the following:

- OFF
- FATAL
- ERROR
- WARN
- INFO
- DEBUG
- TRACE

Default:ERROR

AWSRegion

AWS region containing the S3 bucket from which to read files. This parameter can only be configured with one region at a time. If you need to access buckets in multiple regions, change the parameter each time you change regions.

If you do not set the correct region, you might experience a delay before queries fail because Vertica retries several times before giving up.
Default: us-east-1

AWSSessionToken

Temporary security token generated by running the get-session-token command, which generates temporary credentials you can use to configure multi-factor authentication.

Set this parameter in a user session using ALTER SESSION . You can set this parameter at the database level, but be aware that session tokens are temporary. When the token expires, any attempt to access AWS fails.

> **Note**
> If you use session tokens at the session level, you must set all parameters at the session level, even if some of them are set at the database level. Use ALTER SESSION to set session parameters.

AWSStreamingConnectionPercentage

Controls the number of connections to the communal storage that Vertica uses for streaming reads. In a cloud environment, this setting helps prevent streaming data from communal storage using up all available file handles. It leaves some file handles available for other communal storage operations.

Due to the low latency of on-premises object stores, this option is unnecessary for an Eon Mode database that uses on-premises communal storage. In this case, disable the parameter by setting it to 0.

S3BucketConfig

Contains S3 bucket configuration information as a JSON object with the following properties. Each property other than the bucket name has a corresponding configuration parameter (shown in parentheses). If both the property in S3BucketConfig and the configuration parameter are set, the S3BucketConfig property takes precedence.

Properties:

- bucket : Name of the bucket
- region (AWSRegion): Name of the region
- protocol (AWSEnableHttps): Connection protocol, either http or https
- endpoint (AWSEndpoint): Endpoint URL or IP address
- enableVirtualAddressing (S3BucketCredentials): Whether to rewrite the S3 URL to use a virtual hosted path
- requesterPays (S3RequesterPays): Whether requester (instead of bucket owner) pays the cost of accessing data on the bucket
- serverSideEncryption (S3ServerSideEncryption): Encryption algorithm if using SSE-S3 or SSE-KMS, one of AES256 , aws:kms , or an empty string
- sseCustomerAlgorithm (S3SseCustomerAlgorithm): Encryption algorithm if using SSE-C; must be AES256
- sseCustomerKey (S3SseCustomerKey): Key if using SSE-C encryption, either 32-character plaintext or 44-character base64-encoded
- sseKmsKeyId (S3SseKmsKeyId): Key ID if using SSE-KMS encryption

The configuration properties for a given bucket might differ based on its type. For example, the following S3BucketConfig is for an AWS bucket AWSBucket and a Pure Storage bucket PureStorageBucket . AWSBucket doesn't specify an endpoint, so Vertica uses the value of AWSEndpoint, which defaults to s3.amazonaws.com :

```
ALTER DATABASE DEFAULT SET S3BucketConfig=
'[
   {
      "bucket": "AWSBucket",
      "region": "us-east-2",
      "protocol": "https",
      "requesterPays": true
   },
   {
      "bucket": "PureStorageBucket",
      "endpoint": "pure.mycorp.net:1234",
      "protocol": "http",
      "enableVirtualAddressing": false
   }
]';
```

S3BucketCredentials

Contains credentials for accessing an S3 bucket. Each property in S3BucketCredentials has an equivalent parameter (shown in parentheses). When set, S3BucketCredentials takes precedence over both AWSAuth and AWSSessionToken.

Providing credentials for more than one bucket authenticates to them simultaneously, allowing you to perform cross-endpoint joins, export from one bucket to another, etc.
Properties:

- bucket : Name of the bucket
- accessKey : Access key for the bucket (the *ID* in AWSAuth)
- secretAccessKey : Secret access key for the bucket (the *secret* in AWSAuth)
- sessionToken : Session token, only used when S3BucketCredentials is set at the session level (AWSSessionToken)

For example, the following S3BucketCredentials is for an AWS bucket AWSBucket and a Pure Storage bucket PureStorageBucket and sets all possible properties:

```
ALTER SESSION SET S3BucketCredentials='
[
   {
      "bucket": "AWSBucket",
      "accessKey": "<AK0>",
      "secretAccessKey": "<SAK0>",
      "sessionToken": "1234567890"
   },
   {
      "bucket": "PureStorageBucket",
      "accessKey": "<AK1>",
      "secretAccessKey": "<SAK1>"
   }
]';
```

This parameter is only visible to the superuser. Users can set this parameter at the session level with ALTER SESSION .

S3EnableVirtualAddressing

Boolean, specifies whether to rewrite S3 URLs to use virtual-hosted paths. For example, if you use AWS, the S3 URLs change to bucketname.s3.amazonaws.com instead of s3.amazonaws.com/bucketname . This configuration setting takes effect only when you have specified a value for AWSEndpoint .

If you set AWSEndpoint to a FIPS-compliant S3 Endpoint , you must enable S3EnableVirtualAddressing in auth_params.conf:

```
AWSEndpoint = s3-fips.dualstack.us-east-1.amazonaws.com
S3EnableVirtualAddressing = 1
```

The value of this parameter does not affect how you specify S3 paths.

Default: 0 (disabled)

> **Note**
> As of September 30, 2020, AWS requires virtual address paths for newly created buckets.

S3RequesterPays

Boolean, specifies whether requester (instead of bucket owner) pays the cost of accessing data on the bucket. When true, the bucket owner is only responsible for paying the cost of storing the data, rather than all costs associated with the bucket; must be set in order to access S3 buckets configured as Requester Pays buckets. By setting this property to true, you are accepting the charges for accessing data. If not specified, the default value is false.

S3ServerSideEncryption

String, encryption algorithm to use when reading or writing to S3. The value depends on which type of encryption at rest is configured for S3:

- AES256 : Use for SSE-S3 encryption
- aws:kms : Use for SSE-KMS encryption
- Empty string (""): No encryption

SSE-C encryption does not use this parameter. Instead, see S3SseCustomerAlgorithm.
For details on using SSE parameters, see S3 object store .

Default: "" (no encryption)

S3SseCustomerAlgorithm

String, the encryption algorithm to use when reading or writing to S3 using SSE-C encryption. The only supported values are AES256 and "" .

For SSE-S3 and SSE-KMS, instead use S3ServerSideEncryption.
Default: "" (no encryption)

S3SseCustomerKey

If using SSE-C encryption, the client key for S3 access.

S3SseKmsKeyId

If using SSE-KMS encryption, the key identifier (not the key) to pass to the Key Management Server. Vertica must have permission to use the key, which is managed through KMS.

SNS parameters

The following parameters configure Amazon Simple Notification Service (SNS) notifiers . These parameters can only be set at the database level and some, if unset, fall back to their equivalent S3 parameters .

Notifiers must be disabled and then reenabled for these parameters to take effect:

```
=> ALTER NOTIFIER sns_notifier DISABLE;
=> ALTER NOTIFIER sns_notifier ENABLE;
```

Parameter	Description	Falls back to
SNSAuth	ID and secret key for authentication, equivalent to the AccessKeyID and SecretAccessKey in AWS. For example: => ALTER DATABASE DEFAULT SET SNSAuth=' *ID* : *secret* '; **Default** : "" (empty string)	AWSAuth

SNSCAFile	File name of the TLS server certificate bundle to use. Setting this parameter overrides the Vertica default CA bundle path specified in the SystemCABundlePath parameter. If set, this parameter overrides the Vertica default CA bundle path specified in the SystemCABundlePath parameter. => ALTER DATABASE DEFAULT SET SNSCAFile = '/etc/ssl/ca-bundle.pem'; **Default** : "" (empty string)	AWSCAFile
SNSCAPath	Path Vertica uses to look up TLS server certificates. The file name of the TLS server certificate bundle to use. If set, this parameter overrides the Vertica default CA bundle path specified in the SystemCABundlePath parameter. => ALTER DATABASE DEFAULT SET SNSCAPath = '/etc/ssl/'; **Default** : "" (empty string)	AWSCAPath
SNSEnableHttps	Boolean, specifies whether to use the HTTPS protocol when connecting to S3, can be set only at the database level with ALTER DATABASE . If you choose not to use TLS, this parameter must be set to 0. **Default** : 1 (enabled)	None
SNSEndpoint	URL of the SNS API endpoint. If this parameter is set to an empty string and the region is specified (either by SNSRegion or its fallback to AWSRegion), Vertica automatically infers the appropriate endpoint. If you use FIPS, you must manually specify a FIPS-compliant endpoint . **Default** : "" (empty string)	None
SNSRegion	AWS region for the SNSEndpoint. This parameter can only be configured with one region at a time. **Default** : "" (empty string)	AWSRegion

Security parameters

Use these client authentication configuration parameters and general security parameters to configure TLS.

- To configure Vertica for client-server TLS, see Configuring client-server TLS .
- To configure JDBC and ODBC clients for TLS, see Configuring TLS for JDBC clients and Configuring TLS for ODBC Clients .
- For Kerberos-related parameters, see Kerberos parameters .

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.
Database parameters

DataSSLParams

This parameter has been deprecated. Use the data_channel TLS Configuration instead.

Enables encryption using SSL on the data channel. The value of this parameter is a comma-separated list of the following:

- An SSL certificate (chainable)
- The corresponding SSL private key
- The SSL CA (Certificate Authority) certificate.

You should set EncryptSpreadComm before setting this parameter.

In the following example, the SSL Certificate contains two certificates, where the certificate for the non-root CA verifies the certificate for the cluster. This is called an SSL Certificate Chain.

```
=> ALTER DATABASE DEFAULT SET PARAMETER DataSSLParams =
'----BEGIN CERTIFICATE-----<certificate for Cluster>-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----<certificate for non-root CA>-----END CERTIFICATE-----,
-----BEGIN RSA PRIVATE KEY-----<private key for Cluster>-----END RSA PRIVATE KEY-----,
-----BEGIN CERTIFICATE-----<certificate for public CA>-----END CERTIFICATE-----';
```

DefaultIdleSessionTimeout

Indicates a default session timeout value for all users where IDLESESSIONTIMEOUT is not set. For example:

```
=> ALTER DATABASE DEFAULT SET defaultidlesessiontimeout = '300 secs';
```

DHParams

String, a Diffie-Hellman group of at least 2048 bits in the form:

```
-----BEGIN DH PARAMETERS-----...-----END DH PARAMETERS-----
```

You can generate your own or use the pre-calculated Modular Exponential (MODP) Diffie-Hellman groups specified in RFC 3526.
Changes to this parameter do not take effect until you restart the database.

Default: RFC 3526 2048-bit MODP Group 14:

```
-----BEGIN DH PARAMETERS-----MIIBCAKCAQEA//////////
JD9qiIWjCNMTGYouA3BzRKQJOCIpnzHQCC76mOxObIIFKCHmONAT
d75UZs806QxswKwpt8l8UN0/hNW1tUcJF5IW1dmJefsb0TELppjf
tawv/XLb0Brft7jhr+1qJn6WunyQRfEsf5kkoZlHs5Fs9wgB8uKF
jvwWY2kg2HFXTmmkWP6j9JM9fg2VdI9yjrZYcYvNWIIVSu57VKQd
wlpZtZww1Tkq8mATxdGwIyhghfDKQXkYuNs474553LBgOhgObJ4O
i7Aeij7XFXfBvTFLJ3ivL9pVYFxg5lUl86pVq5RXSJhiY+gUQFXK
OWoqsqmj//////////wIBAg==-----END DH PARAMETERS-----
```

DoUserSpecificFilteringInSysTables

Boolean, specifies whether a non-superuser can view details of another user:

- 0: Users can view details of other users.
- 1: Users can only view details about themselves.

Default: 0

EnableAllRolesOnLogin

Boolean, specifies whether to automatically enable all roles granted to a user on login:

- 0: Do not automatically enable roles
- 1: Automatically enable roles. With this setting, users do not need to run SET ROLE.

Default: 0 (disable)

EnabledCipherSuites

Specifies which SSL cipher suites to use for secure client-server communication. Changes to this parameter apply only to new connections.

Default: Vertica uses the Microsoft Schannel default cipher suites. For more information, see the Schannel documentation.

EnableOAuth2JITCleanup

If enabled, users created by just-in-time OAuth provisioning are automatically dropped if the user does not log in after the number of days specified by OAuth2UserExpiredInterval.

To view provisioned users, see USERS.
Default : 0 (disable)

EncryptSpreadComm

Enables Spread encryption on the control channel, set to one of the following strings:

- vertica : Specifies that Vertica generates the Spread encryption key for the database cluster.
- aws-kms| *key-name* , where *key-name* is a named key in the iAWS Key Management Service (KMS). On database restart, Vertica fetches the named key from the KMS instead of generating its own.

If the parameter is empty, Spread communication is unencrypted. In general, you should enable this parameter before modifying other security parameters.
Enabling this parameter requires database restart.

GlobalHeirUsername

A string that specifies which user inherits objects after their owners are dropped. This setting ensures preservation of data that would otherwise be lost.

Set this parameter to one of the following string values:

- Empty string: Objects of dropped users are removed from the database.
- *username* : Reassigns objects of dropped users to *username* . If *username* does not exist, Vertica creates that user and sets GlobalHeirUsername to it.
- <auto> : Reassigns objects of dropped LDAP or just-in-time-provisioned users to the dbadmin user. The brackets (< and >) are required for this option.

For more information about usage, see Examples .

Default: <auto>

ImportExportTLSMode

When using CONNECT TO VERTICA to connect to another Vertica cluster for import or export, specifies the degree of stringency for using TLS. Possible values are:

- PREFER : Try TLS but fall back to plaintext if TLS fails.
- REQUIRE : Use TLS and fail if the server does not support TLS.
- VERIFY_CA : Require TLS (as with REQUIRE), and also validate the other server's certificate using the CA specified by the "server" TLS Configuration's CA certificates (in this case, "ca_cert" and "ica_cert"):

```
=> SELECT name, certificate, ca_certificate, mode FROM tls_configurations WHERE name = 'server';
 name |   certificate    |   ca_certificate   |   mode
--------+------------------+--------------------+-----------
 server | server_cert      | ca_cert,ica_cert   | VERIFY_CA
(1 row)
```

- VERIFY_FULL : Require TLS and validate the certificate (as with VERIFY_CA), and also validate the server certificate's hostname.
- REQUIRE_FORCE , VERIFY_CA_FORCE , and VERIFY_FULL_FORCE : Same behavior as REQUIRE , VERIFY_CA , and VERIFY_FULL , respectively, and cannot be overridden by CONNECT TO VERTICA .

Default: PREFER

InternodeTLSConfig

The TLS Configuration to use for internode encryption .

For example:

```
=> ALTER DATABASE DEFAULT SET InternodeTLSConfig = my_tls_config;
```

Default: data_channel

LDAPAuthTLSConfig

The TLS Configuration to use for TLS with LDAP authentication .

For example:

```
=> ALTER DATABASE DEFAULT SET LDAPAuthTLSConfig = my_tls_config;
```

Default: ldapauth

LDAPLinkTLSConfig

The TLS Configuration to use for TLS for the LDAP Link service .

For example:

```
=> ALTER DATABASE DEFAULT SET LDAPLinkTLSConfig = my_tls_config;
```

Default: ldaplink

OAuth2JITClient

The client/application name that contains Vertica roles in the identity provider. The Vertica roles under resource_access. *OAuth2JITClient* .roles are automatically granted to and set as default roles for users created by just-in-time provisioning. Roles that do not exist in Vertica are ignored.

For details, see Just-in-time user provisioning .

Default : vertica

OAuth2UserExpiredInterval

If EnableOAuthJITCleanup is enabled, users created by just-in-time OAuth provisioning are automatically dropped after not logging in for the number of days specified by OAuth2UserExpiredInterval. The number of days the user has not logged in is calculated relative to the LAST_LOGIN_TIME column in the USERS system table.

> **Note**
>
> The LAST_LOGIN_TIME as recorded by the USERS system table is not persistent; if the database is restarted, the LAST_LOGIN_TIME for users created by just-in-time user provisioning is set to the database start time (this appears as an empty value in LAST_LOGIN_TIME).
>
> You can view the database start time by querying the DATABASES system table:
>
> ```
> => SELECT database_name, start_time FROM databases;
> database_name |          start_time
> ---------------+-------------------------------
> VMart         | 2023-02-06 14:26:50.630054-05
> (1 row)
> ```

Default : 14

PasswordLockTimeUnit

The time units for which an account is locked by PASSWORD_LOCK_TIME after FAILED_LOGIN_ATTEMPTS , one of the following:

- 'd' : days (default)
- 'h' : hours
- 'm' : minutes
- 's' : seconds

For example, to configure the default profile to lock user accounts for 30 minutes after three unsuccessful login attempts:

```
=> ALTER DATABASE DEFAULT SET PasswordLockTimeUnit = 'm'
=> ALTER PROFILE DEFAULT LIMIT PASSWORD_LOCK_TIME 30;
```

RequireFIPS

Boolean, specifies whether the FIPS mode is enabled:

- 0 (disable)
- 1: (enable)

On startup, Vertica automatically sets this parameter from the contents of the file crypto.fips_enabled . You cannot modify this parameter. For details, see FIPS compliance for the Vertica server .

Default: 0

SecurityAlgorithm

Sets the algorithm for the function that hash authentication uses, one of the following:

- SHA512
- MD5

For example:

```
=> ALTER DATABASE DEFAULT SET SecurityAlgorithm = 'SHA512';
```

Default: SHA512

ServerTLSConfig

The TLS Configuration to use for client-server TLS .

For example:

```
=> ALTER DATABASE DEFAULT SET ServerTLSConfig = my_tls_config;
```

Default: server

SystemCABundlePath

The absolute path to a certificate bundle of trusted CAs. This CA bundle is used when establishing TLS connections to external services such as

AWS or Azure through their respective SDKs and libcurl. The CA bundle file must be in the same location on all nodes.

If this parameter is empty, Vertica searches the "standard" paths for the CA bundles, which differs between distributions:

- Red Hat-based: /etc/pki/tls/certs/ca-bundle.crt
- Debian-based: /etc/ssl/certs/ca-certificates.crt
- SUSE: /var/lib/ca-certificates/ca-bundle.pem

Example :

```
=> ALTER DATABASE DEFAULT SET SystemCABundlePath = 'path/to/ca_bundle.pem';
```

Default: Empty

TLS parameters

To set your Vertica database's TLSMode, private key, server certificate, and CA certificate(s), see TLS configurations . In versions prior to 11.0.0, these parameters were known as EnableSSL, SSLPrivateKey, SSLCertificate, and SSLCA, respectively.

Examples

Set the database parameter GlobalHeirUsername :

```
=> \du
     List of users
 User name | Is Superuser
-----------+--------------
 Joe       | f
 SuzyQ     | f
 dbadmin   | t
(3 rows)

=> ALTER DATABASE DEFAULT SET PARAMETER GlobalHeirUsername='SuzyQ';
ALTER DATABASE
=>  \c - Joe
You are now connected as user "Joe".
=> CREATE TABLE t1 (a int);
CREATE TABLE

=> \c
You are now connected as user "dbadmin".
=> \dt t1
            List of tables
 Schema | Name | Kind  | Owner | Comment
--------+------+-------+-------+---------
 public | t1   | table | Joe   |
(1 row)

=> DROP USER Joe;
NOTICE 4927:  The Table t1 depends on User Joe
ROLLBACK 3128:  DROP failed due to dependencies
DETAIL:  Cannot drop User Joe because other objects depend on it
HINT:  Use DROP ... CASCADE to drop the dependent objects too
=> DROP USER Joe CASCADE;
DROP USER
=> \dt t1
            List of tables
 Schema | Name | Kind  | Owner | Comment
--------+------+-------+-------+---------
 public | t1   | table | SuzyQ |
(1 row)
```

Stored procedure parameters

The following parameters configure the behavior of stored procedures and triggers.

PLvSQLCoerceNull

Boolean, whether to allow NULL-to-false type coercion to improve compatibility with PLpgSQL. For details, see PL/pgSQL to PL/vSQL migration guide.

Default: 0

EnableStoredProcedureScheduler

Boolean, whether to enable the scheduler. For details, see Scheduled execution.

Default: 1

Text search parameters

You can configure Vertica for text search using the following parameter.

TextIndexMaxTokenLength

Controls the maximum size of a token in a text index.

For example:

```
ALTER DATABASE database_name SET PARAMETER TextIndexMaxTokenLength=760;
```

If the parameter is set to a value greater than 65000 characters, then the tokenizer truncates the token at 65000 characters.

> **Caution**
> Avoid setting this parameter near its maximum value, 65000. Doing so can result in a significant decrease in performance. For optimal performance, set this parameter to the maximum token value of your tokenizer.

Default: 128 (characters)

Tuple mover parameters

These parameters control how the Tuple Mover operates.

Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

ActivePartitionCount

Sets the number of active partitions. The active partitions are those most recently created. For example:

```
=> ALTER DATABASE DEFAULT SET ActivePartitionCount = 2;
```

For information about how the Tuple Mover treats active and inactive partitions during a mergeout operation, see Partition mergeout.

Default: 1

CancelTMTimeout

When partition, copy table, and rebalance operations encounter a conflict with an internal Tuple Mover job, those operations attempt to cancel the conflicting Tuple Mover job. This parameter specifies the amount of time, in seconds, that the blocked operation waits for the Tuple Mover cancellation to take effect. If the operation is unable to cancel the Tuple Mover job within limit specified by this parameter, the operation displays an error and rolls back.

Default: 300

EnableTMOnRecoveringNode

Boolean, specifies whether Tuple Mover performs mergeout activities on nodes with a node state of RECOVERING. Enabling Tuple Mover reduces the number of ROS containers generated during recovery. Having fewer than 1024 ROS containers per projection allows Vertica to maintain optimal recovery performance.

Default: 1 (enabled)

MaxMrgOutROSSizeMB

Specifies in MB the maximum size of ROS containers that are candidates for mergeout operations. The Tuple Mover avoids merging ROS containers that are larger than this setting.

> **Note**
> After a rebalance operation, Tuple Mover groups ROS containers in batches that are smaller than MaxMrgOutROSSizeMB. ROS containers

that are larger than MaxMrgOutROSSizeMB are merged individually

Default: -1 (no maximum limit)

MergeOutInterval

Specifies in seconds how frequently the Tuple Mover checks the mergeout request queue for pending requests:

1. If the queue contains mergeout requests, the Tuple Mover does nothing and goes back to sleep.
2. If the queue is empty, the Tuple Mover:
   - Processes pending storage location move requests.
   - Checks for new unqueued purge requests and adds them to the queue.

   It then goes back to sleep.

Default: 600

PurgeMergeoutPercent

Specifies as a percentage the threshold of deleted records in a ROS container that invokes an automatic mergeout operation, to purge those records. Vertica only counts the number of 'aged-out' delete vectors—that is, delete vectors that are as 'old' or older than the ancient history mark (AHM) epoch.

This threshold applies to all ROS containers for non-partitioned tables. It also applies to ROS containers of all inactive partitions. In both cases, aged-out delete vectors are permanently purged from the ROS container.

> **Note**
> This configuration parameter only applies to automatic mergeout operations. It does not apply to manual mergeout operations that are invoked by calling meta-functions DO_TM_TASK('mergeout') and PURGE .

Default: 20 (percent)

File systems and object stores

Vertica supports access to several file systems and object stores in addition to the Linux file system. The reference pages in this section provide information on URI syntax, configuration parameters, and authentication.

Vertica accesses the file systems in this section in one of two ways:

- If user-provided credentials are present, Vertica uses them to access the storage. Note that on HDFS, user credentials are always present because Vertica accesses HDFS using the Vertica user identity.
- If user-provided credentials are not present, or if the UseServerIdentityOverUserIdentity configuration parameter is set, Vertica checks for a configured USER storage location. When access is managed through USER storage locations, Vertica uses the server credential to access the file system. For more information about USER storage locations, see CREATE LOCATION .

Not all file systems are supported in all contexts. See the documentation of specific features for the file systems those features support.

In this section

- Azure Blob Storage object store
- Google Cloud Storage (GCS) object store
- HDFS file system
- S3 object store

Azure Blob Storage object store

Azure has several interfaces for accessing data. Vertica reads and always writes Block Blobs in Azure Storage. Vertica can read external data created using ADLS Gen2, and data that Vertica exports can be read using ADLS Gen2.

URI format

One of the following:

- `azb://` *account* `/` *container* `/` *path*
- `azb://[` *account* `@]` *host* `[:` *port* `]/` *container* `/` *path*

In the first version, a URI like 'azb://myaccount/mycontainer/path' treats the first token after the '//' as the account name. In the second version, you can specify account and must specify host explicitly.

The following rules apply to the second form:

- If *account* is not specified, the first label of the host is used. For example, if the URI is 'azb://myaccount.blob.core.windows.net/mycontainer/my/object', then 'myaccount' is used for *account* .
- If *account* is not specified and *host* has a single label and no port, the endpoint is *host* .blob.core.windows.net . Otherwise, the endpoint is the host and port specified in the URI.

The protocol (HTTP or HTTPS) is specified in the AzureStorageEndpointConfig configuration parameter.

Authentication

If you are using Azure managed identities, no further configuration in Vertica is needed. If your Azure storage uses multiple managed identities, you must tag the one to be used. Vertica looks for an Azure tag with a key of VerticaManagedIdentityClientId, the value of which must be the client_id attribute of the managed identity to be used. If you update the Azure tag, call AZURE_TOKEN_CACHE_CLEAR .

If you are not using managed identities, use the AzureStorageCredentials configuration parameter to provide credentials to Azure. If loading data, you can set the parameter at the session level. If using Eon Mode communal storage on Azure, you must set this configuration parameter at the database level.

In Azure you must also grant access to the containers for the identities used from Vertica.

Configuration parameters

The following database configuration parameters apply to the Azure blob file system. You can set parameters at different levels with the appropriate ALTER statement, such as ALTER SESSION...SET PARAMETER . Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

For external tables using highly partitioned data in an object store, see the ObjectStoreGlobStrategy configuration parameter and Partitions on Object Stores .

AzureStorageCredentials

Collection of JSON objects, each of which specifies connection credentials for one endpoint. This parameter takes precedence over Azure managed identities.

The collection must contain at least one object and may contain more. Each object must specify at least one of accountName or blobEndpoint , and at least one of accountKey or sharedAccessSignature .

- accountName : If not specified, uses the label of blobEndpoint .
- blobEndpoint : Host name with optional port (host:port). If not specified, uses *account* .blob.core.windows.net .
- accountKey : Access key for the account or endpoint.
- sharedAccessSignature : Access token for finer-grained access control, if being used by the Azure endpoint.

AzureStorageEndpointConfig

Collection of JSON objects, each of which specifies configuration elements for one endpoint. Each object must specify at least one of accountName or blobEndpoint .

- accountName : If not specified, uses the label of blobEndpoint .
- blobEndpoint : Host name with optional port (host:port). If not specified, uses *account* .blob.core.windows.net .
- protocol : HTTPS (default) or HTTP.
- isMultiAccountEndpoint : true if the endpoint supports multiple accounts, false otherwise (default is false). To use multiple-account access, you must include the account name in the URI. If a URI path contains an account, this value is assumed to be true unless explicitly set to false.

Examples

The following examples use these values for the configuration parameters. AzureStorageCredentials contains sensitive information and is set at the session level in this example.

```
=> ALTER SESSION SET AzureStorageCredentials =
   '[{"accountName": "myaccount", "accountKey": "REAL_KEY"},
     {"accountName": "myaccount", "blobEndpoint": "localhost:8080", "accountKey": "TEST_KEY"}]';

=> ALTER DATABASE default SET AzureStorageEndpointConfig =
   '[{"accountName": "myaccount", "blobEndpoint": "localhost:8080", "protocol": "http"}]';
```

The following example creates an external table using data from Azure. The URI specifies an account name of "myaccount".

```
=> CREATE EXTERNAL TABLE users (id INT, name VARCHAR(20))
   AS COPY FROM 'azb://myaccount/mycontainer/my/object/*';
```

Vertica uses AzureStorageEndpointConfig and the account name to produce the following location for the files:

```
https://myaccount.blob.core.windows.net/mycontainer/my/object/*
```

Data is accessed using the REAL_KEY credential.

If the URI in the COPY statement is instead azb://myaccount.blob.core.windows.net/mycontainer/my/object , then the resulting location is https://myaccount.blob.core.windows.net/mycontainer/my/object , again using the REAL_KEY credential.

However, if the URI in the COPY statement is azb://myaccount@localhost:8080/mycontainer/my/object , then the host and port specify a different endpoint: http://localhost:8080/myaccount/mycontainer/my/object . This endpoint is configured to use a different credential, TEST_KEY.

Google Cloud Storage (GCS) object store

File system using the Google Cloud Storage platform.

URI format

```
gs://bucket/path
```

Authentication

To access data in Google Cloud Storage (GCS) you must first do the following tasks:

- Create a default project, obtain a developer key, and enable S3 interoperability mode as described in the GCS documentation .
- Set the GCSAuth configuration parameter as in the following example.

```
=> ALTER SESSION SET GCSAuth='id:secret';
```

Configuration parameters

The following database configuration parameters apply to the GCS file system. You can set parameters at different levels with the appropriate ALTER statement, such as ALTER SESSION...SET PARAMETER . Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter. For information about all parameters related to GCS, see Google Cloud Storage parameters .

For external tables using highly partitioned data in an object store, see the ObjectStoreGlobStrategy configuration parameter and Partitions on Object Stores .

GCSAuth

An ID and secret key to authenticate to GCS. For extra security, do not store credentials in the database; instead, use ALTER SESSION...SET PARAMETER to set this value for the current session only.

GCSEnableHttps

Boolean, whether to use the HTTPS protocol when connecting to GCS, can be set only at the database level with ALTER DATABASE...SET PARAMETER .

Default: 1 (enabled)

GCSEndpoint

The connection endpoint address.

Default: storage.googleapis.com

Examples

The following example loads data from GCS:

```
=> ALTER SESSION SET GCSAuth='my_id:my_secret_key';

=> COPY t FROM 'gs://DataLake/clicks.parquet' PARQUET;
```

HDFS file system

HDFS is the Hadoop Distributed File System. You can use the webhdfs and swebhdfs schemes to access data through the WebHDFS service. Vertica also supports the hdfs scheme, which by default uses WebHDFS. To have hdfs URIs use the deprecated LibHDFS++ package, set the HDFSUseWebHDFS configuration parameter to 0 (disabled).

If you specify a webhdfs URI but the Hadoop HTTP policy (dfs.http.policy) is set to HTTPS_ONLY, Vertica automatically uses swebhdfs instead.

If you use LibHDFS++, the WebHDFS service must still be available because Vertica falls back to WebHDFS for operations not supported by LibHDFS++.

> **Deprecated**
> Support for LibHDFS++ is deprecated. In the future, HDFSUseWebHDFS will be enabled in all cases and hdfs URIs will be equivalent to webhdfs URIs.

URI format

URIs in the webhdfs , swebhdfs , and hdfs schemes all have two formats, depending on whether you specify a name service or the host and port of a name node:

- [[s]web]hdfs://[*nameservice*]/ *path*
- [[s]web]hdfs:// *namenode-host:port* / *path*

Characters may be URL-encoded (%NN where NN is a two-digit hexadecimal number) but are not required to be, except that the '%' character must be encoded.

To use the default name service specified in the HDFS configuration files, omit *nameservice* . Use this shorthand only for reading external data, not for creating a storage location.

Always specify a name service or host explicitly when using Vertica with more than one HDFS cluster. The name service or host name must be globally unique. Using [web]hdfs:/// could produce unexpected results because Vertica uses the first value of fs.defaultFS that it finds.

Authentication

Vertica can use Kerberos authentication with Cloudera or Hortonworks HDFS clusters. See Accessing kerberized HDFS data .

For loading and exporting data, Vertica can access HDFS clusters protected by mTLS through the swebhdfs scheme. You must create a certificate and key and set the WebhdfsClientCertConf configuration parameter.

You can use CREATE KEY and CREATE CERTIFICATE to create temporary, session-scoped values if you specify the TEMPORARY keyword. Temporary keys and certificates are stored in memory, not on disk.

The WebhdfsClientCertConf configuration parameter holds client credentials for one or more HDFS clusters. The value is a JSON string listing name services or authorities and their corresponding keys. You can set the configuration parameter at the session or database level. Setting the parameter at the database level has the following additional requirements:

- The UseServerIdentityOverUserIdentity configuration parameter must be set to 1 (true).
- The user must be dbadmin or must have access to the user storage location on HDFS.

The following example shows how to use mTLS. The key and certificate values themselves are not shown, just the beginning and end markers:

```
=> CREATE TEMPORARY KEY client_key TYPE 'RSA'
   AS '-----BEGIN PRIVATE KEY-----...-----END PRIVATE KEY-----';

-> CREATE TEMPORARY CERTIFICATE client_cert
   AS '-----BEGIN CERTIFICATE-----...-----END CERTIFICATE-----' key client_key;

=> ALTER SESSION SET WebhdfsClientCertConf =
   '[{"authority": "my.hdfs.namenode1:50088", "certName": "client_cert"}]';

=> COPY people FROM 'swebhdfs://my.hdfs.namenode1:50088/path/to/file/1.txt';
Rows Loaded
-------------
1
(1 row)
```

To configure access to more than one HDFS cluster, define the keys and certificates and then include one object per cluster in the value of WebhdfsClientCertConf:

```
=> ALTER SESSION SET WebhdfsClientCertConf =
    '[{"authority" : "my.authority.com:50070", "certName" : "myCert"},
      {"nameservice" : "prod", "certName" : "prodCert"}]';
```

Configuration parameters

The following database configuration parameters apply to the HDFS file system. You can set parameters at different levels with the appropriate ALTER statement, such as ALTER SESSION...SET PARAMETER . Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter. For information about all parameters related to Hadoop, see Hadoop parameters .

EnableHDFSBlockInfoCache
: Boolean, whether to distribute block location metadata collected during planning on the initiator to all database nodes for execution, reducing name node contention. Disabled by default.

HadoopConfDir
: Directory path containing the XML configuration files copied from Hadoop. The same path must be valid on every Vertica node. The files are accessed by the Linux user under which the Vertica server process runs.

HadoopImpersonationConfig
: Session parameter specifying the delegation token or Hadoop user for HDFS access. See HadoopImpersonationConfig format for information about the value of this parameter and Proxy users and delegation tokens for more general context.

HDFSUseWebHDFS
: Boolean. If true (the default), URIs in the hdfs scheme are treated as if they were in the webhdfs scheme. If false, Vertica uses LibHDFS++ where possible, though some operations can still use WebHDFS if not supported by LibHDFS++.

WebhdfsClientCertConf
: mTLS configurations for accessing one or more WebHDFS servers, a JSON string. Each object must specify either a nameservice or authority field and a certName field. See Authentication .

Configuration files

The path specified in HadoopConfDir must include a directory containing the files listed in the following table. Vertica reads these files at database start time. If you do not set a value, Vertica looks for the files in /etc/hadoop/conf.

If a property is not defined, Vertica uses the defaults shown in the table. If no default is specified for a property, the configuration files must specify a value.

File	Properties	Default
core-site.xml	fs.defaultFS	none
	(for doAs users:) hadoop.proxyuser. *username* .users	none
	(for doAs users:) hadoop.proxyuser. *username* .hosts	none
hdfs-site.xml	dfs.client.failover.max.attempts	15
	dfs.client.failover.sleep.base.millis	500
	dfs.client.failover.sleep.max.millis	15000
	(For HA NN:) dfs.nameservices	none
	(WebHDFS:) dfs.namenode.http-address or dfs.namenode.https-address	none
	(WebHDFS:) dfs.datanode.http.address or dfs.datanode.https.address	none
	(WebHDFS:) dfs.http.policy	HTTP_ONLY

If using High Availability (HA) Name Nodes, the individual name nodes must also be defined in hdfs-site.xml.

> **Note**
> If you are using Eon Mode with communal storage on HDFS, then if you set dfs.encrypt.data.transfer you must use the swebhdfs scheme for communal storage.

To verify that Vertica can find configuration files in HadoopConfDir, use the VERIFY_HADOOP_CONF_DIR function.

To test access through the hdfs scheme, use the HDFS_CLUSTER_CONFIG_CHECK function.

For more information about testing your configuration, see Verifying HDFS configuration .

To reread the configuration files, use the CLEAR_HDFS_CACHES function.

Name nodes and name services

You can access HDFS data using the default name node by not specifying a name node or name service:

```
=> COPY users FROM 'webhdfs:///data/users.csv';
```

Vertica uses the fs.defaultFS Hadoop configuration parameter to find the name node. (It then uses that name node to locate the data.) You can instead specify a host and port explicitly using the following format:

```
webhdfs://nn-host:nn-port/
```

The specified host is the name node, not an individual data node. If you are using High Availability (HA) Name Nodes you should not use an explicit host because high availability is provided through name services instead.

If the HDFS cluster uses High Availability Name Nodes or defines name services, use the name service instead of the host and port, in the format webhdfs://nameservice/ . The name service you specify must be defined in hdfs-site.xml .

The following example shows how you can use a name service, hadoopNS:

```
=> CREATE EXTERNAL TABLE users (id INT, name VARCHAR(20))
   AS COPY FROM 'webhdfs://hadoopNS/data/users.csv';
```

If you are using Vertica to access data from more than one HDFS cluster, always use explicit name services or hosts in the URL. Using the /// shorthand could produce unexpected results because Vertica uses the first value of fs.defaultFS that it finds. To access multiple HDFS clusters, you must use host and service names that are globally unique. See Configuring HDFS access for more information.

S3 object store

File systems using the S3 protocol, including AWS, Pure Storage, and MinIO.

URI format

```
s3://bucket/path
```

For AWS, specify the region using the AWSRegion configuration parameter, not the URI. If the region is incorrect, you might experience a delay before the load fails because Vertica retries several times before giving up. The default region is us-east-1 .

Authentication

For AWS:

- To access S3 you must create an IAM role and grant that role permission to access your S3 resources.
- By default, bucket access is restricted to the communal storage bucket. Use an AWS access key to load data from non-communal storage buckets.
- Either set the AWSAuth configuration parameter to provide credentials or create a USER storage location for the S3 path (see CREATE LOCATION) and grant users access.
- You can use AWS STS temporary session tokens to load data. Because they are session tokens, do not use them for access to storage locations.
- You can configure S3 buckets individually with the per-bucket parameters S3BucketConfig and S3BucketCredentials. For details, see Per-bucket S3 configurations .

Configuration parameters

The following database configuration parameters apply to the S3 file system. You can set parameters at different levels with the appropriate ALTER statement, such as ALTER SESSION...SET PARAMETER . Query the CONFIGURATION_PARAMETERS system table to determine what levels (node, session, user, database) are valid for a given parameter.

You can configure individual buckets using the S3BucketConfig and S3BucketCredentials parameters instead of the global parameters.

For details about all parameters related to S3, see S3 parameters .

For external tables using highly partitioned data in an object store, see the ObjectStoreGlobStrategy configuration parameter and Partitions on Object Stores .

AWSAuth

An ID and secret key for authentication. AWS calls these AccessKeyID and SecretAccessKey. For extra security, do not store credentials in the database; use ALTER SESSION...SET PARAMETER to set this value for the current session only.

AWSCAFile

The file name of the TLS server certificate bundle to use. You must set a value when installing a CA certificate on a SUSE Linux Enterprise Server.

AWSCAPath

The path Vertica uses to look up TLS server certificates. You must set a value when installing a CA certificate on a SUSE Linux Enterprise Server.

AWSEnableHttps

Boolean, whether to use the HTTPS protocol when connecting to S3. Can be set only at the database level. You can set the prototol for individual buckets using S3BucketConfig.

Default: 1 (enabled)

AWSEndpoint

String, the endpoint host for all S3 URLs, set as follows:

- AWS: *hostname_or_IP* : *port* . Do not include the scheme (http(s)).
- AWS with a FIPS-compliant S3 Endpoint: Hostname of a FIPS-compliant S3 endpoint . You must also enable S3EnableVirtualAddressing.
- On-premises/Pure: IP address of the Pure Storage server.

If not set, Vertica uses virtual-hosted request URLs.
Default: 's3.amazonaws.com'

AWSLogLevel

The log level, one of: OFF, FATAL, ERROR, WARN, INFO, DEBUG, or TRACE.

Default: ERROR

AWSRegion

The AWS region containing the S3 bucket from which to read files. This parameter can only be configured with one region at a time. Failure to set the correct region can lead to a delay before queries fail.

Default: 'us-east-1'

AWSSessionToken

A temporary security token generated by running the get-session-token command, used to configure multi-factor authentication.

> **Note**
> If you use session tokens at the session level, you must set all parameters at the session level, even if some of them are set at the database level.

AWSStreamingConnectionPercentage

In Eon Mode, the number of connections to the communal storage to use for streaming reads. In a cloud environment, this setting helps prevent streaming data from using up all available file handles. This setting is unnecessary when using on-premises object stores because of their lower latency.

S3BucketConfig

A JSON object specifying per-bucket configuration. Each property other than the bucket name has a corresponding configuration parameter. If both the database-level parameter and its equivalent in S3BucketConfig are set, the value in S3BucketConfig takes precedence.

Properties:

- bucket : Bucket name
- region (AWSRegion)
- protocol : Scheme, one of http or https ; overrides AWSEnableHttps
- endpoint (AWSEndpoint)
- enableVirtualAddressing : Boolean, whether to rewrite the S3 URL to use a virtual hosted path (S3BucketCredentials)
- requesterPays (S3RequesterPays)
- serverSideEncryption (S3ServerSideEncryption)
- sseCustomerAlgorithm (S3SseCustomerAlgorithm)
- sseCustomerKey (S3SseCustomerKey)
- sseKmsKeyId (S3SseKmsKeyId)

S3BucketCredentials

A JSON object specifying per-bucket credentials. Each property other than the bucket name has a corresponding configuration parameter. If both the database-level parameter and its equivalent in S3BucketCredentials are set, the value in S3BucketCredentials takes precedence.

Properties:

- bucket : Bucket name

- accessKey : Access key for the bucket (the *ID* in AWSAuth)
- secretAccessKey : Secret access key for the bucket (the *secret* in AWSAuth)
- sessionToken : Session token, only used when S3BucketCredentials is set at the session level (AWSSessionToken)

This parameter is only visible to superusers. Users can set this parameter at the session level with ALTER SESSION .

S3EnableVirtualAddressing
Boolean, whether to rewrite S3 URLs to use virtual-hosted paths (disabled by default). This configuration setting takes effect only when you have specified a value for AWSEndpoint.

If you set AWSEndpoint to a FIPS-compliant S3 endpoint , you must enable S3EnableVirtualAddressing.
The value of this parameter does not affect how you specify S3 paths.

S3RequesterPays
Boolean, whether requester (instead of bucket owner) pays the cost of accessing data on the bucket.

S3ServerSideEncryption
String, encryption algorithm to use when reading or writing to S3. Supported values are AES256 (for SSE-S3), aws:kms (for SSE-KMS), and an empty string (for no encryption). See Server-Side Encryption .

Default: "" (no encryption)

S3SseCustomerAlgorithm
String, the encryption algorithm to use when reading or writing to S3 using SSE-C encryption. The only supported values are AES256 and "" . For SSE-S3 and SSE-KMS, instead use S3ServerSideEncryption.

Default: "" (no encryption)

S3SseCustomerKey
If using SSE-C encryption, the client key for S3 access.

S3SseKmsKeyId
If using SSE-KMS encryption, the key identifier (not the key) to pass to the Key Management Service. Vertica must have permission to use the key, which is managed through KMS.

Server-side encryption

By default, Vertica reads and writes S3 data that is not encrypted. If the S3 bucket uses server-side encryption (SSE), you can configure Vertica to access it. S3 supports three types of server-side encryption: SSE-S3, SSE-KMS, and SSE-C.

Vertica must also have read or write permissions (depending on the operation) on the bucket.

SSE-S3

With SSE-S3, the S3 service manages encryption keys. Reads do not require additional configuration. To write to S3, the client (Vertica, in this case) must specify only the encryption algorithm.

If the S3 bucket is configured with the default encryption settings, Vertica can read and write data to them with no further changes. If the bucket does not use the default encryption settings, set the S3ServerSideEncryption configuration parameter or the serverSideEncryption field in S3BucketConfig to AES256 .

SSE-KMS

With SSE-KMS, encryption keys are managed by the Key Management Service (KMS). The client must supply a KMS key identifier (not the actual key) when writing data. For all operations, the client must have permission to use the KMS key. These permissions are managed in KMS, not in Vertica.

To use SSE-KMS:

- Set the S3ServerSideEncryption configuration parameter or the serverSideEncryption field in S3BucketConfig to aws:kms .
- Set the S3SseKmsKeyId configuration parameter or the sseKmsKeyId field in S3BucketConfig to the key ID.

SSE-C

With SSE-C, the client manages encryption keys and provides them to S3 for each operation.

To use SSE-C:

- Set the S3SseCustomerAlgorithm configuration parameter or the sseCustomerAlgorithm field in S3BucketConfig to AES256 .
- Set the S3SseCustomerKey configuration parameter or the sseCustomerKey field in S3BucketConfig to the access key. The value can be either a 32-character plaintext key or a 44-character base64-encoded key.

Examples

The following example sets a database-wide AWS region and credentials:

```
=> ALTER DATABASE DEFAULT SET AWSRegion='us-west-1';
=> ALTER DATABASE DEFAULT SET AWSAuth = 'myaccesskeyid123456:mysecretaccesskey1234567890123456789012345678901234';
```

The following example loads data from S3. You can use a glob if all files in the glob can be loaded together. In the following example, AWS_DataLake contains only ORC files.

```
=> COPY t FROM 's3://datalake/*' ORC;
```

You can specify a list of comma-separated S3 buckets as in the following example. All buckets must be in the same region. To load from more than one region, use separate COPY statements and change the value of AWSRegion between calls.

```
=> COPY t FROM 's3://AWS_Data_1/sales.parquet', 's3://AWS_Data_2/sales.parquet' PARQUET;
```

The following example creates a user storage location and a role, so that users without their own S3 credentials can read data from S3 using the server credential.

```
   --- set database-level credential (once):
=> ALTER DATABASE DEFAULT SET AWSAuth = 'myaccesskeyid123456:mysecretaccesskey1234567890123456789012345678901234';

=> CREATE LOCATION 's3://datalake' SHARED USAGE 'USER' LABEL 's3user';

=> CREATE ROLE ExtUsers;
   --- Assign users to this role using GRANT (Role).

=> GRANT READ ON LOCATION 's3://datalake' TO ExtUsers;
```

The configuration properties for a given bucket may differ based on its type. The following S3BucketConfig setting is for an AWS bucket (AWSBucket) and a Pure Storage bucket (PureStorageBucket). AWSBucket doesn't specify an endpoint, so Vertica uses the AWSEndpoint configuration parameter, which defaults to s3.amazonaws.com :

```
=> ALTER DATABASE DEFAULT SET S3BucketConfig=
'[
   {
      "bucket": "AWSBucket",
      "region": "us-east-2",
      "protocol": "https",
      "requesterPays": true,
      "serverSideEncryption": "aes256"
   },
   {
      "bucket": "PureStorageBucket",
      "endpoint": "pure.mycorp.net:1234",
      "protocol": "http",
      "enableVirtualAddressing": false
   }
]';
```

The following example sets S3BucketCredentials for these two buckets:

```
=> ALTER SESSION SET S3BucketCredentials='
[
   {
      "bucket": "AWSBucket",
      "accessKey": "<AK0>",
      "secretAccessKey": "<SAK0>",
      "sessionToken": "1234567890"
   },
   {
      "bucket": "PureStorageBucket",
      "accessKey": "<AK1>",
      "secretAccessKey": "<SAK1>"
   }
]';
```

The following example sets an STS temporary session token. Vertica uses the session token to access S3 with the specified credentials and bypasses checking for a USER storage location.

```
$ aws sts get-session-token
 {
   "Credentials": {
      "AccessKeyId": "ASIAJZQNDVS727EHDHOQ",
      "SecretAccessKey": "F+xnpkHbst6UPorlLGj/ilJhO5J2n3Yo7Mp4vYvd",
      "SessionToken":
"FQoDYXdzEKv//////////wEaDMWKxakEkCyuDH0UjyKsAe6/3REgW5VbWtpuYyVvSnEK1jzGPHi/jPOPNT7Kd+ftSnD3qdaQ7j28SUW9YYbD50lcXikz/HPlusPu
X9sAJJb7w5oiwdg+ZasIS/+ejFgCzLeNE3kDAzLxKKsunvwuo7EhTTyqmlLkLtIWu9zFykzrR+3Tl76X7EUMOaoL31HOYsVEL5d9I9KInF0gE12ZB1yN16MsQV>
pSCavOFHQsj/05zbxOQ4o0erY1gU=",
      "Expiration": "2018-07-18T05:56:33Z"
   }
 }

$ vsql
=> ALTER SESSION SET AWSAuth = 'ASIAJZQNDVS727EHDHOQ:F+xnpkHbst6UPorlLGj/ilJhO5J2n3Yo7Mp4vYvd';
=> ALTER SESSION SET AWSSessionToken =
'FQoDYXdzEKv//////////wEaDMWKxakEkCyuDH0UjyKsAe6/3REgW5VbWtpuYyVvSnEK1jzGPHi/jPOPNT7Kd+ftSnD3qdaQ7j28SUW9YYbD50lcXikz/HPlusPu
X9sAJJb7w5oiwdg+ZasIS/+ejFgCzLeNE3kDAzLxKKsunvwuo7EhTTyqmlLkLtIWu9zFykzrR+3Tl76X7EUMOaoL31HOYsVEL5d9I9KInF0gE12ZB1yN16MsQV>
pSCavOFHQsj/05zbxOQ4o0erY1gU=';
```

See also

Per-Bucket S3 Configurations

In this section

- Per-bucket S3 configurations

Per-bucket S3 configurations

You can manage configurations and credentials for individual buckets with the S3BucketConfig and S3BucketCredentials configuration parameters. These parameters each take a JSON object, whose respective properties behave like the related S3 configuration parameters.

For example, you can create a different configuration for each of your S3 buckets by setting S3BucketConfig at the database level with ALTER DATABASE . The following S3BucketConfig specifies several common bucket properties:

```
=> ALTER DATABASE DEFAULT SET S3BucketConfig='
[
   {
      "bucket": "exampleAWS",
      "region": "us-east-2",
      "protocol": "https",
      "requesterPays": true
   },
   {
      "bucket": "examplePureStorage",
      "endpoint": "pure.mycorp.net:1234",
      "protocol": "http",
      "enableVirtualAddressing": false
   }
]';
```

Users can then access a bucket by setting S3BucketCredentials at the session level with ALTER SESSION . The following S3BucketCredentials specifies all properties and authenticates to both exampleAWS and examplePureStorage simultaneously:

```
=> ALTER SESSION SET S3BucketCredentials='
[
   {
      "bucket": "exampleAWS",
      "accessKey": "<AK0>",
      "secretAccessKey": "<SAK0>",
      "sessionToken": "1234567890"
   },
   {
      "bucket": "examplePureStorage",
      "accessKey": "<AK1>",
      "secretAccessKey": "<SAK1>",
   }
]';
```

Recommended usage

The recommended usage is as follows:

- Define in your S3 storage system one set of credentials per principal, per storage system.
- It is often most convenient to set S3BucketConfig once at the database level and have users authenticate by setting S3BucketCredentials at the session level.
- To access buckets outside those configured at the database level, set both S3BucketConfig and S3BucketCredentials at the session level.

If you cannot define credentials for your S3 storage, you can set S3BucketCredentials or AWSAuth at the database level with ALTER DATABASE , but this comes with certain drawbacks:

- Storing credentials statically in another location (in this case, in the Vertica catalog) always incurs additional risk.
- This increases overhead for the dbadmin, who needs to create user storage locations and grant access to each user or role.
- Users share one set of credentials, increasing the potential impact if the credentials are compromised.

> **Note**
> If you set AWSEndpoint to a non-Amazon S3 storage system like Pure Storage or MinIO and you want to configure S3BucketConfig for real Amazon S3, the following requirements apply:
>
> - If your real Amazon S3 region is not us-east-1 (the default), you must specify the region .
> - Set endpoint to an empty string ("").

```
=> ALTER DATABASE DEFAULT SET S3BucketConfig='
[
   {
      "bucket": "additionalAWSBucket",
      "region": "us-east-2",
      "endpoint": ""
   }
]';
```

Precedence of per-bucket and standard parameters

Vertica uses the following rules to determine the effective set of properties for an S3 connection:

- If set, S3BucketCredentials takes priority over its standard parameters. S3BucketCredentials is checked first at the session level and then at the database level.
- The level/source of the S3 credential parameters determines the source of the S3 configuration parameters:
  - If credentials come from the session level, then the configuration can come from either the session or database level (with the session level taking priority).
  - If your credentials come from the database level, then the configuration can only come from the database level.
- If S3BucketConfig is set, it takes priority over its standard parameters. If an S3BucketConfig property isn't specified, Vertica falls back to the missing property's equivalent parameter. For example, if S3BucketConfig specifies every property except protocol , Vertica falls back to the standard parameter AWSEnableHttps .

Examples

Multiple buckets

This example configures a real Amazon S3 bucket AWSBucket and a Pure Storage bucket PureStorageBucket with S3BucketConfig.

AWSBucket does not specify an endpoint or protocol, so Vertica falls back to AWSEndpoint (defaults to s3.amazonaws.com) and AWSEnableHttps (defaults to 1).

In this example environment, access to the PureStorageBucket is over a secure network, so HTTPS is disabled:

```
=> ALTER DATABASE DEFAULT SET S3BucketConfig='
[
   {
      "bucket": "AWSBucket",
      "region": "us-east-2"
   },
   {
      "bucket": "PureStorageBucket",
      "endpoint": "pure.mycorp.net:1234",
      "protocol": "http",
      "enableVirtualAddressing": false
   }
]';
```

Bob can then set S3BucketCredentials at the session level to authenticate to AWSBucket :

```
=> ALTER SESSION SET S3BucketCredentials='
[
   {
      "bucket": "AWSBucket",
      "accessKey": "<AK0>",
      "secretAccessKey": "<SAK0>",
      "sessionToken": "1234567890"
   }
]';
```

Similarly, Alice can authenticate to PureStorageBucket :

```
=> ALTER SESSION SET S3BucketCredentials='
[
   {
      "bucket": "PureStorageBucket",
      "accessKey": "<AK1>",
      "secretAccessKey": "<SAK1>"
   }
]';
```

Charlie provides credentials for both AWSBucket and PureStorageBucket and authenticates to them simultaneously. This allows him to perform cross-endpoint joins, export from one bucket to another, etc.

```
=> ALTER SESSION SET S3BucketCredentials='
[
   {
      "bucket": "AWSBucket",
      "accessKey": "<AK0>",
      "secretAccessKey": "<SAK0>",
      "sessionToken": "1234567890"
   },
   {
      "bucket": "PureStorageBucket",
      "accessKey": "<AK1>",
      "secretAccessKey": "<SAK1>"
   }
]';
```

S3 server-side encryption

S3 has three types of server-side encryption: SSE-S3, SSE-KMS, and SSE-C. The following example configures access using SSE-KMS:

```
=> ALTER DATABASE DEFAULT SET S3BucketConfig='
[
   {
      "bucket": "AWSBucket",
      "region": "us-east-2",
      "serverSideEncryption": "aws:kms",
      "sseKmsKeyId": "1234abcd-12ab-34cd-56ef-1234567890ab"
   }
]';
```

For more information, see Server-Side Encryption .

Non-amazon S3 storage with AWSEndpoint and S3BucketConfig

If AWSEndpoint is set to a non-Amazon S3 bucket like Pure Storage or MinIO and you want to configure S3BucketConfig for a real Amazon S3 bucket, the following requirements apply:

- If your real Amazon S3 region is not us-east-1 (the default), you must specify the region .
- Set endpoint to an empty string ("").

In this example, AWSEndpoint is set to a Pure Storage bucket.

```
=> ALTER DATABASE DEFAULT SET AWSEndpoint='pure.mycorp.net:1234';
```

To configure S3BucketConfig for a real Amazon S3 bucket realAmazonS3Bucket in region " us-east-2 ":

```
=> ALTER DATABASE DEFAULT SET S3BucketConfig='
[
   {
      "bucket": "realAmazonS3Bucket",
      "region": "us-east-2",
      "endpoint": ""
   },
]';
```

Functions

Functions return information from the database. This section describes functions that Vertica supports. Except for meta-functions, you can use a function anywhere an expression is allowed.

Meta-functions usually access the internal state of Vertica. They can be used in a top-level SELECT statement only, and the statement cannot contain other clauses such as FROM or WHERE. Meta-functions are labeled on their reference pages.

The Behavior Type section on each reference page categorizes the function's return behavior as one or more of the following:

- **Immutable** (invariant): When run with a given set of arguments, immutable functions always produce the same result, regardless of environment or session settings such as locale.
- **Stable** : When run with a given set of arguments, stable functions produce the same result within a single query or scan operation. However, a stable function can produce different results when issued under different environments or at different times, such as change of locale and time zone—for example, SYSDATE .
- **Volatile** : Regardless of their arguments or environment, volatile functions can return a different result with each invocation—for example, UUID_GENERATE .

List of all functions

The following list contains all Vertica SQL functions.

Jump to letter: A - B - C - D - E - F - G - H - I - J - K - L - M - N - O - P - Q - R - S - T - U - V - W - X - Y - Z

A

ABS
Returns the absolute value of the argument. [Mathematical functions]

ACOS
Returns a DOUBLE PRECISION value representing the trigonometric inverse cosine of the argument. [Mathematical functions]

ACOSH
Returns a DOUBLE PRECISION value that represents the inverse (arc) hyperbolic cosine of the function argument. [Mathematical functions]

ACTIVE_SCHEDULER_NODE
Returns the active scheduler node. [Stored procedure functions]

ADD_MONTHS
Adds the specified number of months to a date and returns the sum as a DATE. [Date/time functions]

ADVANCE_EPOCH
Manually closes the current epoch and begins a new epoch. [Epoch functions]

AGE_IN_MONTHS
Returns the difference in months between two dates, expressed as an integer. [Date/time functions]

AGE_IN_YEARS
Returns the difference in years between two dates, expressed as an integer. [Date/time functions]

ALTER_LOCATION_LABEL
Adds a label to a storage location, or changes or removes an existing label. [Storage functions]

ALTER_LOCATION_SIZE
Resizes on one node, all nodes in a subcluster, or all nodes in the database. [Eon Mode functions]

ALTER_LOCATION_USE
Alters the type of data that a storage location holds. [Storage functions]

ANALYZE_CONSTRAINTS
Analyzes and reports on constraint violations within the specified scope. [Table functions]

ANALYZE_CORRELATIONS
This function is deprecated and will be removed in a future release. [Table functions]

ANALYZE_EXTERNAL_ROW_COUNT
Calculates the exact number of rows in an external table. [Statistics management functions]

ANALYZE_STATISTICS
Collects and aggregates data samples and storage information from all nodes that store projections associated with the specified table. [Statistics management functions]

ANALYZE_STATISTICS_PARTITION
Collects and aggregates data samples and storage information for a range of partitions in the specified table. [Statistics management functions]

ANALYZE_WORKLOAD
Runs Workload Analyzer, a utility that analyzes system information held in system tables. [Workload management functions]

APPLY_AVG
Returns the average of all elements in a with numeric values. [Collection functions]

APPLY_BISECTING_KMEANS
Applies a trained bisecting k-means model to an input relation, and assigns each new data point to the closest matching cluster in the trained model. [Transformation functions]

APPLY_COUNT (ARRAY_COUNT)
Returns the total number of non-null elements in a. [Collection functions]

APPLY_COUNT_ELEMENTS (ARRAY_LENGTH)
Returns the total number of elements in a , including NULLs. [Collection functions]

APPLY_IFOREST
Applies an isolation forest (iForest) model to an input relation. [Transformation functions]

APPLY_INVERSE_PCA
Inverts the APPLY_PCA-generated transform back to the original coordinate system. [Transformation functions]

APPLY_INVERSE_SVD
Transforms the data back to the original domain. [Transformation functions]

APPLY_KMEANS
Assigns each row of an input relation to a cluster center from an existing k-means model. [Transformation functions]

APPLY_KPROTOTYPES
Assigns each row of an input relation to a cluster center from an existing k-prototypes model. [Transformation functions]

APPLY_MAX
Returns the largest non-null element in a. [Collection functions]

APPLY_MIN
Returns the smallest non-null element in a. [Collection functions]

APPLY_NORMALIZE
A UDTF function that applies the normalization parameters saved in a model to a set of specified input columns. [Transformation functions]

APPLY_ONE_HOT_ENCODER
A user-defined transform function (UDTF) that loads the one hot encoder model and writes out a table that contains the encoded columns. [Transformation functions]

APPLY_PCA
Transforms the data using a PCA model. [Transformation functions]

APPLY_SUM
Computes the sum of all elements in a of numeric values (INTEGER, FLOAT, NUMERIC, or INTERVAL). [Collection functions]

APPLY_SVD
Transforms the data using an SVD model. [Transformation functions]

APPROXIMATE_COUNT_DISTINCT
Returns the number of distinct non-NULL values in a data set. [Aggregate functions]

APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS
Calculates the number of distinct non-NULL values from the synopsis objects created by APPROXIMATE_COUNT_DISTINCT_SYNOPSIS. [Aggregate functions]

APPROXIMATE_COUNT_DISTINCT_SYNOPSIS
Summarizes the information of distinct non-NULL values and materializes the result set in a VARBINARY or LONG VARBINARY synopsis object. [Aggregate functions]

APPROXIMATE_COUNT_DISTINCT_SYNOPSIS_MERGE
Aggregates multiple synopses into one new synopsis. [Aggregate functions]

APPROXIMATE_MEDIAN [aggregate]
Computes the approximate median of an expression over a group of rows. [Aggregate functions]

APPROXIMATE_PERCENTILE [aggregate]
Computes the approximate percentile of an expression over a group of rows. [Aggregate functions]

APPROXIMATE_QUANTILES
Computes an array of weighted, approximate percentiles of a column within some user-specified error. [Aggregate functions]

ARGMAX [analytic]
This function is patterned after the mathematical function argmax(f(x)), which returns the value of x that maximizes f(x). [Analytic functions]

ARGMAX_AGG
Takes two arguments target and arg, where both are columns or column expressions in the queried dataset. [Aggregate functions]

ARGMIN [analytic]
This function is patterned after the mathematical function argmin(f(x)), which returns the value of x that minimizes f(x). [Analytic functions]

ARGMIN_AGG
Takes two arguments target and arg, where both are columns or column expressions in the queried dataset. [Aggregate functions]

ARIMA
Creates and trains an autoregressive integrated moving average (ARIMA) model from a time series with consistent timesteps. [Machine learning algorithms]

ARRAY_CAT
Concatenates two arrays of the same element type and dimensionality. [Collection functions]

ARRAY_CONTAINS
Returns true if the specified element is found in the array and false if not. [Collection functions]

ARRAY_DIMS
Returns the dimensionality of the input array. [Collection functions]

ARRAY_FIND
Returns the ordinal position of a specified element in an array, or -1 if not found. [Collection functions]

ASCII
Converts the first character of a VARCHAR datatype to an INTEGER. [String functions]

ASIN
Returns a DOUBLE PRECISION value representing the trigonometric inverse sine of the argument. [Mathematical functions]

ASINH
Returns a DOUBLE PRECISION value that represents the inverse (arc) hyperbolic sine of the function argument. [Mathematical functions]

ATAN
Returns a DOUBLE PRECISION value representing the trigonometric inverse tangent of the argument. [Mathematical functions]

ATAN2
Returns a DOUBLE PRECISION value representing the trigonometric inverse tangent of the arithmetic dividend of the arguments. [Mathematical functions]

ATANH
Returns a DOUBLE PRECISION value that represents the inverse hyperbolic tangent of the function argument. [Mathematical functions]

AUDIT
Returns the raw data size (in bytes) of a database, schema, or table as it is counted in an audit of the database size. [License functions]

AUDIT_FLEX
Returns the estimated ROS size of __raw__ columns, equivalent to the export size of the flex data in the audited objects. [License functions]

AUDIT_LICENSE_SIZE
Triggers an immediate audit of the database size to determine if it is in compliance with the raw data storage allowance included in your Vertica licenses. [License functions]

AUDIT_LICENSE_TERM
Triggers an immediate audit to determine if the Vertica license has expired. [License functions]

AUTOREGRESSOR
Creates an autoregressive (AR) model from a stationary time series with consistent timesteps that can then be used for prediction via PREDICT_AR. [Machine learning algorithms]

AVG [aggregate]
Computes the average (arithmetic mean) of an expression over a group of rows. [Aggregate functions]

AVG [analytic]
Computes an average of an expression in a group within a. [Analytic functions]

AZURE_TOKEN_CACHE_CLEAR
Clears the cached access token for Azure. [Cloud functions]

B

BACKGROUND_DEPOT_WARMING
Vertica version 10.0.0 removes support for foreground depot warming. [Eon Mode functions]

BALANCE
Returns a view with an equal distribution of the input data based on the response_column. [Data preparation]

BISECTING_KMEANS
Executes the bisecting k-means algorithm on an input relation. [Machine learning algorithms]

BIT_AND
Takes the bitwise AND of all non-null input values. [Aggregate functions]

BIT_LENGTH
Returns the length of the string expression in bits (bytes * 8) as an INTEGER. [String functions]

BIT_OR
Takes the bitwise OR of all non-null input values. [Aggregate functions]

BIT_XOR
Takes the bitwise XOR of all non-null input values. [Aggregate functions]

BITCOUNT
Returns the number of one-bits (sometimes referred to as set-bits) in the given VARBINARY value. [String functions]

BITSTRING_TO_BINARY
Translates the given VARCHAR bitstring representation into a VARBINARY value. [String functions]

BOOL_AND [aggregate]
Processes Boolean values and returns a Boolean value result. [Aggregate functions]

BOOL_AND [analytic]
Returns the Boolean value of an expression within a. [Analytic functions]

BOOL_OR [aggregate]
Processes Boolean values and returns a Boolean value result. [Aggregate functions]

BOOL_OR [analytic]
Returns the Boolean value of an expression within a. [Analytic functions]

BOOL_XOR [aggregate]
Processes Boolean values and returns a Boolean value result. [Aggregate functions]

BOOL_XOR [analytic]
Returns the Boolean value of an expression within a. [Analytic functions]

BTRIM
Removes the longest string consisting only of specified characters from the start and end of a string. [String functions]

BUILD_FLEXTABLE_VIEW
Creates, or re-creates, a view for a default or user-defined keys table, ignoring any empty keys. [Flex data functions]

C

CALENDAR_HIERARCHY_DAY
Specifies to group DATE partition keys into a hierarchy of years, months, and days. [Partition functions]

CANCEL_DEPOT_WARMING
Cancels depot warming on a node. [Eon Mode functions]

CANCEL_DRAIN_SUBCLUSTER
Cancels the draining of a subcluster or subclusters. [Eon Mode functions]

CANCEL_REBALANCE_CLUSTER
Stops any rebalance task that is currently in progress or is waiting to execute. [Cluster functions]

CANCEL_REFRESH
Cancels refresh-related internal operations initiated by START_REFRESH and REFRESH. [Session functions]

CBRT
Returns the cube root of the argument. [Mathematical functions]

CEILING
Rounds up the returned value up to the next whole number. [Mathematical functions]

CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY
Changes the run-time priority of an active query. [Workload management functions]

CHANGE_MODEL_STATUS
Changes the status of a registered model. [Model management]

CHANGE_RUNTIME_PRIORITY
Changes the run-time priority of a query that is actively running. [Workload management functions]

CHARACTER_LENGTH
The CHARACTER_LENGTH() function:. [String functions]

CHR
Converts the first character of an INTEGER datatype to a VARCHAR. [String functions]

CLEAN_COMMUNAL_STORAGE
Marks for deletion invalid data in communal storage, often data that leaked due to an event where Vertica cleanup mechanisms failed. [Eon Mode functions]

CLEAR_CACHES
Clears the Vertica internal cache files. [Storage functions]

CLEAR_DATA_COLLECTOR
Clears all memory and disk records from Data Collector tables and logs, and resets collection statistics in system table DATA_COLLECTOR. [Data collector functions]

CLEAR_DATA_DEPOT
Deletes the specified depot data. [Eon Mode functions]

CLEAR_DEPOT_PIN_POLICY_PARTITION
Clears a depot pinning policy from the specified table or projection partitions. [Eon Mode functions]

CLEAR_DEPOT_PIN_POLICY_PROJECTION
Clears a depot pinning policy from the specified projection. [Eon Mode functions]

CLEAR_DEPOT_PIN_POLICY_TABLE
Clears a depot pinning policy from the specified table. [Eon Mode functions]

CLEAR_FETCH_QUEUE
Removes all entries or entries for a specific transaction from the queue of fetch requests of data from the communal storage. [Eon Mode functions]

CLEAR_HDFS_CACHES
Clears the configuration information copied from HDFS and any cached connections. [Hadoop functions]

CLEAR_OBJECT_STORAGE_POLICY
Removes a user-defined storage policy from the specified database, schema or table. [Storage functions]

CLEAR_PROFILING
Clears from memory data for the specified profiling type. [Profiling functions]

CLEAR_PROJECTION_REFRESHES
Clears information projection refresh history from system table PROJECTION_REFRESHES. [Projection functions]

CLEAR_RESOURCE_REJECTIONS
Clears the content of the RESOURCE_REJECTIONS and DISK_RESOURCE_REJECTIONS system tables. [Database functions]

CLOCK_TIMESTAMP
Returns a value of type TIMESTAMP WITH TIMEZONE that represents the current system-clock time. [Date/time functions]

CLOSE_ALL_RESULTSETS
Closes all result set sessions within Multiple Active Result Sets (MARS) and frees the MARS storage for other result sets. [Client connection functions]

CLOSE_ALL_SESSIONS
Closes all external sessions except the one that issues this function. [Session functions]

CLOSE_RESULTSET
Closes a specific result set within Multiple Active Result Sets (MARS) and frees the MARS storage for other result sets. [Client connection functions]

CLOSE_SESSION
Interrupts the specified external session, rolls back the current transaction if any, and closes the socket. [Session functions]

CLOSE_USER_SESSIONS
Stops the session for a user, rolls back any transaction currently running, and closes the connection. [Session functions]

COALESCE
Returns the value of the first non-null expression in the list. [NULL-handling functions]

COLLATION
Applies a collation to two or more strings. [String functions]

COMPACT_STORAGE
Bundles existing data (.fdb) and index (.pidx) files into the .gt file format. [Database functions]

COMPUTE_FLEXTABLE_KEYS
Computes the virtual columns (keys and values) from flex table VMap data. [Flex data functions]

COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW
Combines the functionality of BUILD_FLEXTABLE_VIEW and COMPUTE_FLEXTABLE_KEYS to compute virtual columns (keys) from the VMap data of a flex table and construct a view. [Flex data functions]

CONCAT
Concatenates two strings and returns a varchar data type. [String functions]

CONDITIONAL_CHANGE_EVENT [analytic]
Assigns an event window number to each row, starting from 0, and increments by 1 when the result of evaluating the argument expression on the current row differs from that on the previous row. [Analytic functions]

CONDITIONAL_TRUE_EVENT [analytic]
Assigns an event window number to each row, starting from 0, and increments the number by 1 when the result of the boolean argument expression evaluates true. [Analytic functions]

CONFUSION_MATRIX
Computes the confusion matrix of a table with observed and predicted values of a response variable. [Model evaluation]

CONTAINS
Returns true if the specified element is found in the collection and false if not. [Collection functions]

COPY_PARTITIONS_TO_TABLE
Copies partitions from one table to another. [Partition functions]

COPY_TABLE
Copies one table to another. [Table functions]

CORR
Returns the DOUBLE PRECISION coefficient of correlation of a set of expression pairs, as per the Pearson correlation coefficient. [Aggregate functions]

CORR_MATRIX
Takes an input relation with numeric columns, and calculates the Pearson Correlation Coefficient between each pair of its input columns. [Data preparation]

COS
Returns a DOUBLE PRECISION value tat represents the trigonometric cosine of the passed parameter. [Mathematical functions]

COSH
Returns a DOUBLE PRECISION value that represents the hyperbolic cosine of the passed parameter. [Mathematical functions]

COT
Returns a DOUBLE PRECISION value representing the trigonometric cotangent of the argument. [Mathematical functions]

COUNT [aggregate]
Returns as a BIGINT the number of rows in each group where the expression is not NULL. [Aggregate functions]

COUNT [analytic]
Counts occurrences within a group within a. [Analytic functions]

COVAR_POP
Returns the population covariance for a set of expression pairs. [Aggregate functions]

COVAR_SAMP
Returns the sample covariance for a set of expression pairs. [Aggregate functions]

CROSS_VALIDATE
Performs k-fold cross validation on a learning algorithm using an input relation, and grid search for hyper parameters. [Model evaluation]

CUME_DIST [analytic]
Calculates the cumulative distribution, or relative rank, of the current row with regard to other rows in the same partition within a . [Analytic functions]

CURRENT_DATABASE
Returns the name of the current database, equivalent to DBNAME. [System information functions]

CURRENT_DATE
Returns the date (date-type value) on which the current transaction started. [Date/time functions]

CURRENT_LOAD_SOURCE

When called within the scope of a COPY statement, returns the file name used for the load. [System information functions]

CURRENT_SCHEMA
Returns the name of the current schema. [System information functions]

CURRENT_SESSION
Returns the ID of the current client session. [System information functions]

CURRENT_TIME
Returns a value of type TIME WITH TIMEZONE that represents the start of the current transaction. [Date/time functions]

CURRENT_TIMESTAMP
Returns a value of type TIME WITH TIMEZONE that represents the start of the current transaction. [Date/time functions]

CURRENT_TRANS_ID
Returns the ID of the transaction currently in progress. [System information functions]

CURRENT_USER
Returns a VARCHAR containing the name of the user who initiated the current database connection. [System information functions]

CURRVAL
Returns the last value across all nodes that was set by NEXTVAL on this sequence in the current session. [Sequence functions]

D

DATA_COLLECTOR_HELP
Returns online usage instructions about the Data Collector, the V_MONITOR.DATA_COLLECTOR system table, and the Data Collector control functions. [Data collector functions]

DATE
Converts the input value to a DATE data type. [Date/time functions]

DATE_PART
Extracts a sub-field such as year or hour from a date/time expression, equivalent to the the SQL-standard function EXTRACT. [Date/time functions]

DATE_TRUNC
Truncates date and time values to the specified precision. [Date/time functions]

DATEDIFF
Returns the time span between two dates, in the intervals specified. [Date/time functions]

DAY
Returns as an integer the day of the month from the input value. [Date/time functions]

DAYOFMONTH
Returns the day of the month as an integer. [Date/time functions]

DAYOFWEEK
Returns the day of the week as an integer, where Sunday is day 1. [Date/time functions]

DAYOFWEEK_ISO
Returns the ISO 8061 day of the week as an integer, where Monday is day 1. [Date/time functions]

DAYOFYEAR
Returns the day of the year as an integer, where January 1 is day 1. [Date/time functions]

DAYS
Returns the integer value of the specified date, where 1 AD is 1. [Date/time functions]

DBNAME (function)
Returns the name of the current database, equivalent to CURRENT_DATABASE. [System information functions]

DECODE
Compares expression to each search value one by one. [String functions]

DEGREES
Converts an expression from radians to fractional degrees, or from degrees, minutes, and seconds to fractional degrees. [Mathematical functions]

DELETE_TOKENIZER_CONFIG_FILE
Deletes a tokenizer configuration file. [Text search functions]

DEMOTE_SUBCLUSTER_TO_SECONDARY
Converts a to a . [Eon Mode functions]

DENSE_RANK [analytic]

Within each window partition, ranks all rows in the query results set according to the order specified by the window's ORDER BY clause. [Analytic functions]

DESCRIBE_LOAD_BALANCE_DECISION
Evaluates if any load balancing routing rules apply to a given IP address and This function is useful when you are evaluating connection load balancing policies you have created, to ensure they work the way you expect them to. [Client connection functions]

DESIGNER_ADD_DESIGN_QUERIES
Reads and evaluates queries from an input file, and adds the queries that it accepts to the specified design. [Database Designer functions]

DESIGNER_ADD_DESIGN_QUERIES_FROM_RESULTS
Executes the specified query and evaluates results in the following columns:. [Database Designer functions]

DESIGNER_ADD_DESIGN_QUERY
Reads and parses the specified query, and if accepted, adds it to the design. [Database Designer functions]

DESIGNER_ADD_DESIGN_TABLES
Adds the specified tables to a design. [Database Designer functions]

DESIGNER_CANCEL_POPULATE_DESIGN
Cancels population or deployment operation for the specified design if it is currently running. [Database Designer functions]

DESIGNER_CREATE_DESIGN
Creates a design with the specified name. [Database Designer functions]

DESIGNER_DESIGN_PROJECTION_ENCODINGS
Analyzes encoding in the specified projections, creates a script to implement encoding recommendations, and optionally deploys the recommendations. [Database Designer functions]

DESIGNER_DROP_ALL_DESIGNS
Removes all Database Designer-related schemas associated with the current user. [Database Designer functions]

DESIGNER_DROP_DESIGN
Removes the schema associated with the specified design and all its contents. [Database Designer functions]

DESIGNER_OUTPUT_ALL_DESIGN_PROJECTIONS
Displays the DDL statements that define the design projections to standard output. [Database Designer functions]

DESIGNER_OUTPUT_DEPLOYMENT_SCRIPT
Displays the deployment script for the specified design to standard output. [Database Designer functions]

DESIGNER_RESET_DESIGN
Discards all run-specific information of the previous Database Designer build or deployment of the specified design but keeps its configuration. [Database Designer functions]

DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY
Populates the design and creates the design and deployment scripts. [Database Designer functions]

DESIGNER_SET_DESIGN_KSAFETY
Sets K-safety for a comprehensive design and stores the K-safety value in the DESIGNS table. [Database Designer functions]

DESIGNER_SET_DESIGN_TYPE
Specifies whether Database Designer creates a comprehensive or incremental design. [Database Designer functions]

DESIGNER_SET_OPTIMIZATION_OBJECTIVE
Valid only for comprehensive database designs, specifies the optimization objective Database Designer uses. [Database Designer functions]

DESIGNER_SET_PROPOSE_UNSEGMENTED_PROJECTIONS
Specifies whether a design can include unsegmented projections. [Database Designer functions]

DESIGNER_SINGLE_RUN
Evaluates all queries that completed execution within the specified timespan, and returns with a design that is ready for deployment. [Database Designer functions]

DESIGNER_WAIT_FOR_DESIGN
Waits for completion of operations that are populating and deploying the design. [Database Designer functions]

DETECT_OUTLIERS
Returns the outliers in a data set based on the outlier threshold. [Data preparation]

DISABLE_DUPLICATE_KEY_ERROR
Disables error messaging when Vertica finds duplicate primary or unique key values at run time (for use with key constraints that are not automatically enabled). [Table functions]

DISABLE_LOCAL_SEGMENTS
Disables local data segmentation, which breaks projections segments on nodes into containers that can be easily moved to other nodes. [

Cluster functions]

DISABLE_PROFILING
Disables for the current session collection of profiling data of the specified type. [Profiling functions]

DISPLAY_LICENSE
Returns the terms of your Vertica license. [License functions]

DISTANCE
Returns the distance (in kilometers) between two points. [Mathematical functions]

DISTANCEV
Returns the distance (in kilometers) between two points using the Vincenty formula. [Mathematical functions]

DO_TM_TASK
Runs a (TM) operation and commits current transactions. [Storage functions]

DROP_EXTERNAL_ROW_COUNT
Removes external table row count statistics compiled by ANALYZE_EXTERNAL_ROW_COUNT. [Statistics management functions]

DROP_LICENSE
Drops a license key from the global catalog. [Catalog functions]

DROP_LOCATION
Permanently removes a retired storage location. [Storage functions]

DROP_PARTITIONS
Drops the specified table partition keys. [Partition functions]

DROP_STATISTICS
Removes statistical data on database projections previously generated by ANALYZE_STATISTICS. [Statistics management functions]

DROP_STATISTICS_PARTITION
Removes statistical data on database projections previously generated by ANALYZE_STATISTICS_PARTITION. [Statistics management functions]

DUMP_CATALOG
Returns an internal representation of the Vertica catalog. [Catalog functions]

DUMP_LOCKTABLE
Returns information about deadlocked clients and the resources they are waiting for. [Database functions]

DUMP_PARTITION_KEYS
Dumps the partition keys of all projections in the system. [Database functions]

DUMP_PROJECTION_PARTITION_KEYS
Dumps the partition keys of the specified projection. [Partition functions]

DUMP_TABLE_PARTITION_KEYS
Dumps the partition keys of all projections for the specified table. [Partition functions]

E

EDIT_DISTANCE
Calculates and returns the Levenshtein distance between two strings. [String functions]

EMPTYMAP
Constructs a new VMap with one row but without keys or data. [Flex map functions]

ENABLE_ELASTIC_CLUSTER
Enables elastic cluster scaling, which makes enlarging or reducing the size of your database cluster more efficient by segmenting a node's data into chunks that can be easily moved to other hosts. [Cluster functions]

ENABLE_LOCAL_SEGMENTS
Enables local storage segmentation, which breaks projections segments on nodes into containers that can be easily moved to other nodes. [Cluster functions]

ENABLE_PROFILING
Enables collection of profiling data of the specified type for the current session. [Profiling functions]

ENABLE_SCHEDULE
Enables or disables a schedule. [Stored procedure functions]

ENABLE_TRIGGER
Enables or disables a trigger. [Stored procedure functions]

ENABLED_ROLE
Checks whether a Vertica user role is enabled, and returns true or false. [Privileges and access functions]

ENFORCE_OBJECT_STORAGE_POLICY
Applies storage policies of the specified object immediately. [Storage functions]

ERROR_RATE
Using an input table, returns a table that calculates the rate of incorrect classifications and displays them as FLOAT values. [Model evaluation]

EVALUATE_DELETE_PERFORMANCE
Evaluates projections for potential DELETE and UPDATE performance issues. [Projection functions]

EVENT_NAME
Returns a VARCHAR value representing the name of the event that matched the row. [MATCH clause functions]

EXECUTE_TRIGGER
Manually executes the stored procedure attached to a trigger. [Stored procedure functions]

EXP
Returns the exponential function, e to the power of a number. [Mathematical functions]

EXPLODE
Expands the elements of one or more collection columns (ARRAY or SET) into individual table rows, one row per element. [Collection functions]

EXPONENTIAL_MOVING_AVERAGE [analytic]
Calculates the exponential moving average (EMA) of expression E with smoothing factor X. [Analytic functions]

EXPORT_CATALOG
This function and EXPORT_OBJECTS return equivalent output. [Catalog functions]

EXPORT_DIRECTED_QUERIES
Generates SQL for creating directed queries from a set of input queries. [Directed queries functions]

EXPORT_MODELS
Exports machine learning models. [Model management]

EXPORT_OBJECTS
This function and EXPORT_CATALOG return equivalent output. [Catalog functions]

EXPORT_STATISTICS
Generates statistics in XML format from data previously collected by ANALYZE_STATISTICS. [Statistics management functions]

EXPORT_STATISTICS_PARTITION
Generates partition-level statistics in XML format from data previously collected by ANALYZE_STATISTICS_PARTITION. [Statistics management functions]

EXPORT_TABLES
Generates a SQL script that can be used to recreate a logical schema—schemas, tables, constraints, and views—on another cluster. [Catalog functions]

EXTERNAL_CONFIG_CHECK
Tests the Hadoop configuration of a Vertica cluster. [Hadoop functions]

EXTRACT
Retrieves sub-fields such as year or hour from date/time values and returns values of type NUMERIC. [Date/time functions]

F

FILTER
Takes an input array and returns an array containing only elements that meet a specified condition. [Collection functions]

FINISH_FETCHING_FILES
Fetches to the depot all files that are queued for download from communal storage. [Eon Mode functions]

FIRST_VALUE [analytic]
Lets you select the first value of a table or partition (determined by the window-order-clause) without having to use a self join. [Analytic functions]

FLOOR
Rounds down the returned value to the previous whole number. [Mathematical functions]

FLUSH_DATA_COLLECTOR
Waits until memory logs are moved to disk and then flushes the Data Collector, synchronizing the log with disk storage. [Data collector functions]

FLUSH_REAPER_QUEUE
Deletes all data marked for deletion in the database. [Eon Mode functions]

G

GET_AHM_EPOCH
Returns the number of the in which the is located. [Epoch functions]

GET_AHM_TIME
Returns a TIMESTAMP value representing the. [Epoch functions]

GET_AUDIT_TIME
Reports the time when the automatic audit of database size occurs. [License functions]

GET_CLIENT_LABEL
Returns the client connection label for the current session. [Client connection functions]

GET_COMPLIANCE_STATUS
Displays whether your database is in compliance with your Vertica license agreement. [License functions]

GET_CONFIG_PARAMETER
Gets the value of a configuration parameter at the specified level. [Database functions]

GET_CURRENT_EPOCH
Returns the number of the current epoch. [Epoch functions]

GET_DATA_COLLECTOR_NOTIFY_POLICY
Lists any notification policies set on a component. [Notifier functions]

GET_DATA_COLLECTOR_POLICY
Retrieves a brief statement about the retention policy for the specified component. [Data collector functions]

GET_LAST_GOOD_EPOCH
Returns the number. [Epoch functions]

GET_METADATA
Returns the metadata of a Parquet file. [Hadoop functions]

GET_MODEL_ATTRIBUTE
Extracts either a specific attribute from a model or all attributes from a model. [Model management]

GET_MODEL_SUMMARY
Returns summary information of a model. [Model management]

GET_NUM_ACCEPTED_ROWS
Returns the number of rows loaded into the database for the last completed load for the current session. [Session functions]

GET_NUM_REJECTED_ROWS
Returns the number of rows that were rejected during the last completed load for the current session. [Session functions]

GET_PRIVILEGES_DESCRIPTION
Returns the effective privileges the current user has on an object, including explicit, implicit, inherited, and role-based privileges. [Privileges and access functions]

GET_PROJECTION_SORT_ORDER
Returns the order of columns in a projection's ORDER BY clause. [Projection functions]

GET_PROJECTION_STATUS
Returns information relevant to the status of a :. [Projection functions]

GET_PROJECTIONS
Returns contextual and projection information about projections of the specified anchor table. [Projection functions]

GET_TOKENIZER_PARAMETER
Returns the configuration parameter for a given tokenizer. [Text search functions]

GETDATE
Returns the current statement's start date and time as a TIMESTAMP value. [Date/time functions]

GETUTCDATE
Returns the current statement's start date and time as a TIMESTAMP value. [Date/time functions]

GREATEST
Returns the largest value in a list of expressions of any data type. [String functions]

GREATESTB
Returns the largest value in a list of expressions of any data type, using binary ordering. [String functions]

GROUP_ID
Uniquely identifies duplicate sets for GROUP BY queries that return duplicate grouping sets. [Aggregate functions]

GROUPING

Disambiguates the use of NULL values when GROUP BY queries with multilevel aggregates generate NULL values to identify subtotals in grouping columns. [Aggregate functions]

GROUPING_ID
Concatenates the set of Boolean values generated by the GROUPING function into a bit vector. [Aggregate functions]

H

HADOOP_IMPERSONATION_CONFIG_CHECK
Reports the delegation tokens Vertica will use when accessing Kerberized data in HDFS. [Hadoop functions]

HAS_ROLE
Checks whether a Vertica user role is granted to the specified user or role, and returns true or false. [Privileges and access functions]

HAS_TABLE_PRIVILEGE
Returns true or false to verify whether a user has the specified privilege on a table. [System information functions]

HASH
Calculates a hash value over the function arguments, producing a value in the range 0 <= x < 263. [Mathematical functions]

HASH_EXTERNAL_TOKEN
Returns a hash of a string token, for use with HADOOP_IMPERSONATION_CONFIG_CHECK. [Hadoop functions]

HCATALOGCONNECTOR_CONFIG_CHECK
Tests the configuration of a Vertica cluster that uses the HCatalog Connector to access Hive data. [Hadoop functions]

HDFS_CLUSTER_CONFIG_CHECK
Tests the configuration of a Vertica cluster that uses HDFS. [Hadoop functions]

HEX_TO_BINARY
Translates the given VARCHAR hexadecimal representation into a VARBINARY value. [String functions]

HEX_TO_INTEGER
Translates the given VARCHAR hexadecimal representation into an INTEGER value. [String functions]

HOUR
Returns the hour portion of the specified date as an integer, where 0 is 00:00 to 00:59. [Date/time functions]

I

IFNULL
Returns the value of the first non-null expression in the list. [NULL-handling functions]

IFOREST
Trains and returns an isolation forest (iForest) model. [Data preparation]

IMPLODE
Takes a column of any scalar type and returns an unbounded array. [Collection functions]

IMPORT_DIRECTED_QUERIES
Imports to the database catalog directed queries from a SQL file that was generated by EXPORT_DIRECTED_QUERIES. [Directed queries functions]

IMPORT_MODELS
Imports models into Vertica, either Vertica models that were exported with EXPORT_MODELS, or models in Predictive Model Markup Language (PMML) or TensorFlow format. [Model management]

IMPORT_STATISTICS
Imports statistics from the XML file that was generated by EXPORT_STATISTICS. [Statistics management functions]

IMPUTE
Imputes missing values in a data set with either the mean or the mode, based on observed values for a variable in each column. [Data preparation]

INET_ATON
Converts a string that contains a dotted-quad representation of an IPv4 network address to an INTEGER. [IP address functions]

INET_NTOA
Converts an INTEGER value into a VARCHAR dotted-quad representation of an IPv4 network address. [IP address functions]

INFER_EXTERNAL_TABLE_DDL
This function is deprecated and will be removed in a future release. [Table functions]

INFER_TABLE_DDL
Inspects a file in Parquet, ORC, JSON, or Avro format and returns a CREATE TABLE or CREATE EXTERNAL TABLE statement based on its contents. [Table functions]

INITCAP
Capitalizes first letter of each alphanumeric word and puts the rest in lowercase. [String functions]

INITCAPB
Capitalizes first letter of each alphanumeric word and puts the rest in lowercase. [String functions]

INSERT
Inserts a character string into a specified location in another character string. [String functions]

INSTALL_LICENSE
Installs the license key in the global catalog. [Catalog functions]

INSTR
Searches string for substring and returns an integer indicating the position of the character in string that is the first character of this occurrence. [String functions]

INSTRB
Searches string for substring and returns an integer indicating the octet position within string that is the first occurrence. [String functions]

INTERRUPT_STATEMENT
Interrupts the specified statement in a user session, rolls back the current transaction, and writes a success or failure message to the log file. [Session functions]

ISFINITE
Tests for the special TIMESTAMP constant INFINITY and returns a value of type BOOLEAN. [Date/time functions]

ISNULL
Returns the value of the first non-null expression in the list. [NULL-handling functions]

ISUTF8
Tests whether a string is a valid UTF-8 string. [String functions]

J

JARO_DISTANCE
Calculates and returns the Jaro similarity, an edit distance between two sequences. [String functions]

JARO_WINKLER_DISTANCE
Calculates and returns the Jaro-Winkler similarity, an edit distance between two sequences. [String functions]

JULIAN_DAY
Returns the integer value of the specified day according to the Julian calendar, where day 1 is the first day of the Julian period, January 1, 4713 BC (on the Gregorian calendar, November 24, 4714 BC). [Date/time functions]

K

KERBEROS_CONFIG_CHECK
Tests the Kerberos configuration of a Vertica cluster. [Database functions]

KERBEROS_HDFS_CONFIG_CHECK
This function is deprecated and will be removed in a future release. [Hadoop functions]

KMEANS
Executes the k-means algorithm on an input relation. [Machine learning algorithms]

KPROTOTYPES
Executes the k-prototypes algorithm on an input relation. [Machine learning algorithms]

L

LAG [analytic]
Returns the value of the input expression at the given offset before the current row within a. [Analytic functions]

LAST_DAY
Returns the last day of the month in the specified date. [Date/time functions]

LAST_INSERT_ID
Returns the last value of an IDENTITY column. [Table functions]

LAST_VALUE [analytic]
Lets you select the last value of a table or partition (determined by the window-order-clause) without having to use a self join. [Analytic functions]

LDAP_LINK_DRYRUN_CONNECT
Takes a set of LDAP Link connection parameters as arguments and begins a dry run connection between the LDAP server and Vertica. [LDAP link functions]

LDAP_LINK_DRYRUN_SEARCH

Takes a set of LDAP Link connection and search parameters as arguments and begins a dry run search for users and groups that would get imported from the LDAP server. [LDAP link functions]

LDAP_LINK_DRYRUN_SYNC

Takes a set of LDAP Link connection and search parameters as arguments and begins a dry run synchronization between the database and the LDAP server, which maps and synchronizes the LDAP server's users and groups with their equivalents in Vertica. [LDAP link functions]

LDAP_LINK_SYNC_CANCEL

Cancels in-progress LDAP Link synchronizations (including those started by LDAP_LINK_DRYRUN_SYNC) between the LDAP server and Vertica. [LDAP link functions]

LDAP_LINK_SYNC_START

Begins the synchronization between the LDAP server and Vertica immediately rather than waiting for the interval set in LDAPLinkInterval. [LDAP link functions]

LEAD [analytic]

Returns values from the row after the current row within a , letting you access more than one row in a table at the same time. [Analytic functions]

LEAST

Returns the smallest value in a list of expressions of any data type. [String functions]

LEASTB

Returns the smallest value in a list of expressions of any data type, using binary ordering. [String functions]

LEFT

Returns the specified characters from the left side of a string. [String functions]

LENGTH

Returns the length of a string. [String functions]

LIFT_TABLE

Returns a table that compares the predictive quality of a machine learning model. [Model evaluation]

LINEAR_REG

Executes linear regression on an input relation, and returns a linear regression model. [Machine learning algorithms]

LIST_ENABLED_CIPHERS

Returns a list of enabled cipher suites, which are sets of algorithms used to secure TLS/SSL connections. [System information functions]

LISTAGG

Transforms non-null values from a group of rows into a list of values that are delimited by commas (default) or a configurable separator. [Aggregate functions]

LN

Returns the natural logarithm of the argument. [Mathematical functions]

LOCALTIME

Returns a value of type TIME that represents the start of the current transaction. [Date/time functions]

LOCALTIMESTAMP

Returns a value of type TIMESTAMP/TIMESTAMPTZ that represents the start of the current transaction, and remains unchanged until the transaction is closed. [Date/time functions]

LOG

Returns the logarithm to the specified base of the argument. [Mathematical functions]

LOG10

Returns the base 10 logarithm of the argument, also known as the common logarithm. [Mathematical functions]

LOGISTIC_REG

Executes logistic regression on an input relation. [Machine learning algorithms]

LOWER

Takes a string value and returns a VARCHAR value converted to lowercase. [String functions]

LOWERB

Returns a character string with each ASCII character converted to lowercase. [String functions]

LPAD

Returns a VARCHAR value representing a string of a specific length filled on the left with specific characters. [String functions]

LTRIM

Returns a VARCHAR value representing a string with leading blanks removed from the left side (beginning). [String functions]

M

MAKE_AHM_NOW
Sets the (AHM) to the greatest allowable value. [Epoch functions]

MAKEUTF8
Coerces a string to UTF-8 by removing or replacing non-UTF-8 characters. [String functions]

MAPAGGREGATE
Returns a LONG VARBINARY VMap with key and value pairs supplied from two VARCHAR input columns. [Flex map functions]

MAPCONTAINSKEY
Determines whether a VMap contains a virtual column (key). [Flex map functions]

MAPCONTAINSVALUE
Determines whether a VMap contains a specific value. [Flex map functions]

MAPDELIMITEDEXTRACTOR
Extracts data with a delimiter character and other optional arguments, returning a single VMap value. [Flex extractor functions]

MAPITEMS
Returns information about items in a VMap. [Flex map functions]

MAPJSONEXTRACTOR
Extracts content of repeated JSON data objects,, including nested maps, or data with an outer list of JSON elements. [Flex extractor functions]

MAPKEYS
Returns the virtual columns (and values) present in any VMap data. [Flex map functions]

MAPKEYSINFO
Returns virtual column information from a given map. [Flex map functions]

MAPLOOKUP
Returns single-key values from VMAP data. [Flex map functions]

MAPPUT
Accepts a VMap and one or more key/value pairs and returns a new VMap with the key/value pairs added. [Flex map functions]

MAPREGEXEXTRACTOR
Extracts data with a regular expression and returns results as a VMap. [Flex extractor functions]

MAPSIZE
Returns the number of virtual columns present in any VMap data. [Flex map functions]

MAPTOSTRING
Recursively builds a string representation of VMap data, including nested JSON maps. [Flex map functions]

MAPVALUES
Returns a string representation of the top-level values from a VMap. [Flex map functions]

MAPVERSION
Returns the version or invalidity of any map data. [Flex map functions]

MARK_DESIGN_KSAFE
Enables or disables high availability in your environment, in case of a failure. [Catalog functions]

MATCH_COLUMNS
Specified as an element in a SELECT list, returns all columns in queried tables that match the specified pattern. [Regular expression functions]

MATCH_ID
Returns a successful pattern match as an INTEGER value. [MATCH clause functions]

MATERIALIZE_FLEXTABLE_COLUMNS
Materializes virtual columns listed as key_names in the flextable_keys table you compute using either COMPUTE_FLEXTABLE_KEYS or COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW. [Flex data functions]

MAX [aggregate]
Returns the greatest value of an expression over a group of rows. [Aggregate functions]

MAX [analytic]
Returns the maximum value of an expression within a. [Analytic functions]

MD5
Calculates the MD5 hash of string, returning the result as a VARCHAR string in hexadecimal. [String functions]

MEASURE_LOCATION_PERFORMANCE
Measures a storage location's disk performance. [Storage functions]

MEDIAN [analytic]
For each row, returns the median value of a value set within each partition. [Analytic functions]

MEMORY_TRIM
Calls glibc function malloc_trim() to reclaim free memory from malloc and return it to the operating system. [Database functions]

MICROSECOND
Returns the microsecond portion of the specified date as an integer. [Date/time functions]

MIDNIGHT_SECONDS
Within the specified date, returns the number of seconds between midnight and the date's time portion. [Date/time functions]

MIGRATE_ENTERPRISE_TO_EON
Migrates an Enterprise database to an Eon Mode database. [Eon Mode functions]

MIN [aggregate]
Returns the smallest value of an expression over a group of rows. [Aggregate functions]

MIN [analytic]
Returns the minimum value of an expression within a. [Analytic functions]

MINUTE
Returns the minute portion of the specified date as an integer. [Date/time functions]

MOD
Returns the remainder of a division operation. [Mathematical functions]

MONTH
Returns the month portion of the specified date as an integer. [Date/time functions]

MONTHS_BETWEEN
Returns the number of months between two dates. [Date/time functions]

MOVE_PARTITIONS_TO_TABLE
Moves partitions from one table to another. [Partition functions]

MOVE_RETIRED_LOCATION_DATA
Moves all data from the specified retired storage location or from all retired storage locations in the database. [Storage functions]

MOVE_STATEMENT_TO_RESOURCE_POOL
Attempts to move the specified query to the specified target pool. [Workload management functions]

MOVING_AVERAGE
Creates a moving-average (MA) model from a stationary time series with consistent timesteps that can then be used for prediction via PREDICT_MOVING_AVERAGE. [Machine learning algorithms]

MSE
Returns a table that displays the mean squared error of the prediction and response columns in a machine learning model. [Model evaluation]

N

NAIVE_BAYES
Executes the Naive Bayes algorithm on an input relation and returns a Naive Bayes model. [Machine learning algorithms]

NEW_TIME
Converts a timestamp value from one time zone to another and returns a TIMESTAMP. [Date/time functions]

NEXT_DAY
Returns the date of the first instance of a particular day of the week that follows the specified date. [Date/time functions]

NEXTVAL
Returns the next value in a sequence. [Sequence functions]

NORMALIZE
Runs a normalization algorithm on an input relation. [Data preparation]

NORMALIZE_FIT
This function differs from NORMALIZE, which directly outputs a view with normalized results, rather than storing normalization parameters into a model for later operation. [Data preparation]

NOTIFY
Sends a specified message to a NOTIFIER. [Notifier functions]

NOW [date/time]
Returns a value of type TIMESTAMP WITH TIME ZONE representing the start of the current transaction. [Date/time functions]

NTH_VALUE [analytic]

Returns the value evaluated at the row that is the nth row of the window (counting from 1). [Analytic functions]

NTILE [analytic]
Equally divides an ordered data set (partition) into a {value} number of subsets within a , where the subsets are numbered 1 through the value in parameter constant-value. [Analytic functions]

NULLIF
Compares two expressions. [NULL-handling functions]

NULLIFZERO
Evaluates to NULL if the value in the column is 0. [NULL-handling functions]

NVL
Returns the value of the first non-null expression in the list. [NULL-handling functions]

NVL2
Takes three arguments. [NULL-handling functions]

O

OCTET_LENGTH
Takes one argument as an input and returns the string length in octets for all string types. [String functions]

ONE_HOT_ENCODER_FIT
Generates a sorted list of each of the category levels for each feature to be encoded, and stores the model. [Data preparation]

OVERLAPS
Evaluates two time periods and returns true when they overlap, false otherwise. [Date/time functions]

OVERLAY
Replaces part of a string with another string and returns the new string value as a VARCHAR. [String functions]

OVERLAYB
Replaces part of a string with another string and returns the new string as an octet value. [String functions]

P

PARTITION_PROJECTION
Splits containers for a specified projection. [Partition functions]

PARTITION_TABLE
Invokes the to reorganize ROS storage containers as needed to conform with the current partitioning policy. [Partition functions]

PATTERN_ID
Returns an integer value that is a partition-wide unique identifier for the instance of the pattern that matched. [MATCH clause functions]

PCA
Computes principal components from the input table/view. [Data preparation]

PERCENT_RANK [analytic]
Calculates the relative rank of a row for a given row in a group within a by dividing that row's rank less 1 by the number of rows in the partition, also less 1. [Analytic functions]

PERCENTILE_CONT [analytic]
An inverse distribution function where, for each row, PERCENTILE_CONT returns the value that would fall into the specified percentile among a set of values in each partition within a. [Analytic functions]

PERCENTILE_DISC [analytic]
An inverse distribution function where, for each row, PERCENTILE_DISC returns the value that would fall into the specified percentile among a set of values in each partition within a. [Analytic functions]

PI
Returns the constant pi (P), the ratio of any circle's circumference to its diameter in Euclidean geometry The return type is DOUBLE PRECISION. [Mathematical functions]

POISSON_REG
Executes Poisson regression on an input relation, and returns a Poisson regression model. [Machine learning algorithms]

POSITION
Returns an INTEGER value representing the character location of a specified substring with a string (counting from one). [String functions]

POSITIONB
Returns an INTEGER value representing the octet location of a specified substring with a string (counting from one). [String functions]

POWER
Returns a DOUBLE PRECISION value representing one number raised to the power of another number. [Mathematical functions]

PRC
Returns a table that displays the points on a receiver precision recall (PR) curve. [Model evaluation]

PREDICT_ARIMA
Applies an autoregressive integrated moving average (ARIMA) model to an input relation or makes predictions using the in-sample data. [Transformation functions]

PREDICT_AUTOREGRESSOR
Applies an autoregressor (AR) model to an input relation. [Transformation functions]

PREDICT_LINEAR_REG
Applies a linear regression model on an input relation and returns the predicted value as a FLOAT. [Transformation functions]

PREDICT_LOGISTIC_REG
Applies a logistic regression model on an input relation. [Transformation functions]

PREDICT_MOVING_AVERAGE
Applies a moving-average (MA) model, created by MOVING_AVERAGE, to an input relation. [Transformation functions]

PREDICT_NAIVE_BAYES
Applies a Naive Bayes model on an input relation. [Transformation functions]

PREDICT_NAIVE_BAYES_CLASSES
Applies a Naive Bayes model on an input relation and returns the probabilities of classes:. [Transformation functions]

PREDICT_PMML
Applies an imported PMML model on an input relation. [Transformation functions]

PREDICT_POISSON_REG
Applies a Poisson regression model on an input relation and returns the predicted value as a FLOAT. [Transformation functions]

PREDICT_RF_CLASSIFIER
Applies a random forest model on an input relation. [Transformation functions]

PREDICT_RF_CLASSIFIER_CLASSES
Applies a random forest model on an input relation and returns the probabilities of classes:. [Transformation functions]

PREDICT_RF_REGRESSOR
Applies a random forest model on an input relation, and returns with a FLOAT data type that specifies the predicted value of the random forest model—the average of the prediction of the trees in the forest. [Transformation functions]

PREDICT_SVM_CLASSIFIER
Uses an SVM model to predict class labels for samples in an input relation, and returns the predicted value as a FLOAT data type. [Transformation functions]

PREDICT_SVM_REGRESSOR
Uses an SVM model to perform regression on samples in an input relation, and returns the predicted value as a FLOAT data type. [Transformation functions]

PREDICT_TENSORFLOW
Applies a TensorFlow model on an input relation, and returns with the result expected for the encoded model type. [Transformation functions]

PREDICT_XGB_CLASSIFIER
Applies an XGBoost classifier model on an input relation. [Transformation functions]

PREDICT_XGB_CLASSIFIER_CLASSES
Applies an XGBoost classifier model on an input relation and returns the probabilities of classes:. [Transformation functions]

PREDICT_XGB_REGRESSOR
Applies an XGBoost regressor model on an input relation. [Transformation functions]

PROMOTE_SUBCLUSTER_TO_PRIMARY
Converts a secondary subcluster to a. [Eon Mode functions]

PURGE
Permanently removes delete vectors from ROS storage containers so disk space can be reused. [Database functions]

PURGE_PARTITION
Purges a table partition of deleted rows. [Partition functions]

PURGE_PROJECTION
PURGE_PROJECTION can use significant disk space while purging the data. [Projection functions]

PURGE_TABLE
This function was formerly named PURGE_TABLE_PROJECTIONS(). [Table functions]

Q

QUARTER
Returns calendar quarter of the specified date as an integer, where the January-March quarter is 1. [Date/time functions]

QUOTE_IDENT
Returns the specified string argument in the format required to use the string as an identifier in an SQL statement. [String functions]

QUOTE_LITERAL
Returns the given string suitably quoted for use as a string literal in a SQL statement string. [String functions]

QUOTE_NULLABLE
Returns the given string suitably quoted for use as a string literal in an SQL statement string; or if the argument is null, returns the unquoted string NULL. [String functions]

R

RADIANS
Returns a DOUBLE PRECISION value representing an angle expressed in radians. [Mathematical functions]

RANDOM
Returns a uniformly-distributed random DOUBLE PRECISION value x, where 0 <= x < 1. [Mathematical functions]

RANDOMINT
Accepts and returns an integer between 0 and the integer argument expression-1. [Mathematical functions]

RANDOMINT_CRYPTO
Accepts and returns an INTEGER value from a set of values between 0 and the specified function argument -1. [Mathematical functions]

RANK [analytic]
Within each window partition, ranks all rows in the query results set according to the order specified by the window's ORDER BY clause. [Analytic functions]

READ_CONFIG_FILE
Reads and returns the key-value pairs of all the parameters of a given tokenizer. [Text search functions]

READ_TREE
Reads the contents of trees within the random forest or XGBoost model. [Model evaluation]

REALIGN_CONTROL_NODES
Causes Vertica to re-evaluate which nodes in the cluster or subcluster are and which nodes are assigned to them as dependents when large cluster is enabled. [Cluster functions]

REBALANCE_CLUSTER
Rebalances the database cluster synchronously as a session foreground task. [Cluster functions]

REBALANCE_SHARDS
Rebalances shard assignments in a subcluster or across the entire cluster in Eon Mode. [Eon Mode functions]

REBALANCE_TABLE
Synchronously rebalances data in the specified table. [Table functions]

REENABLE_DUPLICATE_KEY_ERROR
Restores the default behavior of error reporting by reversing the effects of DISABLE_DUPLICATE_KEY_ERROR. [Table functions]

REFRESH
Synchronously refreshes one or more table projections in the foreground, and updates the PROJECTION_REFRESHES system table. [Projection functions]

REFRESH_COLUMNS
Refreshes table columns that are defined with the constraint SET USING or DEFAULT USING. [Projection functions]

REGEXP_COUNT
Returns the number times a regular expression matches a string. [Regular expression functions]

REGEXP_ILIKE
Returns true if the string contains a match for the regular expression. [Regular expression functions]

REGEXP_INSTR
Returns the starting or ending position in a string where a regular expression matches. [Regular expression functions]

REGEXP_LIKE
Returns true if the string matches the regular expression. [Regular expression functions]

REGEXP_NOT_ILIKE
Returns true if the string does not match the case-insensitive regular expression. [Regular expression functions]

REGEXP_NOT_LIKE
Returns true if the string does not contain a match for the regular expression. [Regular expression functions]

REGEXP_REPLACE
Replaces all occurrences of a substring that match a regular expression with another substring. [Regular expression functions]

REGEXP_SUBSTR
Returns the substring that matches a regular expression within a string. [Regular expression functions]

REGISTER_MODEL
Registers a trained model and adds it to Model Versioning environment with a status of 'under_review'. [Model management]

REGR_AVGX
Returns the DOUBLE PRECISION average of the independent expression in an expression pair. [Aggregate functions]

REGR_AVGY
Returns the DOUBLE PRECISION average of the dependent expression in an expression pair. [Aggregate functions]

REGR_COUNT
Returns the count of all rows in an expression pair. [Aggregate functions]

REGR_INTERCEPT
Returns the y-intercept of the regression line determined by a set of expression pairs. [Aggregate functions]

REGR_R2
Returns the square of the correlation coefficient of a set of expression pairs. [Aggregate functions]

REGR_SLOPE
Returns the slope of the regression line, determined by a set of expression pairs. [Aggregate functions]

REGR_SXX
Returns the sum of squares of the difference between the independent expression (expression2) and its average. [Aggregate functions]

REGR_SXY
Returns the sum of products of the difference between the dependent expression (expression1) and its average and the difference between the independent expression (expression2) and its average. [Aggregate functions]

REGR_SYY
Returns the sum of squares of the difference between the dependent expression (expression1) and its average. [Aggregate functions]

RELEASE_ALL_JVM_MEMORY
Forces all sessions to release the memory consumed by their Java Virtual Machines (JVM). [Session functions]

RELEASE_JVM_MEMORY
Terminates a Java Virtual Machine (JVM), making available the memory the JVM was using. [Session functions]

RELEASE_SYSTEM_TABLES_ACCESS
Enables non-superuser access to all system tables. [Privileges and access functions]

RELOAD_ADMINTOOLS_CONF
Updates the admintools.conf on each UP node in the cluster. [Catalog functions]

RELOAD_SPREAD
Updates cluster changes to the catalog's Spread configuration file. [Cluster functions]

REPEAT
Replicates a string the specified number of times and concatenates the replicated values as a single string. [String functions]

REPLACE
Replaces all occurrences of characters in a string with another set of characters. [String functions]

RESERVE_SESSION_RESOURCE
Reserves memory resources from the general resource pool for the exclusive use of the Vertica backup and restore process. [Session functions]

RESET_LOAD_BALANCE_POLICY
Resets the counter each host in the cluster maintains, to track which host it will refer a client to when the native connection load balancing scheme is set to ROUNDROBIN. [Client connection functions]

RESET_SESSION
Applies your default connection string configuration settings to your current session. [Session functions]

RESHARD_DATABASE
Changes the number of shards in a database. [Eon Mode functions]

RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW
Restores the keys table and the view. [Flex data functions]

RESTORE_LOCATION
Restores a storage location that was previously retired with RETIRE_LOCATION. [Storage functions]

RESTRICT_SYSTEM_TABLES_ACCESS
Checks system table SYSTEM_TABLES to determine which system tables non-superusers can access. [Privileges and access functions]

RETIRE_LOCATION
Deactivates the specified storage location. [Storage functions]

REVERSE_NORMALIZE
Reverses the normalization transformation on normalized data, thereby de-normalizing the normalized data. [Transformation functions]

RF_CLASSIFIER
Trains a random forest model for classification on an input relation. [Machine learning algorithms]

RF_PREDICTOR_IMPORTANCE
Measures the importance of the predictors in a random forest model using the Mean Decrease Impurity (MDI) approach. [Model evaluation]

RF_REGRESSOR
Trains a random forest model for regression on an input relation. [Machine learning algorithms]

RIGHT
Returns the specified characters from the right side of a string. [String functions]

ROC
Returns a table that displays the points on a receiver operating characteristic curve. [Model evaluation]

ROUND
Rounds the specified date or time. [Date/time functions]

ROUND
Rounds a value to a specified number of decimal places, retaining the original precision and scale. [Mathematical functions]

ROW_NUMBER [analytic]
Assigns a sequence of unique numbers to each row in a partition, starting with 1. [Analytic functions]

RPAD
Returns a VARCHAR value representing a string of a specific length filled on the right with specific characters. [String functions]

RSQUARED
Returns a table with the R-squared value of the predictions in a regression model. [Model evaluation]

RTRIM
Returns a VARCHAR value representing a string with trailing blanks removed from the right side (end). [String functions]

RUN_INDEX_TOOL
Runs the Index tool on a Vertica database to perform one of these tasks:. [Database functions]

S

SANDBOX_SUBCLUSTER
Creates a sandbox for a secondary subcluster. [Eon Mode functions]

SAVE_PLANS
Creates optimizer-generated directed queries from the most frequently executed queries, up to the maximum specified. [Directed queries functions]

SECOND
Returns the seconds portion of the specified date as an integer. [Date/time functions]

SECURITY_CONFIG_CHECK
Returns the status of various security-related parameters. [Database functions]

SESSION_USER
Returns a VARCHAR containing the name of the user who initiated the current database session. [System information functions]

SET_AHM_EPOCH
Sets the (AHM) to the specified epoch. [Epoch functions]

SET_AHM_TIME
Sets the (AHM) to the epoch corresponding to the specified time on the initiator node. [Epoch functions]

SET_AUDIT_TIME
Sets the time that Vertica performs automatic database size audit to determine if the size of the database is compliant with the raw data allowance in your Vertica license. [License functions]

SET_CLIENT_LABEL

Assigns a label to a client connection for the current session. [Client connection functions]

SET_CONFIG_PARAMETER
Sets or clears a configuration parameter at the specified level. [Database functions]

SET_CONTROL_SET_SIZE
Sets the number of that participate in the spread service when large cluster is enabled. [Cluster functions]

SET_DATA_COLLECTOR_NOTIFY_POLICY
Creates/enables notification policies for a component. [Notifier functions]

SET_DATA_COLLECTOR_POLICY
Updates the following retention policy properties for the specified component:. [Data collector functions]

SET_DATA_COLLECTOR_TIME_POLICY
Updates the retention policy property INTERVAL_TIME for the specified component. [Data collector functions]

SET_DEPOT_PIN_POLICY_PARTITION
Pins the specified partitions of a table or projection to a subcluster depot, or all database depots, to reduce exposure to depot eviction. [Eon Mode functions]

SET_DEPOT_PIN_POLICY_PROJECTION
Pins a projection to a subcluster depot, or all database depots, to reduce its exposure to depot eviction. [Eon Mode functions]

SET_DEPOT_PIN_POLICY_TABLE
Pins a table to a subcluster depot, or all database depots, to reduce its exposure to depot eviction. [Eon Mode functions]

SET_LOAD_BALANCE_POLICY
Sets how native connection load balancing chooses a host to handle a client connection. [Client connection functions]

SET_LOCATION_PERFORMANCE
Sets disk performance for a storage location. [Storage functions]

SET_OBJECT_STORAGE_POLICY
Creates or changes the storage policy of a database object by assigning it a labeled storage location. [Storage functions]

SET_SCALING_FACTOR
Sets the scaling factor that determines the number of storage containers used when rebalancing the database and when using local data segmentation is enabled. [Cluster functions]

SET_SPREAD_OPTION
Changes daemon settings. [Database functions]

SET_TOKENIZER_PARAMETER
Configures the tokenizer parameters. [Text search functions]

SET_UNION
Returns a SET containing all elements of two input sets. [Collection functions]

SHA1
Uses the US Secure Hash Algorithm 1 to calculate the SHA1 hash of string. [String functions]

SHA224
Uses the US Secure Hash Algorithm 2 to calculate the SHA224 hash of string. [String functions]

SHA256
Uses the US Secure Hash Algorithm 2 to calculate the SHA256 hash of string. [String functions]

SHA384
Uses the US Secure Hash Algorithm 2 to calculate the SHA384 hash of string. [String functions]

SHA512
Uses the US Secure Hash Algorithm 2 to calculate the SHA512 hash of string. [String functions]

SHOW_PROFILING_CONFIG
Shows whether profiling is enabled. [Profiling functions]

SHUTDOWN
Shuts down a Vertica database. [Database functions]

SHUTDOWN_SUBCLUSTER
Shuts down a subcluster. [Eon Mode functions]

SHUTDOWN_WITH_DRAIN
Gracefully shuts down a subcluster or subclusters. [Eon Mode functions]

SIGN

Returns a DOUBLE PRECISION value of -1, 0, or 1 representing the arithmetic sign of the argument. [Mathematical functions]

SIN

Returns a DOUBLE PRECISION value that represents the trigonometric sine of the passed parameter. [Mathematical functions]

SINH

Returns a DOUBLE PRECISION value that represents the hyperbolic sine of the passed parameter. [Mathematical functions]

SLEEP

Waits a specified number of seconds before executing another statement or command. [Workload management functions]

SOUNDEX

Takes a VARCHAR argument and returns a four-character code that enables comparison of that argument with other SOUNDEX-encoded strings that are spelled differently in English, but are phonetically similar. [String functions]

SOUNDEX_MATCHES

Compares the Soundex encodings of two strings. [String functions]

SPACE

Returns the specified number of blank spaces, typically for insertion into a character string. [String functions]

SPLIT_PART

Splits string on the delimiter and returns the string at the location of the beginning of the specified field (counting from 1). [String functions]

SPLIT_PARTB

Divides an input string on a delimiter character and returns the Nth segment, counting from 1. [String functions]

SQRT

Returns a DOUBLE PRECISION value representing the arithmetic square root of the argument. [Mathematical functions]

ST_Area

Calculates the area of a spatial object. [Geospatial functions]

ST_AsBinary

Creates the Well-Known Binary (WKB) representation of a spatial object. [Geospatial functions]

ST_AsText

Creates the Well-Known Text (WKT) representation of a spatial object. [Geospatial functions]

ST_Boundary

Calculates the boundary of the specified GEOMETRY object. [Geospatial functions]

ST_Buffer

Creates a GEOMETRY object greater than or equal to a specified distance from the boundary of a spatial object. [Geospatial functions]

ST_Centroid

Calculates the geometric center—the centroid—of a spatial object. [Geospatial functions]

ST_Contains

Determines if a spatial object is entirely inside another spatial object without existing only on its boundary. [Geospatial functions]

ST_ConvexHull

Calculates the smallest convex GEOMETRY object that contains a GEOMETRY object. [Geospatial functions]

ST_Crosses

Determines if one GEOMETRY object spatially crosses another GEOMETRY object. [Geospatial functions]

ST_Difference

Calculates the part of a spatial object that does not intersect with another spatial object. [Geospatial functions]

ST_Disjoint

Determines if two GEOMETRY objects do not intersect or touch. [Geospatial functions]

ST_Distance

Calculates the shortest distance between two spatial objects. [Geospatial functions]

ST_Envelope

Calculates the minimum bounding rectangle that contains the specified GEOMETRY object. [Geospatial functions]

ST_Equals

Determines if two spatial objects are spatially equivalent. [Geospatial functions]

ST_GeographyFromText

Converts a Well-Known Text (WKT) string into its corresponding GEOGRAPHY object. [Geospatial functions]

ST_GeographyFromWKB

Converts a Well-Known Binary (WKB) value into its corresponding GEOGRAPHY object. [Geospatial functions]

ST_GeoHash
Returns a GeoHash in the shape of the specified geometry. [Geospatial functions]

ST_GeometryN
Returns the n geometry within a geometry object. [Geospatial functions]

ST_GeometryType
Determines the class of a spatial object. [Geospatial functions]

ST_GeomFromGeoHash
Returns a polygon in the shape of the specified GeoHash. [Geospatial functions]

ST_GeomFromGeoJSON
Converts the geometry portion of a GeoJSON record in the standard format into a GEOMETRY object. [Geospatial functions]

ST_GeomFromText
Converts a Well-Known Text (WKT) string into its corresponding GEOMETRY object. [Geospatial functions]

ST_GeomFromWKB
Converts the Well-Known Binary (WKB) value to its corresponding GEOMETRY object. [Geospatial functions]

ST_Intersection
Calculates the set of points shared by two GEOMETRY objects. [Geospatial functions]

ST_Intersects
Determines if two GEOMETRY or GEOGRAPHY objects intersect or touch at a single point. [Geospatial functions]

ST_IsEmpty
Determines if a spatial object represents the empty set. [Geospatial functions]

ST_IsSimple
Determines if a spatial object does not intersect itself or touch its own boundary at any point. [Geospatial functions]

ST_IsValid
Determines if a spatial object is well formed or valid. [Geospatial functions]

ST_Length
Calculates the length of a spatial object. [Geospatial functions]

ST_NumGeometries
Returns the number of geometries contained within a spatial object. [Geospatial functions]

ST_NumPoints
Calculates the number of vertices of a spatial object, empty objects return NULL. [Geospatial functions]

ST_Overlaps
Determines if a GEOMETRY object shares space with another GEOMETRY object, but is not completely contained within that object. [Geospatial functions]

ST_PointFromGeoHash
Returns the center point of the specified GeoHash. [Geospatial functions]

ST_PointN
Finds the n point of a spatial object. [Geospatial functions]

ST_Relate
Determines if a given GEOMETRY object is spatially related to another GEOMETRY object, based on the specified DE-9IM pattern matrix string. [Geospatial functions]

ST_SRID
Identifies the spatial reference system identifier (SRID) stored with a spatial object. [Geospatial functions]

ST_SymDifference
Calculates all the points in two GEOMETRY objects except for the points they have in common, but including the boundaries of both objects. [Geospatial functions]

ST_Touches
Determines if two GEOMETRY objects touch at a single point or along a boundary, but do not have interiors that intersect. [Geospatial functions]

ST_Transform
Returns a new GEOMETRY with its coordinates converted to the spatial reference system identifier (SRID) used by the srid argument. [Geospatial functions]

ST_Union
Calculates the union of all points in two spatial objects. [Geospatial functions]

ST_Within
If spatial object g1 is completely inside of spatial object g2, then ST_Within returns true. [Geospatial functions]

ST_X
Determines the x- coordinate for a GEOMETRY point or the longitude value for a GEOGRAPHY point. [Geospatial functions]

ST_XMax
Returns the maximum x-coordinate of the minimum bounding rectangle of the GEOMETRY or GEOGRAPHY object. [Geospatial functions]

ST_XMin
Returns the minimum x-coordinate of the minimum bounding rectangle of the GEOMETRY or GEOGRAPHY object. [Geospatial functions]

ST_Y
Determines the y-coordinate for a GEOMETRY point or the latitude value for a GEOGRAPHY point. [Geospatial functions]

ST_YMax
Returns the maximum y-coordinate of the minimum bounding rectangle of the GEOMETRY or GEOGRAPHY object. [Geospatial functions]

ST_YMin
Returns the minimum y-coordinate of the minimum bounding rectangle of the GEOMETRY or GEOGRAPHY object. [Geospatial functions]

START_DRAIN_SUBCLUSTER
Drains a subcluster or subclusters. [Eon Mode functions]

START_REAPING_FILES
Starts the disk file deletion in the background as an asynchronous function. [Eon Mode functions]

START_REBALANCE_CLUSTER
Asynchronously rebalances the database cluster as a background task. [Cluster functions]

START_REFRESH
Refreshes projections in the current schema with the latest data of their respective. [Projection functions]

STATEMENT_TIMESTAMP
Similar to TRANSACTION_TIMESTAMP, returns a value of type TIMESTAMP WITH TIME ZONE that represents the start of the current statement. [Date/time functions]

STDDEV [aggregate]
Evaluates the statistical sample standard deviation for each member of the group. [Aggregate functions]

STDDEV [analytic]
Computes the statistical sample standard deviation of the current row with respect to the group within a. [Analytic functions]

STDDEV_POP [aggregate]
Evaluates the statistical population standard deviation for each member of the group. [Aggregate functions]

STDDEV_POP [analytic]
Evaluates the statistical population standard deviation for each member of the group. [Analytic functions]

STDDEV_SAMP [aggregate]
Evaluates the statistical sample standard deviation for each member of the group. [Aggregate functions]

STDDEV_SAMP [analytic]
Computes the statistical sample standard deviation of the current row with respect to the group within a. [Analytic functions]

STRING_TO_ARRAY
Splits a string containing array values and returns a native one-dimensional array. [Collection functions]

STRPOS
Returns an INTEGER value that represents the location of a specified substring within a string (counting from one). [String functions]

STRPOSB
Returns an INTEGER value representing the location of a specified substring within a string, counting from one, where each octet in the string is counted (as opposed to characters). [String functions]

STV_AsGeoJSON
Returns the geometry or geography argument as a Geometry Javascript Object Notation (GeoJSON) object. [Geospatial functions]

STV_Create_Index
Creates a spatial index on a set of polygons to speed up spatial intersection with a set of points. [Geospatial functions]

STV_Describe_Index
Retrieves information about an index that contains a set of polygons. [Geospatial functions]

STV_Drop_Index
Deletes a spatial index. [Geospatial functions]

STV_DWithin
Determines if the shortest distance from the boundary of one spatial object to the boundary of another object is within a specified distance. [Geospatial functions]

STV_Export2Shapefile
Exports GEOGRAPHY or GEOMETRY data from a database table or a subquery to a shapefile. [Geospatial functions]

STV_Extent
Returns a bounding box containing all of the input data. [Geospatial functions]

STV_ForceLHR
Alters the order of the vertices of a spatial object to follow the left-hand-rule. [Geospatial functions]

STV_Geography
Casts a GEOMETRY object into a GEOGRAPHY object. [Geospatial functions]

STV_GeographyPoint
Returns a GEOGRAPHY point based on the input values. [Geospatial functions]

STV_Geometry
Casts a GEOGRAPHY object into a GEOMETRY object. [Geospatial functions]

STV_GeometryPoint
Returns a GEOMETRY point, based on the input values. [Geospatial functions]

STV_GetExportShapefileDirectory
Returns the path of the export directory. [Geospatial functions]

STV_Intersect scalar function
Spatially intersects a point or points with a set of polygons. [Geospatial functions]

STV_Intersect transform function
Spatially intersects points and polygons. [Geospatial functions]

STV_IsValidReason
Determines if a spatial object is well formed or valid. [Geospatial functions]

STV_LineStringPoint
Retrieves the vertices of a linestring or multilinestring. [Geospatial functions]

STV_MemSize
Returns the length of the spatial object in bytes as an INTEGER. [Geospatial functions]

STV_NN
Calculates the distance of spatial objects from a reference object and returns (object, distance) pairs in ascending order by distance from the reference object. [Geospatial functions]

STV_PolygonPoint
Retrieves the vertices of a polygon as individual points. [Geospatial functions]

STV_Refresh_Index
Appends newly added or updated polygons and removes deleted polygons from an existing spatial index. [Geospatial functions]

STV_Rename_Index
Renames a spatial index. [Geospatial functions]

STV_Reverse
Reverses the order of the vertices of a spatial object. [Geospatial functions]

STV_SetExportShapefileDirectory
Specifies the directory to export GEOMETRY or GEOGRAPHY data to a shapefile. [Geospatial functions]

STV_ShpCreateTable
Returns a CREATE TABLE statement with the columns and types of the attributes found in the specified shapefile. [Geospatial functions]

STV_ShpSource and STV_ShpParser
These two functions work with COPY to parse and load geometries and attributes from a shapefile into a Vertica table, and convert them to the appropriate GEOMETRY data type. [Geospatial functions]

SUBSTR
Returns VARCHAR or VARBINARY value representing a substring of a specified string. [String functions]

SUBSTRB
Returns an octet value representing the substring of a specified string. [String functions]

SUBSTRING
Returns a value representing a substring of the specified string at the given position, given a value, a position, and an optional length. [String

functions]

SUM [aggregate]
Computes the sum of an expression over a group of rows. [Aggregate functions]

SUM [analytic]
Computes the sum of an expression over a group of rows within a. [Analytic functions]

SUM_FLOAT [aggregate]
Computes the sum of an expression over a group of rows and returns a DOUBLE PRECISION value. [Aggregate functions]

SUMMARIZE_CATCOL
Returns a statistical summary of categorical data input, in three columns:. [Data preparation]

SUMMARIZE_NUMCOL
Returns a statistical summary of columns in a Vertica table:. [Data preparation]

SVD
Computes singular values (the diagonal of the S matrix) and right singular vectors (the V matrix) of an SVD decomposition of the input relation. [Data preparation]

SVM_CLASSIFIER
Trains the SVM model on an input relation. [Machine learning algorithms]

SVM_REGRESSOR
Trains the SVM model on an input relation. [Machine learning algorithms]

SWAP_PARTITIONS_BETWEEN_TABLES
Swaps partitions between two tables. [Partition functions]

SYNC_CATALOG
Synchronizes the catalog to communal storage to enable reviving the current catalog version in the case of an imminent crash. [Eon Mode functions]

SYNC_WITH_HCATALOG_SCHEMA
Copies the structure of a Hive database schema available through the HCatalog Connector to a Vertica schema. [Hadoop functions]

SYNC_WITH_HCATALOG_SCHEMA_TABLE
Copies the structure of a single table in a Hive database schema available through the HCatalog Connector to a Vertica table. [Hadoop functions]

SYSDATE
Returns the current statement's start date and time as a TIMESTAMP value. [Date/time functions]

T

TAN
Returns a DOUBLE PRECISION value that represents the trigonometric tangent of the passed parameter. [Mathematical functions]

TANH
Returns a DOUBLE PRECISION value that represents the hyperbolic tangent of the passed parameter. [Mathematical functions]

Template patterns for date/time formatting
In an output template string (for TO_CHAR), certain patterns are recognized and replaced with appropriately formatted data from the value to format. [Formatting functions]

Template patterns for numeric formatting
A sign formatted using SG, PL, or MI is not anchored to the number. [Formatting functions]

THROW_ERROR
Returns a user-defined error message. [Error-handling functions]

TIME_SLICE
Aggregates data by different fixed-time intervals and returns a rounded-up input TIMESTAMP value to a value that corresponds with the start or end of the time slice interval. [Date/time functions]

TIMEOFDAY
Returns the wall-clock time as a text string. [Date/time functions]

TIMESTAMP_ROUND
Rounds the specified TIMESTAMP. [Date/time functions]

TIMESTAMP_TRUNC
Truncates the specified TIMESTAMP. [Date/time functions]

TIMESTAMPADD
Adds the specified number of intervals to a TIMESTAMP or TIMESTAMPTZ value and returns a result of the same data type. [Date/time functions

]

TIMESTAMPDIFF
Returns the time span between two TIMESTAMP or TIMESTAMPTZ values, in the intervals specified. [Date/time functions]

TO_BITSTRING
This topic is shared in two locations: Formatting Functions and String Functions. [Formatting functions]

TO_CHAR
Converts date/time and numeric values into text strings. [Formatting functions]

TO_DATE
This topic shared in two places: Date/Time functions and Formatting Functions. [Formatting functions]

TO_HEX
This topic is shared in two locations: Formatting Functions and String Functions. [Formatting functions]

TO_JSON
Returns the JSON representation of a complex-type argument, including mixed and nested complex types. [Collection functions]

TO_NUMBER
Converts a string value to DOUBLE PRECISION. [Formatting functions]

TO_TIMESTAMP
Converts a string value or a UNIX/POSIX epoch value to a TIMESTAMP type. [Formatting functions]

TO_TIMESTAMP_TZ
Converts a string value or a UNIX/POSIX epoch value to a TIMESTAMP WITH TIME ZONE type. [Formatting functions]

TRANSACTION_TIMESTAMP
Returns a value of type TIME WITH TIMEZONE that represents the start of the current transaction. [Date/time functions]

TRANSLATE
Replaces individual characters in string_to_replace with other characters. [String functions]

TRIM
Combines the BTRIM, LTRIM, and RTRIM functions into a single function. [String functions]

TRUNC
Truncates the specified date or time. [Date/time functions]

TRUNC
Returns the expression value fully truncated (toward zero). [Mathematical functions]

TS_FIRST_VALUE
Processes the data that belongs to each time slice. [Aggregate functions]

TS_LAST_VALUE
Processes the data that belongs to each time slice. [Aggregate functions]

U

UNNEST
Expands the elements of one or more collection columns (ARRAY or SET) into individual rows. [Collection functions]

UNSANDBOX_SUBCLUSTER
Removes a subcluster from a sandbox. [Eon Mode functions]

UPGRADE_MODEL
Upgrades a model from a previous Vertica version. [Model management]

UPPER
Returns a VARCHAR value containing the argument converted to uppercase letters. [String functions]

UPPERB
Returns a character string with each ASCII character converted to uppercase. [String functions]

URI_PERCENT_DECODE
Decodes a percent-encoded Universal Resource Identifier (URI) according to the RFC 3986 standard. [URI functions]

URI_PERCENT_ENCODE
Encodes a Universal Resource Identifier (URI) according to the RFC 3986 standard for percent encoding. [URI functions]

USER
Returns a VARCHAR containing the name of the user who initiated the current database connection. [System information functions]

USERNAME
Returns a VARCHAR containing the name of the user who initiated the current database connection. [System information functions]

UUID_GENERATE
Returns a new universally unique identifier (UUID) that is generated based on high-quality randomness from /dev/urandom. [UUID functions]

V

V6_ATON
Converts a string containing a colon-delimited IPv6 network address into a VARBINARY string. [IP address functions]

V6_NTOA
Converts an IPv6 address represented as varbinary to a character string. [IP address functions]

V6_SUBNETA
Returns a VARCHAR containing a subnet address in CIDR (Classless Inter-Domain Routing) format from a binary or alphanumeric IPv6 address. [IP address functions]

V6_SUBNETN
Calculates a subnet address in CIDR (Classless Inter-Domain Routing) format from a varbinary or alphanumeric IPv6 address. [IP address functions]

V6_TYPE
Returns an INTEGER value that classifies the type of the network address passed to it as defined in IETF RFC 4291 section 2.4. [IP address functions]

VALIDATE_STATISTICS
Validates statistics in the XML file generated by EXPORT_STATISTICS. [Statistics management functions]

VAR_POP [aggregate]
Evaluates the population variance for each member of the group. [Aggregate functions]

VAR_POP [analytic]
Returns the statistical population variance of a non-null set of numbers (nulls are ignored) in a group within a. [Analytic functions]

VAR_SAMP [aggregate]
Evaluates the sample variance for each row of the group. [Aggregate functions]

VAR_SAMP [analytic]
Returns the sample variance of a non-NULL set of numbers (NULL values in the set are ignored) for each row of the group within a. [Analytic functions]

VARIANCE [aggregate]
Evaluates the sample variance for each row of the group. [Aggregate functions]

VARIANCE [analytic]
Returns the sample variance of a non-NULL set of numbers (NULL values in the set are ignored) for each row of the group within a. [Analytic functions]

VERIFY_HADOOP_CONF_DIR
Verifies that the Hadoop configuration that is used to access HDFS is valid on all Vertica nodes. [Hadoop functions]

VERSION
Returns a VARCHAR containing a Vertica node's version information. [System information functions]

W

WEEK
Returns the week of the year for the specified date as an integer, where the first week begins on the first Sunday on or preceding January 1. [Date/time functions]

WEEK_ISO
Returns the week of the year for the specified date as an integer, where the first week starts on Monday and contains January 4. [Date/time functions]

WIDTH_BUCKET
Constructs equiwidth histograms, in which the histogram range is divided into intervals (buckets) of identical sizes. [Mathematical functions]

WITHIN GROUP ORDER BY clause
Specifies how to sort rows that are grouped by aggregate functions, one of the following:. [Aggregate functions]

X

XGB_CLASSIFIER
Trains an XGBoost model for classification on an input relation. [Machine learning algorithms]

XGB_PREDICTOR_IMPORTANCE
Measures the importance of the predictors in an XGBoost model. [Model evaluation]

XGB_REGRESSOR
Trains an XGBoost model for regression on an input relation. [Machine learning algorithms]

Y

YEAR
Returns an integer that represents the year portion of the specified date. [Date/time functions]

YEAR_ISO
Returns an integer that represents the year portion of the specified date. [Date/time functions]

Z

ZEROIFNULL
Evaluates to 0 if the column is NULL. [NULL-handling functions]

In this section

- Aggregate functions
- Analytic functions
- Client connection functions
- Data-type-specific functions
- Database Designer functions
- Directed queries functions
- Error-handling functions
- Flex functions
- Formatting functions
- Geospatial functions
- Hadoop functions
- Machine learning functions
- Management functions
- Match and search functions
- Mathematical functions
- NULL-handling functions
- Performance analysis functions
- Stored procedure functions
- System information functions

Aggregate functions

> **Note**
> All functions in this section that have an analytic function counterpart are appended with [Aggregate] to avoid confusion between the two.

Aggregate functions summarize data over groups of rows from a query result set. The groups are specified using the GROUP BY clause. They are allowed only in the select list and in the HAVING and ORDER BY clauses of a SELECT statement (as described in Aggregate expressions).

Notes

- Except for COUNT, these functions return a null value when no rows are selected. In particular, SUM of no rows returns NULL, not zero.
- In some cases you can replace an expression that includes multiple aggregates with an single aggregate of an expression. For example SUM(x) + SUM(y) can be expressed as as SUM(x+y) (where x and y are NOT NULL).
- Vertica does not support nested aggregate functions.

You can also use some of the simple aggregate functions as analytic (window) functions. See Analytic functions for details. See also SQL analytics .

In this section

- APPROXIMATE_COUNT_DISTINCT
- APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS
- APPROXIMATE_COUNT_DISTINCT_SYNOPSIS
- APPROXIMATE_COUNT_DISTINCT_SYNOPSIS_MERGE
- APPROXIMATE_MEDIAN [aggregate]
- APPROXIMATE_PERCENTILE [aggregate]
- APPROXIMATE_QUANTILES
- ARGMAX_AGG
- ARGMIN_AGG

- AVG [aggregate]
- BIT_AND
- BIT_OR
- BIT_XOR
- BOOL_AND [aggregate]
- BOOL_OR [aggregate]
- BOOL_XOR [aggregate]
- CORR
- COUNT [aggregate]
- COVAR_POP
- COVAR_SAMP
- GROUP_ID
- GROUPING
- GROUPING_ID
- LISTAGG
- MAX [aggregate]
- MIN [aggregate]
- REGR_AVGX
- REGR_AVGY
- REGR_COUNT
- REGR_INTERCEPT
- REGR_R2
- REGR_SLOPE
- REGR_SXX
- REGR_SXY
- REGR_SYY
- STDDEV [aggregate]
- STDDEV_POP [aggregate]
- STDDEV_SAMP [aggregate]
- SUM [aggregate]
- SUM_FLOAT [aggregate]
- TS_FIRST_VALUE
- TS_LAST_VALUE
- VAR_POP [aggregate]
- VAR_SAMP [aggregate]
- VARIANCE [aggregate]
- WITHIN GROUP ORDER BY clause

APPROXIMATE_COUNT_DISTINCT

Returns the number of distinct non-NULL values in a data set.

Behavior type

Immutable

Syntax

```
APPROXIMATE_COUNT_DISTINCT ( expression[, error-tolerance ] )
```

Parameters

expression

Value to be evaluated using any data type that supports equality comparison.

error-tolerance

Numeric value that represents the desired percentage of error tolerance, distributed around the value returned by this function. The smaller the error tolerance, the closer the approximation.
You can set error-tolerance to a minimum value of 0.88. Vertica imposes no maximum restriction, but any value greater than 5 is implemented with 5% error tolerance.

If you omit this argument, Vertica uses an error tolerance of 1.25(%).

Restrictions

APPROXIMATE_COUNT_DISTINCT and DISTINCT aggregates cannot be in the same query block.

Error tolerance

APPROXIMATE_COUNT_DISTINCT(*x* , *error-tolerance*) returns a value equal to COUNT(DISTINCT *x*) , with an error that is lognormally distributed with standard deviation.

Parameter *error-tolerance* is optional. Supply this argument to specify the desired standard deviation. *error-tolerance* is defined as 2.17 standard deviations, which corresponds to a 97 percent confidence interval:

```
standard-deviation = error-tolerance / 2.17
```

For example:

- *error-tolerance* = 1
  Default setting, corresponds to a standard deviation
  97 percent of the time, APPROXIMATE_COUNT_DISTINCT(*x,5*) returns a value between:
  - COUNT(DISTINCT *x*) * 0.99
  - COUNT(DISTINCT *x*) * 1.01
- *error-tolerance* = 5
  97 percent of the time, APPROXIMATE_COUNT_DISTINCT(*x*) returns a value between:
  - COUNT(DISTINCT *x*) * 0.95
  - COUNT(DISTINCT *x*) * 1.05

A 99 percent confidence interval corresponds to 2.58 standard deviations. To set *error-tolerance* confidence level corresponding to 99 (instead of a 97) percent , multiply *error-tolerance* by 2.17 / 2.58 = 0.841 .

For example, if you specify *error-tolerance* as 5 * 0.841 = 4.2 , APPROXIMATE_COUNT_DISTINCT(*x,4.2*) returns values 99 percent of the time between:

- COUNT (DISTINCT *x*) * 0.95
- COUNT (DISTINCT *x*) * 1.05

Examples

Count the total number of distinct values in column product_key from table store.store_sales_fact :

```
=> SELECT COUNT(DISTINCT product_key) FROM store.store_sales_fact;
 COUNT
-------
 19982
(1 row)
```

Count the approximate number of distinct values in product_key with various error tolerances. The smaller the error tolerance, the closer the approximation:

```
=> SELECT APPROXIMATE_COUNT_DISTINCT(product_key,5) AS five_pct_accuracy,
   APPROXIMATE_COUNT_DISTINCT(product_key,1) AS one_pct_accuracy,
   APPROXIMATE_COUNT_DISTINCT(product_key,.88) AS point_eighteight_pct_accuracy
   FROM store.store_sales_fact;

 five_pct_accuracy | one_pct_accuracy | point_eighteight_pct_accuracy
-------------------+------------------+-------------------------------
             19431 |            19921 |                         19921
(1 row)
```

See also

Approximate count distinct functions

APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS

Calculates the number of distinct non-NULL values from the synopsis objects created by APPROXIMATE_COUNT_DISTINCT_SYNOPSIS .

Behavior type

Immutable

Syntax

```
APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS ( synopsis-obj[, error-tolerance ] )
```

Parameters

synopsis-obj

A synopsis object created by APPROXIMATE_COUNT_DISTINCT_SYNOPSIS.

error-tolerance

Numeric value that represents the desired percentage of error tolerance, distributed around the value returned by this function. The smaller the error tolerance, the closer the approximation.
You can set error-tolerance to a minimum value of 0.88. Vertica imposes no maximum restriction, but any value greater than 5 is implemented with 5% error tolerance.

If you omit this argument, Vertica uses an error tolerance of 1.25(%).

For more details, see APPROXIMATE_COUNT_DISTINCT .

Restrictions

APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS and DISTINCT aggregates cannot be in the same query block.

Examples

The following examples review and compare different ways to obtain a count of unique values in a table column:

Return an exact count of unique values in column product_key, from table store.store_sales_fact :

```
=> \timing
Timing is on.
=> SELECT COUNT(DISTINCT product_key) from store.store_sales_fact;
 count
-------
 19982
(1 row)

Time: First fetch (1 row): 553.033 ms. All rows formatted: 553.075 ms
```

Return an approximate count of unique values in column product_key :

```
=> SELECT APPROXIMATE_COUNT_DISTINCT(product_key) as unique_product_keys
   FROM store.store_sales_fact;
 unique_product_keys
---------------------
               19921
(1 row)

Time: First fetch (1 row): 394.562 ms. All rows formatted: 394.600 ms
```

Create a synopsis object that represents a set of store.store_sales_fact data with unique product_key values, store the synopsis in the new table my_summary :

```
=> CREATE TABLE my_summary AS SELECT APPROXIMATE_COUNT_DISTINCT_SYNOPSIS (product_key) syn
   FROM store.store_sales_fact;
CREATE TABLE
Time: First fetch (0 rows): 582.662 ms. All rows formatted: 582.682 ms
```

Return a count from the saved synopsis:

```
=> SELECT APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS(syn) FROM my_summary;
 ApproxCountDistinctOfSynopsis
-------------------------------
                         19921
(1 row)

Time: First fetch (1 row): 105.295 ms. All rows formatted: 105.335 ms
```

See also

Approximate count distinct functions

APPROXIMATE_COUNT_DISTINCT_SYNOPSIS

Summarizes the information of distinct non-NULL values and materializes the result set in a VARBINARY or LONG VARBINARY *synopsis* object. The calculated result is within a specified range of error tolerance. You save the synopsis object in a Vertica table for use by APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS .

Behavior type

Immutable

Syntax

```
APPROXIMATE_COUNT_DISTINCT_SYNOPSIS ( expression[, error-tolerance] )
```

Parameters

expression

Value to evaluate using any data type that supports equality comparison.

error-tolerance

Numeric value that represents the desired percentage of error tolerance, distributed around the value returned by this function. The smaller the error tolerance, the closer the approximation.
You can set error-tolerance to a minimum value of 0.88. Vertica imposes no maximum restriction, but any value greater than 5 is implemented with 5% error tolerance.

If you omit this argument, Vertica uses an error tolerance of 1.25(%).

For more details, see APPROXIMATE_COUNT_DISTINCT .

Restrictions

APPROXIMATE_COUNT_DISTINCT_SYNOPSIS and DISTINCT aggregates cannot be in the same query block.

Examples

See APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS .

See also

Approximate count distinct functions

APPROXIMATE_COUNT_DISTINCT_SYNOPSIS_MERGE

Aggregates multiple synopses into one new synopsis. This function is similar to APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS but returns one synopsis instead of the count estimate. The benefit of this function is that it speeds up final estimation when calling APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS.

For example, if you need to regularly estimate count distinct of users for a long period of time (such as several years) you can pre-accumulate synopses of days into one synopsis for a year.

Behavior type

Immutable

Syntax

```
APPROXIMATE_COUNT_DISTINCT_SYNOPSIS_MERGE ( synopsis-obj [, error-tolerance] )
```

Parameters

synopsis-obj

An expression that can be evaluated to one or more synopses. Typically a *synopsis-obj* is generated as a binary string by either the APPROXIMATE_COUNT_DISTINCT or APPROXIMATE_COUNT_DISTINCT_SYNOPSIS_MERGE function and is stored in a table column of type VARBINARY or LONG VARBINARY.

error-tolerance

Numeric value that represents the desired percentage of error tolerance, distributed around the value returned by this function. The smaller the error tolerance, the closer the approximation.
You can set error-tolerance to a minimum value of 0.88. Vertica imposes no maximum restriction, but any value greater than 5 is implemented with 5% error tolerance.

If you omit this argument, Vertica uses an error tolerance of 1.25(%).

For more details, see APPROXIMATE_COUNT_DISTINCT .

Examples

See Approximate count distinct functions .

APPROXIMATE_MEDIAN [aggregate]

Computes the approximate median of an expression over a group of rows. The function returns a FLOAT value.

APPROXIMATE_MEDIAN is an alias of APPROXIMATE_PERCENTILE [aggregate] with a parameter of 0.5.

> **Note**
> **Note** : This function is best suited for large groups of data. If you have a small group of data, use the exact MEDIAN [analytic] function.

Behavior type

Immutable

Syntax

```
APPROXIMATE_MEDIAN ( expression )
```

Parameters

expression

Any FLOAT or INTEGER data type. The function returns the approximate middle value or an interpolated value that would be the approximate middle value once the values are sorted. Null values are ignored in the calculation.

Examples

> **Tip**
> For optimal performance when using GROUP BY in your query, verify that your table is sorted on the GROUP BY column.

The following examples uses this table:

```
CREATE TABLE allsales(state VARCHAR(20), name VARCHAR(20), sales INT) ORDER BY state;
INSERT INTO allsales VALUES('MA', 'A', 60);
INSERT INTO allsales VALUES('NY', 'B', 20);
INSERT INTO allsales VALUES('NY', 'C', 15);
INSERT INTO allsales VALUES('MA', 'D', 20);
INSERT INTO allsales VALUES('MA', 'E', 50);
INSERT INTO allsales VALUES('NY', 'F', 40);
INSERT INTO allsales VALUES('MA', 'G', 10);
COMMIT;
```

Calculate the approximate median of all sales in this table:

```
=> SELECT APPROXIMATE_MEDIAN (sales) FROM allsales;
APROXIMATE_MEDIAN
--------------------
                20
(1 row)
```

Modify the query to group sales by state, and obtain the approximate median for each one:

```
=> SELECT state, APPROXIMATE_MEDIAN(sales) FROM allsales GROUP BY state;
 state | APPROXIMATE_MEDIAN
-------+--------------------
 MA    |                 35
 NY    |                 20
(2 rows)
```

See also

- MEDIAN [analytic]
- PERCENTILE_CONT [analytic]
- APPROXIMATE_PERCENTILE [aggregate]
- SQL analytics

APPROXIMATE_PERCENTILE [aggregate]

Computes the approximate percentile of an expression over a group of rows. This function returns a FLOAT value.

> **Note**
> **Note** : Use this function when many rows are aggregated into groups. If the number of aggregated rows is small, use the analytic function PERCENTILE_CONT .

Behavior type

Immutable

Syntax

```
APPROXIMATE_PERCENTILE ( column-expression USING PARAMETERS percentiles='percentile-values' )
```

Arguments

column-expression

A column of FLOAT or INTEGER data types whose percentiles will be calculated. NULL values are ignored.

Parameters

percentiles

One or more (up to 1000) comma-separated FLOAT constants ranging from 0 to 1 inclusive, specifying the percentile values to be calculated.

> **Note**
> **Note** : The deprecated parameter percentile , which takes only a single float, continues to be supported for backwards-compatibility.

Examples

> **Tip**
> For optimal performance when using GROUP BY in your query, verify that your table is sorted on the GROUP BY column.

The following examples use this table:

```
=> CREATE TABLE allsales(state VARCHAR(20), name VARCHAR(20), sales INT) ORDER BY state;
INSERT INTO allsales VALUES('MA', 'A', 60);
INSERT INTO allsales VALUES('NY', 'B', 20);
INSERT INTO allsales VALUES('NY', 'C', 15);
INSERT INTO allsales VALUES('MA', 'D', 20);
INSERT INTO allsales VALUES('MA', 'E', 50);
INSERT INTO allsales VALUES('NY', 'F', 40);
INSERT INTO allsales VALUES('MA', 'G', 10);
COMMIT;

=> SELECT * FROM allsales;
 state | name | sales
-------+------+-------
 MA    | A    |    60
 NY    | B    |    20
 NY    | C    |    15
 NY    | F    |    40
 MA    | D    |    20
 MA    | E    |    50
 MA    | G    |    10
(7 rows)
```

Calculate the approximate percentile for sales in each state:

```
=> SELECT state, APPROXIMATE_PERCENTILE(sales USING PARAMETERS percentiles='0.5') AS median
FROM allsales GROUP BY state;
 state | median
-------+--------
 MA    |     35
 NY    |     20
(2 rows)
```

Calculate multiple approximate percentiles for sales in each state:

```
=> SELECT state, APPROXIMATE_PERCENTILE(sales USING PARAMETERS percentiles='0.5,1.0')
FROM allsales GROUP BY state;
 state | APPROXIMATE_PERCENTILE
-------+--------
 MA    |     [35.0,60.0]
 NY    |     [20.0,40.0]
(2 rows)
```

Calculate multiple approximate percentiles for sales in each state and show results for each percentile in separate columns:

```
=> SELECT ps[0] as q0, ps[1] as q1, ps[2] as q2, ps[3] as q3, ps[4] as q4
FROM (SELECT APPROXIMATE_PERCENTILE(sales USING PARAMETERS percentiles='0, 0.25, 0.5, 0.75, 1')
AS ps FROM allsales GROUP BY state) as s1;
  q0  |  q1  |  q2  |  q3  |  q4
------+------+------+------+------
 10.0 | 17.5 | 35.0 | 52.5 | 60.0
 15.0 | 17.5 | 20.0 | 30.0 | 40.0
(2 rows)
```

See also

- APPROXIMATE_QUANTILES
- MEDIAN
- PERCENTILE_CONT
- SQL analytics

APPROXIMATE_QUANTILES

Computes an array of weighted, approximate percentiles of a column within some user-specified error. This algorithm is similar to APPROXIMATE_PERCENTILE [aggregate] , which instead returns a single percentile.

The performance of this function depends entirely on the specified epsilon and the size of the provided array.

The OVER clause for this function must be empty.

Behavior type

Immutable

Syntax

```
APPROXIMATE_QUANTILES ( column USING PARAMETERS [nquantiles=n], [epsilon=error] ) OVER() FROM table
```

Parameters

column

The INTEGER or FLOAT column for which to calculate the percentiles. NULL values are ignored.

n

An integer that specifies the number of desired quantiles in the returned array.

Default: 11

error

The allowed error for any returned percentile. Specifically, for an array of size N , the specified error ε (epsilon) for the φ-quantile guarantees that the rank r of the return value with respect to the rank $\lfloor \varphi N \rfloor$ of the exact value is such that:

$\lfloor (\varphi-\varepsilon)N \rfloor \leq r \leq \lfloor (\varphi+\varepsilon)N \rfloor$

For n quantiles, if the error ε is specified such that $\varepsilon > 1/n$, this function will return non-deterministic results.

Default: 0.001

table

The table containing *column* .

Examples

The following example uses this table:

```
=> CREATE TABLE allsales(state VARCHAR(20), name VARCHAR(20), sales INT) ORDER BY state;
INSERT INTO allsales VALUES('MA', 'A', 60);
INSERT INTO allsales VALUES('NY', 'B', 20);
INSERT INTO allsales VALUES('NY', 'C', 15);
INSERT INTO allsales VALUES('MA', 'D', 20);
INSERT INTO allsales VALUES('MA', 'E', 50);
INSERT INTO allsales VALUES('NY', 'F', 40);
INSERT INTO allsales VALUES('MA', 'G', 10);
COMMIT;

=> SELECT * FROM allsales;
 state | name | sales
-------+------+-------
 MA    | A    |    60
 NY    | B    |    20
 NY    | C    |    15
 NY    | F    |    40
 MA    | D    |    20
 MA    | E    |    50
 MA    | G    |    10
(7 rows)
```

This call to APPROXIMATE_QUANTILES returns a 6-element array of approximate percentiles, one for each quantile. Each quantile relates to the percentile by a factor of 100. For example, the second entry in the output indicates that 15 is the 0.2-quantile of the input column, so 15 is the 20th percentile of the input column.

```
=> SELECT APPROXIMATE_QUANTILES(sales USING PARAMETERS nquantiles=6) OVER() FROM allsales;
 Quantile | Value
----------+-------
        0 |    10
      0.2 |    15
      0.4 |    20
      0.6 |    40
      0.8 |    50
        1 |    60
(6 rows)
```

ARGMAX_AGG

Takes two arguments *target* and *arg* , where both are columns or column expressions in the queried dataset. ARGMAX_AGG finds the row with the highest non-null value in *target* and returns the value of *arg* in that row. If multiple rows contain the highest *target* value, ARGMAX_AGG returns *arg* from the first row that it finds. Use the WITHIN GROUP ORDER BY clause to control which row ARGMAX_AGG finds first.

Behavior type

Immutable if the WITHIN GROUP ORDER BY clause specifies a column or set of columns that resolves to unique values within the group; otherwise Volatile .

Syntax

```
ARGMAX_AGG ( target, arg ) [ within-group-order-by-clause ]
```

Arguments

target* , *arg

Columns in the queried dataset.

> **Note**
> The *target* argument cannot reference a spatial data type column, GEOMETRY or GEOGRAPHY.

[within-group-order-by-clause](/sql-reference/functions/aggregate-functions/within-group-order-by-clause.html)

Sorts target values within each group of rows:

```
WITHIN GROUP (ORDER BY { column-expression[ sort-qualifiers ] }[,...])
```

sort-qualifiers :

```
 { ASC | DESC [ NULLS { FIRST | LAST | AUTO } ] }
```

Use this clause to determine which row is returned when multiple rows contain the highest target value; otherwise, results are likely to vary with each iteration of the same query.

> **Tip**
> WITHIN GROUP ORDER BY can consume a large amount of memory per group. To minimize memory consumption, create projections that support GROUPBY PIPELINED .

Examples

The following example calls ARGMAX_AGG in a WITH clause to find which employees in each region are at or near retirement age. If multiple employees within each region have the same age, ARGMAX_AGG chooses the employees with the highest salary level and returns with their IDs. The primary query returns with details on the employees selected from each region:

```
=> WITH r AS (SELECT employee_region, ARGMAX_AGG(employee_age, employee_key)
    WITHIN GROUP (ORDER BY annual_salary DESC) emp_id
    FROM employee_dim GROUP BY employee_region ORDER BY employee_region)
  SELECT r.employee_region, ed.annual_salary AS highest_salary, employee_key,
    ed.employee_first_name||' '||ed.employee_last_name AS employee_name, ed.employee_age
    FROM r JOIN employee_dim ed ON r.emp_id = ed.employee_key ORDER BY ed.employee_region;
     employee_region          | highest_salary | employee_key |  employee_name   | employee_age
----------------------------------+---------------+-------------+-----------------+--------------
 East                         |        927335 |          70 | Sally Gauthier   |          65
 MidWest                      |        177716 |         869 | Rebecca McCabe   |          65
 NorthWest                    |        100300 |        7597 | Kim Jefferson    |          65
 South                        |        196454 |         275 | Alexandra Harris |          65
 SouthWest                    |        198669 |        1043 | Seth Stein       |          65
 West                         |        197203 |         681 | Seth Jones       |          65
(6 rows)
```

See also

ARGMIN_AGG

ARGMIN_AGG

Takes two arguments *target* and *arg* , where both are columns or column expressions in the queried dataset. ARGMIN_AGG finds the row with the lowest non-null value in *target* and returns the value of *arg* in that row. If multiple rows contain the lowest *target* value, ARGMIN_AGG returns *arg* from the first row that it finds. Use the WITHIN GROUP ORDER BY clause to control which row ARGMMIN_AGG finds first.

Behavior type

Immutable if the WITHIN GROUP ORDER BY clause specifies a column or set of columns that resolves to unique values within the group; otherwise Volatile .

Syntax

```
ARGMIN_AGG ( target, arg ) [ within-group-order-by-clause ]
```

Arguments

target* , *arg

Columns in the queried dataset.

> **Note**

> The *target* argument cannot reference a spatial data type column, GEOMETRY or GEOGRAPHY.

[within-group-order-by-clause](/sql-reference/functions/aggregate-functions/within-group-order-by-clause.html)

Sorts target values within each group of rows:

```
WITHIN GROUP (ORDER BY { column-expression[ sort-qualifiers ] }[,...])
```

sort-qualifiers :

```
 { ASC | DESC [ NULLS { FIRST | LAST | AUTO } ] }
```

Use this clause to determine which row is returned when multiple rows contain the lowest target value; otherwise, results are likely to vary with each iteration of the same query.

> **Tip**
> WITHIN GROUP ORDER BY can consume a large amount of memory per group. To minimize memory consumption, create projections that support GROUPBY PIPELINED .

Examples

The following example calls ARGMIN_AGG in a WITH clause to find the lowest salary among all employees in each region, and returns with the lowest-paid employee IDs. The primary query returns with the salary amounts and employee names:

```
=> WITH msr (employee_region, emp_id) AS
   (SELECT employee_region, argmin_agg(annual_salary, employee_key) lowest_paid_employee FROM employee_dim GROUP BY employee_region)
   SELECT msr.employee_region, ed.annual_salary AS lowest_salary, ed.employee_first_name||' '||ed.employee_last_name AS employee_name
    FROM msr JOIN employee_dim ed ON msr.emp_id = ed.employee_key ORDER BY annual_salary DESC;
      employee_region          | lowest_salary |  employee_name
-----------------------------------+---------------+-----------------
 NorthWest                         |         20913 | Raja Garnett
 SouthWest                         |         20750 | Seth Moore
 West                              |         20443 | Midori Taylor
 South                             |         20363 | David Bauer
 East                              |         20306 | Craig Jefferson
 MidWest                           |         20264 | Dean Vu
(6 rows)
```

See also

ARGMAX_AGG

AVG [aggregate]

Computes the average (arithmetic mean) of an expression over a group of rows. AVG always returns a DOUBLE PRECISION value.

The AVG aggregate function differs from the AVG analytic function, which computes the average of an expression over a group of rows within a window .

Behavior type

Immutable

Syntax

```
AVG ( [ ALL | DISTINCT ] expression )
```

Parameters

ALL

Invokes the aggregate function for all rows in the group (default).

DISTINCT

Invokes the aggregate function for all distinct non-null values of the expression found in the group.

expression

The value whose average is calculated over a set of rows, any expression that can have a DOUBLE PRECISION result.

Overflow handling

By default, Vertica allows silent numeric overflow when you call this function on numeric data types. For more information on this behavior and how to change it, see Numeric data type overflow with SUM, SUM_FLOAT, and AVG .

Examples

The following query returns the average income from the customer table:

```
=> SELECT AVG(annual_income) FROM customer_dimension;
    AVG
--------------
 2104270.6485
(1 row)
```

See also

- COUNT [aggregate]
- SUM [aggregate]
- Numeric data types

BIT_AND

Takes the bitwise AND of all non-null input values. If the input parameter is NULL, the return value is also NULL.

Behavior type

Immutable

Syntax

```
BIT_AND ( expression )
```

Parameters

expression

The BINARY or VARBINARY input value to evaluate. BIT_AND operates on VARBINARY types explicitly and on BINARY types implicitly through casts .

Returns

BIT_AND returns:

- The same value as the argument data type.
- 1 for each bit compared, if all bits are 1; otherwise 0.

If the columns are different lengths, the return values are treated as though they are all equal in length and are right-extended with zero bytes. For example, given a group containing hex values ff , null , and f , BIT_AND ignores the null value and extends the value f to f0 .

Examples

The example that follows uses table t with a single column of VARBINARY data type:

```
=> CREATE TABLE t ( c VARBINARY(2) );
=> INSERT INTO t values(HEX_TO_BINARY('0xFF00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFFFF'));
=> INSERT INTO t values(HEX_TO_BINARY('0xF00F'));
```

Query table t to see column c output:

```
=> SELECT TO_HEX(c) FROM t;
 TO_HEX
--------
 ff00
 ffff
 f00f
(3 rows)
```

Query table t to get the AND value for column c :

```
=> SELECT TO_HEX(BIT_AND(c)) FROM t;
 TO_HEX
--------
 f000
(1 row)
```

The function is applied pairwise to all values in the group, resulting in f000 , which is determined as follows:

1. ff00 (record 1) is compared with ffff (record 2), which results in ff00 .
2. The result from the previous comparison is compared with f00f (record 3), which results in f000 .

See also

Binary data types (BINARY and VARBINARY)

BIT_OR

Takes the bitwise OR of all non-null input values. If the input parameter is NULL, the return value is also NULL.

Behavior type

Immutable

Syntax

```
BIT_OR ( expression )
```

Parameters

expression

The BINARY or VARBINARY input value to evaluate. BIT_OR operates on VARBINARY types explicitly and on BINARY types implicitly through casts .

Returns

BIT_OR returns:

- The same value as the argument data type.
- 1 for each bit compared, if any bit is 1; otherwise 0.

If the columns are different lengths, the return values are treated as though they are all equal in length and are right-extended with zero bytes. For example, given a group containing hex values ff , null , and f , the function ignores the null value and extends the value f to f0 .

Examples

The example that follows uses table t with a single column of VARBINARY data type:

```
=> CREATE TABLE t ( c VARBINARY(2) );
=> INSERT INTO t values(HEX_TO_BINARY('0xFF00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFFFF'));
=> INSERT INTO t values(HEX_TO_BINARY('0xF00F'));
```

Query table t to see column c output:

```
=> SELECT TO_HEX(c) FROM t;
 TO_HEX
--------
 ff00
 ffff
 f00f
(3 rows)
```

Query table t to get the OR value for column c :

```
=> SELECT TO_HEX(BIT_OR(c)) FROM t;
 TO_HEX
--------
 ffff
(1 row)
```

The function is applied pairwise to all values in the group, resulting in ffff , which is determined as follows:

1. ff00 (record 1) is compared with ffff , which results in ffff .
2. The ff00 result from the previous comparison is compared with f00f (record 3), which results in ffff .

See also

Binary data types (BINARY and VARBINARY)

BIT_XOR

Takes the bitwise XOR of all non-null input values. If the input parameter is NULL , the return value is also NULL .

Behavior type

Immutable

Syntax

```
BIT_XOR ( expression )
```

Parameters

expression

The BINARY or VARBINARY input value to evaluate. BIT_XOR operates on VARBINARY types explicitly and on BINARY types implicitly through casts .

Returns

BIT_XOR returns:

- The same value as the argument data type.
- 1 for each bit compared, if there are an odd number of arguments with set bits; otherwise 0.

If the columns are different lengths, the return values are treated as though they are all equal in length and are right-extended with zero bytes. For example, given a group containing hex values ff , null , and f , the function ignores the null value and extends the value f to f0 .

Examples

First create a sample table and projections with binary columns:

The example that follows uses table t with a single column of VARBINARY data type:

```
=> CREATE TABLE t ( c VARBINARY(2) );
=> INSERT INTO t values(HEX_TO_BINARY('0xFF00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFFFF'));
=> INSERT INTO t values(HEX_TO_BINARY('0xF00F'));
```

Query table t to see column c output:

```
=> SELECT TO_HEX(c) FROM t;
 TO_HEX
--------
 ff00
 ffff
 f00f
(3 rows)
```

Query table t to get the XOR value for column c :

```
=> SELECT TO_HEX(BIT_XOR(c)) FROM t;
 TO_HEX
--------
 f0f0
(1 row)
```

See also

Binary data types (BINARY and VARBINARY)

BOOL_AND [aggregate]

Processes Boolean values and returns a Boolean value result. If all input values are true, BOOL_AND returns t . Otherwise it returns f (false).

Behavior type

Immutable

Syntax

```
BOOL_AND ( expression )
```

Parameters

expression

A Boolean data type or any non-Boolean data type that can be implicitly coerced to a Boolean data type.

Examples

The following example shows how to use aggregate functions BOOL_AND , BOOL_OR , and BOOL_XOR . The sample table mixers includes columns for models and colors.

```
=> CREATE TABLE mixers(model VARCHAR(20), colors VARCHAR(20));
CREATE TABLE
```

Insert sample data into the table. The sample adds two color fields for each model.

```
=> INSERT INTO mixers
SELECT 'beginner', 'green'
UNION ALL
SELECT 'intermediate', 'blue'
UNION ALL
SELECT 'intermediate', 'blue'
UNION ALL
SELECT 'advanced', 'green'
UNION ALL
SELECT 'advanced', 'blue'
UNION ALL
SELECT 'professional', 'blue'
UNION ALL
SELECT 'professional', 'green'
UNION ALL
SELECT 'beginner', 'green';
 OUTPUT
--------
      8
(1 row)
```

Query the table. The result shows models that have two blue (BOOL_AND), one or two blue (BOOL_OR), and specifically not more than one blue (BOOL_XOR) mixer.

```
=> SELECT model,
BOOL_AND(colors= 'blue')AS two_blue,
BOOL_OR(colors= 'blue')AS one_or_two_blue,
BOOL_XOR(colors= 'blue')AS specifically_not_more_than_one_blue
FROM mixers
GROUP BY model;

    model     | two_blue | one_or_two_blue | specifically_not_more_than_one_blue
--------------+----------+-----------------+-------------------------------------
 advanced     | f        | t               | t
 beginner     | f        | f               | f
 intermediate | t        | t               | f
 professional | f        | t               | t
(4 rows)
```

See also

- BOOL_AND [analytic]
- BOOL_OR [aggregate]
- BOOL_XOR [aggregate]
- Boolean data type

BOOL_OR [aggregate]

Processes Boolean values and returns a Boolean value result. If at least one input value is true, BOOL_OR returns t . Otherwise, it returns f .

Behavior type

Immutable

Syntax

```
BOOL_OR ( expression )
```

Parameters

expression

A Boolean data type or any non-Boolean data type that can be implicitly coerced to a Boolean data type.

Examples

The following example shows how to use aggregate functions BOOL_AND , BOOL_OR , and BOOL_XOR . The sample table mixers includes columns for models and colors.

```
=> CREATE TABLE mixers(model VARCHAR(20), colors VARCHAR(20));
CREATE TABLE
```

Insert sample data into the table. The sample adds two color fields for each model.

```
=> INSERT INTO mixers
SELECT 'beginner', 'green'
UNION ALL
SELECT 'intermediate', 'blue'
UNION ALL
SELECT 'intermediate', 'blue'
UNION ALL
SELECT 'advanced', 'green'
UNION ALL
SELECT 'advanced', 'blue'
UNION ALL
SELECT 'professional', 'blue'
UNION ALL
SELECT 'professional', 'green'
UNION ALL
SELECT 'beginner', 'green';
 OUTPUT
--------
      8
(1 row)
```

Query the table. The result shows models that have two blue (BOOL_AND), one or two blue (BOOL_OR), and specifically not more than one blue (BOOL_XOR) mixer.

```
=> SELECT model,
BOOL_AND(colors= 'blue')AS two_blue,
BOOL_OR(colors= 'blue')AS one_or_two_blue,
BOOL_XOR(colors= 'blue')AS specifically_not_more_than_one_blue
FROM mixers
GROUP BY model;

    model     | two_blue | one_or_two_blue | specifically_not_more_than_one_blue
--------------+----------+-----------------+-------------------------------------
 advanced     | f        | t               | t
 beginner     | f        | f               | f
 intermediate | t        | t               | f
 professional | f        | t               | t
(4 rows)
```

See also

- BOOL_OR [analytic]
- BOOL_AND [aggregate]
- BOOL_XOR [aggregate]
- Boolean data type

BOOL_XOR [aggregate]

Processes Boolean values and returns a Boolean value result. If specifically only one input value is true, BOOL_XOR returns t . Otherwise, it returns f .

Behavior type

Immutable

Syntax

```
BOOL_XOR ( expression )
```

Parameters

expression

A Boolean data type or any non-Boolean data type that can be implicitly coerced to a Boolean data type.

Examples

The following example shows how to use aggregate functions BOOL_AND , BOOL_OR , and BOOL_XOR . The sample table mixers includes columns for models and colors.

```
=> CREATE TABLE mixers(model VARCHAR(20), colors VARCHAR(20));
CREATE TABLE
```

Insert sample data into the table. The sample adds two color fields for each model.

```
=> INSERT INTO mixers
SELECT 'beginner', 'green'
UNION ALL
SELECT 'intermediate', 'blue'
UNION ALL
SELECT 'intermediate', 'blue'
UNION ALL
SELECT 'advanced', 'green'
UNION ALL
SELECT 'advanced', 'blue'
UNION ALL
SELECT 'professional', 'blue'
UNION ALL
SELECT 'professional', 'green'
UNION ALL
SELECT 'beginner', 'green';
 OUTPUT
--------
      8
(1 row)
```

Query the table. The result shows models that have two blue (BOOL_AND), one or two blue (BOOL_OR), and specifically not more than one blue (BOOL_XOR) mixer.

```
=> SELECT model,
BOOL_AND(colors= 'blue')AS two_blue,
BOOL_OR(colors= 'blue')AS one_or_two_blue,
BOOL_XOR(colors= 'blue')AS specifically_not_more_than_one_blue
FROM mixers
GROUP BY model;

    model     | two_blue | one_or_two_blue | specifically_not_more_than_one_blue
--------------+----------+-----------------+-------------------------------------
 advanced     | f        | t               | t
 beginner     | f        | f               | f
 intermediate | t        | t               | f
 professional | f        | t               | t
(4 rows)
```

See also

- BOOL_XOR [analytic]
- BOOL_AND [aggregate]
- BOOL_OR [aggregate]
- Boolean data type

CORR

Returns the DOUBLE PRECISION coefficient of correlation of a set of expression pairs, as per the Pearson correlation coefficient . CORR eliminates expression pairs where either expression in the pair is NULL . If no rows remain, the function returns NULL .

Syntax

```
CORR ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT CORR (Annual_salary, Employee_age) FROM employee_dimension;
       CORR
----------------------
 -0.00719153413192422
(1 row)
```

COUNT [aggregate]

Returns as a BIGINT the number of rows in each group where the expression is not NULL. If the query has no GROUP BY clause, COUNT returns the number of table rows.

The COUNT aggregate function differs from the COUNT analytic function, which returns the number over a group of rows within a window .

Behavior type

Immutable

Syntax

```
COUNT ( [ * ] [ ALL | DISTINCT ] expression )
```

Parameters

*

Specifies to count all rows in the specified table or each group.

ALL | DISTINCT

Specifies how to count rows where *expression* has a non-null value:

- ALL (default): Counts all rows where *expression* evaluates to a non-null value.
- DISTINCT : Counts all rows where *expression* evaluates to a distinct non-null value.

expression

The column or expression whose non-null values are counted.

Examples

The following query returns the number of distinct values in a column:

```
=> SELECT COUNT (DISTINCT date_key) FROM date_dimension;

 COUNT
-------
  1826
(1 row)
```

This example returns the number of distinct return values from an expression:

```
=> SELECT COUNT (DISTINCT date_key + product_key) FROM inventory_fact;

 COUNT
-------
 21560
(1 row)
```

You can create an equivalent query using the LIMIT keyword to restrict the number of rows returned:

```
=> SELECT COUNT(date_key + product_key) FROM inventory_fact GROUP BY date_key LIMIT 10;

 COUNT
-------
   173
    31
   321
   113
   286
    84
   244
   238
   145
   202
(10 rows)
```

The following query uses GROUP BY to count distinct values within groups:

```
=> SELECT product_key, COUNT (DISTINCT date_key) FROM inventory_fact
   GROUP BY product_key LIMIT 10;

 product_key | count
-------------+-------
           1 |    12
           2 |    18
           3 |    13
           4 |    17
           5 |    11
           6 |    14
           7 |    13
           8 |    17
           9 |    15
          10 |    12
(10 rows)
```

The following query returns the number of distinct products and the total inventory within each date key:

```
=> SELECT date_key, COUNT (DISTINCT product_key), SUM(qty_in_stock) FROM inventory_fact
   GROUP BY date_key LIMIT 10;

 date_key | count |  sum
----------+-------+--------
        1 |   173 |  88953
        2 |    31 |  16315
        3 |   318 | 156003
        4 |   113 |  53341
        5 |   285 | 148380
        6 |    84 |  42421
        7 |   241 | 119315
        8 |   238 | 122380
        9 |   142 |  70151
       10 |   202 |  95274
(10 rows)
```

This query selects each distinct product_key value and then counts the number of distinct date_key values for all records with the specific product_key value. It also counts the number of distinct warehouse_key values in all records with the specific product_key value:

```
=> SELECT product_key, COUNT (DISTINCT date_key), COUNT (DISTINCT warehouse_key) FROM inventory_fact
   GROUP BY product_key LIMIT 15;

 product_key | count | count
-------------+-------+-------
           1 |    12 |    12
           2 |    18 |    18
           3 |    13 |    12
           4 |    17 |    18
           5 |    11 |     9
           6 |    14 |    13
           7 |    13 |    13
           8 |    17 |    15
           9 |    15 |    14
          10 |    12 |    12
          11 |    11 |    11
          12 |    13 |    12
          13 |     9 |     7
          14 |    13 |    13
          15 |    18 |    17
(15 rows)
```

This query selects each distinct product_key value, counts the number of distinct date_key and warehouse_key values for all records with the specific product_key value, and then sums all qty_in_stock values in records with the specific product_key value. It then returns the number of product_version values in records with the specific product_key value:

```
=> SELECT product_key, COUNT (DISTINCT date_key),
      COUNT (DISTINCT warehouse_key),
      SUM (qty_in_stock),
      COUNT (product_version)
      FROM inventory_fact GROUP BY product_key LIMIT 15;

 product_key | count | count |  sum   | count
-------------+-------+-------+--------+-------
           1 |    12 |    12 |  5530 |    12
           2 |    18 |    18 |  9605 |    18
           3 |    13 |    12 |  8404 |    13
           4 |    17 |    18 | 10006 |    18
           5 |    11 |     9 |  4794 |    11
           6 |    14 |    13 |  7359 |    14
           7 |    13 |    13 |  7828 |    13
           8 |    17 |    15 |  9074 |    17
           9 |    15 |    14 |  7032 |    15
          10 |    12 |    12 |  5359 |    12
          11 |    11 |    11 |  6049 |    11
          12 |    13 |    12 |  6075 |    13
          13 |     9 |     7 |  3470 |     9
          14 |    13 |    13 |  5125 |    13
          15 |    18 |    17 |  9277 |    18
(15 rows)
```

See also

- Analytic functions
- AVG [aggregate]
- SUM [aggregate]
- SQL analytics
- APPROXIMATE_COUNT_DISTINCT
- APPROXIMATE_COUNT_DISTINCT_SYNOPSIS
- APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS

COVAR_POP

Returns the population covariance for a set of expression pairs. The return value is of type DOUBLE PRECISION . COVAR_POP eliminates expression pairs where either expression in the pair is NULL . If no rows remain, the function returns NULL .

Syntax

```
SELECT COVAR_POP ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT COVAR_POP (Annual_salary, Employee_age)
     FROM employee_dimension;
    COVAR_POP
-------------------
 -9032.34810730019
(1 row)
```

COVAR_SAMP

Returns the sample covariance for a set of expression pairs. The return value is of type DOUBLE PRECISION . COVAR_SAMP eliminates expression pairs where either expression in the pair is NULL . If no rows remain, the function returns NULL .

Syntax

```
SELECT COVAR_SAMP ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT COVAR_SAMP (Annual_salary, Employee_age)
     FROM employee_dimension;
   COVAR_SAMP
-------------------
 -9033.25143244343
(1 row)
```

GROUP_ID

Uniquely identifies duplicate sets for GROUP BY queries that return duplicate grouping sets. This function returns one or more integers, starting with zero (0), as identifiers.

For the number of duplicates *n* for a particular grouping, GROUP_ID returns a range of sequential numbers, 0 to *n* –1. For the first each unique group it encounters, GROUP_ID returns the value 0. If GROUP_ID finds the same grouping again, the function returns 1, then returns 2 for the next found grouping, and so on.

> **Note**
> Use GROUP_ID only in SELECT statements that contain a GROUP BY aggregate: CUBE , GROUPING SETS , and ROLLUP .

Behavior type

Immutable

Syntax

```
GROUP_ID ()
```

Examples

This example shows how GROUP_ID creates unique identifiers when a query produces duplicate groupings. For an expenses table, the following query groups the results by category of expense and year and rolls up the sum for those two columns. The results have duplicate groupings for category and NULL. The first grouping has a GROUP_ID of 0, and the second grouping has a GROUP_ID of 1.

```
=> SELECT Category, Year, SUM(Amount), GROUPING_ID(Category, Year),
   GROUP_ID() FROM expenses GROUP BY Category, ROLLUP(Category,Year)
   ORDER BY Category, Year, GROUPING_ID();
  Category   | Year |  SUM   | GROUPING_ID | GROUP_ID
-------------+------+--------+-------------+----------
 Books       | 2005 |  39.98 |           0 |        0
 Books       | 2007 |  29.99 |           0 |        0
 Books       | 2008 |  29.99 |           0 |        0
 Books       |      |  99.96 |           1 |        0
 Books       |      |  99.96 |           1 |        1
 Electricity | 2005 | 109.99 |           0 |        0
 Electricity | 2006 | 109.99 |           0 |        0
 Electricity | 2007 | 229.98 |           0 |        0
 Electricity |      | 449.96 |           1 |        1
 Electricity |      | 449.96 |           1 |        0
```

See also

- CUBE aggregate
- GROUPING
- GROUPING_ID
- GROUPING SETS aggregate
- GROUP BY clause
- ROLLUP aggregate

GROUPING

Disambiguates the use of NULL values when GROUP BY queries with multilevel aggregates generate NULL values to identify subtotals in grouping columns. Such NULL values from the original data can also occur in rows. GROUPING returns 1, if the value of *expression* is:

- NULL , representing an aggregated value
- 0 for any other value, including NULL values in rows

> **Note**
> Use GROUPING only in SELECT statements that contain a GROUP BY aggregate: CUBE , GROUPING SETS , and ROLLUP .

Behavior type

Immutable

Syntax

```
GROUPING ( expression )
```

Parameters

expression

An expression in the GROUP BY clause

Examples

The following query uses the GROUPING function, taking one of the GROUP BY expressions as an argument. For each row, GROUPING returns one of the following:

- 0 : The column is part of the group for that row
- 1 : The column is not part of the group for that row

The 1 in the GROUPING(Year) column for electricity and books indicates that these values are subtotals. The right-most column values for both GROUPING(Category) and GROUPING(Year) are 1 . This value indicates that neither column contributed to the GROUP BY . The final row represents the total sales.

```
=> SELECT Category, Year, SUM(Amount),
  GROUPING(Category), GROUPING(Year) FROM expenses
  GROUP BY ROLLUP(Category, Year) ORDER BY Category, Year, GROUPING_ID();
  Category  | Year |  SUM   | GROUPING | GROUPING
------------+------+--------+----------+----------
 Books       | 2005 |  39.98 |        0 |        0
 Books       | 2007 |  29.99 |        0 |        0
 Books       | 2008 |  29.99 |        0 |        0
 Books       |      |  99.96 |        0 |        1
 Electricity | 2005 | 109.99 |        0 |        0
 Electricity | 2006 | 109.99 |        0 |        0
 Electricity | 2007 | 229.98 |        0 |        0
 Electricity |      | 449.96 |        0 |        1
             |      | 549.92 |        1 |        1
```

See also

- CUBE aggregate
- GROUP_ID
- GROUPING_ID
- GROUPING SETS aggregate
- GROUP BY clause
- ROLLUP aggregate

GROUPING_ID

Concatenates the set of Boolean values generated by the GROUPING function into a bit vector. GROUPING_ID treats the bit vector as a binary number and returns it as a base-10 value that identifies the grouping set combination.

By using GROUPING_ID you avoid the need for multiple, individual GROUPING functions. GROUPING_ID simplifies row-filtering conditions, because rows of interest are identified using a single return from GROUPING_ID = *n* . Use GROUPING_ID to identify grouping combinations.

> **Note**
> Use GROUPING_ID only in SELECT statements that contain a GROUP BY aggregate: CUBE , GROUPING SETS , and ROLLUP .

Behavior type

Immutable

Syntax

```
GROUPING_ID ( [expression[,...] ] )
```

expression

An expression that matches one of the expressions in the GROUP BY clause.

If the GROUP BY clause includes a list of expressions, GROUPING_ID returns a number corresponding to the GROUPING bit vector associated with a row.

Examples

This example shows how calling GROUPING_ID without an expression returns the GROUPING bit vector associated with a full set of multilevel aggregate expressions. The GROUPING_ID value is comparable to GROUPING_ID(a,b) because GROUPING_ID() includes all columns in the GROUP BY ROLLUP :

```
=> SELECT a,b,COUNT(*), GROUPING_ID() FROM T GROUP BY ROLLUP(a,b);
```

In the following query, the GROUPING(Category) and GROUPING(Year) columns have three combinations:

- 0,0
- 0,1
- 1,1

```
=> SELECT Category, Year, SUM(Amount),
  GROUPING(Category), GROUPING(Year) FROM expenses
  GROUP BY ROLLUP(Category, Year) ORDER BY Category, Year, GROUPING_ID();
 Category   | Year |  SUM   | GROUPING | GROUPING
-------------+------+--------+----------+----------
 Books       | 2005 |  39.98 |        0 |        0
 Books       | 2007 |  29.99 |        0 |        0
 Books       | 2008 |  29.99 |        0 |        0
 Books       |      |  99.96 |        0 |        1
 Electricity | 2005 | 109.99 |        0 |        0
 Electricity | 2006 | 109.99 |        0 |        0
 Electricity | 2007 | 229.98 |        0 |        0
 Electricity |      | 449.96 |        0 |        1
             |      | 549.92 |        1 |        1
```

GROUPING_ID converts these values as follows:

Binary Set Values	**Decimal Equivalents**
00	0
01	1
11	3
0	Category, Year

The following query returns the single number for each GROUP BY level that appears in the gr_id column:

```
=> SELECT Category, Year, SUM(Amount),
  GROUPING(Category),GROUPING(Year),GROUPING_ID(Category,Year) AS gr_id
  FROM expenses GROUP BY ROLLUP(Category, Year);
 Category   | Year |  SUM   | GROUPING | GROUPING | gr_id
-------------+------+--------+----------+----------+-------
 Books       | 2008 |  29.99 |        0 |        0 |     0
 Books       | 2005 |  39.98 |        0 |        0 |     0
 Electricity | 2007 | 229.98 |        0 |        0 |     0
 Books       | 2007 |  29.99 |        0 |        0 |     0
 Electricity | 2005 | 109.99 |        0 |        0 |     0
 Electricity |      | 449.96 |        0 |        1 |     1
             |      | 549.92 |        1 |        1 |     3
 Electricity | 2006 | 109.99 |        0 |        0 |     0
 Books       |      |  99.96 |        0 |        1 |     1
```

The gr_id value determines the GROUP BY level for each row:

GROUP BY Level

GROUP BY Row Level

3

Total sum

1

Category

0

Category, year

You can also use the DECODE function to give the values more meaning by comparing each search value individually:

```
=> SELECT Category, Year, SUM(AMOUNT), DECODE(GROUPING_ID(Category, Year),
      3, 'Total',
      1, 'Category',
      0, 'Category,Year')
   AS GROUP_NAME FROM expenses GROUP BY ROLLUP(Category, Year);
 Category   | Year |  SUM   |  GROUP_NAME
------------+------+--------+---------------
Electricity | 2006 | 109.99 | Category,Year
Books       |      |  99.96 | Category
Electricity | 2007 | 229.98 | Category,Year
Books       | 2007 |  29.99 | Category,Year
Electricity | 2005 | 109.99 | Category,Year
Electricity |      | 449.96 | Category
            |      | 549.92 | Total
Books       | 2005 |  39.98 | Category,Year
Books       | 2008 |  29.99 | Category,Year
```

See also

- CUBE aggregate
- GROUP_ID
- GROUPING
- GROUPING SETS aggregate
- GROUP BY clause
- ROLLUP aggregate

LISTAGG

Transforms non-null values from a group of rows into a list of values that are delimited by commas (default) or a configurable separator. LISTAGG can be used to denormalize rows into a string of concatenated values.

Behavior type

Immutable if the WITHIN GROUP ORDER BY clause specifies a column or set of columns that resolves to unique values within the aggregated list; otherwise Volatile .

Syntax

```
LISTAGG ( aggregate-expression [ USING PARAMETERS parameter=value][,...] ] ) [ within-group-order-by-clause ]
```

Arguments

aggregate-expression

Aggregation of one or more columns or column expressions to select from the source table or view.

LISTAGG does not support spatial data types directly. In order to pass column data of this type, convert the data to strings with the geospatial function ST_AsText .

> **Caution**
> Converted spatial data frequently contains commas. LISTAGG uses comma as the default separator character. To avoid ambiguous output, override this default by setting the function's separator parameter to another character.

[within-group-order-by-clause](/sql-reference/functions/aggregate-functions/within-group-order-by-clause.html)

Sorts aggregated values within each group of rows, where *column-expression* is typically a column in *aggregate-expression* :

```
WITHIN GROUP (ORDER BY { column-expression[ sort-qualifiers ] }[,...])
```

sort-qualifiers :

```
 { ASC | DESC [ NULLS { FIRST | LAST | AUTO } ] }
```

> **Tip**
> WITHIN GROUP ORDER BY can consume a large amount of memory per group. Including wide strings in the aggregate expression can also adversely affect performance. To minimize memory consumption, create projections that support GROUPBY PIPELINED .

Parameters

Parameter name	Set to...
max_length	An integer or integer expression that specifies in bytes the maximum length of the result, up to 32M. **Default:** 1024
separator	Separator string of length 0 to 80, inclusive. A length of 0 concatenates the output with no separators. **Default:** comma (,)
on_overflow	Specifies behavior when the result overflows the max_length setting, one of the following strings: • ERROR (default): Return an error when overflow occurs. • TRUNCATE : Remove any characters that exceed max_length setting from the query result, and return the truncated string.

Privileges

None

Examples

In the following query, the aggregated results in the CityStat e column use the string " | " as a separator. The outer GROUP BY clause groups the output rows according to their Region values. Within each group, the aggregated list items are sorted according to their city values, as per the WITHIN GROUP ORDER BY clause:

```
=> \x
Expanded display is on.
=> WITH cd AS (SELECT DISTINCT (customer_city) city, customer_state, customer_region FROM customer_dimension)
SELECT customer_region Region, LISTAGG(city||', '||customer_state USING PARAMETERS separator=' | ')
   WITH GROUP (ORDER BY city) CityAndState FROM cd GROUP BY region ORDER BY region;
-[ RECORD 1 ]+-------------------
```

Parameters

expression

Any expression for which the maximum value is calculated, typically a column reference.

Examples

The following query returns the largest value in column sales_dollar_amount .

```
=> SELECT MAX(sales_dollar_amount) AS highest_sale FROM store.store_sales_fact;
 highest_sale
--------------
          600
(1 row)
```

The following example shows you the difference between the MIN and MAX aggregate functions when you use them with a Boolean value. The sample creates a table, adds two rows of data, and shows sample output for MIN and MAX .

```
=> CREATE TABLE min_max_functions (torf BOOL);

=> INSERT INTO min_max_functions VALUES (1);
=> INSERT INTO min_max_functions VALUES (0);

=> SELECT * FROM min_max_functions;
 torf
------
 t
 f
(2 rows)

=> SELECT min(torf) FROM min_max_functions;
 min
-----
 f
(1 row)

=> SELECT max(torf) FROM min_max_functions;
 max
-----
 t
(1 row)
```

See also

Data aggregation

MIN [aggregate]

Returns the smallest value of an expression over a group of rows. The return value has the same type as the expression data type.

The MIN analytic function differs from the aggregate function, in that it returns the minimum value of an expression over a group of rows within a window .

Aggregate functions MIN and MAX can operate with Boolean values. MAX can act upon a Boolean data type or a value that can be implicitly converted to a Boolean. If at least one input value is true, MAX returns t (true). Otherwise, it returns f (false). In the same scenario, MIN returns t (true) if all input values are true. Otherwise it returns f .

Behavior type

Immutable

Syntax

```
MIN ( expression )
```

Parameters

expression

Any expression for which the minimum value is calculated, typically a column reference .

Examples

The following query returns the lowest salary from the employee dimension table.

This example shows how you can query to return the lowest salary from the employee dimension table.

```
=> SELECT MIN(annual_salary) AS lowest_paid FROM employee_dimension;
 lowest_paid
-------------
        1200
(1 row)
```

The following example shows you the difference between the MIN and MAX aggregate functions when you use them with a Boolean value. The sample creates a table, adds two rows of data, and shows sample output for MIN and MAX .

```
=> CREATE TABLE min_max_functions (torf BOOL);

=> INSERT INTO min_max_functions VALUES (1);
=> INSERT INTO min_max_functions VALUES (0);

=> SELECT * FROM min_max_functions;
  torf
------
 t
 f
(2 rows)

=> SELECT min(torf) FROM min_max_functions;
 min
-----
 f
(1 row)

=> SELECT max(torf) FROM min_max_functions;
 max
-----
 t
(1 row)
```

See also

Data aggregation

REGR_AVGX

Returns the DOUBLE PRECISION average of the independent expression in an expression pair. REGR_AVGX eliminates expression pairs where either expression in the pair is NULL . If no rows remain, REGR_AVGX returns NULL .

Syntax

```
SELECT REGR_AVGX ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT REGR_AVGX (Annual_salary, Employee_age)
    FROM employee_dimension;
 REGR_AVGX
-----------
   39.321
(1 row)
```

REGR_AVGY

Returns the DOUBLE PRECISION average of the dependent expression in an expression pair. The function eliminates expression pairs where either expression in the pair is NULL . If no rows remain, the function returns NULL .

Syntax

```
REGR_AVGY ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT REGR_AVGY (Annual_salary, Employee_age)
    FROM employee_dimension;
 REGR_AVGY
------------
 58354.4913
(1 row)
```

REGR_COUNT

Returns the count of all rows in an expression pair. The function eliminates expression pairs where either expression in the pair is NULL . If no rows remain, the function returns 0 .

Syntax

```
SELECT REGR_COUNT ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT REGR_COUNT (Annual_salary, Employee_age) FROM employee_dimension;
 REGR_COUNT
------------
    10000
(1 row)
```

REGR_INTERCEPT

Returns the y-intercept of the regression line determined by a set of expression pairs. The return value is of type DOUBLE PRECISION . REGR_INTERCEPT eliminates expression pairs where either expression in the pair is NULL . If no rows remain, REGR_INTERCEPT returns NULL .

Syntax

```
SELECT REGR_INTERCEPT ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT REGR_INTERCEPT (Annual_salary, Employee_age) FROM employee_dimension;
 REGR_INTERCEPT
------------------
 59929.5490163437
(1 row)
```

REGR_R2

Returns the square of the correlation coefficient of a set of expression pairs. The return value is of type DOUBLE PRECISION . REGR_R2 eliminates expression pairs where either expression in the pair is NULL . If no rows remain, REGR_R2 returns NULL .

Syntax

```
SELECT REGR_R2 ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT REGR_R2 (Annual_salary, Employee_age) FROM employee_dimension;
     REGR_R2
----------------------
 5.17181631706311e-05
(1 row)
```

REGR_SLOPE

Returns the slope of the regression line, determined by a set of expression pairs. The return value is of type DOUBLE PRECISION . REGR_SLOPE eliminates expression pairs where either expression in the pair is NULL . If no rows remain, REGR_SLOPE returns NULL .

Syntax

```
SELECT REGR_SLOPE ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT REGR_SLOPE (Annual_salary, Employee_age) FROM employee_dimension;
   REGR_SLOPE
------------------
 -40.056400303749
(1 row)
```

REGR_SXX

Returns the sum of squares of the difference between the independent expression (*expression2*) and its average.

That is, REGR_SXX returns: ∑[(*expression2* - average(*expression2*)(*expression2* - average(*expression2*)]

The return value is of type DOUBLE PRECISION . REGR_SXX eliminates expression pairs where either expression in the pair is NULL . If no rows remain, REGR_SXX returns NULL .

Syntax

```
SELECT REGR_SXX ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT REGR_SXX (Annual_salary, Employee_age) FROM employee_dimension;
  REGR_SXX
------------
 2254907.59
(1 row)
```

REGR_SXY

Returns the sum of products of the difference between the dependent expression (*expression1*) and its average and the difference between the independent expression (*expression2*) and its average.

That is, REGR_SXY returns: ∑[(*expression1* - average(*expression1*)(*expression2* - average(*expression2*))]

The return value is of type DOUBLE PRECISION . REGR_SXY eliminates expression pairs where either expression in the pair is NULL . If no rows remain, REGR_SXY returns NULL .

Syntax

```
SELECT REGR_SXY ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT REGR_SXY (Annual_salary, Employee_age) FROM employee_dimension;
    REGR_SXY
--------------------
 -90323481.0730019
(1 row)
```

REGR_SYY

Returns the sum of squares of the difference between the dependent expression (*expression1*) and its average.

That is, REGR_SYY returns: ∑[(*expression1* - average(*expression1*)(*expression1* - average(*expression1*)]

The return value is of type DOUBLE PRECISION . REGR_SYY eliminates expression pairs where either expression in the pair is NULL . If no rows remain, REGR_SYY returns NULL .

Syntax

```
SELECT REGR_SYY ( expression1, expression2 )
```

Parameters

expression1

The dependent DOUBLE PRECISION expression

expression2

The independent DOUBLE PRECISION expression

Examples

```
=> SELECT REGR_SYY (Annual_salary, Employee_age) FROM employee_dimension;
    REGR_SYY
------------------
 69956728794707.2
(1 row)
```

STDDEV [aggregate]

Evaluates the statistical sample standard deviation for each member of the group. The return value is the same as the square root of VAR_SAMP :

```
STDDEV(expression) = SQRT(VAR_SAMP(expression))
```

Behavior type

Immutable

Syntax

```
STDDEV ( expression )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. STDDEV returns the same data type as *expression* .

Related functions

- Nonstandard function STDDEV is provided for compatibility with other databases. It is semantically identical to STDDEV_SAMP .
- This aggregate function differs from analytic function STDDEV , which computes the statistical sample standard deviation of the current row with respect to the group of rows within a window .
- When VAR_SAMP returns NULL , STDDEV returns NULL .

Examples

The following example returns the statistical sample standard deviation for each household ID from the customer_dimension table of the VMart example database:

```
=> SELECT STDDEV(household_id) FROM customer_dimension;
  STDDEV
-----------------
 8651.5084240071
```

STDDEV_POP [aggregate]

Evaluates the statistical population standard deviation for each member of the group.

Behavior type

Immutable

Syntax

```
STDDEV_POP ( expression )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. STDDEV_POP returns the same data type as *expression* .

Related functions

- This function differs from the analytic function STDDEV_POP , which evaluates the statistical population standard deviation for each member of the group of rows within a window .
- STDDEV_POP returns the same value as the square root of VAR_POP :

  ```
  STDDEV_POP(expression) = SQRT(VAR_POP(expression))
  ```

- When VAR_SAMP returns NULL , this function returns NULL .

Examples

The following example returns the statistical population standard deviation for each household ID in the customer table.

```
=> SELECT STDDEV_POP(household_id) FROM customer_dimension;
  STDDEV_POP
------------------
 8651.41895973367
(1 row)
```

See also

- Analytic functions
- SQL analytics

STDDEV_SAMP [aggregate]

Evaluates the statistical sample standard deviation for each member of the group. The return value is the same as the square root of VAR_SAMP :

```
STDDEV_SAMP(expression) = SQRT(VAR_SAMP(expression))
```

Behavior type

Immutable

Syntax

```
STDDEV_SAMP ( expression )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. STDDEV_SAMP returns the same data type as *expression* .

Related functions

- STDDEV_SAMP is semantically identical to nonstandard function STDDEV , which is provided for compatibility with other databases.
- This aggregate function differs from analytic function STDDEV_SAMP , which computes the statistical sample standard deviation of the current row with respect to the group of rows within a window .
- When VAR_SAMP returns NULL , STDDEV_SAMP returns NULL .

Examples

The following example returns the statistical sample standard deviation for each household ID from the customer dimension table.

```
=> SELECT STDDEV_SAMP(household_id) FROM customer_dimension;
  stddev_samp
------------------
 8651.50842400771
(1 row)
```

SUM [aggregate]

Computes the sum of an expression over a group of rows. SUM returns a DOUBLE PRECISION value for a floating-point expression. Otherwise, the return value is the same as the expression data type.

The SUM aggregate function differs from the SUM analytic function, which computes the sum of an expression over a group of rows within a window .

Behavior type

Immutable

Syntax

```
SUM ( [ ALL | DISTINCT ] expression )
```

Parameters

ALL

Invokes the aggregate function for all rows in the group (default)

DISTINCT

Invokes the aggregate function for all distinct non-null values of the expression found in the group

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

Overflow handling

If you encounter data overflow when using SUM() , use SUM_FLOAT which converts the data to a floating point.

By default, Vertica allows silent numeric overflow when you call this function on numeric data types. For more information on this behavior and how to change it, see Numeric data type overflow with SUM, SUM_FLOAT, and AVG .

Examples

The following query returns the total sum of the product_cost column.

```
=> SELECT SUM(product_cost) AS cost FROM product_dimension;
  cost
---------
 9042850
(1 row)
```

See also

- AVG [aggregate]
- COUNT [aggregate]

SUM_FLOAT [aggregate]

Computes the sum of an expression over a group of rows and returns a DOUBLE PRECISION value.

Behavior type

Immutable

Syntax

```
SUM_FLOAT ( [ ALL | DISTINCT ] expression )
```

Parameters

ALL

Invokes the aggregate function for all rows in the group (default).

DISTINCT

Invokes the aggregate function for all distinct non-null values of the expression found in the group.

expression

Any expression whose result is type DOUBLE PRECISION .

Overflow handling

By default, Vertica allows silent numeric overflow when you call this function on numeric data types. For more information on this behavior and how to change it, see Numeric data type overflow with SUM, SUM_FLOAT, and AVG .

Examples

The following query returns the floating-point sum of the average price from the product table:

```
=> SELECT SUM_FLOAT(average_competitor_price) AS cost FROM product_dimension;
  cost
----------
 18181102
(1 row)
```

TS_FIRST_VALUE

Processes the data that belongs to each time slice. A time series aggregate (TSA) function, TS_FIRST_VALUE returns the value at the start of the time slice, where an interpolation scheme is applied if the timeslice is missing, in which case the value is determined by the values corresponding to the previous (and next) timeslices based on the interpolation scheme of const (linear).

TS_FIRST_VALUE returns one output row per time slice, or one output row per partition per time slice if partition expressions are specified

Behavior type

Immutable

Syntax

```
TS_FIRST_VALUE ( expression [ IGNORE NULLS ] [, { 'CONST' | 'LINEAR' } ] )
```

Parameters

expression

An INTEGER or FLOAT expression on which to aggregate and interpolate.

IGNORE NULLS

The IGNORE NULLS behavior changes depending on a CONST or LINEAR interpolation scheme. See When Time Series Data Contains Nulls in Analyzing Data for details.

'CONST' | 'LINEAR'

Specifies the interpolation value as constant or linear:

- CONST (default): New value is interpolated based on previous input records.
- LINEAR : Values are interpolated in a linear slope based on the specified time slice.

Requirements

You must use an ORDER BY clause with a TIMESTAMP column.

Multiple time series aggregate functions

The same query can call multiple time series aggregate functions. They share the same gap-filling policy as defined by the TIMESERIES clause ; however, each time series aggregate function can specify its own interpolation policy. For example:

```
=> SELECT slice_time, symbol,
TS_FIRST_VALUE(bid, 'const') fv_c,
    TS_FIRST_VALUE(bid, 'linear') fv_l,
    TS_LAST_VALUE(bid, 'const') lv_c
FROM TickStore
TIMESERIES slice_time AS '3 seconds'
OVER(PARTITION BY symbol ORDER BY ts);
```

Examples

See Gap Filling and Interpolation in Analyzing Data.

See also

- TS_LAST_VALUE
- Time series analytics

TS_LAST_VALUE

Processes the data that belongs to each time slice. A time series aggregate (TSA) function, TS_LAST_VALUE returns the value at the end of the time slice, where an interpolation scheme is applied if the timeslice is missing. In this case the value is determined by the values corresponding to the previous (and next) timeslices based on the interpolation scheme of const (linear).

TS_LAST_VALUE returns one output row per time slice, or one output row per partition per time slice if partition expressions are specified.

Behavior type

Immutable

Syntax

```
TS_LAST_VALUE ( expression [ IGNORE NULLS ] [, { 'CONST' | 'LINEAR' } ] )
```

Parameters

expression

An INTEGER or FLOAT expression on which to aggregate and interpolate.

IGNORE NULLS

The IGNORE NULLS behavior changes depending on a CONST or LINEAR interpolation scheme. See When Time Series Data Contains Nulls in Analyzing Data for details.

'CONST' | 'LINEAR'

Specifies the interpolation value as constant or linear:

- CONST (default): New value is interpolated based on previous input records.
- LINEAR : Values are interpolated in a linear slope based on the specified time slice.

Requirements

You must use the ORDER BY clause with a TIMESTAMP column.

Multiple time series aggregate functions

The same query can call multiple time series aggregate functions. They share the same gap-filling policy as defined by the TIMESERIES clause ; however, each time series aggregate function can specify its own interpolation policy. For example:

```
=> SELECT slice_time, symbol,
TS_FIRST_VALUE(bid, 'const') fv_c,
      TS_FIRST_VALUE(bid, 'linear') fv_l,
      TS_LAST_VALUE(bid, 'const') lv_c
FROM TickStore
TIMESERIES slice_time AS '3 seconds'
OVER(PARTITION BY symbol ORDER BY ts);
```

Examples

See Gap Filling and Interpolation in Analyzing Data.

See also

- TS_FIRST_VALUE
- Time series analytics

VAR_POP [aggregate]

Evaluates the population variance for each member of the group. This is defined as the sum of squares of the difference of * expression *from the mean of *expression* , divided by the number of remaining rows:

```
(SUM(expression*expression) - SUM(expression)*SUM(expression) / COUNT(expression)) / COUNT(expression)
```

Behavior type

Immutable

Syntax

```
VAR_POP ( expression )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. VAR_POP returns the same data type as *expression* .

Related functions

This aggregate function differs from analytic function VAR_POP , which computes the population variance of the current row with respect to the group of rows within a window .

Examples

The following example returns the population variance for each household ID in the customer table.

```
=> SELECT VAR_POP(household_id) FROM customer_dimension;
   var_pop
------------------
 74847050.0168393
(1 row)
```

VAR_SAMP [aggregate]

Evaluates the sample variance for each row of the group. This is defined as the sum of squares of the difference of *expression* from the mean of *expression* divided by the number of remaining rows minus 1:

```
(SUM(expression*expression) - SUM(expression) *SUM(expression) / COUNT(expression)) / (COUNT(expression) -1)
```

Behavior type

Immutable

Syntax

```
VAR_SAMP ( expression )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. VAR_SAMP returns the same data type as *expression* .

Related functions

- VAR_SAMP is semantically identical to nonstandard function VARIANCE , which is provided for compatibility with other databases.
- This aggregate function differs from analytic function VAR_SAMP , which computes the sample variance of the current row with respect to the group of rows within a window .

Examples

The following example returns the sample variance for each household ID in the customer table.

```
=> SELECT VAR_SAMP(household_id) FROM customer_dimension;
   var_samp
------------------
 74848598.0106764
(1 row)
```

See also

VARIANCE [aggregate]

VARIANCE [aggregate]

Evaluates the sample variance for each row of the group. This is defined as the sum of squares of the difference of *expression* from the mean of *expression* divided by the number of remaining rows minus 1.

```
(SUM(expression*expression) - SUM(expression) *SUM(expression) /COUNT(expression)) / (COUNT(expression) -1)
```

Behavior type

Immutable

Syntax

```
VARIANCE ( expression )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. VARIANCE returns the same data type as *expression* .

Related functions

The nonstandard function VARIANCE is provided for compatibility with other databases. It is semantically identical to VAR_SAMP .

This aggregate function differs from analytic function VARIANCE , which computes the sample variance of the current row with respect to the group of rows within a window .

Examples

The following example returns the sample variance for each household ID in the customer table.

```
=> SELECT VARIANCE(household_id) FROM customer_dimension;
   variance
------------------
 74848598.0106764
(1 row)
```

See also

- Analytic functions
- VAR_SAMP [aggregate]
- SQL analytics

WITHIN GROUP ORDER BY clause

Specifies how to sort rows that are grouped by aggregate functions, one of the following:

- ARGMAX_AGG
- ARGMIN_AGG
- IMPLODE
- LISTAGG

This clause is also supported for user-defined aggregate functions.

The order clause only specifies order within the result set of each group. The query can have its own ORDER BY clause, which has precedence over order that is specified by WITHIN GROUP ORDER BY, and orders the final result set.

Syntax

```
WITHIN GROUP (ORDER BY
 { column-expression [ ASC | DESC [ NULLS { FIRST | LAST | AUTO } ] ]
 }[,...])
```

Parameters

column-expression

A column, constant, or arbitrary expression formed on columns, on which to sort grouped rows.

ASC | DESC

Specifies the ordering sequence as ascending (default) or descending.

NULLS {FIRST | LAST | AUTO}

Specifies whether to position null values first or last. Default positioning depends on whether the sort order is ascending or descending:

- Ascending default: NULLS LAST
- Descending default: NULLS FIRST

If you specify NULLS AUTO , Vertica chooses the positioning that is most efficient for this query, either NULLS FIRST or NULLS LAST .

If you omit all sort qualifiers, Vertica uses ASC NULLS LAST .

Examples

For usage examples, see these functions:

- ARGMAX_AGG
- ARGMIN_AGG
- IMPLODE
- LISTAGG

Analytic functions

> **Note**
> All analytic functions in this section with an aggregate counterpart are appended with [Analytics] in the heading to avoid confusion between the two function types.

Vertica analytics are SQL functions based on the ANSI 99 standard. These functions handle complex analysis and reporting tasks—for example:

- Rank the longest-standing customers in a particular state.
- Calculate the moving average of retail volume over a specified time.
- Find the highest score among all students in the same grade.
- Compare the current sales bonus that salespersons received against their previous bonus.

Analytic functions return aggregate results but they do not group the result set. They return the group value multiple times, once per record. You can sort group values, or partitions, using a window ORDER BY clause, but the order affects only the function result set, not the entire query result set.

Syntax

General

```
analytic-function(arguments) OVER(
 [ window-partition-clause ]
 [ window-order-clause [ window-frame-clause ] ]
)
```

With named window

```
analytic-function(arguments) OVER(
  [ named-window [ window-frame-clause ] ]
)
```

Parameters

***analytic-function* (*arguments*)**

A Vertica analytic function and its arguments.

OVER

Specifies how to partition, sort, and window frame function input with respect to the current row. The input data is the result set that the query returns after it evaluates FROM , WHERE , GROUP BY , and HAVING clauses.

An empty OVER clause provides the best performance for single threaded queries on a single node.

window-partition-clause

Groups input rows according to one or more columns or expressions.

If you omit this clause, no grouping occurs and the analytic function processes all input rows as a single partition.

window-order-clause

Optionally specifies how to sort rows that are supplied to the analytic function. If the OVER clause also includes a partition clause, rows are sorted within each partition.

window-frame-clause

Only valid for some analytic functions, specifies as input a set of rows relative to the row that is currently being evaluated by the analytic function. After the function processes that row and its window, Vertica advances the current row and adjusts the window boundaries accordingly.

named-window

The name of a window that you define in the same query with a window-name-clause . This definition encapsulates window partitioning and sorting. Named windows are useful when the query invokes multiple analytic functions with similar OVER clauses.

A window name clause cannot specify a window frame clause. However, you can qualify the named window in an OVER clause with a window frame clause.

Requirements

The following requirements apply to analytic functions:

- All require an OVER clause. Each function has its own OVER clause requirements. For example, you can supply an empty OVER clause for some analytic aggregate functions such as SUM . For other functions, window frame and order clauses might be required, or might be invalid.
- Analytic functions can be invoked only in a query's SELECT and ORDER BY clauses.
- Analytic functions cannot be nested. For example, the following query is not allowed:
  ```
  => SELECT MEDIAN(RANK() OVER(ORDER BY sal) OVER()).
  ```
- WHERE , GROUP BY and HAVING operators are technically not part of the analytic function. However, they determine input to that function.

See also

- SQL analytics
- GROUP BY queries

In this section

- ARGMAX [analytic]
- ARGMIN [analytic]
- AVG [analytic]
- BOOL_AND [analytic]
- BOOL_OR [analytic]
- BOOL_XOR [analytic]
- CONDITIONAL_CHANGE_EVENT [analytic]
- CONDITIONAL_TRUE_EVENT [analytic]
- COUNT [analytic]
- CUME_DIST [analytic]
- DENSE_RANK [analytic]
- EXPONENTIAL_MOVING_AVERAGE [analytic]
- FIRST_VALUE [analytic]
- LAG [analytic]

- LAST_VALUE [analytic]
- LEAD [analytic]
- MAX [analytic]
- MEDIAN [analytic]
- MIN [analytic]
- NTH_VALUE [analytic]
- NTILE [analytic]
- PERCENT_RANK [analytic]
- PERCENTILE_CONT [analytic]
- PERCENTILE_DISC [analytic]
- RANK [analytic]
- ROW_NUMBER [analytic]
- STDDEV [analytic]
- STDDEV_POP [analytic]
- STDDEV_SAMP [analytic]
- SUM [analytic]
- VAR_POP [analytic]
- VAR_SAMP [analytic]
- VARIANCE [analytic]

ARGMAX [analytic]

This function is patterned after the mathematical function argmax(*f* (*x*)) , which returns the value of *x* that maximizes *f* (*x*) . Similarly, ARGMAX takes two arguments *target* and *arg* , where both are columns or column expressions in the queried dataset. ARGMAX finds the row with the largest non-null value in *target* and returns the value of *arg* in that row. If multiple rows contain the largest *target* value, ARGMAX returns *arg* from the first row that it finds.

Behavior type

Immutable

Syntax

```
ARGMAX ( target, arg ) OVER ( [ PARTITION BY expression[,...] ] [ window-order-clause ] )
```

Arguments

target* , *arg

Columns in the queried dataset.

> **Note**
> ARGMAX does not support spatial data types : GEOMETRY and GEOGRAPHY.

OVER()

Specifies the following window clauses:

- PARTITION BY *expression* : Groups (partitions) input rows according to the values in *expression* , which resolves to one or more columns in the queried dataset. If you omit this clause, ARGMAX processes all input rows as a single partition.
- *window-order-clause* : Specifies how to sort input rows. If the OVER clause also includes a partition clause, rows are sorted separately within each partition.
  Important
  To ensure consistent results when multiple rows contain the largest *target* value, include a window order clause that sorts on *arg* .

For details, see Analytic Functions .

Examples

Create and populate table service_info , which contains information on various services, their respective development groups, and their userbase. A NULL in the users column indicates that the service has not been released, and so it cannot have users.

```
=> CREATE TABLE service_info(dev_group VARCHAR(10), product_name VARCHAR(30), users INT);
=> COPY t FROM stdin NULL AS 'null';
>> iris|chat|48193
>> aspen|trading|3000
>> orchid|cloud|990322
>> iris|video call| 10203
>> daffodil|streaming|44123
>> hydrangea|password manager|null
>> hydrangea|totp|1837363
>> daffodil|clip share|3000
>> hydrangea|e2e sms|null
>> rose|crypto|null
>> iris|forum|48193
>> \.
```

ARGMAX returns the value in the product_name column that maximizes the value in the users column. In this case, ARGMAX returns totp , which indicates that the totp service has the largest user base:

```
=> SELECT dev_group, product_name, users, ARGMAX(users, product_name) OVER (ORDER BY dev_group ASC) FROM service_info;
 dev_group |   product_name   |  users  | ARGMAX
-----------+------------------+---------+--------
 aspen     | trading          |    3000 | totp
 daffodil  | clip share       |    3000 | totp
 daffodil  | streaming        |   44123 | totp
 hydrangea | e2e sms          |         | totp
 hydrangea | password manager |         | totp
 hydrangea | totp             | 1837363 | totp
 iris      | chat             |   48193 | totp
 iris      | forum            |   48193 | totp
 iris      | video call       |   10203 | totp
 orchid    | cloud            |  990322 | totp
 rose      | crypto           |         | totp
(11 rows)
```

The next query partitions the data on dev_group to identify the most popular service created by each development group. ARGMAX returns NULL if the partition's users column contains only NULL values and breaks ties using the first value in product_name from the top of the partition.

```
=> SELECT dev_group, product_name, users, ARGMAX(users, product_name) OVER (PARTITION BY dev_group ORDER BY product_name ASC) FROM
service_info;
 dev_group |   product_name   |  users  |  ARGMAX
-----------+------------------+---------+-----------
 iris      | chat             |   48193 | chat
 iris      | forum            |   48193 | chat
 iris      | video call       |   10203 | chat
 orchid    | cloud            |  990322 | cloud
 aspen     | trading          |    3000 | trading
 daffodil  | clip share       |    3000 | streaming
 daffodil  | streaming        |   44123 | streaming
 rose      | crypto           |         |
 hydrangea | e2e sms          |         | totp
 hydrangea | password manager |         | totp
 hydrangea | totp             | 1837363 | totp
(11 rows)
```

See also

ARGMIN [analytic]
ARGMIN [analytic]

This function is patterned after the mathematical function argmin(*f* (*x*)) , which returns the value of *x* that minimizes *f* (*x*) . Similarly, ARGMIN takes two arguments *target* and *arg* , where both are columns or column expressions in the queried dataset. ARGMIN finds the row with the smallest non-null value in *target* and returns the value of *arg* in that row. If multiple rows contain the smallest *target* value, ARGMIN returns *arg* from the first row that it finds.

Behavior type

Immutable

Syntax

```
ARGMIN ( target, arg )  OVER ( [ PARTITION BY expression[,...] ] [ window-order-clause ] )
```

Arguments

target* , *arg

Columns in the queried dataset.

> **Note**
> ARGMIN does not support spatial data types : GEOMETRY and GEOGRAPHY.

OVER()

Specifies the following window clauses:

- PARTITION BY *expression* : Groups (partitions) input rows according to the values in *expression* , which resolves to one or more columns in the queried dataset. If you omit this clause, ARGMIN processes all input rows as a single partition.
- *window-order-clause* : Specifies how to sort input rows. If the OVER clause also includes a partition clause, rows are sorted separately within each partition.
  Important
  To ensure consistent results when multiple rows contain the smallest *target* value, include a window order clause that sorts on *arg* .

For details, see Analytic Functions .

Examples

Create and populate table service_info , which contains information on various services, their respective development groups, and their userbase. A NULL in the users column indicates that the service has not been released, and so it cannot have users.

```
=> CREATE TABLE service_info(dev_group VARCHAR(10), product_name VARCHAR(30), users INT);
=> COPY t FROM stdin NULL AS 'null';
>> iris|chat|48193
>> aspen|trading|3000
>> orchid|cloud|990322
>> iris|video call| 10203
>> daffodil|streaming|44123
>> hydrangea|password manager|null
>> hydrangea|totp|1837363
>> daffodil|clip share|3000
>> hydrangea|e2e sms|null
>> rose|crypto|null
>> iris|forum|48193
>> \.
```

ARGMIN returns the value in the product_name column that minimizes the value in the users column. In this case, ARGMIN returns totp , which indicates that the totp service has the smallest user base:

```
=> SELECT dev_group, product_name, users, ARGMIN(users, product_name) OVER (ORDER BY dev_group ASC) FROM service_info;
 dev_group |   product_name   |  users  | ARGMIN
-----------+------------------+---------+---------
 aspen     | trading          |    3000 | trading
 daffodil  | clip share       |    3000 | trading
 daffodil  | streaming        |   44123 | trading
 hydrangea | e2e sms          |         | trading
 hydrangea | password manager |         | trading
 hydrangea | totp             | 1837363 | trading
 iris      | chat             |   48193 | trading
 iris      | forum            |   48193 | trading
 iris      | video call       |   10203 | trading
 orchid    | cloud            |  990322 | trading
 rose      | crypto           |         | trading
(11 rows)
```

The next query partitions the data on dev_group to identify the least popular service created by each development group. ARGMIN returns NULL if the partition's users column contains only NULL values and breaks ties using the first value in product_name from the top of the partition.

```
=> SELECT dev_group, product_name, users, ARGMIN(users, product_name) OVER (PARTITION BY dev_group ORDER BY product_name ASC) FROM
service_info;
 dev_group |   product_name   |  users  |   ARGMIN
-----------+------------------+---------+------------
 iris      | chat             |   48193 | video call
 iris      | forum            |   48193 | video call
 iris      | video call       |   10203 | video call
 orchid    | cloud            |  990322 | cloud
 aspen     | trading          |    3000 | trading
 daffodil  | clip share       |    3000 | clip share
 daffodil  | streaming        |   44123 | clip share
 rose      | crypto           |         |
 hydrangea | e2e sms          |         | totp
 hydrangea | password manager |         | totp
 hydrangea | totp             | 1837363 | totp
(11 rows)
```

See also

ARGMAX [analytic]

AVG [analytic]

Computes an average of an expression in a group within a window. AVG returns the same data type as the expression's numeric data type.

The AVG analytic function differs from the AVG aggregate function, which computes the average of an expression over a group of rows.

Behavior type

Immutable

Syntax

```
AVG ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Any data that can be implicitly converted to a numeric data type.

OVER()

See Analytic Functions.

Overflow handling

By default, Vertica allows silent numeric overflow when you call this function on numeric data types. For more information on this behavior and how to change it, see Numeric data type overflow with SUM, SUM_FLOAT, and AVG .

Examples

The following query finds the sales for that calendar month and returns a running/cumulative average (sometimes called a moving average) using the default window of RANGE UNBOUNDED PRECEDING AND CURRENT ROW :

```
=> SELECT calendar_month_number_in_year Mo, SUM(product_price) Sales,
   AVG(SUM(product_price)) OVER (ORDER BY calendar_month_number_in_year)::INTEGER Average
   FROM product_dimension pd, date_dimension dm, inventory_fact if
   WHERE dm.date_key = if.date_key AND pd.product_key = if.product_key GROUP BY Mo;
 Mo |  Sales   | Average
----+----------+----------
  1 | 23869547 | 23869547
  2 | 19604661 | 21737104
  3 | 22877913 | 22117374
  4 | 22901263 | 22313346
  5 | 23670676 | 22584812
  6 | 22507600 | 22571943
  7 | 21514089 | 22420821
  8 | 24860684 | 22725804
  9 | 21687795 | 22610470
 10 | 23648921 | 22714315
 11 | 21115910 | 22569005
 12 | 24708317 | 22747281
(12 rows)
```

To return a moving average that is not a running (cumulative) average, the window can specify ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING :

```
=> SELECT calendar_month_number_in_year Mo, SUM(product_price) Sales,
   AVG(SUM(product_price)) OVER (ORDER BY calendar_month_number_in_year
     ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)::INTEGER Average
   FROM product_dimension pd, date_dimension dm, inventory_fact if
   WHERE dm.date_key = if.date_key AND pd.product_key = if.product_key GROUP BY Mo;
 Mo |  Sales   | Average
----+----------+----------
  1 | 23869547 | 22117374
  2 | 19604661 | 22313346
  3 | 22877913 | 22584812
  4 | 22901263 | 22312423
  5 | 23670676 | 22694308
  6 | 22507600 | 23090862
  7 | 21514089 | 22848169
  8 | 24860684 | 22843818
  9 | 21687795 | 22565480
 10 | 23648921 | 23204325
 11 | 21115910 | 22790236
 12 | 24708317 | 23157716
(12 rows)
```

See also

- COUNT [analytic]
- SUM [analytic]
- SQL analytics

BOOL_AND [analytic]

Returns the Boolean value of an expression within a window . If all input values are true, BOOL_AND returns t . Otherwise, it returns f .

Behavior type

Immutable

Syntax

```
BOOL_AND ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

A Boolean data type or any non-Boolean data type that can be implicitly converted to a Boolean data type. The function returns a Boolean value.

OVER()

See Analytic Functions.

Examples

The following example illustrates how you can use the BOOL_AND , BOOL_OR , and BOOL_XOR analytic functions. The sample table, employee, includes a column for type of employee and years paid.

```
=> CREATE TABLE employee(emptype VARCHAR, yearspaid VARCHAR);
CREATE TABLE
```

Insert sample data into the table to show years paid. In more than one case, an employee could be paid more than once within one year.

```
=> INSERT INTO employee
SELECT 'contractor1', '2014'
UNION ALL
SELECT 'contractor2', '2015'
UNION ALL
SELECT 'contractor3', '2014'
UNION ALL
SELECT 'contractor1', '2014'
UNION ALL
SELECT 'contractor2', '2014'
UNION ALL
SELECT 'contractor3', '2015'
UNION ALL
SELECT 'contractor4', '2014'
UNION ALL
SELECT 'contractor4', '2014'
UNION ALL
SELECT 'contractor5', '2015'
UNION ALL
SELECT 'contractor5', '2016';
 OUTPUT
--------
    10
(1 row)
```

Query the table. The result shows employees that were paid twice in 2014 (BOOL_AND), once or twice in 2014 (BOOL_OR), and specifically not more than once in 2014 (BOOL_XOR).

```
=> SELECT DISTINCT emptype,
BOOL_AND(yearspaid='2014') OVER (PARTITION BY emptype) AS paidtwicein2014,
BOOL_OR(yearspaid='2014') OVER (PARTITION BY emptype) AS paidonceortwicein2014,
BOOL_XOR(yearspaid='2014') OVER (PARTITION BY emptype) AS paidjustoncein2014
FROM employee;


  emptype   | paidtwicein2014 | paidonceortwicein2014 | paidjustoncein2014
------------+-----------------+-----------------------+--------------------
 contractor1 | t               | t                     | f
 contractor2 | f               | t                     | t
 contractor3 | f               | t                     | t
 contractor4 | t               | t                     | f
 contractor5 | f               | f                     | f
(5 rows)
```

See also

- BOOL_AND [aggregate]
- BOOL_OR [analytic]
- BOOL_XOR [analytic]
- Boolean data type

BOOL_OR [analytic]

Returns the Boolean value of an expression within a window . If at least one input value is true, BOOL_OR returns t . Otherwise, it returns f .

Behavior type

Immutable

Syntax

```
BOOL_OR ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

A Boolean data type or any non-Boolean data type that can be implicitly converted to a Boolean data type. The function returns a Boolean value.

OVER()

See Analytic Functions.

Examples

The following example illustrates how you can use the BOOL_AND , BOOL_OR , and BOOL_XOR analytic functions. The sample table, employee, includes a column for type of employee and years paid.

```
=> CREATE TABLE employee(emptype VARCHAR, yearspaid VARCHAR);
CREATE TABLE
```

Insert sample data into the table to show years paid. In more than one case, an employee could be paid more than once within one year.

```
=> INSERT INTO employee
SELECT 'contractor1', '2014'
UNION ALL
SELECT 'contractor2', '2015'
UNION ALL
SELECT 'contractor3', '2014'
UNION ALL
SELECT 'contractor1', '2014'
UNION ALL
SELECT 'contractor2', '2014'
UNION ALL
SELECT 'contractor3', '2015'
UNION ALL
SELECT 'contractor4', '2014'
UNION ALL
SELECT 'contractor4', '2014'
UNION ALL
SELECT 'contractor5', '2015'
UNION ALL
SELECT 'contractor5', '2016';
 OUTPUT
--------
     10
(1 row)
```

Query the table. The result shows employees that were paid twice in 2014 (BOOL_AND), once or twice in 2014 (BOOL_OR), and specifically not more than once in 2014 (BOOL_XOR).

```
=> SELECT DISTINCT emptype,
BOOL_AND(yearspaid='2014') OVER (PARTITION BY emptype) AS paidtwicein2014,
BOOL_OR(yearspaid='2014') OVER (PARTITION BY emptype) AS paidonceortwicein2014,
BOOL_XOR(yearspaid='2014') OVER (PARTITION BY emptype) AS paidjustoncein2014
FROM employee;


   emptype   | paidtwicein2014 | paidonceortwicein2014 | paidjustoncein2014
-------------+-----------------+-----------------------+--------------------
 contractor1 | t               | t                     | f
 contractor2 | f               | t                     | t
 contractor3 | f               | t                     | t
 contractor4 | t               | t                     | f
 contractor5 | f               | f                     | f
(5 rows)
```

See also

- BOOL_OR [aggregate]
- BOOL_AND [analytic]
- BOOL_XOR [analytic]
- Boolean data type

BOOL_XOR [analytic]

Returns the Boolean value of an expression within a window . If only one input value is true, BOOL_XOR returns t . Otherwise, it returns f .

Behavior type

Immutable

Syntax

```
BOOL_XOR ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

A Boolean data type or any non-Boolean data type that can be implicitly converted to a Boolean data type. The function returns a Boolean value.

OVER()

See Analytic Functions.

Examples

The following example illustrates how you can use the BOOL_AND , BOOL_OR , and BOOL_XOR analytic functions. The sample table, employee, includes a column for type of employee and years paid.

```
=> CREATE TABLE employee(emptype VARCHAR, yearspaid VARCHAR);
CREATE TABLE
```

Insert sample data into the table to show years paid. In more than one case, an employee could be paid more than once within one year.

```
=> INSERT INTO employee
SELECT 'contractor1', '2014'
UNION ALL
SELECT 'contractor2', '2015'
UNION ALL
SELECT 'contractor3', '2014'
UNION ALL
SELECT 'contractor1', '2014'
UNION ALL
SELECT 'contractor2', '2014'
UNION ALL
SELECT 'contractor3', '2015'
UNION ALL
SELECT 'contractor4', '2014'
UNION ALL
SELECT 'contractor4', '2014'
UNION ALL
SELECT 'contractor5', '2015'
UNION ALL
SELECT 'contractor5', '2016';
 OUTPUT
--------
    10
(1 row)
```

Query the table. The result shows employees that were paid twice in 2014 (BOOL_AND), once or twice in 2014 (BOOL_OR), and specifically not more than once in 2014 (BOOL_XOR).

```
=> SELECT DISTINCT emptype,
BOOL_AND(yearspaid='2014') OVER (PARTITION BY emptype) AS paidtwicein2014,
BOOL_OR(yearspaid='2014') OVER (PARTITION BY emptype) AS paidonceortwicein2014,
BOOL_XOR(yearspaid='2014') OVER (PARTITION BY emptype) AS paidjustoncein2014
FROM employee;


  emptype   | paidtwicein2014 | paidonceortwicein2014 | paidjustoncein2014
------------+-----------------+-----------------------+--------------------
 contractor1 | t               | t                     | f
 contractor2 | f               | t                     | t
 contractor3 | f               | t                     | t
 contractor4 | t               | t                     | f
 contractor5 | f               | f                     | f
(5 rows)
```

See also

- BOOL_XOR [aggregate]

- BOOL_AND [analytic]
- BOOL_OR [analytic]
- Boolean data type

CONDITIONAL_CHANGE_EVENT [analytic]

Assigns an event window number to each row, starting from 0, and increments by 1 when the result of evaluating the argument expression on the current row differs from that on the previous row.

Behavior type

Immutable

Syntax

```
CONDITIONAL_CHANGE_EVENT ( expression ) OVER (
  [ window-partition-clause ]
  window-order-clause )
```

Parameters

expression

SQL scalar expression that is evaluated on an input record. The result of * expression *can be of any data type.

OVER()

See Analytic Functions.

Notes

The analytic *window-order-clause* is required but the *window-partition-clause* is optional.

Examples

```
=> SELECT CONDITIONAL_CHANGE_EVENT(bid)
       OVER (PARTITION BY symbol ORDER BY ts) AS cce
  FROM TickStore;
```

The system returns an error when no ORDER BY clause is present:

```
=> SELECT CONDITIONAL_CHANGE_EVENT(bid)
       OVER (PARTITION BY symbol) AS cce
  FROM TickStore;

ERROR:  conditional_change_event must contain an
ORDER BY clause within its analytic clause
```

For more examples, see Event-based windows .

See also

- CONDITIONAL_TRUE_EVENT [analytic]
- ROW_NUMBER [analytic]
- Time series analytics
- Event-based windows

CONDITIONAL_TRUE_EVENT [analytic]

Assigns an event window number to each row, starting from 0, and increments the number by 1 when the result of the boolean argument expression evaluates true. For example, given a sequence of values for column a, as follows:

```
( 1, 2, 3, 4, 5, 6 )
```

CONDITIONAL_TRUE_EVENT(a > 3) returns 0, 0, 0, 1, 2, 3 .

Behavior type

Immutable

Syntax

```
CONDITIONAL_TRUE_EVENT ( boolean-expression ) OVER (
  [ window-partition-clause ]
  window-order-clause )
```

Parameters

boolean-expression

SQL scalar expression that is evaluated on an input record, type BOOLEAN.

OVER()

See Analytic functions .

Notes

The analytic *window-order-clause* is required but the *window-partition-clause* is optional.

Examples

```
> SELECT CONDITIONAL_TRUE_EVENT(bid > 10.6)
    OVER(PARTITION BY bid ORDER BY ts) AS cte
  FROM Tickstore;
```

The system returns an error if the ORDER BY clause is omitted:

```
> SELECT CONDITIONAL_TRUE_EVENT(bid > 10.6)
     OVER(PARTITION BY bid) AS cte
  FROM Tickstore;

ERROR:  conditional_true_event must contain an ORDER BY
clause within its analytic clause
```

For more examples, see Event-based windows .

See also

- CONDITIONAL_CHANGE_EVENT [analytic]
- Time series analytics
- Event-based windows

COUNT [analytic]

Counts occurrences within a group within a window . If you specify * or some non-null constant, COUNT() counts all rows.

Behavior type

Immutable

Syntax

```
COUNT ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Returns the number of rows in each group for which the *expression* is not null. Can be any expression resulting in BIGINT.

OVER()

See Analytic Functions.

Examples

Using the schema defined in Window framing , the following COUNT function omits window order and window frame clauses; otherwise Vertica would treat it as a window aggregate. Think of the window of reporting aggregates as RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW .

```
=> SELECT deptno, sal, empno, COUNT(sal)
    OVER (PARTITION BY deptno) AS count FROM emp;

 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     2
     10 | 104 |     4 |     2
     20 | 110 |    10 |     6
     20 | 110 |     9 |     6
     20 | 109 |     7 |     6
     20 | 109 |     6 |     6
     20 | 109 |     8 |     6
     20 | 109 |    11 |     6
     30 | 105 |     5 |     3
     30 | 103 |     3 |     3
     30 | 102 |     2 |     3
```

Using ORDER BY sal creates a moving window query with default window: RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW .

```
=> SELECT deptno, sal, empno, COUNT(sal)
    OVER (PARTITION BY deptno ORDER BY sal) AS count
  FROM emp;

 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     4
     20 | 109 |     6 |     4
     20 | 109 |     8 |     4
     20 | 110 |    10 |     6
     20 | 110 |     9 |     6
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

Using the VMart schema, the following query finds the number of employees who make less than or equivalent to the hourly rate of the current employee. The query returns a running/cumulative average (sometimes called a moving average) using the default window of RANGE UNBOUNDED PRECEDING AND CURRENT ROW :

```
=> SELECT employee_last_name AS "last_name", hourly_rate, COUNT(*)
  OVER (ORDER BY hourly_rate) AS moving_count from employee_dimension;

 last_name  | hourly_rate | moving_count
------------+-------------+--------------
 Gauthier   |           6 |            4
 Taylor     |           6 |            4
 Jefferson  |           6 |            4
 Nielson    |           6 |            4
 McNulty    |        6.01 |           11
 Robinson   |        6.01 |           11
 Dobisz     |        6.01 |           11
 Williams   |        6.01 |           11
 Kramer     |        6.01 |           11
 Miller     |        6.01 |           11
 Wilson     |        6.01 |           11
 Vogel      |        6.02 |           14
 Moore      |        6.02 |           14
 Vogel      |        6.02 |           14
 Carcetti   |        6.03 |           19
...
```

To return a moving average that is not also a running (cumulative) average, the window should specify ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING :

```
=> SELECT employee_last_name AS "last_name", hourly_rate, COUNT(*)
    OVER (ORDER BY hourly_rate ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)
  AS moving_count from employee_dimension;
```

See also

- COUNT [aggregate]
- AVG [analytic]
- SUM [analytic]
- SQL analytics

CUME_DIST [analytic]

Calculates the cumulative distribution, or relative rank, of the current row with regard to other rows in the same partition within a window .

CUME_DIST() returns a number greater then 0 and less then or equal to 1, where the number represents the relative position of the specified row within a group of *n* rows. For a row x (assuming ASC ordering), the CUME_DIST of x is the number of rows with values lower than or equal to the value of x , divided by the number of rows in the partition. For example, in a group of three rows, the cumulative distribution values returned would be 1/3, 2/3, and 3/3.

> **Note**
> Because the result for a given row depends on the number of rows preceding that row in the same partition, you should always specify a *window-order-clause* when you call this function.

Behavior type

Immutable

Syntax

```
CUME_DIST ( ) OVER (
   [ window-partition-clause ]
   window-order-clause  )
```

Parameters

OVER()

See Analytic Functions.

Examples

The following example returns the cumulative distribution of sales for different transaction types within each month of the first quarter.

```
=> SELECT calendar_month_name AS month, tender_type, SUM(sales_quantity),
      CUME_DIST()
   OVER (PARTITION BY calendar_month_name ORDER BY SUM(sales_quantity)) AS
CUME_DIST
   FROM store.store_sales_fact JOIN date_dimension
   USING(date_key) WHERE calendar_month_name IN ('January','February','March')
   AND tender_type NOT LIKE 'Other'
   GROUP BY calendar_month_name, tender_type;


  month   | tender_type |  SUM   | CUME_DIST
----------+-------------+--------+-----------
 March    | Credit      | 469858 |      0.25
 March    | Cash        | 470449 |       0.5
 March    | Check       | 473033 |      0.75
 March    | Debit       | 475103 |         1
 January  | Cash        | 441730 |      0.25
 January  | Debit       | 443922 |       0.5
 January  | Check       | 446297 |      0.75
 January  | Credit      | 450994 |         1
 February | Check       | 425665 |      0.25
 February | Debit       | 426726 |       0.5
 February | Credit      | 430010 |      0.75
 February | Cash        | 430767 |         1
(12 rows)
```

See also

- PERCENT_RANK [analytic]
- PERCENTILE_DISC [analytic]
- SQL analytics

DENSE_RANK [analytic]

Within each window partition, ranks all rows in the query results set according to the order specified by the window's ORDER BY clause. A DENSE_RANK function returns a sequence of ranking numbers without any gaps.

DENSE_RANK executes as follows:

1. Sorts partition rows as specified by the ORDER BY clause.
2. Compares the ORDER BY values of the preceding row and current row and ranks the current row as follows:
   - If ORDER BY values are the same, the current row gets the same ranking as the preceding row.

     > **Note**
     > Null values are considered equal. For detailed information on how null values are sorted, see NULL sort order .

   - If the ORDER BY values are different, DENSE_RANK increments or decrements the current row's ranking by 1, depending whether sort order is ascending or descending.

DENSE_RANK always changes the ranking by 1, so no gaps appear in the ranking sequence. The largest rank value is the number of unique ORDER BY values returned by the query.

Behavior type

Immutable

Syntax

```
DENSE_RANK() OVER (
   [ window-partition-clause ]
   window-order-clause  )
```

Parameters

OVER()

See Analytic Functions.

See Analytic Functions

Compared with RANK

RANK leaves gaps in the ranking sequence, while DENSE_RANK does not. The example below compares the behavior of the two functions.

Examples

The following query invokes RANK and DENSE_RANK to rank customers by annual income. The two functions return different rankings, as follows:

- If annual_salary contains duplicate values, RANK() inserts duplicate rankings and then skips one or more values—for example, from 4 to 6 and 7 to 9.
- In the parallel column Dense Rank , DENSE_RANK() also inserts duplicate rankings, but leaves no gaps in the rankings sequence:

```
=>  SELECT employee_region region, employee_key, annual_salary,
    RANK() OVER (PARTITION BY employee_region ORDER BY annual_salary) Rank,
    DENSE_RANK() OVER (PARTITION BY employee_region ORDER BY annual_salary) "Dense Rank"
    FROM employee_dimension;
          region           | employee_key | annual_salary | Rank | Dense Rank
-----------------------------------+--------------+---------------+------+------------
 West                              |         5248 |          1200 |    1 |          1
 West                              |         6880 |          1204 |    2 |          2
 West                              |         5700 |          1214 |    3 |          3
 West                              |         9857 |          1218 |    4 |          4
 West                              |         6014 |          1218 |    4 |          4
 West                              |         9221 |          1220 |    6 |          5
 West                              |         7646 |          1222 |    7 |          6
 West                              |         6621 |          1222 |    7 |          6
 West                              |         6488 |          1224 |    9 |          7
 West                              |         7659 |          1226 |   10 |          8
 West                              |         7432 |          1226 |   10 |          8
 West                              |         9905 |          1226 |   10 |          8
 West                              |         9021 |          1228 |   13 |          9
...
 West                              |           56 |        963104 | 2794 |       2152
 West                              |          100 |        992363 | 2795 |       2153
 East                              |         8353 |          1200 |    1 |          1
 East                              |         9743 |          1202 |    2 |          2
 East                              |         9975 |          1202 |    2 |          2
 East                              |         9205 |          1204 |    4 |          3
 East                              |         8894 |          1206 |    5 |          4
 East                              |         7740 |          1206 |    5 |          4
 East                              |         7324 |          1208 |    7 |          5
 East                              |         6505 |          1208 |    7 |          5
 East                              |         5404 |          1208 |    7 |          5
 East                              |         5010 |          1208 |    7 |          5
 East                              |         9114 |          1212 |   11 |          6
...
```

See also

SQL analytics

EXPONENTIAL_MOVING_AVERAGE [analytic]

Calculates the exponential moving average (EMA) of expression *E* with smoothing factor *X* . An EMA differs from a simple moving average in that it provides a more stable picture of changes to data over time.

The EMA is calculated by adding the previous EMA value to the current data point scaled by the smoothing factor, as in the following formula:

*EMA = EMA0 + (X * (E - EMA0))*

where:

- *E* is the current data point
- *EMA0* is the previous row's EMA value.
- *X* is the smoothing factor.

This function also works at the row level. For example, EMA assumes the data in a given column is sampled at uniform intervals. If the users' data points are sampled at non-uniform intervals, they should run the time series gap filling and interpolation (GFI) operations before EMA()

Behavior type

Immutable

Syntax

```
EXPONENTIAL_MOVING_AVERAGE ( E, X ) OVER (
  [ window-partition-clause ]
  window-order-clause  )
```

Parameters

E

The value whose average is calculated over a set of rows. Can be INTEGER , FLOAT or NUMERIC type and must be a constant.

X

A positive FLOAT value between 0 and 1 that is used as the smoothing factor.

OVER()

See Analytic Functions.

Examples

The following example uses time series gap filling and interpolation (GFI) first in a subquery, and then performs an EXPONENTIAL_MOVING_AVERAGE operation on the subquery result.

Create a simple four-column table:

```
=> CREATE TABLE ticker(
   time TIMESTAMP,
   symbol VARCHAR(8),
   bid1 FLOAT,
   bid2 FLOAT );
```

Insert some data, including nulls, so GFI can do its interpolation and gap filling:

```
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:00', 'ABC', 60.45, 60.44);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:01', 'ABC', 60.49, 65.12);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:02', 'ABC', 57.78, 59.25);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:03', 'ABC', null, 65.12);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:04', 'ABC', 67.88, null);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:00', 'XYZ', 47.55, 40.15);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:01', 'XYZ', 44.35, 46.78);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:02', 'XYZ', 71.56, 75.78);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:03', 'XYZ', 85.55, 70.21);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:04', 'XYZ', 45.55, 58.65);
=> COMMIT;
```

> **Note**
> During gap filling and interpolation, Vertica takes the closest non null value on either side of the time slice and uses that value. For example, if you use a linear interpolation scheme and you do not specify IGNORE NULLS , and your data has one real value and one null, the result is null. If the value on either side is null, the result is null. See When Time Series Data Contains Nulls for details.

Query the table that you just created to you can see the output:

```
=> SELECT * FROM ticker;
       time         | symbol | bid1  | bid2
--------------------+--------+-------+-------
 2009-07-12 03:00:00 | ABC    | 60.45 | 60.44
 2009-07-12 03:00:01 | ABC    | 60.49 | 65.12
 2009-07-12 03:00:02 | ABC    | 57.78 | 59.25
 2009-07-12 03:00:03 | ABC    |       | 65.12
 2009-07-12 03:00:04 | ABC    | 67.88 |
 2009-07-12 03:00:00 | XYZ    | 47.55 | 40.15
 2009-07-12 03:00:01 | XYZ    | 44.35 | 46.78
 2009-07-12 03:00:02 | XYZ    | 71.56 | 75.78
 2009-07-12 03:00:03 | XYZ    | 85.55 | 70.21
 2009-07-12 03:00:04 | XYZ    | 45.55 | 58.65
(10 rows)
```

The following query processes the first and last values that belong to each 2-second time slice in table trades ' column a . The query then calculates the exponential moving average of expression fv and lv with a smoothing factor of 50%:

```
=> SELECT symbol, slice_time, fv, lv,
    EXPONENTIAL_MOVING_AVERAGE(fv, 0.5)
     OVER (PARTITION BY symbol ORDER BY slice_time) AS ema_first,
  EXPONENTIAL_MOVING_AVERAGE(lv, 0.5)
     OVER (PARTITION BY symbol ORDER BY slice_time) AS ema_last
  FROM (
    SELECT symbol, slice_time,
      TS_FIRST_VALUE(bid1 IGNORE NULLS) as fv,
      TS_LAST_VALUE(bid2 IGNORE NULLS) AS lv
     FROM ticker TIMESERIES slice_time AS '2 seconds'
     OVER (PARTITION BY symbol ORDER BY time) ) AS sq;


 symbol |     slice_time      |  fv   |  lv   | ema_first | ema_last
--------+---------------------+-------+-------+-----------+----------
 ABC    | 2009-07-12 03:00:00 | 60.45 | 65.12 |     60.45 |    65.12
 ABC    | 2009-07-12 03:00:02 | 57.78 | 65.12 |    59.115 |    65.12
 ABC    | 2009-07-12 03:00:04 | 67.88 | 65.12 |   63.4975 |    65.12
 XYZ    | 2009-07-12 03:00:00 | 47.55 | 46.78 |     47.55 |    46.78
 XYZ    | 2009-07-12 03:00:02 | 71.56 | 70.21 |    59.555 |   58.495
 XYZ    | 2009-07-12 03:00:04 | 45.55 | 58.65 |   52.5525 |  58.5725
(6 rows)
```

See also

- TIMESERIES clause
- Time series analytics
- SQL analytics

FIRST_VALUE [analytic]

Lets you select the first value of a table or partition (determined by the *window-order-clause*) without having to use a self join. This function is useful when you want to use the first value as a baseline in calculations.

Use FIRST_VALUE() with the *window-order-clause* to produce deterministic results. If no window is specified for the current row, the default window is UNBOUNDED PRECEDING AND CURRENT ROW .

Behavior type

Immutable

Syntax

```
FIRST_VALUE ( expression [ IGNORE NULLS ] ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Expression to evaluate—or example, a constant, column, nonanalytic function, function expression, or expressions involving any of these.

IGNORE NULLS

Specifies to return the first non-null value in the set, or NULL if all values are NULL . If you omit this option and the first value in the set is null, the function returns NULL .

OVER()

See Analytic Functions.

Examples

The following query asks for the first value in the partitioned day of week, and illustrates the potential nondeterministic nature of FIRST_VALUE() :

```
=> SELECT calendar_year, date_key, day_of_week, full_date_description,
  FIRST_VALUE(full_date_description)
    OVER(PARTITION BY calendar_month_number_in_year ORDER BY day_of_week)
    AS "first_value"
  FROM date_dimension
  WHERE calendar_year=2003 AND calendar_month_number_in_year=1;
```

The first value returned is January 31, 2003; however, the next time the same query is run, the first value might be January 24 or January 3, or the 10th or 17th. This is because the analytic ORDER BY column day_of_week returns rows that contain ties (multiple Fridays). These repeated values make the ORDER BY evaluation result nondeterministic, because rows that contain ties can be ordered in any way, and any one of those rows qualifies as being the first value of day_of_week .

```
calendar_year | date_key | day_of_week | full_date_description |    first_value
--------------+----------+-------------+-----------------------+------------------
         2003 |       31 | Friday      | January 31, 2003      | January 31, 2003
         2003 |       24 | Friday      | January 24, 2003      | January 31, 2003
         2003 |        3 | Friday      | January 3, 2003       | January 31, 2003
         2003 |       10 | Friday      | January 10, 2003      | January 31, 2003
         2003 |       17 | Friday      | January 17, 2003      | January 31, 2003
         2003 |        6 | Monday      | January 6, 2003       | January 31, 2003
         2003 |       27 | Monday      | January 27, 2003      | January 31, 2003
         2003 |       13 | Monday      | January 13, 2003      | January 31, 2003
         2003 |       20 | Monday      | January 20, 2003      | January 31, 2003
         2003 |       11 | Saturday    | January 11, 2003      | January 31, 2003
         2003 |       18 | Saturday    | January 18, 2003      | January 31, 2003
         2003 |       25 | Saturday    | January 25, 2003      | January 31, 2003
         2003 |        4 | Saturday    | January 4, 2003       | January 31, 2003
         2003 |       12 | Sunday      | January 12, 2003      | January 31, 2003
         2003 |       26 | Sunday      | January 26, 2003      | January 31, 2003
         2003 |        5 | Sunday      | January 5, 2003       | January 31, 2003
         2003 |       19 | Sunday      | January 19, 2003      | January 31, 2003
         2003 |       23 | Thursday    | January 23, 2003      | January 31, 2003
         2003 |        2 | Thursday    | January 2, 2003       | January 31, 2003
         2003 |        9 | Thursday    | January 9, 2003       | January 31, 2003
         2003 |       16 | Thursday    | January 16, 2003      | January 31, 2003
         2003 |       30 | Thursday    | January 30, 2003      | January 31, 2003
         2003 |       21 | Tuesday     | January 21, 2003      | January 31, 2003
         2003 |       14 | Tuesday     | January 14, 2003      | January 31, 2003
         2003 |        7 | Tuesday     | January 7, 2003       | January 31, 2003
         2003 |       28 | Tuesday     | January 28, 2003      | January 31, 2003
         2003 |       22 | Wednesday   | January 22, 2003      | January 31, 2003
         2003 |       29 | Wednesday   | January 29, 2003      | January 31, 2003
         2003 |       15 | Wednesday   | January 15, 2003      | January 31, 2003
         2003 |        1 | Wednesday   | January 1, 2003       | January 31, 2003
         2003 |        8 | Wednesday   | January 8, 2003       | January 31, 2003
(31 rows)
```

Note

> The day_of_week results are returned in alphabetical order because of lexical rules. The fact that each day does not appear ordered by the 7-day week cycle (for example, starting with Sunday followed by Monday, Tuesday, and so on) has no affect on results.

To return deterministic results, modify the query so that it performs its analytic ORDER BY operations on a **unique** field, such as date_key :

```
=> SELECT calendar_year, date_key, day_of_week, full_date_description,
  FIRST_VALUE(full_date_description) OVER
    (PARTITION BY calendar_month_number_in_year ORDER BY date_key) AS "first_value"
  FROM date_dimension WHERE calendar_year=2003;
```

FIRST_VALUE() returns a first value of January 1 for the January partition and the first value of February 1 for the February partition. Also, the full_date_description column contains no ties:

```
 calendar_year | date_key | day_of_week | full_date_description | first_value
---------------+----------+-------------+-----------------------+-------------
          2003 |        1 | Wednesday   | January 1, 2003       | January 1, 2003
          2003 |        2 | Thursday    | January 2, 2003       | January 1, 2003
          2003 |        3 | Friday      | January 3, 2003       | January 1, 2003
          2003 |        4 | Saturday    | January 4, 2003       | January 1, 2003
          2003 |        5 | Sunday      | January 5, 2003       | January 1, 2003
          2003 |        6 | Monday      | January 6, 2003       | January 1, 2003
          2003 |        7 | Tuesday     | January 7, 2003       | January 1, 2003
          2003 |        8 | Wednesday   | January 8, 2003       | January 1, 2003
          2003 |        9 | Thursday    | January 9, 2003       | January 1, 2003
          2003 |       10 | Friday      | January 10, 2003      | January 1, 2003
          2003 |       11 | Saturday    | January 11, 2003      | January 1, 2003
          2003 |       12 | Sunday      | January 12, 2003      | January 1, 2003
          2003 |       13 | Monday      | January 13, 2003      | January 1, 2003
          2003 |       14 | Tuesday     | January 14, 2003      | January 1, 2003
          2003 |       15 | Wednesday   | January 15, 2003      | January 1, 2003
          2003 |       16 | Thursday    | January 16, 2003      | January 1, 2003
          2003 |       17 | Friday      | January 17, 2003      | January 1, 2003
          2003 |       18 | Saturday    | January 18, 2003      | January 1, 2003
          2003 |       19 | Sunday      | January 19, 2003      | January 1, 2003
          2003 |       20 | Monday      | January 20, 2003      | January 1, 2003
          2003 |       21 | Tuesday     | January 21, 2003      | January 1, 2003
          2003 |       22 | Wednesday   | January 22, 2003      | January 1, 2003
          2003 |       23 | Thursday    | January 23, 2003      | January 1, 2003
          2003 |       24 | Friday      | January 24, 2003      | January 1, 2003
          2003 |       25 | Saturday    | January 25, 2003      | January 1, 2003
          2003 |       26 | Sunday      | January 26, 2003      | January 1, 2003
          2003 |       27 | Monday      | January 27, 2003      | January 1, 2003
          2003 |       28 | Tuesday     | January 28, 2003      | January 1, 2003
          2003 |       29 | Wednesday   | January 29, 2003      | January 1, 2003
          2003 |       30 | Thursday    | January 30, 2003      | January 1, 2003
          2003 |       31 | Friday      | January 31, 2003      | January 1, 2003
          2003 |       32 | Saturday    | February 1, 2003      | February 1, 2003
          2003 |       33 | Sunday      | February 2, 2003      | February 1,2003
     ...
(365 rows)
```

See also

- LAST_VALUE [analytic]
- TIME_SLICE
- SQL analytics

LAG [analytic]

Returns the value of the input expression at the given offset before the current row within a window . This function lets you access more than one row in a table at the same time. This is useful for comparing values when the relative positions of rows can be reliably known. It also lets you avoid the more costly self join, which enhances query processing speed.

For information on getting the rows that follow, see LEAD .

Behavior type

Immutable

Syntax

```
LAG ( expression[, offset ] [, default ] ) OVER (
   [ window-partition-clause ]
   window-order-clause )
```

Parameters

expression

The expression to evaluate—for example, a constant, column, non-analytic function, function expression, or expressions involving any of these.

offset

Indicates how great is the lag. The default value is 1 (the previous row). This parameter must evaluate to a constant positive integer.

default

The value returned if *offset* falls outside the bounds of the table or partition. This value must be a constant value or an expression that can be evaluated to a constant; its data type is coercible to that of the first argument.

Examples

This example sums the current balance by date in a table and also sums the previous balance from the last day. Given the inputs that follow, the data satisfies the following conditions:

- For each some_id , there is exactly 1 row for each date represented by month_date .
- For each some_id , the set of dates is consecutive; that is, if there is a row for February 24 and a row for February 26, there would also be a row for February 25.
- Each some_id has the same set of dates.

```
=> CREATE TABLE balances (
     month_date DATE,
     current_bal INT,
     some_id INT);
=> INSERT INTO balances values ('2009-02-24', 10, 1);
=> INSERT INTO balances values ('2009-02-25', 10, 1);
=> INSERT INTO balances values ('2009-02-26', 10, 1);
=> INSERT INTO balances values ('2009-02-24', 20, 2);
=> INSERT INTO balances values ('2009-02-25', 20, 2);
=> INSERT INTO balances values ('2009-02-26', 20, 2);
=> INSERT INTO balances values ('2009-02-24', 30, 3);
=> INSERT INTO balances values ('2009-02-25', 20, 3);
=> INSERT INTO balances values ('2009-02-26', 30, 3);
```

Now run LAG to sum the current balance for each date and sum the previous balance from the last day:

```
=> SELECT month_date,
    SUM(current_bal) as current_bal_sum,
    SUM(previous_bal) as previous_bal_sum FROM
     (SELECT month_date, current_bal,
    LAG(current_bal, 1, 0) OVER
     (PARTITION BY some_id ORDER BY month_date)
    AS previous_bal FROM balances) AS subQ
    GROUP BY month_date ORDER BY month_date;
month_date  | current_bal_sum | previous_bal_sum
------------+-----------------+------------------
 2009-02-24 |              60 |                0
 2009-02-25 |              50 |               60
 2009-02-26 |              60 |               50
(3 rows)
```

Using the same example data, the following query would not be allowed because LAG is nested inside an aggregate function:

```
=> SELECT month_date,
    SUM(current_bal) as current_bal_sum,
  SUM(LAG(current_bal, 1, 0) OVER
     (PARTITION BY some_id ORDER BY month_date)) AS previous_bal_sum
  FROM some_table GROUP BY month_date ORDER BY month_date;
```

The following example uses the VMart database . LAG first returns the annual income from the previous row, and then it calculates the difference between the income in the current row from the income in the previous row:

```
=> SELECT occupation, customer_key, customer_name, annual_income,
  LAG(annual_income, 1, 0) OVER (PARTITION BY occupation
  ORDER BY annual_income) AS prev_income, annual_income -
  LAG(annual_income, 1, 0) OVER (PARTITION BY occupation
  ORDER BY annual_income) AS difference
  FROM customer_dimension ORDER BY occupation, customer_key LIMIT 20;
 occupation | customer_key |    customer_name     | annual_income | prev_income | difference
------------+--------------+----------------------+---------------+-------------+------------
 Accountant |           15 | Midori V. Peterson   |        692610 |      692535 |         75
 Accountant |           43 | Midori S. Rodriguez  |        282359 |      280976 |       1383
 Accountant |           93 | Robert P. Campbell   |        471722 |      471355 |        367
 Accountant |          102 | Sam T. McNulty       |        901636 |      901561 |         75
 Accountant |          134 | Martha B. Overstreet |        705146 |      704335 |        811
 Accountant |          165 | James C. Kramer      |        376841 |      376474 |        367
 Accountant |          225 | Ben W. Farmer        |         70574 |       70449 |        125
 Accountant |          270 | Jessica S. Lang      |        684204 |      682274 |       1930
 Accountant |          273 | Mark X. Lampert      |        723294 |      722737 |        557
 Accountant |          295 | Sharon K. Gauthier   |         29033 |       28412 |        621
 Accountant |          338 | Anna S. Jackson      |        816858 |      815557 |       1301
 Accountant |          377 | William I. Jones     |        915149 |      914872 |        277
 Accountant |          438 | Joanna A. McCabe     |        147396 |      144482 |       2914
 Accountant |          452 | Kim P. Brown         |        126023 |      124797 |       1226
 Accountant |          467 | Meghan K. Carcetti   |        810528 |      810284 |        244
 Accountant |          478 | Tanya E. Greenwood   |        639649 |      639029 |        620
 Accountant |          511 | Midori P. Vogel      |        187246 |      185539 |       1707
 Accountant |          525 | Alexander K. Moore   |        677433 |      677050 |        383
 Accountant |          550 | Sam P. Reyes         |        735691 |      735355 |        336
 Accountant |          577 | Robert U. Vu         |        616101 |      615439 |        662
(20 rows)
```

The next example uses LEAD and LAG to return the third row after the salary in the current row and fifth salary before the salary in the current row:

```
=> SELECT hire_date, employee_key, employee_last_name,
  LEAD(hire_date, 1) OVER (ORDER BY hire_date) AS "next_hired" ,
  LAG(hire_date, 1) OVER (ORDER BY hire_date) AS "last_hired"
  FROM employee_dimension ORDER BY hire_date, employee_key;
 hire_date  | employee_key | employee_last_name | next_hired | last_hired
------------+--------------+--------------------+------------+------------
 1956-04-11 |         2694 | Farmer             | 1956-05-12 |
 1956-05-12 |         5486 | Winkler            | 1956-09-18 | 1956-04-11
 1956-09-18 |         5525 | McCabe             | 1957-01-15 | 1956-05-12
 1957-01-15 |          560 | Greenwood          | 1957-02-06 | 1956-09-18
 1957-02-06 |         9781 | Bauer              | 1957-05-25 | 1957-01-15
 1957-05-25 |         9506 | Webber             | 1957-07-04 | 1957-02-06
 1957-07-04 |         6723 | Kramer             | 1957-07-07 | 1957-05-25
 1957-07-07 |         5827 | Garnett            | 1957-11-11 | 1957-07-04
 1957-11-11 |          373 | Reyes              | 1957-11-21 | 1957-07-07
 1957-11-21 |         3874 | Martin             | 1958-02-06 | 1957-11-11
(10 rows)
```

See also

- [LEAD](/sql-reference/functions/analytic-functions/lead-analytic.html)
- SQL analytics

LAST_VALUE [analytic]

Lets you select the last value of a table or partition (determined by the *window-order-clause*) without having to use a self join. LAST_VALUE takes the last record from the partition after the window order clause. The function then computes the expression against the last record, and returns the results. This function is useful when you want to use the last value as a baseline in calculations.

Use LAST_VALUE() with the *window-order-clause* to produce deterministic results. If no window is specified for the current row, the default window is UNBOUNDED PRECEDING AND CURRENT ROW .

> **Tip**
> Due to default window semantics, LAST_VALUE does not always return the last value of a partition. If you omit *window-frame-clause* from the analytic clause, LAST_VALUE operates on this default window. Although results can seem non-intuitive by not returning the bottom of the current partition, it returns the bottom of the window, which continues to change along with the current input row being processed. If you want to return the last value of a partition, use UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING . See examples below.

Behavior type

Immutable

Syntax

```
LAST_VALUE ( expression [ IGNORE NULLS ] ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Expression to evaluate—for example, a constant, column, nonanalytic function, function expression, or expressions involving any of these.

IGNORE NULLS

Specifies to return the last non-null value in the set, or NULL if all values are NULL . If you omit this option and the last value in the set is null, the function returns NULL .

OVER()

See Analytic Functions.

Examples

Using the schema defined in Window framing in Analyzing Data, the following query does not show the highest salary value by department; instead it shows the highest salary value by department by salary.

```
=> SELECT deptno, sal, empno, LAST_VALUE(sal)
     OVER (PARTITION BY deptno ORDER BY sal) AS lv
   FROM emp;
 deptno | sal | empno |    lv
--------+-----+-------+--------
     10 | 101 |     1 |    101
     10 | 104 |     4 |    104
     20 | 100 |    11 |    100
     20 | 109 |     7 |    109
     20 | 109 |     6 |    109
     20 | 109 |     8 |    109
     20 | 110 |    10 |    110
     20 | 110 |     9 |    110
     30 | 102 |     2 |    102
     30 | 103 |     3 |    103
     30 | 105 |     5 |    105
```

If you include the window frame clause ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING , LAST_VALUE() returns the highest salary by department, an accurate representation of the information:

```
=> SELECT deptno, sal, empno, LAST_VALUE(sal)
     OVER (PARTITION BY deptno ORDER BY sal
          ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS lv
  FROM emp;
 deptno | sal | empno |    lv
--------+-----+-------+--------
     10 | 101 |     1 |    104
     10 | 104 |     4 |    104
     20 | 100 |    11 |    110
     20 | 109 |     7 |    110
     20 | 109 |     6 |    110
     20 | 109 |     8 |    110
     20 | 110 |    10 |    110
     20 | 110 |     9 |    110
     30 | 102 |     2 |    105
     30 | 103 |     3 |    105
     30 | 105 |     5 |    105
```

For more examples, see FIRST_VALUE() .

See also

- FIRST_VALUE [analytic]
- TIME_SLICE
- SQL analytics

LEAD [analytic]

Returns values from the row after the current row within a window , letting you access more than one row in a table at the same time. This is useful for comparing values when the relative positions of rows can be reliably known. It also lets you avoid the more costly self join, which enhances query processing speed.

Behavior type

Immutable

Syntax

```
LEAD ( expression[, offset ] [, default ] ) OVER (
   [ window-partition-clause ]
   window-order-clause )
```

Parameters

expression

The expression to evaluate—for example, a constant, column, non-analytic function, function expression, or expressions involving any of these.

offset

Is an optional parameter that defaults to 1 (the next row). This parameter must evaluate to a constant positive integer.

default

The value returned if *offset* falls outside the bounds of the table or partition. This value must be a constant value or an expression that can be evaluated to a constant; its data type is coercible to that of the first argument.

Examples

LEAD finds the hire date of the employee hired just after the current row:

```
=> SELECT employee_region, hire_date, employee_key, employee_last_name,
   LEAD(hire_date, 1) OVER (PARTITION BY employee_region ORDER BY hire_date) AS "next_hired"
   FROM employee_dimension ORDER BY employee_region, hire_date, employee_key;
 employee_region  | hire_date  | employee_key | employee_last_name | next_hired
------------------+------------+--------------+--------------------+------------
 East             | 1956-04-08 |         9218 | Harris             | 1957-02-06
 East             | 1957-02-06 |         7799 | Stein              | 1957-05-25
 East             | 1957-05-25 |         3687 | Farmer             | 1957-06-26
 East             | 1957-06-26 |         9474 | Bauer              | 1957-08-18
 East             | 1957-08-18 |          570 | Jefferson          | 1957-08-24
 East             | 1957-08-24 |         4363 | Wilson             | 1958-02-17
 East             | 1958-02-17 |         6457 | McCabe             | 1958-06-26
 East             | 1958-06-26 |         6196 | Li                 | 1958-07-16
 East             | 1958-07-16 |         7749 | Harris             | 1958-09-18
 East             | 1958-09-18 |         9678 | Sanchez            | 1958-11-10
(10 rows)
```

The next example uses LEAD and LAG to return the third row after the salary in the current row and fifth salary before the salary in the current row.

```
=> SELECT hire_date, employee_key, employee_last_name,
   LEAD(hire_date, 1) OVER (ORDER BY hire_date) AS "next_hired" ,
   LAG(hire_date, 1) OVER (ORDER BY hire_date) AS "last_hired"
   FROM employee_dimension ORDER BY hire_date, employee_key;
 hire_date  | employee_key | employee_last_name | next_hired | last_hired
------------+--------------+--------------------+------------+------------
 1956-04-11 |         2694 | Farmer             | 1956-05-12 |
 1956-05-12 |         5486 | Winkler            | 1956-09-18 | 1956-04-11
 1956-09-18 |         5525 | McCabe             | 1957-01-15 | 1956-05-12
 1957-01-15 |          560 | Greenwood          | 1957-02-06 | 1956-09-18
 1957-02-06 |         9781 | Bauer              | 1957-05-25 | 1957-01-15
 1957-05-25 |         9506 | Webber             | 1957-07-04 | 1957-02-06
 1957-07-04 |         6723 | Kramer             | 1957-07-07 | 1957-05-25
 1957-07-07 |         5827 | Garnett            | 1957-11-11 | 1957-07-04
 1957-11-11 |          373 | Reyes              | 1957-11-21 | 1957-07-07
 1957-11-21 |         3874 | Martin             | 1958-02-06 | 1957-11-11
(10 rows)
```

The following example returns employee name and salary, along with the next highest and lowest salaries.

```
=> SELECT employee_last_name, annual_salary,
      NVL(LEAD(annual_salary) OVER (ORDER BY annual_salary),
        MIN(annual_salary) OVER()) "Next Highest",
      NVL(LAG(annual_salary) OVER (ORDER BY annual_salary),
        MAX(annual_salary)  OVER()) "Next Lowest"
   FROM employee_dimension;
 employee_last_name | annual_salary | Next Highest | Next Lowest
--------------------+---------------+--------------+-------------
 Nielson            |          1200 |         1200 |      995533
 Lewis              |          1200 |         1200 |        1200
 Harris             |          1200 |         1202 |        1200
 Robinson           |          1202 |         1202 |        1200
 Garnett            |          1202 |         1202 |        1202
 Weaver             |          1202 |         1202 |        1202
 Nielson            |          1202 |         1202 |        1202
 McNulty            |          1202 |         1204 |        1202
 Farmer             |          1204 |         1204 |        1202
 Martin             |          1204 |         1204 |        1204
(10 rows)
```

The next example returns, for each assistant director in the employees table, the hire date of the director hired just after the director on the current row. For example, Jackson was hired on 2016-12-28, and the next director hired was Bauer:

```
=> SELECT employee_last_name, hire_date,
     LEAD(hire_date, 1) OVER (ORDER BY hire_date DESC) as "NextHired"
  FROM employee_dimension WHERE job_title = 'Assistant Director';
 employee_last_name | hire_date  | NextHired
--------------------+------------+------------
 Jackson            | 2016-12-28 | 2016-12-26
 Bauer              | 2016-12-26 | 2016-12-11
 Miller             | 2016-12-11 | 2016-12-07
 Fortin             | 2016-12-07 | 2016-11-27
 Harris             | 2016-11-27 | 2016-11-15
 Goldberg           | 2016-11-15 |
(5 rows)
```

See also

- LAG
- SQL analytics

MAX [analytic]

Returns the maximum value of an expression within a window . The return value has the same type as the expression data type.

The analytic functions MIN() and MAX() can operate with Boolean values. The MAX() function acts upon a Boolean data type or a value that can be implicitly converted to a Boolean value. If at least one input value is true, MAX() returns t (true). Otherwise, it returns f (false). In the same scenario, the MIN() function returns t (true) if all input values are true. Otherwise, it returns f .

Behavior type

Immutable

Syntax

```
MAX ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Any expression for which the maximum value is calculated, typically a column reference .

OVER()

See Analytic Functions.

Examples

The following query computes the deviation between the employees' annual salary and the maximum annual salary in Massachusetts:

```
=> SELECT employee_state, annual_salary,
    MAX(annual_salary)
     OVER(PARTITION BY employee_state ORDER BY employee_key) max,
       annual_salary- MAX(annual_salary)
     OVER(PARTITION BY employee_state ORDER BY employee_key) diff
  FROM employee_dimension
  WHERE employee_state = 'MA';
 employee_state | annual_salary |  max   |  diff
----------------+---------------+--------+---------
 MA             |          1918 | 995533 | -993615
 MA             |          2058 | 995533 | -993475
 MA             |          2586 | 995533 | -992947
 MA             |          2500 | 995533 | -993033
 MA             |          1318 | 995533 | -994215
 MA             |          2072 | 995533 | -993461
 MA             |          2656 | 995533 | -992877
 MA             |          2148 | 995533 | -993385
 MA             |          2366 | 995533 | -993167
 MA             |          2664 | 995533 | -992869
(10 rows)
```

The following example shows you the difference between the MIN and MAX analytic functions when you use them with a Boolean value. The sample creates a table with two columns, adds two rows of data, and shows sample output for MIN and MAX .

```
CREATE TABLE min_max_functions (emp VARCHAR, torf BOOL);

INSERT INTO min_max_functions VALUES ('emp1', 1);
INSERT INTO min_max_functions VALUES ('emp1', 0);

SELECT DISTINCT emp,
min(torf) OVER (PARTITION BY emp) AS worksasbooleanand,
Max(torf) OVER (PARTITION BY emp) AS worksasbooleanor
FROM min_max_functions;

 emp  | worksasbooleanand | worksasbooleanor
------+-------------------+------------------
 emp1 | f                 | t
(1 row)
```

See also

- SQL analytics
- MAX [aggregate]
- MIN [analytic]

MEDIAN [analytic]

For each row, returns the median value of a value set within each partition. MEDIAN determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

MEDIAN is an alias of PERCENTILE_CONT [analytic] with an argument of 0.5 (50%).

Behavior type

Immutable

Syntax

```
MEDIAN ( expression ) OVER ( [ window-partition-clause ] )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the middle value or an interpolated value that would be the middle value once the values are sorted. Null values are ignored in the calculation.

OVER()

If the OVER clause specifies *window-partition-clause* , MEDIAN groups input rows according to one or more columns or expressions. If this clause is omitted, no grouping occurs and MEDIAN processes all input rows as a single partition.

Examples

See Calculating a median value

See also

- PERCENTILE_CONT [analytic]
- SQL analytics

MIN [analytic]

Returns the minimum value of an expression within a window . The return value has the same type as the expression data type.

The analytic functions MIN() and MAX() can operate with Boolean values. The MAX() function acts upon a Boolean data type or a value that can be implicitly converted to a Boolean value. If at least one input value is true, MAX() returns t (true). Otherwise, it returns f (false). In the same scenario, the MIN() function returns t (true) if all input values are true. Otherwise, it returns f .

Behavior type

Immutable

Syntax

```
MIN ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Any expression for which the minimum value is calculated, typically a column reference .

OVER()

See Analytic Functions.

Examples

The following example shows how you can query to determine the deviation between the employees' annual salary and the minimum annual salary in Massachusetts:

```
=> SELECT employee_state, annual_salary,
     MIN(annual_salary)
     OVER(PARTITION BY employee_state ORDER BY employee_key) min,
       annual_salary- MIN(annual_salary)
     OVER(PARTITION BY employee_state ORDER BY employee_key) diff
   FROM employee_dimension
   WHERE employee_state = 'MA';
 employee_state | annual_salary | min  | diff
----------------+---------------+------+------
 MA             |          1918 | 1204 |  714
 MA             |          2058 | 1204 |  854
 MA             |          2586 | 1204 | 1382
 MA             |          2500 | 1204 | 1296
 MA             |          1318 | 1204 |  114
 MA             |          2072 | 1204 |  868
 MA             |          2656 | 1204 | 1452
 MA             |          2148 | 1204 |  944
 MA             |          2366 | 1204 | 1162
 MA             |          2664 | 1204 | 1460
(10 rows)
```

The following example shows you the difference between the MIN and MAX analytic functions when you use them with a Boolean value. The sample creates a table with two columns, adds two rows of data, and shows sample output for MIN and MAX .

```
CREATE TABLE min_max_functions (emp VARCHAR, torf BOOL);

INSERT INTO min_max_functions VALUES ('emp1', 1);
INSERT INTO min_max_functions VALUES ('emp1', 0);

SELECT DISTINCT emp,
min(torf) OVER (PARTITION BY emp) AS worksasbooleanand,
Max(torf) OVER (PARTITION BY emp) AS worksasbooleanor
FROM min_max_functions;

 emp  | worksasbooleanand | worksasbooleanor
------+-------------------+------------------
 emp1 | f                 | t
(1 row)
```

See also

- SQL analytics
- MIN [aggregate]
- MAX [analytic]

NTH_VALUE [analytic]

Returns the value evaluated at the row that is the * n *th row of the window (counting from 1). If the specified row does not exist, NTH_VALUE returns NULL .

Behavior type

Immutable

Syntax

```
NTH_VALUE ( expression, row-number [ IGNORE NULLS ] ) OVER (
   [ window-frame-clause ]
   [ window-order-clause ])
```

Parameters

expression

Expression to evaluate. The expression can be a constant, column name, nonanalytic function, function expression, or expressions that include any of these.

row-number

Specifies the row to evaluate, where *row-number* evaluates to an integer ≥ 1.

IGNORE NULLS

Specifies to return the first non- NULL value in the set, or NULL if all values are NULL .

OVER()

See Analytic Functions.

Examples

In the following example, for each tuple (current row) in table t1 , the window frame clause defines the window as follows:

```
ORDER BY b ROWS BETWEEN 3 PRECEDING AND CURRENT ROW
```

For each window, *n* for * n *th value is a+1 . a is the value of column a in the tuple.

NTH_VALUE returns the result of the expression b+1 , where b is the value of column b in the * n *th row, which is the a+1 row within the window.

```
=> SELECT * FROM t1 ORDER BY a;
 a | b
---+----
 1 | 10
 2 | 20
 2 | 21
 3 | 30
 4 | 40
 5 | 50
 6 | 60
(7 rows)

=> SELECT NTH_VALUE(b+1, a+1) OVER
    (ORDER BY b ROWS BETWEEN 3 PRECEDING AND CURRENT ROW) FROM t1;
 ?column?
----------



      22
      31




(7 rows)
```

NTILE [analytic]

Equally divides an ordered data set (partition) into a { *value* } number of subsets within a window , where the subsets are numbered 1 through the value in parameter *constant-value* . For example, if *constant-value* = 4 and the partition contains 20 rows, NTILE divides the partition rows into four equal subsets of five rows. NTILE assigns each row to a subset by giving row a number from 1 to 4. The rows in the first subset are assigned 1, the next five are assigned 2, and so on.

If the number of partition rows is not evenly divisible by the number of subsets, the rows are distributed so no subset is more than one row larger than any other subset, and the lowest subsets have extra rows. For example, if *constant-value* = 4 and the number of rows = 21, the first subset has six rows, the second subset has five rows, and so on.

If the number of subsets is greater than the number of rows, then a number of subsets equal to the number of rows is filled, and the remaining subsets are empty.

Behavior type

Immutable

Syntax

```
NTILE ( constant-value ) OVER (
   [ window-partition-clause ]
   window-order-clause )
```

Parameters

constant-value

Specifies the number of subsets , where *constant-value* must resolve to a positive constant for each partition.

OVER()

See Analytic Functions.

Examples

The following query assigns each month's sales total into one of four subsets:

```
=> SELECT calendar_month_name AS MONTH, SUM(sales_quantity),
     NTILE(4) OVER (ORDER BY SUM(sales_quantity)) AS NTILE
   FROM store.store_sales_fact JOIN date_dimension
   USING(date_key)
   GROUP BY calendar_month_name
   ORDER BY NTILE;
   MONTH   |   SUM   | NTILE
-----------+---------+-------
 November  | 2040726 |     1
 June      | 2088528 |     1
 February  | 2134708 |     1
 April     | 2181767 |     2
 January   | 2229220 |     2
 October   | 2316363 |     2
 September | 2323914 |     3
 March     | 2354409 |     3
 August    | 2387017 |     3
 July      | 2417239 |     4
 May       | 2492182 |     4
 December  | 2531842 |     4
(12 rows)
```

See also

- PERCENTILE_CONT [analytic]
- WIDTH_BUCKET
- SQL analytics

PERCENT_RANK [analytic]

Calculates the relative rank of a row for a given row in a group within a window by dividing that row's rank less 1 by the number of rows in the partition, also less 1. PERCENT_RANK always returns values from 0 to 1 inclusive. The first row in any set has a PERCENT_RANK of 0. The return value is NUMBER .

```
( rank - 1 ) / ( [ rows ] - 1 )
```

In the preceding formula, rank is the rank position of a row in the group and rows is the total number of rows in the partition defined by the OVER() clause.

Behavior type

Immutable

Syntax

```
PERCENT_RANK ( ) OVER (
   [ window-partition-clause ]
   window-order-clause  )
```

Parameters

OVER()

See Analytic Functions

Examples

The following example finds the percent rank of gross profit for different states within each month of the first quarter:

```
=> SELECT calendar_month_name AS MONTH, store_state,
     SUM(gross_profit_dollar_amount),
     PERCENT_RANK() OVER (PARTITION BY calendar_month_name
     ORDER BY SUM(gross_profit_dollar_amount)) AS PERCENT_RANK
   FROM store.store_sales_fact JOIN date_dimension
   USING(date_key)
   JOIN store.store_dimension
   USING (store_key)
   WHERE calendar_month_name IN ('January','February','March')
   AND store_state IN ('OR','IA','DC','NV','WI')
   GROUP BY calendar_month_name, store_state
   ORDER BY calendar_month_name, PERCENT_RANK;
  MONTH   | store_state |  SUM   | PERCENT_RANK
----------+-------------+--------+--------------
 February | IA          | 418490 |            0
 February | OR          | 460588 |         0.25
 February | DC          | 616553 |          0.5
 February | WI          | 619204 |         0.75
 February | NV          | 838039 |            1
 January  | OR          | 446528 |            0
 January  | IA          | 474501 |         0.25
 January  | DC          | 628496 |          0.5
 January  | WI          | 679382 |         0.75
 January  | NV          | 871824 |            1
 March    | IA          | 460282 |            0
 March    | OR          | 481935 |         0.25
 March    | DC          | 716063 |          0.5
 March    | WI          | 771575 |         0.75
 March    | NV          | 970878 |            1
(15 rows)
```

The following example calculates, for each employee, the percent rank of the employee's salary by their job title:

```
=> SELECT job_title, employee_last_name, annual_salary,
     PERCENT_RANK()
    OVER (PARTITION BY job_title ORDER BY annual_salary DESC) AS percent_rank
  FROM employee_dimension
  ORDER BY percent_rank, annual_salary;
    job_title      | employee_last_name | annual_salary |    percent_rank
-------------------+--------------------+---------------+--------------------
 Cashier           | Fortin             |          3196 |                  0
 Delivery Person   | Garnett            |          3196 |                  0
 Cashier           | Vogel              |          3196 |                  0
 Customer Service  | Sanchez            |          3198 |                  0
 Shelf Stocker     | Jones              |          3198 |                  0
 Custodian         | Li                 |          3198 |                  0
 Customer Service  | Kramer             |          3198 |                  0
 Greeter           | McNulty            |          3198 |                  0
 Greeter           | Greenwood          |          3198 |                  0
 Shift Manager     | Miller             |         99817 |                  0
 Advertising       | Vu                 |         99853 |                  0
 Branch Manager    | Jackson            |         99858 |                  0
 Marketing         | Taylor             |         99928 |                  0
 Assistant Director | King              |         99973 |                  0
 Sales             | Kramer             |         99973 |                  0
 Head of PR        | Goldberg           |        199067 |                  0
 Regional Manager  | Gauthier           |        199744 |                  0
 Director of HR    | Moore              |        199896 |                  0
 Head of Marketing | Overstreet         |        199955 |                  0
 VP of Advertising | Meyer              |        199975 |                  0
 VP of Sales       | Sanchez            |        199992 |                  0
 Founder           | Gauthier           |        927335 |                  0
 CEO               | Taylor             |        953373 |                  0
 Investor          | Garnett            |        963104 |                  0
 Co-Founder        | Vu                 |        977716 |                  0
 CFO               | Vogel              |        983634 |                  0
 President         | Sanchez            |        992363 |                  0
 Delivery Person   | Li                 |          3194 | 0.00114155251141553
 Delivery Person   | Robinson           |          3194 | 0.00114155251141553
 Custodian         | McCabe             |          3192 | 0.00126582278481013
 Shelf Stocker     | Moore              |          3196 | 0.00128040973111396
 Branch Manager    | Moore              |         99716 | 0.00186567164179104
...
```

See also

- CUME_DIST [analytic]
- SQL analytics

PERCENTILE_CONT [analytic]

An inverse distribution function where, for each row, PERCENTILE_CONT returns the value that would fall into the specified percentile among a set of values in each partition within a window . For example, if the argument to the function is 0.5, the result of the function is the median of the data set (50th percentile). PERCENTILE_CONT assumes a continuous distribution data model. NULL values are ignored.

PERCENTILE_CONT computes the percentile by first computing the row number where the percentile row would exist. For example:

```
row-number = 1 + percentile-value * (num-partition-rows -1)
```

If *row-number* is a whole number (within an error of 0.00001), the percentile is the value of row *row-number* .

Otherwise, Vertica interpolates the percentile value between the value of the CEILING(*row-number*) row and the value of the FLOOR(*row-number*) row. In other words, the percentile is calculated as follows:

```
 ( CEILING( row-number) - row-number ) * ( value of FLOOR(row-number) row )
+ ( row-number - FLOOR(row-number) ) * ( value of CEILING(row-number) row)
```

Note
If the percentile value is 0.5, PERCENTILE_CONT returns the same result set as the function MEDIAN .

Behavior type

Immutable

Syntax

```
PERCENTILE_CONT ( percentile ) WITHIN GROUP ( ORDER BY expression [ ASC | DESC ] ) OVER ( [ window-partition-clause ] )
```

Parameters

percentile

Percentile value, a FLOAT constant that ranges from 0 to 1 (inclusive).

WITHIN GROUP (ORDER BY *expression*)

Specifies how to sort data within each group. ORDER BY takes only one column/expression that must be INTEGER, FLOAT, INTERVAL, or NUMERIC data type. NULL values are discarded.

The WITHIN GROUP(ORDER BY) clause does not guarantee the order of the SQL result. To order the final result , use the SQL ORDER BY clause set.

ASC | DESC

Specifies the ordering sequence as ascending (default) or descending.

Specifying ASC or DESC in the WITHIN GROUP clause affects results as long as the *percentile* is not 0.5 .

OVER()

See Analytic Functions

Examples

This query computes the median annual income per group for the first 300 customers in Wisconsin and the District of Columbia.

```
=> SELECT customer_state, customer_key, annual_income, PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY annual_income)
     OVER (PARTITION BY customer_state) AS PERCENTILE_CONT
   FROM customer_dimension WHERE customer_state IN ('DC','WI') AND customer_key < 300
   ORDER BY customer_state, customer_key;
 customer_state | customer_key | annual_income | PERCENTILE_CONT
----------------+--------------+---------------+-----------------
 DC             |           52 |        168312 |        483266.5
 DC             |          118 |        798221 |        483266.5
 WI             |           62 |        283043 |         377691
 WI             |          139 |        472339 |         377691
(4 rows)
```

This query computes the median annual income per group for all customers in Wisconsin and the District of Columbia.

```
=> SELECT customer_state, customer_key, annual_income, PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY annual_income)
    OVER (PARTITION BY customer_state) AS PERCENTILE_CONT
  FROM customer_dimension WHERE customer_state IN ('DC','WI') ORDER BY customer_state, customer_key;
 customer_state | customer_key | annual_income | PERCENTILE_CONT
----------------+--------------+---------------+-----------------
 DC             |           52 |        168312 |        483266.5
 DC             |          118 |        798221 |        483266.5
 DC             |          622 |        220782 |         555088
 DC             |          951 |        178453 |         555088
 DC             |          972 |        961582 |         555088
 DC             |         1286 |        760445 |         555088
 DC             |         1434 |         44836 |         555088
 ...

 WI             |           62 |        283043 |         377691
 WI             |          139 |        472339 |         377691
 WI             |          359 |         42242 |         517717
 WI             |          364 |        867543 |         517717
 WI             |          403 |        509031 |         517717
 WI             |          455 |         32000 |         517717
 WI             |          485 |        373129 |         517717
 ...

(1353 rows)
```

See also

- MEDIAN [analytic]
- SQL analytics

PERCENTILE_DISC [analytic]

An inverse distribution function where, for each row, PERCENTILE_DISC returns the value that would fall into the specified percentile among a set of values in each partition within a window . PERCENTILE_DISC() assumes a discrete distribution data model. NULL values are ignored.

PERCENTILE_DISC examines the cumulative distribution values in each group until it finds one that is greater than or equal to the specified percentile. Vertica computes the percentile where, for each row, PERCENTILE_DISC outputs the first value of the WITHIN GROUP(ORDER BY) column whose CUME_DIST (cumulative distribution) value is >= the argument FLOAT value—for example, 0.4 :

```
PERCENTILE_DIST(0.4) WITHIN GROUP (ORDER BY salary) OVER(PARTITION BY deptno)...
```

Given the following query:

```
SELECT CUME_DIST() OVER(ORDER BY salary) FROM table-name;
```

The smallest CUME_DIST value that is greater than 0.4 is also the PERCENTILE_DISC .

Behavior type

Immutable

Syntax

```
PERCENTILE_DISC ( percentile ) WITHIN GROUP (
  ORDER BY expression [ ASC | DESC ] ) OVER (
  [ window-partition-clause ] )
```

Parameters

percentile

Percentile value, a FLOAT constant that ranges from 0 to 1 (inclusive).

WITHIN GROUP(ORDER BY *expression*)

Specifies how to sort data within each group. ORDER BY takes only one column/expression that must be INTEGER , FLOAT , INTERVAL , or NUMERIC data type. NULL values are discarded.

The WITHIN GROUP(ORDER BY) clause does not guarantee the order of the SQL result. To order the final result , use the SQL ORDER BY clause set.

ASC | DESC

Specifies the ordering sequence as ascending (default) or descending.

OVER()

See Analytic Functions

Examples

This query computes the 20th percentile annual income by group for first 300 customers in Wisconsin and the District of Columbia.

```
=> SELECT customer_state, customer_key, annual_income,
     PERCENTILE_DISC(.2) WITHIN GROUP(ORDER BY annual_income)
     OVER (PARTITION BY customer_state) AS PERCENTILE_DISC
   FROM customer_dimension
   WHERE customer_state IN ('DC','WI')
   AND customer_key < 300
   ORDER BY customer_state, customer_key;
 customer_state | customer_key | annual_income | PERCENTILE_DISC
----------------+--------------+---------------+-----------------
 DC             |          104 |        658383 |          417092
 DC             |          168 |        417092 |          417092
 DC             |          245 |        670205 |          417092
 WI             |          106 |        227279 |          227279
 WI             |          127 |        703889 |          227279
 WI             |          209 |        458607 |          227279
(6 rows)
```

See also

- CUME_DIST [analytic]
- PERCENTILE_CONT [analytic]
- SQL analytics

RANK [analytic]

Within each window partition, ranks all rows in the query results set according to the order specified by the window's ORDER BY clause.

RANK executes as follows:

1. Sorts partition rows as specified by the ORDER BY clause.
2. Compares the ORDER BY values of the preceding row and current row and ranks the current row as follows:
   - If ORDER BY values are the same, the current row gets the same ranking as the preceding row.

     > **Note**
     > Null values are considered equal. For detailed information on how null values are sorted, see NULL sort order .

   - If the ORDER BY values are different, DENSE_RANK increments or decrements the current row's ranking by 1, plus the number of consecutive duplicate values in the rows that precede it.

The largest rank value is the equal to the total number of rows returned by the query.

Behavior type

Immutable

Syntax

```
RANK() OVER (
  [ window-partition-clause ]
  window-order-clause )
```

Parameters

OVER()

See Analytic Functions

Compared with DENSE_RANK

RANK can leave gaps in the ranking sequence, while DENSE_RANK does not.

Examples

The following query ranks by state all company customers that have been customers since 2007. In rows where the customer_since dates are the same, RANK assigns the rows equal ranking. When the customer_since date changes, RANK skips one or more rankings—for example, within CA , from 12 to 14, and from 17 to 19.

```
=> SELECT customer_state, customer_name, customer_since,
   RANK() OVER (PARTITION BY customer_state ORDER BY customer_since) AS rank
   FROM customer_dimension WHERE customer_type='Company' AND customer_since > '01/01/2007'
   ORDER BY customer_state;
 customer_state | customer_name | customer_since | rank
----------------+---------------+----------------+------
 AZ             | Foodshop      | 2007-01-20     |    1
 AZ             | Goldstar      | 2007-08-11     |    2
 CA             | Metahope      | 2007-01-05     |    1
 CA             | Foodgen       | 2007-02-05     |    2
 CA             | Infohope      | 2007-02-09     |    3
 CA             | Foodcom       | 2007-02-19     |    4
 CA             | Amerihope     | 2007-02-22     |    5
 CA             | Infostar      | 2007-03-05     |    6
 CA             | Intracare     | 2007-03-14     |    7
 CA             | Infocare      | 2007-04-07     |    8
...
 CO             | Goldtech      | 2007-02-19     |    1
 CT             | Foodmedia     | 2007-02-11     |    1
 CT             | Metatech      | 2007-02-20     |    2
 CT             | Infocorp      | 2007-04-10     |    3
...
```

See also

SQL analytics

ROW_NUMBER [analytic]

Assigns a sequence of unique numbers to each row in a window partition, starting with 1. ROW_NUMBER and RANK are generally interchangeable, with the following differences:

- ROW_NUMBER assigns a unique ordinal number to each row in the ordered set, starting with 1.
- ROW_NUMBER() is a Vertica extension, while RANK conforms to the SQL-99 standard.

Behavior type

Immutable

Syntax

```
ROW_NUMBER () OVER (
   [ window-partition-clause ]
   [ window-order-clause ] )
```

Parameters

OVER()

See Analytic Functions

Examples

The following ROW_NUMBER query partitions customers in the VMart table customer_dimension by customer_regio n. Within each partition, the function ranks those customers in order of seniority, as specified by its window order clause:

```
=> SELECT * FROM
   (SELECT ROW_NUMBER() OVER (PARTITION BY customer_region ORDER BY customer_since) AS most_senior,
    customer_region, customer_name, customer_since FROM public.customer_dimension WHERE customer_type = 'Individual') sq
  WHERE most_senior <= 5;
 most_senior | customer_region |    customer_name     | customer_since
-------------+-----------------+----------------------+----------------
           1 | West            | Jack Y. Perkins      | 1965-01-01
           2 | West            | Linda Q. Winkler     | 1965-01-02
           3 | West            | Marcus K. Li         | 1965-01-03
           4 | West            | Carla R. Jones       | 1965-01-07
           5 | West            | Seth P. Young        | 1965-01-09
           1 | East            | Kim O. Vu            | 1965-01-01
           2 | East            | Alexandra L. Weaver  | 1965-01-02
           3 | East            | Steve L. Webber      | 1965-01-04
           4 | East            | Thom Y. Li           | 1965-01-05
           5 | East            | Martha B. Farmer     | 1965-01-07
           1 | SouthWest       | Martha V. Gauthier   | 1965-01-01
           2 | SouthWest       | Jessica U. Goldberg  | 1965-01-07
           3 | SouthWest       | Robert O. Stein      | 1965-01-07
           4 | SouthWest       | Emily I. McCabe      | 1965-01-18
           5 | SouthWest       | Jack E. Miller       | 1965-01-25
           1 | NorthWest       | Julie O. Greenwood   | 1965-01-08
           2 | NorthWest       | Amy X. McNulty       | 1965-01-25
           3 | NorthWest       | Kevin S. Carcetti    | 1965-02-09
           4 | NorthWest       | Sam K. Carcetti      | 1965-03-16
           5 | NorthWest       | Alexandra X. Winkler | 1965-04-05
           1 | MidWest         | Michael Y. Meyer     | 1965-01-01
           2 | MidWest         | Joanna W. Bauer      | 1965-01-06
           3 | MidWest         | Amy E. Harris        | 1965-01-08
           4 | MidWest         | Julie W. McCabe      | 1965-01-09
           5 | MidWest         | William . Peterson   | 1965-01-09
           1 | South           | Dean . Martin        | 1965-01-01
           2 | South           | Ruth U. Williams     | 1965-01-02
           3 | South           | Steve Y. Farmer      | 1965-01-03
           4 | South           | Mark V. King         | 1965-01-08
           5 | South           | Lucas Y. Young       | 1965-01-10
(30 rows)
```

See also

- RANK [analytic]
- SQL analytics

STDDEV [analytic]

Computes the statistical sample standard deviation of the current row with respect to the group within a window . STDDEV_SAMP returns the same value as the square root of the variance defined for the VAR_SAMP function:

```
STDDEV( expression ) = SQRT(VAR_SAMP( expression ))
```

When VAR_SAMP returns NULL , this function returns NULL .

> **Note**
> The nonstandard function STDDEV is provided for compatibility with other databases. It is semantically identical to STDDEV_SAMP .

Behavior type

Immutable

Syntax

```
STDDEV ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

OVER()

See Analytic Functions

Examples

The following example returns the standard deviations of salaries in the employee dimension table by job title Assistant Director:

```
=> SELECT employee_last_name, annual_salary,
     STDDEV(annual_salary) OVER (ORDER BY hire_date) as "stddev"
  FROM employee_dimension
  WHERE job_title =  'Assistant Director';
 employee_last_name | annual_salary |      stddev
--------------------+---------------+------------------
 Bauer              |         85003 |              NaN
 Reyes              |         91051 | 4276.58181261624
 Overstreet         |         53296 | 20278.6923394976
 Gauthier           |         97216 | 19543.7184537642
 Jones              |         82320 | 16928.0764028285
 Fortin             |         56166 | 18400.2738421652
 Carcetti           |         71135 | 16968.9453554483
 Weaver             |         74419 | 15729.0709901852
 Stein              |         85689 | 15040.5909495309
 McNulty            |         69423 | 14401.1524291943
 Webber             |         99091 | 15256.3160166536
 Meyer              |         74774 | 14588.6126417355
 Garnett            |         82169 | 14008.7223268494
 Roy                |         76974 | 13466.1270356647
 Dobisz             |         83486 | 13040.4887828347
 Martin             |         99702 | 13637.6804131055
 Martin             |         73589 | 13299.2838158566
 ...
```

See also

- STDDEV [aggregate]
- STDDEV_SAMP [aggregate]
- STDDEV_SAMP [analytic]
- SQL analytics

STDDEV_POP [analytic]

Computes the statistical population standard deviation and returns the square root of the population variance within a window . The STDDEV_POP() return value is the same as the square root of the VAR_POP() function:

```
STDDEV_POP( expression ) = SQRT(VAR_POP( expression ))
```

When VAR_POP returns null, STDDEV_POP returns null.

Behavior type

Immutable

Syntax

```
STDDEV_POP ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

OVER()

See Analytic Functions.

Examples

The following example returns the population standard deviations of salaries in the employee dimension table by job title Assistant Director:

```
=> SELECT employee_last_name, annual_salary,
     STDDEV_POP(annual_salary) OVER (ORDER BY hire_date) as "stddev_pop"
  FROM employee_dimension WHERE job_title =  'Assistant Director';
 employee_last_name | annual_salary |    stddev_pop
--------------------+---------------+------------------
 Goldberg           |         61859 |                0
 Miller             |         79582 |           8861.5
 Goldberg           |         74236 | 7422.74712548456
 Campbell           |         66426 | 6850.22125098891
 Moore              |         66630 | 6322.08223926257
 Nguyen             |         53530 | 8356.55480080699
 Harris             |         74115 | 8122.72288970008
 Lang               |         59981 | 8053.54776538731
 Farmer             |         60597 | 7858.70140687825
 Nguyen             |         78941 | 8360.63150784682
```

See also

- STDDEV_POP [aggregate]
- SQL analytics

STDDEV_SAMP [analytic]

Computes the statistical sample standard deviation of the current row with respect to the group within a window . STDDEV_SAM 's return value is the same as the square root of the variance defined for the VAR_SAMP function:

```
STDDEV( expression ) = SQRT(VAR_SAMP( expression ))
```

When VAR_SAMP returns NULL , STDDEV_SAMP returns NULL.

> **Note**
> STDDEV_SAMP() is semantically identical to the nonstandard function, STDDEV() .

Behavior type

Immutable

Syntax

```
STDDEV_SAMP ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument..

OVER()

See Analytic Functions

Examples

The following example returns the sample standard deviations of salaries in the employee dimension table by job title Assistant Director:

```
=> SELECT employee_last_name, annual_salary,
    STDDEV(annual_salary) OVER (ORDER BY hire_date) as "stddev_samp"
    FROM employee_dimension WHERE job_title =  'Assistant Director';
 employee_last_name | annual_salary |   stddev_samp
--------------------+---------------+------------------
 Bauer              |         85003 |             NaN
 Reyes              |         91051 | 4276.58181261624
 Overstreet         |         53296 | 20278.6923394976
 Gauthier           |         97216 | 19543.7184537642
 Jones              |         82320 | 16928.0764028285
 Fortin             |         56166 | 18400.2738421652
 Carcetti           |         71135 | 16968.9453554483
 Weaver             |         74419 | 15729.0709901852
 Stein              |         85689 | 15040.5909495309
 McNulty            |         69423 | 14401.1524291943
 Webber             |         99091 | 15256.3160166536
 Meyer              |         74774 | 14588.6126417355
 Garnett            |         82169 | 14008.7223268494
 Roy                |         76974 | 13466.1270356647
 Dobisz             |         83486 | 13040.4887828347
 ...
```

See also

- Analytic functions
- STDDEV [analytic]
- STDDEV [aggregate]
- STDDEV_SAMP [aggregate]
- SQL analytics

SUM [analytic]

Computes the sum of an expression over a group of rows within a window . It returns a DOUBLE PRECISION value for a floating-point expression. Otherwise, the return value is the same as the expression data type.

Behavior type

Immutable

Syntax

```
SUM ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

OVER()

See Analytic Functions

Overflow handling

If you encounter data overflow when using SUM , use SUM_FLOAT which converts data to a floating point.

By default, Vertica allows silent numeric overflow when you call this function on numeric data types. For more information on this behavior and how to change it, see Numeric data type overflow with SUM, SUM_FLOAT, and AVG .

Examples

The following query returns the cumulative sum all of the returns made to stores in January:

```
=> SELECT calendar_month_name AS month, transaction_type, sales_quantity,
    SUM(sales_quantity)
    OVER (PARTITION BY calendar_month_name ORDER BY date_dimension.date_key) AS SUM
    FROM store.store_sales_fact JOIN date_dimension
    USING(date_key) WHERE calendar_month_name IN ('January')
    AND transaction_type= 'return';
 month  | transaction_type | sales_quantity | SUM
---------+------------------+----------------+------
January | return           |              7 |  651
January | return           |              3 |  651
January | return           |              7 |  651
January | return           |              7 |  651
January | return           |              7 |  651
January | return           |              3 |  651
January | return           |              7 |  651
January | return           |              5 |  651
January | return           |              1 |  651
January | return           |              6 |  651
January | return           |              6 |  651
January | return           |              3 |  651
January | return           |              9 |  651
January | return           |              7 |  651
January | return           |              6 |  651
January | return           |              8 |  651
January | return           |              7 |  651
January | return           |              2 |  651
January | return           |              4 |  651
January | return           |              5 |  651
January | return           |              7 |  651
January | return           |              8 |  651
January | return           |              4 |  651
January | return           |             10 |  651
January | return           |              6 |  651
...
```

See also

- SUM [aggregate]
- Numeric data types
- SQL analytics

VAR_POP [analytic]

Returns the statistical population variance of a non-null set of numbers (nulls are ignored) in a group within a window . Results are calculated by the sum of squares of the difference of *expression* from the mean of *expression* , divided by the number of rows remaining:

```
(SUM( expression * expression ) - SUM( expression  ) * SUM(  expression ) /   COUNT( expression )) / COUNT( expression )
```

Behavior type

Immutable

Syntax

```
VAR_POP ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument

OVER()

See Analytic Functions

Examples

The following example calculates the cumulative population in the store orders fact table of sales in January 2007:

```
=> SELECT date_ordered,
     VAR_POP(SUM(total_order_cost))
     OVER (ORDER BY date_ordered) "var_pop"
   FROM store.store_orders_fact s
   WHERE date_ordered BETWEEN '2007-01-01' AND '2007-01-31'
   GROUP BY s.date_ordered;
 date_ordered |     var_pop
--------------+------------------
 2007-01-01   |               0
 2007-01-02   |        89870400
 2007-01-03   |      3470302472
 2007-01-04   |  4466755450.6875
 2007-01-05   | 3816904780.80078
 2007-01-06   |   25438212385.25
 2007-01-07   | 22168747513.1016
 2007-01-08   | 23445191012.7344
 2007-01-09   | 39292879603.1113
 2007-01-10   | 48080574326.9609
(10 rows)
```

See also

- VAR_POP [aggregate]
- SQL analytics

VAR_SAMP [analytic]

Returns the sample variance of a non- NULL set of numbers (NULL values in the set are ignored) for each row of the group within a window . Results are calculated as follows:

```
(SUM( expression * expression ) - SUM( expression ) * SUM( expression ) / COUNT( expression ) )
/ (COUNT( expression ) - 1 )
```

This function and VARIANCE differ in one way: given an input set of one element, VARIANCE returns 0 and VAR_SAMP returns NULL .

Behavior type

Immutable

Syntax

```
VAR_SAMP ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument

OVER()

See Analytic Functions

Null handling

- VAR_SAMP returns the sample variance of a set of numbers after it discards the NULL values in the set.
- If the function is applied to an empty set, then it returns NULL .

Examples

The following example calculates the sample variance in the store orders fact table of sales in December 2007:

```
=> SELECT date_ordered,
     VAR_SAMP(SUM(total_order_cost))
     OVER (ORDER BY date_ordered) "var_samp"
   FROM store.store_orders_fact s
   WHERE date_ordered BETWEEN '2007-12-01' AND '2007-12-31'
   GROUP BY s.date_ordered;
 date_ordered |     var_samp
--------------+------------------
 2007-12-01   |              NaN
 2007-12-02   |      90642601088
 2007-12-03   | 48030548449.3359
 2007-12-04   | 32740062504.2461
 2007-12-05   | 32100319112.6992
 2007-12-06   |  26274166814.668
 2007-12-07   | 23017490251.9062
 2007-12-08   | 21099374085.1406
 2007-12-09   | 27462205977.9453
 2007-12-10   | 26288687564.1758
(10 rows)
```

See also

- VARIANCE [analytic]
- VAR_SAMP [aggregate]
- SQL analytics

VARIANCE [analytic]

Returns the sample variance of a non- NULL set of numbers (NULL values in the set are ignored) for each row of the group within a window . Results are calculated as follows:

```
( SUM( expression * expression ) - SUM( expression ) * SUM( expression ) / COUNT( expression )) / (COUNT( expression ) - 1 )
```

VARIANCE returns the variance of *expression* , which is calculated as follows:

- 0 if the number of rows in *expression* = 1
- VAR_SAMP if the number of rows in *expression* > 1

> **Note**
> The nonstandard function VARIANCE is provided for compatibility with other databases. It is semantically identical to VAR_SAMP .

Behavior type

Immutable

Syntax

```
VAR_SAMP ( expression ) OVER (
   [ window-partition-clause ]
   [ window-order-clause ]
   [ window-frame-clause ] )
```

Parameters

expression

Any NUMERIC data type or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

OVER()

See Analytic Functions

Examples

The following example calculates the cumulative variance in the store orders fact table of sales in December 2007:

```
=> SELECT date_ordered,
     VARIANCE(SUM(total_order_cost))
     OVER (ORDER BY date_ordered) "variance"
   FROM store.store_orders_fact s
   WHERE date_ordered BETWEEN '2007-12-01' AND '2007-12-31'
   GROUP BY s.date_ordered;
 date_ordered |     variance
--------------+------------------
 2007-12-01   |              NaN
 2007-12-02   |       2259129762
 2007-12-03   | 1809012182.33301
 2007-12-04   |   35138165568.25
 2007-12-05   | 26644110029.3003
 2007-12-06   |      25943125234
 2007-12-07   | 23178202223.9048
 2007-12-08   | 21940268901.1431
 2007-12-09   | 21487676799.6108
 2007-12-10   | 21521358853.4331
(10 rows)
```

See also

- VAR_SAMP [analytic]
- VARIANCE [aggregate]
- VAR_SAMP [aggregate]
- SQL analytics

Client connection functions

This section contains client connection management functions specific to Vertica.

In this section

- CLOSE_ALL_RESULTSETS
- CLOSE_RESULTSET
- DESCRIBE_LOAD_BALANCE_DECISION
- GET_CLIENT_LABEL
- RESET_LOAD_BALANCE_POLICY
- SET_CLIENT_LABEL
- SET_LOAD_BALANCE_POLICY

CLOSE_ALL_RESULTSETS

Closes all result set sessions within Multiple Active Result Sets (MARS) and frees the MARS storage for other result sets.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SELECT CLOSE_ALL_RESULTSETS ('session_id')
```

Parameters

session_id

A string that specifies the Multiple Active Result Sets session.

Privileges

None; however, without superuser privileges, you can only close your own session's results.

Examples

This example shows how you can view a MARS result set, then close the result set, and then confirm that the result set has been closed.

Query the MARS storage table. One session ID is open and three result sets appear in the output.

```
=> SELECT * FROM SESSION_MARS_STORE;

   node_name     |           session_id            | user_name | resultset_id | row_count | remaining_row_count | bytes_used
-----------------+---------------------------------+-----------+--------------+-----------+---------------------+------------
 v_vmart_node0001 | server1.company.-83046:1y28gu9   | dbadmin   |            7 |    777460 |              776460 |   89692848
 v_vmart_node0001 | server1.company.-83046:1y28gu9   | dbadmin   |            8 |    324349 |              323349 |   81862010
 v_vmart_node0001 | server1.company.-83046:1y28gu9   | dbadmin   |            9 |    277947 |              276947 |   32978280
(1 row)
```

Close all result sets for session server1.company.-83046:1y28gu9:

```
=> SELECT CLOSE_ALL_RESULTSETS('server1.company.-83046:1y28gu9');
            close_all_resultsets
-----------------------------------------------------------------
 Closing all result sets from server1.company.-83046:1y28gu9
(1 row)
```

Query the MARS storage table again for the current status. You can see that the session and result sets have been closed:

```
=> SELECT * FROM SESSION_MARS_STORE;

   node_name     |           session_id            | user_name | resultset_id | row_count | remaining_row_count | bytes_used
-----------------+---------------------------------+-----------+--------------+-----------+---------------------+------------
(0 rows)
```

CLOSE_RESULTSET

Closes a specific result set within Multiple Active Result Sets (MARS) and frees the MARS storage for other result sets.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SELECT CLOSE_RESULTSET ('session_id', ResultSetID)
```

Parameters

session_id

A string that specifies the Multiple Active Result Sets session containing the ResultSetID to close.

ResultSetID

An integer that specifies which result set to close.

Privileges

None; however, without superuser privileges, you can only close your own session's results.

Examples

This example shows a MARS storage table opened. One session_id is currently open, and one result set appears in the output.

```
=> SELECT * FROM SESSION_MARS_STORE;
   node_name     |           session_id            | user_name | resultset_id | row_count | remaining_row_count | bytes_used
-----------------+---------------------------------+-----------+--------------+-----------+---------------------+------------
 v_vmart_node0001 | server1.company.-83046:1y28gu9   | dbadmin   |            1 |    318718 |              312718 |   80441904
(1 row)
```

Close user session server1.company.-83046:1y28gu9 and result set 1:

```
=> SELECT CLOSE_RESULTSET('server1.company.-83046:1y28gu9', 1);
            close_resultset
-----------------------------------------------------------------
 Closing result set 1 from server1.company.-83046:1y28gu9
(1 row)
```

Query the MARS storage table again for current status. You can see that result set 1 is now closed:

```
SELECT * FROM SESSION_MARS_STORE;

   node_name     |           session_id            | user_name | resultset_id | row_count | remaining_row_count | bytes_used
-----------------+---------------------------------+-----------+--------------+-----------+---------------------+------------
(0 rows)
```

DESCRIBE_LOAD_BALANCE_DECISION

Evaluates if any load balancing routing rules apply to a given IP address and describes how the client connection would be handled. This function is useful when you are evaluating connection load balancing policies you have created, to ensure they work the way you expect them to.

You pass this function an IP address of a client connection, and it uses the load balancing routing rules to determine how the connection will be handled. The logic this function uses is the same logic used when Vertica load balances client connections, including determining which nodes are available to handle the client connection.

This function assumes the client connection has opted into being load balanced. If actual clients have not opted into load balancing, the connections will not be redirected. See Load balancing in ADO.NET , Load balancing in JDBC , and Load balancing , for information on enabling load balancing on the client. For vsql, use the -C command-line option to enable load balancing.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESCRIBE_LOAD_BALANCE_DECISION('ip_address')
```

Arguments

' *ip_address* '

An IP address of a client connection to be tested against the load balancing rules. This can be either an IPv4 or IPv6 address.

Return value

A step-by-step description of how the load balancing rules are being evaluated, including the final decision of which node in the database has been chosen to service the connection.

Privileges

None.

Examples

The following example demonstrates calling DESCRIBE_LOAD_BALANCE_DECISION with three different IP addresses, two of which are handled by different routing rules, and one which is not handled by any rule.

```
=> SELECT describe_load_balance_decision('192.168.1.25');
                        describe_load_balance_decision
--------------------------------------------------------------------------------
 Describing load balance decision for address [192.168.1.25]
Load balance cache internal version id (node-local): [2]
Considered rule [etl_rule] source ip filter [10.20.100.0/24]... input address
does not match source ip filter for this rule.
Considered rule [internal_clients] source ip filter [192.168.1.0/24]... input
address matches this rule
Matched to load balance group [group_1] the group has policy [ROUNDROBIN]
number of addresses [2]
(0) LB Address: [10.20.100.247]:5433
(1) LB Address: [10.20.100.248]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [10.20.100.248] port [5433]

(1 row)

=> SELECT describe_load_balance_decision('192.168.2.25');
                        describe_load_balance_decision
--------------------------------------------------------------------------------
 Describing load balance decision for address [192.168.2.25]
Load balance cache internal version id (node-local): [2]
Considered rule [etl_rule] source ip filter [10.20.100.0/24]... input address
does not match source ip filter for this rule.
Considered rule [internal_clients] source ip filter [192.168.1.0/24]... input
address does not match source ip filter for this rule.
Considered rule [subnet_192] source ip filter [192.0.0.0/8]... input address
matches this rule
Matched to load balance group [group_all] the group has policy [ROUNDROBIN]
number of addresses [3]
(0) LB Address: [10.20.100.247]:5433
(1) LB Address: [10.20.100.248]:5433
(2) LB Address: [10.20.100.249]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [10.20.100.248] port [5433]

(1 row)

=> SELECT describe_load_balance_decision('1.2.3.4');
                         describe_load_balance_decision
--------------------------------------------------------------------------------
 Describing load balance decision for address [1.2.3.4]
Load balance cache internal version id (node-local): [2]
Considered rule [etl_rule] source ip filter [10.20.100.0/24]... input address
does not match source ip filter for this rule.
Considered rule [internal_clients] source ip filter [192.168.1.0/24]... input
address does not match source ip filter for this rule.
Considered rule [subnet_192] source ip filter [192.0.0.0/8]... input address
does not match source ip filter for this rule.
Routing table decision: No matching routing rules: input address does not match
any routing rule source filters. Details: [Tried some rules but no matching]
No rules matched. Falling back to classic load balancing.
Classic load balance decision: Classic load balancing considered, but either
the policy was NONE or no target was available. Details: [NONE or invalid]

(1 row)
```

The following example demonstrates calling DESCRIBE_LOAD_BALANCE_DECISION repeatedly with the same IP address. You can see that the load balance group's ROUNDROBIN load balance policy has it switch between the two nodes in the load balance group:

```
=> SELECT describe_load_balance_decision('192.168.1.25');
                    describe_load_balance_decision
-------------------------------------------------------------------------------
 Describing load balance decision for address [192.168.1.25]
Load balance cache internal version id (node-local): [1]
Considered rule [etl_rule] source ip filter [10.20.100.0/24]... input address
does not match source ip filter for this rule.
Considered rule [internal_clients] source ip filter [192.168.1.0/24]... input
address matches this rule
Matched to load balance group [group_1] the group has policy [ROUNDROBIN]
number of addresses [2]
(0) LB Address: [10.20.100.247]:5433
(1) LB Address: [10.20.100.248]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [10.20.100.248]
port [5433]

(1 row)

=> SELECT describe_load_balance_decision('192.168.1.25');
                    describe_load_balance_decision
-------------------------------------------------------------------------------
 Describing load balance decision for address [192.168.1.25]
Load balance cache internal version id (node-local): [1]
Considered rule [etl_rule] source ip filter [10.20.100.0/24]... input address
does not match source ip filter for this rule.
Considered rule [internal_clients] source ip filter [192.168.1.0/24]... input
address matches this rule
Matched to load balance group [group_1] the group has policy [ROUNDROBIN]
number of addresses [2]
(0) LB Address: [10.20.100.247]:5433
(1) LB Address: [10.20.100.248]:5433
Chose address at position [0]
Routing table decision: Success. Load balance redirect to: [10.20.100.247]
port [5433]

(1 row)

=> SELECT describe_load_balance_decision('192.168.1.25');
                    describe_load_balance_decision
-------------------------------------------------------------------------------
 Describing load balance decision for address [192.168.1.25]
Load balance cache internal version id (node-local): [1]
Considered rule [etl_rule] source ip filter [10.20.100.0/24]... input address
does not match source ip filter for this rule.
Considered rule [internal_clients] source ip filter [192.168.1.0/24]... input
address matches this rule
Matched to load balance group [group_1] the group has policy [ROUNDROBIN]
number of addresses [2]
(0) LB Address: [10.20.100.247]:5433
(1) LB Address: [10.20.100.248]:5433
Chose address at position [1]
Routing table decision: Success. Load balance redirect to: [10.20.100.248]
port [5433]

(1 row)
```

See also

- GET_CLIENT_LABEL
- RESET_LOAD_BALANCE_POLICY
- SET_CLIENT_LABEL

- SET_LOAD_BALANCE_POLICY

GET_CLIENT_LABEL

Returns the client connection label for the current session.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_CLIENT_LABEL()
```

Privileges

None

Examples

Return the current client connection label:

```
=> SELECT GET_CLIENT_LABEL();
  GET_CLIENT_LABEL
----------------------
 data_load_application
(1 row)
```

See also

Setting a client connection label

RESET_LOAD_BALANCE_POLICY

Resets the counter each host in the cluster maintains, to track which host it will refer a client to when the native connection load balancing scheme is set to ROUNDROBIN . To reset the counter, run this function on all cluster nodes.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RESET_LOAD_BALANCE_POLICY()
```

Privileges

Superuser

Examples

```
=> SELECT RESET_LOAD_BALANCE_POLICY();

                    RESET_LOAD_BALANCE_POLICY
-------------------------------------------------------------------------
Successfully reset stateful client load balance policies: "roundrobin".
(1 row)
```

SET_CLIENT_LABEL

Assigns a label to a client connection for the current session. You can use this label to distinguish client connections.

Labels appear in the SESSIONS system table. However, only certain Data collector tables show new client labels set by SET_CLIENT_LABEL. For example, DC_REQUESTS_ISSUED reflects changes by SET_CLIENT_LABEL, while DC_SESSION_STARTS, which collects login data before SET_CLIENT_LABEL can be run, does not.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_CLIENT_LABEL('label-name')
```

Parameters

label-name

VARCHAR name assigned to the client connection label.

Privileges

None

Examples

Assign label data_load_application to the current client connection:

```
=> SELECT SET_CLIENT_LABEL('data_load_application');
          SET_CLIENT_LABEL
-------------------------------------------
 client_label set to data_load_application
(1 row)
```

See also

Setting a client connection label

SET_LOAD_BALANCE_POLICY

Sets how native connection load balancing chooses a host to handle a client connection.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_LOAD_BALANCE_POLICY('policy')
```

Parameters

policy

The name of the load balancing policy to use, one of the following:

- NONE (default): Disables native connection load balancing.
- ROUNDROBIN : Chooses the next host from a circular list of hosts in the cluster that are up—for example, in a three-node cluster, iterates over node1, node2, and node3, then wraps back to node1. Each host in the cluster maintains its own pointer to the next host in the circular list, rather than there being a single cluster-wide state.
- RANDOM : Randomly chooses a host from among all hosts in the cluster that are up.

> **Note**
> Even if the load balancing policy is set on the server to something other than NONE , clients must indicate they want their connections to be load balanced by setting a connection property.

Privileges

Superuser

Examples

The following example demonstrates enabling native connection load balancing on the server by setting the load balancing scheme to ROUNDROBIN :

```
=> SELECT SET_LOAD_BALANCE_POLICY('ROUNDROBIN');
             SET_LOAD_BALANCE_POLICY
--------------------------------------------------------------------------------
Successfully changed the client initiator load balancing policy to: roundrobin
(1 row)
```

See also

About native connection load balancing

Data-type-specific functions

Vertica provides functions for use with specific data types, described in this section.

In this section

- Collection functions
- Date/time functions
- IP address functions
- Sequence functions
- String functions
- URI functions
- UUID functions

Collection functions

The functions in this section apply to collection types (arrays and sets).

Some functions apply aggregation operations (such as sum) to collections. These function names all begin with APPLY.

Other functions in this section operate specifically on arrays or sets, as indicated on the individual reference pages. Array functions operate on both native array values and array values in external tables.

Notes

- Arrays are 0-indexed. The first element's ordinal position in 0, second is 1, and so on. Indexes are not meaningful for sets.
- Unless otherwise stated, functions operate on one-dimensional (1D) collections only. To use multidimensional arrays, you must first dereference to a 1D array type. Sets can only be one-dimensional.

In this section

- APPLY_AVG
- APPLY_COUNT (ARRAY_COUNT)
- APPLY_COUNT_ELEMENTS (ARRAY_LENGTH)
- APPLY_MAX
- APPLY_MIN
- APPLY_SUM
- ARRAY_CAT
- ARRAY_CONTAINS
- ARRAY_DIMS
- ARRAY_FIND
- CONTAINS
- EXPLODE
- FILTER
- IMPLODE
- SET_UNION
- STRING_TO_ARRAY
- TO_JSON
- UNNEST

APPLY_AVG

Returns the average of all elements in a collection (array or set) with numeric values.

Behavior type

Immutable

Syntax

```
APPLY_AVG(collection)
```

Arguments

collection

Target collection

Null-handling

The following cases return NULL:

- if the input collection is NULL
- if the input collection contains only null values
- if the input collection is empty

If the input collection contains a mix of null and non-null elements, only the non-null values are considered in the calculation of the average.

Examples

```
=> SELECT apply_avg(ARRAY[1,2.4,5,6]);
apply_avg
-----------
3.6
(1 row)
```

See also

- APPLY_SUM

APPLY_COUNT (ARRAY_COUNT)

Returns the total number of non-null elements in a collection (array or set). To count all elements including nulls, use APPLY_COUNT_ELEMENTS (ARRAY_LENGTH) .

Behavior type

Immutable

Syntax

```
APPLY_COUNT(collection)
```

ARRAY_COUNT is a synonym of APPLY_COUNT .

Arguments

collection

Target collection

Null-handling

Null values are not included in the count.

Examples

The array in this example contains six elements, one of which is null:

```
=> SELECT apply_count(ARRAY[1,NULL,3,7,8,5]);
apply_count
-------------
5
(1 row)
```

APPLY_COUNT_ELEMENTS (ARRAY_LENGTH)

Returns the total number of elements in a collection (array or set), including NULLs. To count only non-null values, use APPLY_COUNT (ARRAY_COUNT) .

Behavior type

Immutable

Syntax

```
APPLY_COUNT_ELEMENTS(collection)
```

ARRAY_LENGTH is a synonym of APPLY_COUNT_ELEMENTS .

Arguments

collection

Target collection

Null-handling

This function counts all members, including nulls.

An empty collection (ARRAY[] or SET[]) has a length of 0. A collection containing a single null (ARRAY[null] or SET[null]) has a length of 1.

Examples

The following array has six elements including one null:

```
=> SELECT apply_count_elements(ARRAY[1,NULL,3,7,8,5]);
apply_count_elements
---------------------
                   6
(1 row)
```

As the previous example shows, a null element is an element. Thus, an array containing only a null element has one element:

```
=> SELECT apply_count_elements(ARRAY[null]);
apply_count_elements
---------------------
                   1
(1 row)
```

A set does not contain duplicates. If you construct a set and pass it directly to this function, the result could differ from the number of inputs:

```
=> SELECT apply_count_elements(SET[1,1,3]);
apply_count_elements
---------------------
                   2
(1 row)
```

APPLY_MAX

Returns the largest non-null element in a collection (array or set). This function is similar to the MAX [aggregate] function; APPLY_MAX operates on elements of a collection and MAX operates on an expression such as a column selection.

Behavior type

Immutable

Syntax

```
APPLY_MAX(collection)
```

Arguments

collection

Target collection

Null-handling

This function ignores null elements. If all elements are null or the collection is empty, this function returns null.

Examples

```
=> SELECT apply_max(ARRAY[1,3.4,15]);
apply_max
-----------
    15.0
(1 row)
```

APPLY_MIN

Returns the smallest non-null element in a collection (array or set). This function is similar to the MIN [aggregate] function; APPLY_MIN operates on elements of a collection and MIN operates on an expression such as a column selection.

Behavior type

Immutable

Syntax

```
APPLY_MIN(collection)
```

Arguments

collection

Target collection

Null-handling

This function ignores null elements. If all elements are null or the collection is empty, this function returns null.

Examples

```
=> SELECT apply_min(ARRAY[1,3.4,15]);
apply_min
-----------
      1.0
(1 row)
```

APPLY_SUM

Computes the sum of all elements in a collection (array or set) of numeric values (INTEGER, FLOAT, NUMERIC, or INTERVAL).

Behavior type

Immutable

Syntax

```
APPLY_SUM(collection)
```

Arguments

collection

Target collection

Null-handling

The following cases return NULL:

- if the input collection is NULL
- if the input collection contains only null values
- if the input collection is empty

Examples

```
=> SELECT apply_sum(ARRAY[12.5,3,4,1]);
apply_sum
-----------
     20.5
(1 row)
```

See also

- APPLY_AVG

ARRAY_CAT

Concatenates two arrays of the same element type and dimensionality. For example, ROW elements must have the same fields.

If the inputs are both bounded, the bound for the result is the sum of the bounds of the inputs.

If any input is unbounded, the result is unbounded with a binary size that is the sum of the sizes of the inputs.

Behavior type

Immutable

Syntax

```
ARRAY_CAT(array1,array2)
```

Arguments

array1 **,** ***array2***

Arrays of matching dimensionality and element type

Null-handling

If either input is NULL, the function returns NULL.

Examples

Types are coerced if necessary, as shown in the second example.

```
=> SELECT array_cat(ARRAY[1,2], ARRAY[3,4,5]);
array_cat
-----------------------
[1,2,3,4,5]
(1 row)

=> SELECT array_cat(ARRAY[1,2], ARRAY[3,4,5.0]);
array_cat
-----------------------
["1.0","2.0","3.0","4.0","5.0"]
(1 row)
```

ARRAY_CONTAINS

Returns true if the specified element is found in the array and false if not. Both arguments must be non-null, but the array may be empty.

> **Deprecated**
> This function has been renamed to CONTAINS .

ARRAY_DIMS

Returns the dimensionality of the input array.

Behavior type

Immutable

Syntax

```
ARRAY_DIMS(array)
```

Arguments

array

Target array

Examples

```
=> SELECT array_dims(ARRAY[[1,2],[2,3]]);
array_dims
------------
      2
(1 row)
```

ARRAY_FIND

Returns the ordinal position of a specified element in an array, or -1 if not found. This function uses null-safe equality checks when testing elements.

Behavior type

Immutable

Syntax

```
ARRAY_FIND(array, { value | lambda-expression })
```

Arguments

array

Target array.

value

Value to search for; type must match or be coercible to the element type of the array.

lambda-expression

Lambda function to apply to each element. The function must return a Boolean value. The first argument to the function is the element, and the optional second element is the index of the element.

Examples

The function returns the first occurrence of the specified element. However, nothing ensures that value is unique in the array:

```
=> SELECT array_find(ARRAY[1,2,7,5,7],7);
 array_find
------------
          2
(1 row)
```

The function returns -1 if the specified element is not found:

```
=> SELECT array_find(ARRAY[1,3,5,7],4);
array_find
------------
        -1
(1 row)
```

You can search for complex element types:

```
=> SELECT ARRAY_FIND(ARRAY[ARRAY[1,2,3],ARRAY[1,null,4]],
                     ARRAY[1,2,3]);
 ARRAY_FIND
------------
          0
(1 row)

=> SELECT ARRAY_FIND(ARRAY[ARRAY[1,2,3],ARRAY[1,null,4]],
                     ARRAY[1,null,4]);
 ARRAY_FIND
------------
          1
(1 row)
```

The second example, comparing arrays with null elements, finds a match because ARRAY_FIND uses a null-safe equality check when evaluating elements.

Lambdas

Consider a table of departments where each department has an array of ROW elements representing employees. The following example searches for a specific employee name in those records. The results show that Alice works (or has worked) for two departments:

```
=> SELECT deptID, ARRAY_FIND(employees, e -> e.name = 'Alice Adams') AS 'has_alice'
   FROM departments;
 deptID | has_alice
--------+-----------
      1 |         0
      2 |        -1
      3 |         0
(3 rows)
```

In the following example, each person in the table has an array of email addresses, and the function locates fake addresses. The function takes one argument, the array element to test, and calls a regular-expression function that returns a Boolean:

```
=> SELECT name, ARRAY_FIND(email, e -> REGEXP_LIKE(e,'example.com','i'))
                AS 'example.com'
   FROM people;
      name       | example.com
-----------------+-------------
 Elaine Jackson |          -1
 Frank Adams    |           0
 Lee Jones      |          -1
 M Smith        |           0
(4 rows)
```

See also

- CONTAINS

- FILTER

CONTAINS

Returns true if the specified element is found in the collection and false if not. This function uses null-safe equality checks when testing elements.

Behavior type

Immutable

Syntax

```
CONTAINS(collection, { value | lambda-expression })
```

Arguments

collection

Target collection (ARRAY or SET).

value

Value to search for; type must match or be coercible to the element type of the collection.

lambda-expression

Lambda function to apply to each element. The function must return a Boolean value. The first argument to the function is the element, and the optional second element is the index of the element.

Examples

```
=> SELECT CONTAINS(SET[1,2,3,4],2);
 contains
----------
t
(1 row)
```

You can search for NULL as an element value:

```
=> SELECT CONTAINS(ARRAY[1,null,2],null);
 contains
----------
 t
(1 row)
```

You can search for complex element types:

```
=> SELECT CONTAINS(ARRAY[ARRAY[1,2,3],ARRAY[1,null,4]],
                   ARRAY[1,2,3]);
 CONTAINS
----------
 t
(1 row)

=> SELECT CONTAINS(ARRAY[ARRAY[1,2,3],ARRAY[1,null,4]],
                   ARRAY[1,null,4]);
 CONTAINS
----------
 t
(1 row)
```

The second example, comparing arrays with null elements, returns true because CONTAINS uses a null-safe equality check when evaluating elements.

In the following example, the orders table has the following definition:

```
=> CREATE EXTERNAL TABLE orders(
 orderid int,
 accountid int,
 shipments Array[
   ROW(
     shipid int,
     address ROW(
       street varchar,
       city varchar,
       zip int
     ),
     shipdate date
   )
 ]
) AS COPY FROM '...' PARQUET;
```

The following query tests for a specific order. When passing a ROW literal as the second argument, cast any ambiguous fields to ensure type matches:

```
=> SELECT CONTAINS(shipments,
          ROW(1,ROW('911 San Marcos St'::VARCHAR,
               'Austin'::VARCHAR, 73344),
          '2020-11-05'::DATE))
  FROM orders;
 CONTAINS
----------
 t
 f
 f
(3 rows)
```

Lambdas

Consider a table of departments where each department has an array of ROW elements representing employees. The following query finds departments with early hires (low employee IDs):

```
=> SELECT deptID FROM departments
   WHERE CONTAINS(employees, e -> e.id < 20);
 deptID
--------
      1
      3
(2 rows)
```

In the following example, a schedules table includes an array of events, where each event is a ROW with several fields:

```
=> CREATE TABLE schedules
      (guest VARCHAR,
      events ARRAY[ROW(e_date DATE, e_name VARCHAR, price NUMERIC(8,2))]);
```

You can use the CONTAINS function with a lambda expression to find people who have more than one event on the same day. The second argument, idx , is the index of the current element:

```
=> SELECT guest FROM schedules
WHERE CONTAINS(events, (e, idx) ->
                    (idx < ARRAY_LENGTH(events) - 1)
                    AND (e.e_date = events[idx + 1].e_date));
   guest
-------------
 Alice Adams
(1 row)
```

See also

- ARRAY_FIND
- FILTER

EXPLODE

Expands the elements of one or more collection columns (ARRAY or SET) into individual table rows, one row per element. For each exploded collection, the results include two columns, one for the element index, and one for the value at that position. If the function explodes a single collection, these columns are named position and value by default. If the function explodes two or more collections, the columns for each collection are named pos_ *column-name* and val_ *column-name* . You can use an AS clause in the SELECT to change these column names.

EXPLODE is similar to UNNEST , which returns values but not positions.

By default, EXPLODE requires an OVER clause. If you set the skip_partitioning parameter to true, an OVER clause is not required and is ignored if present.

Behavior type

Immutable

Syntax

```
EXPLODE (column[,...] [USING PARAMETERS param=value])
[ OVER ( [window-partition-clause] ) ]
```

Arguments

column

Column in the table being queried. You must specify at least as many collection columns as the value of the explode_count parameter. Columns that are not collections are passed through without modification.

Passthrough columns are not needed if skip_partitioning is true.

OVER(...)

How to partition and sort input data. The input data is the result set that the query returns after it evaluates FROM, WHERE, GROUP BY, and HAVING clauses. For EXPLODE, use OVER() or OVER(PARTITION BEST).

This clause is ignored if skip_partitioning is true.

Parameters

explode_count

The number of collection columns to explode. The function checks each column, up to this value, and either explodes it if is a collection or passes it through if it is not a collection or if this limit has been reached. If the value of explode_count is greater than the number of collection columns specified, the function returns an error.

Default: 1

skip_partitioning

Whether to skip partitioning and ignore the OVER clause if present. EXPLODE translates a single row of input into multiple rows of output, one per collection element. There is, therefore, usually no benefit to partitioning the input first. Skipping partitioning can help a query avoid an expensive sort or merge operation. Even so, setting this parameter can negatively affect performance in rare cases.

Default: false

Null-handling

This function expands each element in a collection into a row, including null elements. If the input column is NULL or an empty collection, the function produces no rows for that column.

Joining on results

You can use the output of EXPLODE as if it were a relation by using CROSS JOIN EXPLODE in a query. This operation is not a full Cartesian product because it ignores null or empty arrays. CROSS JOIN EXPLODE requires that you set the skip_partitioning parameter, either in the EXPLODE call or as a session parameter.

Consider the following table of students and exam scores:

```
=> SELECT * FROM tests;
 student |    scores     |    questions
---------+---------------+-----------------
 Bob     | [92,78,79]    | [20,20,100]
 Sam     | [97,98,85]    | [20,20,100]
 Tom     | [68,75,82,91] | [20,20,100,100]
(3 rows)
```

The following query finds the best test scores across all students:

```
=> ALTER SESSION SET UDPARAMETER FOR ComplexTypesLib skip_partitioning = true;

=> SELECT student, score FROM tests
   CROSS JOIN EXPLODE(scores) AS t (pos, score)
   ORDER BY score DESC;
 student | score
---------+-------
 Sam     |    98
 Sam     |    97
 Bob     |    92
 Tom     |    91
 Sam     |    85
 Tom     |    82
 Bob     |    79
 Bob     |    78
 Tom     |    75
 Tom     |    68
(10 rows)
```

The following query returns maximum and average per-question scores, considering both the exam score and the number of questions:

```
=> SELECT student, MAX(score/qcount), AVG(score/qcount) FROM tests
   CROSS JOIN EXPLODE(scores, questions USING PARAMETERS explode_count=2)
      AS t(pos_s, score, pos_q, qcount)
   GROUP BY student;
 student |         MAX          |       AVG
---------+----------------------+------------------
 Bob     | 4.600000000000000000 | 3.04333333333333
 Sam     | 4.900000000000000000 | 3.42222222222222
 Tom     | 4.550000000000000000 |             2.37
(3 rows)
```

Examples

Consider an orders table with the following contents:

```
=> SELECT orderkey, custkey, prodkey, orderprices, email_addrs
   FROM orders LIMIT 5;
  orderkey  | custkey |                  prodkey                  |           orderprices           |                                                  email_addrs
------------+---------+-------------------------------------------+---------------------------------+-------------------------------------------------------------------------------------------------
----
 113-341987 |  342799 | ["MG-7190 ","VA-4028 ","EH-1247 ","MS-7018 "] | ["60.00","67.00","22.00","14.99"] |
["bob@example,com","robert.jones@example.com"]
 111-952000 |  342845 | ["ID-2586 ","IC-9010 ","MH-2401 ","JC-1905 "] | ["22.00","35.00",null,"12.00"]    | ["br92@cs.example.edu"]
 111-345634 |  342536 | ["RS-0731 ","SJ-2021 "]                       | ["50.00",null]                    | [null]
 113-965086 |  342176 | ["GW-1808 "]                                  | ["108.00"]                        | ["joe.smith@example.com"]
 111-335121 |  342321 | ["TF-3556 "]                                  | ["50.00"]                         | ["789123@example-
isp.com","alexjohnson@example.com","monica@eng.example.com","sara@johnson.example.name",null]
(5 rows)
```

The following query explodes the order prices for a single customer. The other two columns are passed through and are repeated for each returned row:

```
=> SELECT EXPLODE(orderprices, custkey, email_addrs
                  USING PARAMETERS skip_partitioning=true)
          AS (position, orderprices, custkey, email_addrs)
   FROM orders WHERE custkey='342845' ORDER BY orderprices;
 position | orderprices | custkey |         email_addrs
----------+-------------+---------+-----------------------------
        2 |             |  342845 | ["br92@cs.example.edu",null]
        3 |       12.00 |  342845 | ["br92@cs.example.edu",null]
        0 |       22.00 |  342845 | ["br92@cs.example.edu",null]
        1 |       35.00 |  342845 | ["br92@cs.example.edu",null]
(4 rows)
```

The previous example uses the skip_partitioning parameter. Instead of setting it for each call to EXPLODE, you can set it as a session parameter. EXPLODE is part of the ComplexTypesLib UDx library. The following example returns the same results:

```
=> ALTER SESSION SET UDPARAMETER FOR ComplexTypesLib skip_partitioning=true;

=> SELECT EXPLODE(orderprices, custkey, email_addrs)
          AS (position, orderprices, custkey, email_addrs)
   FROM orders WHERE custkey='342845' ORDER BY orderprices;
```

You can explode more than one column by specifying the explode_count parameter:

```
=> SELECT EXPLODE(orderkey, prodkey, orderprices
                  USING PARAMETERS explode_count=2, skip_partitioning=true)
         AS (orderkey,pk_idx,pk_val,ord_idx,ord_val)
   FROM orders
   WHERE orderkey='113-341987';
  orderkey  | pk_idx |  pk_val  | ord_idx | ord_val
------------+--------+----------+---------+---------
 113-341987 |      0 | MG-7190  |       0 |   60.00
 113-341987 |      0 | MG-7190  |       1 |   67.00
 113-341987 |      0 | MG-7190  |       2 |   22.00
 113-341987 |      0 | MG-7190  |       3 |   14.99
 113-341987 |      1 | VA-4028  |       0 |   60.00
 113-341987 |      1 | VA-4028  |       1 |   67.00
 113-341987 |      1 | VA-4028  |       2 |   22.00
 113-341987 |      1 | VA-4028  |       3 |   14.99
 113-341987 |      2 | EH-1247  |       0 |   60.00
 113-341987 |      2 | EH-1247  |       1 |   67.00
 113-341987 |      2 | EH-1247  |       2 |   22.00
 113-341987 |      2 | EH-1247  |       3 |   14.99
 113-341987 |      3 | MS-7018  |       0 |   60.00
 113-341987 |      3 | MS-7018  |       1 |   67.00
 113-341987 |      3 | MS-7018  |       2 |   22.00
 113-341987 |      3 | MS-7018  |       3 |   14.99
(16 rows)
```

The following example uses a multi-dimensional array:

```
=> SELECT name, pingtimes FROM network_tests;
 name |                    pingtimes
------+-------------------------------------------------
 eng1 | [[24.24,25.27,27.16,24.97],[23.97,25.01,28.12,29.5]]
 eng2 | [[27.12,27.91,28.11,26.95],[29.01,28.99,30.11,31.56]]
 qa1  | [[23.15,25.11,24.63,23.91],[22.85,22.86,23.91,31.52]]
(3 rows)

=> SELECT EXPLODE(name, pingtimes USING PARAMETERS explode_count=1) OVER()
   FROM network_tests;
 name | position |          value
------+----------+--------------------------
 eng1 |        0 | [24.24,25.27,27.16,24.97]
 eng1 |        1 | [23.97,25.01,28.12,29.5]
 eng2 |        0 | [27.12,27.91,28.11,26.95]
 eng2 |        1 | [29.01,28.99,30.11,31.56]
 qa1  |        0 | [23.15,25.11,24.63,23.91]
 qa1  |        1 | [22.85,22.86,23.91,31.52]
(6 rows)
```

You can rewrite the previous query as follows to produce the same results:

```
=> SELECT name, EXPLODE(pingtimes USING PARAMETERS skip_partitioning=true)
   FROM network_tests;
```

FILTER

Takes an input array and returns an array containing only elements that meet a specified condition. This function uses null-safe equality checks when testing elements.

Behavior type

Immutable

Syntax

```
FILTER(array, lambda-expression )
```

Arguments

array

Input array.

lambda-expression

Lambda function to apply to each element. The function must return a Boolean value. The first argument to the function is the element, and the optional second element is the index of the element.

Examples

Given a table that contains names and arrays of email addresses, the following query filters out fake email addresses and returns the rest:

```
=> SELECT name, FILTER(email, e -> NOT REGEXP_LIKE(e,'example.com','i')) AS 'real_email'
   FROM people;
     name      |                  real_email
---------------+----------------------------------------------
 Elaine Jackson | ["ejackson@somewhere.org","elaine@jackson.com"]
 Frank Adams    | []
 Lee Jones      | ["lee.jones@somewhere.org"]
 M Smith        | ["ms@msmith.com"]
(4 rows)
```

You can use the results in a WHERE clause to exclude rows that no longer contain any email addresses:

```
=> SELECT name, FILTER(email, e -> NOT REGEXP_LIKE(e,'example.com','i')) AS 'real_email'
   FROM people
   WHERE ARRAY_LENGTH(real_email) > 0;
     name      |                  real_email
---------------+-----------------------------------------------
 Elaine Jackson | ["ejackson@somewhere.org","elaine@jackson.com"]
 Lee Jones      | ["lee.jones@somewhere.org"]
 M Smith        | ["ms@msmith.com"]
(3 rows)
```

See also

- CONTAINS
- ARRAY_FIND

IMPLODE

Takes a column of any scalar type and returns an unbounded array. Combined with GROUP BY, this function can be used to reverse an EXPLODE operation.

Behavior type

- Immutable if the WITHIN GROUP ORDER BY clause specifies a column or set of columns that resolves to unique element values within each output array group.
- Volatile otherwise because results are non-commutative.

Syntax

```
IMPLODE (input-column [ USING PARAMETERS param=value[,...] ] )
   [ within-group-order-by-clause ]
```

Arguments

input-column

Column of any scalar type from which to create the array.

[within-group-order-by-clause](/sql-reference/functions/aggregate-functions/within-group-order-by-clause.html)

Sorts elements within each output array group:

```
WITHIN GROUP (ORDER BY { column-expression[ sort-qualifiers ] }[,...])
```

sort-qualifiers : { ASC | DESC [NULLS { FIRST | LAST | AUTO }] }

> **Tip**
> WITHIN GROUP ORDER BY can consume a large amount of memory per group. To minimize memory consumption, create projections that support GROUPBY PIPELINED .

Parameters

allow_truncate

Boolean, if true truncates results when output length exceeds column size. If false (the default), the function returns an error if the output array is too large.

Even if this parameter is set to true, IMPLODE returns an error if any single array element is too large. Truncation removes elements from the output array but does not alter individual elements.

max_binary_size

The maximum binary size in bytes for the returned array. If you omit this parameter, IMPLODE uses the value of the configuration parameter DefaultArrayBinarySize .

Examples

Consider a table with the following contents:

```
=> SELECT * FROM filtered;

 position | itemprice | itemkey
----------+-----------+---------
        0 |     14.99 |     345
        0 |     27.99 |     567
        1 |     18.99 |     567
        1 |     35.99 |     345
        2 |     14.99 |     123
(5 rows)
```

The following query calls IMPLODE to assemble prices into arrays (grouped by keys):

```
=> SELECT itemkey AS key, IMPLODE(itemprice) AS prices
    FROM filtered GROUP BY itemkey ORDER BY itemkey;
 key |      prices
-----+------------------
 123 | ["14.99"]
 345 | ["35.99","14.99"]
 567 | ["27.99","18.99"]
(3 rows)
```

You can modify this query by including a WITHIN GROUP ORDER BY clause, which specifies how to sort array elements within each group:

```
=> SELECT itemkey AS key, IMPLODE(itemprice) WITHIN GROUP (ORDER BY itemprice) AS prices
    FROM filtered GROUP BY itemkey ORDER BY itemkey;
 key |      prices
-----+------------------
 123 | ["14.99"]
 345 | ["14.99","35.99"]
 567 | ["18.99","27.99"]
(3 rows)
```

See Arrays and sets (collections) for a fuller example.

SET_UNION

Returns a SET containing all elements of two input sets.

If the inputs are both bounded, the bound for the result is the sum of the bounds of the inputs.

If any input is unbounded, the result is unbounded with a binary size that is the sum of the sizes of the inputs.

Behavior type

Immutable

Syntax

```
SET_UNION(set1,set2)
```

Arguments

set1 **,** ***set2***

Sets of matching element type

Null-handling

- Null arguments are ignored. If one of the inputs is null, the function returns the non-null input. In other words, an argument of NULL is equivalent to SET[].
- If both inputs are null, the function returns null.

Examples

```
=> SELECT SET_UNION(SET[1,2,4], SET[2,3,4,5.9]);
set_union
----------------------
["1.0","2.0","3.0","4.0","5.9"]
(1 row)
```

STRING_TO_ARRAY

Splits a string containing array values and returns a native one-dimensional array. The output does not include the "ARRAY" keyword. This function does not support nested (multi-dimensional) arrays.

This function returns array elements as strings by default. You can cast to other types, as in the following example:

```
=> SELECT STRING_TO_ARRAY('[1,2,3]')::ARRAY[INT];
```

Behavior

Immutable

Syntax

```
STRING_TO_ARRAY(string [USING PARAMETERS param=value[,...]])
```

The following syntax is deprecated:

```
STRING_TO_ARRAY(string, delimiter)
```

Arguments

string

String representation of a one-dimensional array; can be a VARCHAR or LONG VARCHAR column, a literal string, or the string output of an expression.

Spaces in the string are removed unless elements are individually quoted. For example, ' a,b,c' is equivalent to 'a,b,c' . To preserve the space, use '" a","b","c"' .

Parameters

These parameters behave the same way as the corresponding options when loading delimited data (see DELIMITED).

No parameter may have the same value as any other parameter.

collection_delimiter

The character or character sequence used to separate array elements (VARCHAR(8)). You can use any ASCII values in the range E'\000' to E'\177', inclusive.

Default: Comma (',').

collection_open , collection_close

The characters that mark the beginning and end of the array (VARCHAR(8)). It is an error to use these characters elsewhere within the list of elements without escaping them. These characters can be omitted from the input string.

Default: Square brackets ('[' and ']').

collection_null_element

The string representing a null element value (VARCHAR(65000)). You can specify a null value using any ASCII values in the range E'\001' to E'\177' inclusive (any ASCII value except NULL: E'\000').

Default: 'null'

collection_enclose

An optional quote character within which to enclose individual elements, allowing delimiter characters to be embedded in string values. You can choose any ASCII value in the range E'\001' to E'\177' inclusive (any ASCII character except NULL: E'\000'). Elements do not need to be enclosed by this value.

Default: double quote ('"')

Examples

The function uses comma as the default delimiter. You can specify a different value:

```
=> SELECT STRING_TO_ARRAY('[1,3,5]');
 STRING_TO_ARRAY
-----------------
 ["1","3","5"]
(1 row)

=> SELECT STRING_TO_ARRAY('[t|t|f|t]' USING PARAMETERS collection_delimiter = '|');
  STRING_TO_ARRAY
-------------------
 ["t","t","f","t"]
(1 row)
```

The bounding brackets are optional:

```
=> SELECT STRING_TO_ARRAY('t|t|f|t' USING PARAMETERS collection_delimiter = '|');
  STRING_TO_ARRAY
-------------------
 ["t","t","f","t"]
(1 row)
```

The input can use other characters for open and close:

```
=> SELECT STRING_TO_ARRAY('{NASA-1683,NASA-7867,SPX-76}' USING PARAMETERS collection_open = '{', collection_close = '}');
         STRING_TO_ARRAY
-----------------------------------
 ["NASA-1683","NASA-7867","SPX-76"]
(1 row)
```

By default the string 'null' in input is treated as a null value:

```
=> SELECT STRING_TO_ARRAY('{"us-1672",null,"darpa-1963"}' USING PARAMETERS collection_open = '{', collection_close = '}');
       STRING_TO_ARRAY
-------------------------------
 ["us-1672",null,"darpa-1963"]
(1 row)
```

In the following example, the input comes from a column:

```
=> SELECT STRING_TO_ARRAY(name USING PARAMETERS collection_delimiter=' ') FROM employees;
    STRING_TO_ARRAY
------------------------
 ["Howard","Wolowitz"]
 ["Sheldon","Cooper"]
(2 rows)
```

TO_JSON

Returns the JSON representation of a complex-type argument, including mixed and nested complex types. This is the same format that queries of complex-type columns return.

Behavior

Immutable

Syntax

```
TO_JSON(value)
```

Arguments

value

Column or literal of a complex type

Examples

These examples query the following table:

```
=> SELECT name, contact FROM customers;
       name       |                                                
```

OVER(...)

How to partition and sort input data. The input data is the result set that the query returns after it evaluates FROM, WHERE, GROUP BY, and HAVING clauses.

This clause only applies if skip_partitioning is false.

Parameters

skip_partitioning

Whether to skip partitioning and ignore the OVER clause if present. UNNEST translates a single row of input into multiple rows of output, one per collection element. There is, therefore, usually no benefit to partitioning the input first. Skipping partitioning can help a query avoid an expensive sort or merge operation.

Default: true

Null-handling

This function expands each element in a collection into a row, including null elements. If the input column is NULL or an empty collection, the function produces no rows for that column.

Examples

Consider a table with the following definition:

```
=> CREATE TABLE orders (
      orderkey VARCHAR, custkey INT,
      prodkey ARRAY[VARCHAR], orderprices ARRAY[DECIMAL(12,2)],
      email_addrs ARRAY[VARCHAR]);
```

The following query expands one of the array columns. One of the elements is null:

```
=> SELECT UNNEST(orderprices) AS price, custkey, email_addrs
   FROM orders WHERE custkey='342845' ORDER BY price;
 price | custkey |       email_addrs
-------+---------+--------------------------
       |  342845 | ["br92@cs.example.edu"]
 12.00 |  342845 | ["br92@cs.example.edu"]
 22.00 |  342845 | ["br92@cs.example.edu"]
 35.00 |  342845 | ["br92@cs.example.edu"]
(4 rows)
```

UNNEST can expand more than one column:

```
=> SELECT orderkey, UNNEST(prodkey, orderprices)
   FROM orders WHERE orderkey='113-341987';
  orderkey  | val_prodkey | val_orderprices
------------+-------------+-----------------
 113-341987 | MG-7190     |           60.00
 113-341987 | MG-7190     |           67.00
 113-341987 | MG-7190     |           22.00
 113-341987 | MG-7190     |           14.99
 113-341987 | VA-4028     |           60.00
 113-341987 | VA-4028     |           67.00
 113-341987 | VA-4028     |           22.00
 113-341987 | VA-4028     |           14.99
 113-341987 | EH-1247     |           60.00
 113-341987 | EH-1247     |           67.00
 113-341987 | EH-1247     |           22.00
 113-341987 | EH-1247     |           14.99
 113-341987 | MS-7018     |           60.00
 113-341987 | MS-7018     |           67.00
 113-341987 | MS-7018     |           22.00
 113-341987 | MS-7018     |           14.99
(16 rows)
```

Date/time functions

Date and time functions perform conversion, extraction, or manipulation operations on date and time data types and can return date and time information.

Usage

Functions that take TIME or TIMESTAMP inputs come in two variants:

- TIME WITH TIME ZONE or TIMESTAMP WITH TIME ZONE
- TIME WITHOUT TIME ZONE or TIMESTAMP WITHOUT TIME ZONE

For brevity, these variants are not shown separately.

The + and * operators come in commutative pairs; for example, both DATE + INTEGER and INTEGER + DATE . We show only one of each such pair.

Daylight savings time considerations

When adding an INTERVAL value to (or subtracting an INTERVAL value from) a TIMESTAMP WITH TIME ZONE value, the days component advances (or decrements) the date of the TIMESTAMP WITH TIME ZONE by the indicated number of days. Across daylight saving time changes (with the session time zone set to a time zone that recognizes DST), this means INTERVAL '1 day' does not necessarily equal INTERVAL '24 hours' .

For example, with the session time zone set to CST7CDT :

```
TIMESTAMP WITH TIME ZONE '2014-04-02 12:00-07' + INTERVAL '1 day'
```

produces

```
TIMESTAMP WITH TIME ZONE '2014-04-03 12:00-06'
```

Adding INTERVAL '24 hours' to the same initial TIMESTAMP WITH TIME ZONE produces

```
TIMESTAMP WITH TIME ZONE '2014-04-03 13:00-06',
```

This result occurs because there is a change in daylight saving time at 2014-04-03 02:00 in time zone CST7CDT .

Date/time functions in transactions

Certain date/time functions such as CURRENT_TIMESTAMP and NOW return the start time of the current transaction; for the duration of that transaction, they return the same value. Other date/time functions such as TIMEOFDAY always return the current time.

See also

Template patterns for date/time formatting

In this section

- ADD_MONTHS
- AGE_IN_MONTHS
- AGE_IN_YEARS
- CLOCK_TIMESTAMP
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- DATE
- DATE_PART
- DATE_TRUNC
- DATEDIFF
- DAY
- DAYOFMONTH
- DAYOFWEEK
- DAYOFWEEK_ISO
- DAYOFYEAR
- DAYS
- EXTRACT
- GETDATE
- GETUTCDATE
- HOUR
- ISFINITE
- JULIAN_DAY
- LAST_DAY

- LOCALTIME
- LOCALTIMESTAMP
- MICROSECOND
- MIDNIGHT_SECONDS
- MINUTE
- MONTH
- MONTHS_BETWEEN
- NEW_TIME
- NEXT_DAY
- NOW [date/time]
- OVERLAPS
- QUARTER
- ROUND
- SECOND
- STATEMENT_TIMESTAMP
- SYSDATE
- TIME_SLICE
- TIMEOFDAY
- TIMESTAMP_ROUND
- TIMESTAMP_TRUNC
- TIMESTAMPADD
- TIMESTAMPDIFF
- TRANSACTION_TIMESTAMP
- TRUNC
- WEEK
- WEEK_ISO
- YEAR
- YEAR_ISO

ADD_MONTHS

Adds the specified number of months to a date and returns the sum as a DATE . In general, ADD_MONTHS returns a date with the same day component as the start date. For example:

```
=> SELECT ADD_MONTHS ('2015-09-15'::date, -2) "2 Months Ago";
 2 Months Ago
--------------
 2015-07-15
(1 row)
```

Two exceptions apply:

- If the start date's day component is greater than the last day of the result month, ADD_MONTHS returns the last day of the result month. For example:

  ```
  => SELECT ADD_MONTHS ('31-Jan-2016'::TIMESTAMP, 1) "Leap Month";
   Leap Month
  ------------
   2016-02-29
  (1 row)
  ```

- If the start date's day component is the last day of that month, and the result month has more days than the start date month, ADD_MONTHS returns the last day of the result month. For example:

  ```
  => SELECT ADD_MONTHS ('2015-09-30'::date,-1) "1 Month Ago";
   1 Month Ago
  -------------
   2015-08-31
  (1 row)
  ```

Behavior type

- Immutable if the *start-date* argument is a TIMESTAMP or DATE
- Stable if the *start-date* argument is a TIMESTAMPTZ

Syntax

```
ADD_MONTHS ( start-date, num-months );
```

Parameters

start-date

The date to process, an expression that evaluates to one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMPTZ

num-months

An integer expression that specifies the number of months to add to or subtract from *start-date* .

Examples

Add one month to the current date:

```
=> SELECT CURRENT_DATE Today;
   Today
------------
 2016-05-05
(1 row)

VMart=> SELECT ADD_MONTHS(CURRENT_TIMESTAMP,1);
 ADD_MONTHS
------------
 2016-06-05
(1 row)
```

Subtract four months from the current date:

```
=> SELECT ADD_MONTHS(CURRENT_TIMESTAMP, -4);
 ADD_MONTHS
------------
 2016-01-05
(1 row)
```

Add one month to January 31 2016:

```
=> SELECT ADD_MONTHS('31-Jan-2016'::TIMESTAMP, 1) "Leap Month";
 Leap Month
------------
 2016-02-29
(1 row)
```

The following example sets the timezone to EST; it then adds 24 months to a TIMESTAMPTZ that specifies a PST time zone, so ADD_MONTHS takes into account the time change:

```
=> SET TIME ZONE 'America/New_York';
SET
VMart=> SELECT ADD_MONTHS('2008-02-29 23:30 PST'::TIMESTAMPTZ, 24);
 ADD_MONTHS
------------
 2010-03-01
(1 row)
```

AGE_IN_MONTHS

Returns the difference in months between two dates, expressed as an integer.

Behavior type

- Immutable if both date arguments are of data type TIMESTAMP
- Stable if either date is a TIMESTAMPTZ or only one argument is supplied

Syntax

```
AGE_IN_MONTHS ( [ date1,] date2 )
```

Parameters

date1

date2

Specify the boundaries of the period to measure. If you supply only one argument, Vertica sets *date2* to the current date. Both parameters must evaluate to one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMPTZ

If *date1* < *date2* , AGE_IN_MONTHS returns a negative value.

Examples

Get the age in months of someone born March 2 1972, as of June 21 1990:

```
=> SELECT AGE_IN_MONTHS('1990-06-21'::TIMESTAMP, '1972-03-02'::TIMESTAMP);
 AGE_IN_MONTHS
---------------
           219
(1 row)
```

If the first date is less than the second date, AGE_IN_MONTHS returns a negative value

```
=> SELECT AGE_IN_MONTHS('1972-03-02'::TIMESTAMP, '1990-06-21'::TIMESTAMP);
AGE_IN_MONTHS
---------------
-220
(1 row)
```

Get the age in months of someone who was born November 21 1939, as of today:

```
=> SELECT AGE_IN_MONTHS ('1939-11-21'::DATE);
 AGE_IN_MONTHS
---------------
           930
(1 row)
```

AGE_IN_YEARS

Returns the difference in years between two dates, expressed as an integer.

Behavior type

- Immutable if both date arguments are of data type TIMESTAMP
- Stable if either date is a TIMESTAMPTZ or only one argument is supplied

Syntax

```
AGE_IN_YEARS( [ date1,] date2 )
```

Parameters

date1

date2

Specify the boundaries of the period to measure. If you supply only one argument, Vertica sets *date1* to the current date. Both parameters must evaluate to one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMPTZ

If *date1* < *date2* , AGE_IN_YEARS returns a negative value.

Examples

Get the age of someone born March 2 1972, as of June 21 1990:

```
=> SELECT AGE_IN_YEARS('1990-06-21'::TIMESTAMP, '1972-03-02'::TIMESTAMP);
 AGE_IN_YEARS
--------------
           18
(1 row)
```

If the first date is earlier than the second date, AGE_IN_YEARS returns a negative number:

```
=> SELECT AGE_IN_YEARS('1972-03-02'::TIMESTAMP, '1990-06-21'::TIMESTAMP);
AGE_IN_YEARS
--------------
          -19
(1 row)
```

Get the age of someone who was born November 21 1939, as of today:

```
=> SELECT AGE_IN_YEARS('1939-11-21'::DATE);
 AGE_IN_YEARS
--------------
           77
(1 row)
```

CLOCK_TIMESTAMP

Returns a value of type TIMESTAMP WITH TIMEZONE that represents the current system-clock time.

CLOCK_TIMESTAMP uses the date and time supplied by the operating system on the server to which you are connected, which should be the same across all servers. The value changes each time you call it.

Behavior type

Volatile

Syntax

```
CLOCK_TIMESTAMP()
```

Examples

The following command returns the current time on your system:

```
SELECT CLOCK_TIMESTAMP() "Current Time";
         Current Time
-------------------------------
 2010-09-23 11:41:23.33772-04
(1 row)
```

Each time you call the function, you get a different result. The difference in this example is in microseconds:

```
SELECT CLOCK_TIMESTAMP() "Time 1", CLOCK_TIMESTAMP() "Time 2";
             Time 1              |             Time 2
---------------------------------+---------------------------------
 2010-09-23 11:41:55.369201-04 | 2010-09-23 11:41:55.369202-04
(1 row)
```

See also

- STATEMENT_TIMESTAMP
- TRANSACTION_TIMESTAMP

CURRENT_DATE

Returns the date (date-type value) on which the current transaction started.

Behavior type

Stable

Syntax

```
CURRENT_DATE()
```

> **Note**
> You can call this function without parentheses.

Examples

```
SELECT CURRENT_DATE;
 ?column?
------------
 2010-09-23
(1 row)
```

CURRENT_TIME

Returns a value of type TIME WITH TIMEZONE that represents the start of the current transaction.

The return value does not change during the transaction. Thus, multiple calls to CURRENT_TIME within the same transaction return the same timestamp.

Behavior type

Stable

Syntax

```
CURRENT_TIME  [ ( precision ) ]
```

> **Note**
> If you specify a column label without precision, you must also omit parentheses.

Parameters

precision

An integer value between 0-6, specifies to round the seconds fraction field result to the specified number of digits.

Examples

```
=> SELECT CURRENT_TIME(1) AS Time;
    Time
---------------
 06:51:45.2-07
(1 row)
=> SELECT CURRENT_TIME(5) AS Time;
      Time
-------------------
 06:51:45.18435-07
(1 row)
```

CURRENT_TIMESTAMP

Returns a value of type TIME WITH TIMEZONE that represents the start of the current transaction.

The return value does not change during the transaction. Thus, multiple calls to CURRENT_TIMESTAMP within the same transaction return the same timestamp.

Behavior type

Stable

Syntax

```
CURRENT_TIMESTAMP ( precision )
```

Parameters

precision

An integer value between 0-6, specifies to round the seconds fraction field result to the specified number of digits.

Examples

```
=> SELECT CURRENT_TIMESTAMP(1) AS time;
          time
--------------------------
 2017-03-27 06:50:49.7-07
(1 row)
=> SELECT CURRENT_TIMESTAMP(5) AS time;
           time
-------------------------------
 2017-03-27 06:50:49.69967-07
(1 row)
```

DATE

Converts the input value to a DATE data type.

Behavior type

- Immutable if the input value is a TIMESTAMP , DATE , VARCHAR , or integer
- Stable if the input value is a TIMESTAMPTZ

Syntax

```
DATE ( value )
```

Parameters

value

The value to convert, one of the following:

- TIMESTAMP , TIMESTAMPTZ , VARCHAR , or another DATE .
- Integer: Vertica treats the integer as the number of days since 01/01/0001 and returns the date.

Examples

```
=> SELECT DATE (1);
   DATE
------------
 0001-01-01
(1 row)

=> SELECT DATE (734260);
   DATE
------------
 2011-05-03
(1 row)

=> SELECT DATE('TODAY');
   DATE
------------
 2016-12-07
(1 row)
```

See also

- TO_DATE
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ

DATE_PART

Extracts a sub-field such as year or hour from a date/time expression, equivalent to the the SQL-standard function EXTRACT .

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , or INTERVAL
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
DATE_PART ( 'field', date )
```

Parameters

field

A constant value that specifies the sub-field to extract from *date* (see Field Values below).

date

The date to process, an expression that evaluates to one of the following data types:

- DATE (cast to TIMESTAMP)
- TIMESTAMP/TIMESTAMPTZ
- INTERVAL

Field values

CENTURY

The century number.

The first century starts at 0001-01-01 00:00:00 AD. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from –1 to 1.

DAY

The day (of the month) field (1–31).

DECADE

The year field divided by 10.

DOQ

The day within the current quarter. DOQ recognizes leap year days.

DOW

Zero-based day of the week, where Sunday=0.

> **Note**
> EXTRACT 's day of week numbering differs from the function TO_CHAR .

DOY

The day of the year (1–365/366)

EPOCH

Specifies to return one of the following:

- For DATE and TIMESTAMP values: the number of seconds before or since 1970-01-01 00:00:00-00 (if before, a negative number).
- For INTERVAL values, the total number of seconds in the interval.

HOUR

The hour field (0–23).

ISODOW

The ISO day of the week, an integer between 1 and 7 where Monday is 1.

ISOWEEK

The ISO week of the year, an integer between 1 and 53.

ISOYEAR

The ISO year.

MICROSECONDS

The seconds field, including fractional parts, multiplied by 1,000,000. This includes full seconds.

MILLENNIUM

The millennium number, where the first millennium is 1 and each millenium starts on 01-01- *y* 001 . For example, millennium 2 starts on 01-01-1001.

MILLISECONDS

The seconds field, including fractional parts, multiplied by 1000. This includes full seconds.

MINUTE

The minutes field (0 - 59).

MONTH

For TIMESTAMP values, the number of the month within the year (1 - 12) ; for interval values the number of months, modulo 12 (0 - 11).

QUARTER

The calendar quarter of the specified date as an integer, where the January-March quarter is 1, valid only for TIMESTAMP values.

SECOND

The seconds field, including fractional parts, 0–59, or 0-60 if the operating system implements leap seconds.

TIME ZONE

The time zone offset from UTC, in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC.

TIMEZONE_HOUR

The hour component of the time zone offset.

TIMEZONE_MINUTE

The minute component of the time zone offset.

WEEK

The number of the week of the calendar year that the day is in.

YEAR

The year field. There is no 0 AD , so subtract BC years from AD years accordingly.

Notes

According to the ISO-8601 standard, the week starts on Monday, and the first week of a year contains January 4. Thus, an early January date can sometimes be in the week 52 or 53 of the previous calendar year. For example:

```
=> SELECT YEAR_ISO('01-01-2016'::DATE), WEEK_ISO('01-01-2016'), DAYOFWEEK_ISO('01-01-2016');
 YEAR_ISO | WEEK_ISO | DAYOFWEEK_ISO
----------+----------+---------------
     2015 |       53 |             5
(1 row)
```

Examples

Extract the day value:

```
SELECT DATE_PART('DAY', TIMESTAMP '2009-02-24 20:38:40') "Day";
  Day
-----
  24
(1 row)
```

Extract the month value:

```
SELECT DATE_PART('MONTH', '2009-02-24 20:38:40'::TIMESTAMP) "Month";
  Month
-------
     2
(1 row)
```

Extract the year value:

```
SELECT DATE_PART('YEAR', '2009-02-24 20:38:40'::TIMESTAMP) "Year";
  Year
------
 2009
(1 row)
```

Extract the hours:

```
SELECT DATE_PART('HOUR', '2009-02-24 20:38:40'::TIMESTAMP) "Hour";
  Hour
------
   20
(1 row)
```

Extract the minutes:

```
SELECT DATE_PART('MINUTES', '2009-02-24 20:38:40'::TIMESTAMP) "Minutes";
 Minutes
---------
      38
(1 row)
```

Extract the day of quarter (DOQ):

```
SELECT DATE_PART('DOQ', '2009-02-24 20:38:40'::TIMESTAMP) "DOQ";
 DOQ
-----
  55
(1 row)
```

See also

TO_CHAR

DATE_TRUNC

Truncates date and time values to the specified precision. The return value is the same data type as the input value. All fields that are less than the specified precision are set to 0, or to 1 for day and month.

Behavior type

Stable

Syntax

```
DATE_TRUNC( precision, trunc-target )
```

Parameters

precision

A string constant that specifies precision for the truncated value. See Precision Field Values below. The precision must be valid for the *trunc-target* date or time.

trunc-target

Valid date/time expression.

Precision field values

MILLENNIUM

The millennium number.

CENTURY

The century number.

The first century starts at 0001-01-01 00:00:00 AD. This definition applies to all Gregorian calendar countries.

DECADE

The year field divided by 10.

YEAR

The year field. Keep in mind there is no 0 AD , so subtract BC years from AD years with care.

QUARTER

The calendar quarter of the specified date as an integer, where the January-March quarter is 1.

MONTH

For timestamp values, the number of the month within the year (1–12) ; for interval values the number of months, modulo 12 (0–11).

WEEK

The number of the week of the year that the day is in.

According to the ISO-8601 standard, the week starts on Monday, and the first week of a year contains January 4. Thus, an early January date can sometimes be in the week 52 or 53 of the previous calendar year. For example:

```
=> SELECT YEAR_ISO('01-01-2016'::DATE), WEEK_ISO('01-01-2016'), DAYOFWEEK_ISO('01-01-2016');
 YEAR_ISO | WEEK_ISO | DAYOFWEEK_ISO
----------+----------+---------------
     2015 |       53 |             5
(1 row)
```

DAY

The day (of the month) field (1–31).

HOUR

The hour field (0–23).

MINUTE

The minutes field (0–59).

SECOND

The seconds field, including fractional parts (0–59) (60 if leap seconds are implemented by the operating system).

MILLISECONDS

The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.

MICROSECONDS

The seconds field, including fractional parts, multiplied by 1,000,000. This includes full seconds.

Examples

The following example sets the field value as hour and returns the hour, truncating the minutes and seconds:

```
=> SELECT DATE_TRUNC('HOUR', TIMESTAMP '2012-02-24 13:38:40') AS HOUR;
      HOUR
---------------------
 2012-02-24 13:00:00
(1 row)
```

The following example returns the year from the input timestamptz '2012-02-24 13:38:40' . The function also defaults the month and day to January 1, truncates the hour:minute:second of the timestamp, and appends the time zone (-05):

```
=> SELECT DATE_TRUNC('YEAR', TIMESTAMPTZ '2012-02-24 13:38:40') AS YEAR;
       YEAR
------------------------
 2012-01-01 00:00:00-05
(1 row)
```

The following example returns the year and month and defaults day of month to 1, truncating the rest of the string:

```
=> SELECT DATE_TRUNC('MONTH', TIMESTAMP '2012-02-24 13:38:40') AS MONTH;
      MONTH
---------------------
 2012-02-01 00:00:00
(1 row)
```

DATEDIFF

Returns the time span between two dates, in the intervals specified. DATEDIFF excludes the start date in its calculation.

Behavior type

- Immutable if start and end dates are TIMESTAMP , DATE , TIME , or INTERVAL
- Stable if start and end dates are TIMESTAMPTZ

Syntax

```
DATEDIFF ( datepart, start, end );
```

Parameters

datepart

Specifies the type of date or time intervals that DATEDIFF returns. If *datepart* is an expression, it must be enclosed in parentheses:

```
DATEDIFF((expression), start, end);
```

datepart must evaluate to one of the following string literals, either quoted or unquoted:

- year | yy | yyyy
- quarter | qq | q
- month | mm | m
- day | dayofyear | dd | d | dy | y

- week | wk | ww
- hour | hh
- minute | mi | n
- second | ss | s
- millisecond | ms
- microsecond | mcs | us

start **,** ***end***

Specify the start and end dates, where *start* and *end* evaluate to one of the following data types:

- TIMESTAMP/TIMESTAMPTZ
- DATE
- TIME/TIMETZ
- INTERVAL

If *end* < *start* , DATEDIFF returns a negative value.

> **Note**
> TIME and INTERVAL data types are invalid for start and end dates if *datepart* is set to year , quarter , or month .

Compatible start and end date data types

The following table shows which data types can be matched as start and end dates:

	DATE	TIMESTAMP	TIMESTAMPTZ	TIME	INTERVAL
DATE	•	•	•		
TIMESTAMP	•	•	•		
TIMESTAMPTZ	•	•	•		
TIME				•	
INTERVAL					•

For example, if you set the start date to an INTERVAL data type, the end date must also be an INTERVAL , otherwise Vertica returns an error:

```
SELECT DATEDIFF(day, INTERVAL '26 days', INTERVAL '1 month ');
 datediff
----------
        4
(1 row)
```

Date part intervals

DATEDIFF uses the *datepart* argument to calculate the number of intervals between two dates, rather than the actual amount of time between them. DATEDIFF uses the following cutoff points to calculate those intervals:

- year : January 1
- quarter : January 1, April 1, July 1, October 1
- month : the first day of the month
- week : Sunday at midnight (24:00)

For example, if *datepart* is set to year , DATEDIFF uses January 01 to calculate the number of years between two dates. The following DATEDIFF statement sets *datepart* to year , and specifies a time span 01/01/2005 - 06/15/2008:

```
SELECT DATEDIFF(year, '01-01-2005'::date, '12-31-2008'::date);
 datediff
----------
        3
(1 row)
```

DATEDIFF always excludes the start date when it calculates intervals—in this case, 01/01//2005. DATEDIFF considers only calendar year starts in its calculation, so in this case it only counts years 2006, 2007, and 2008. The function returns 3, although the actual time span is nearly four years.

If you change the start and end dates to 12/31/2004 and 01/01/2009, respectively, DATEDIFF also counts years 2005 and 2009. This time, it returns 5, although the actual time span is just over four years:

```
=> SELECT DATEDIFF(year, '12-31-2004'::date, '01-01-2009'::date);
 datediff
----------
        5
(1 row)
```

Similarly, DATEDIFF uses month start dates when it calculates the number of months between two dates. Thus, given the following statement, DATEDIFF counts months February through September and returns 8:

```
=> SELECT DATEDIFF(month, '01-31-2005'::date, '09-30-2005'::date);
 datediff
----------
        8
(1 row)
```

See also

TIMESTAMPDIFF

DAY

Returns as an integer the day of the month from the input value.

Behavior type

- Immutable if the input value is a TIMESTAMP , DATE , VARCHAR , or INTEGER
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
DAY ( value )
```

Parameters

value

The value to convert, one of the following: TIMESTAMP , TIMESTAMPTZ , INTERVAL , VARCHAR , or INTEGER .

Examples

```
=> SELECT DAY (6);
 DAY
-----
   6
(1 row)

=> SELECT DAY(TIMESTAMP 'sep 22, 2011 12:34');
 DAY
-----
  22
(1 row)

=> SELECT DAY('sep 22, 2011 12:34');
 DAY
-----
  22
(1 row)

=> SELECT DAY(INTERVAL '35 12:34');
 DAY
-----
  35
(1 row)
```

DAYOFMONTH

Returns the day of the month as an integer.

Behavior type

- Immutable if thetarget date is a TIMESTAMP , DATE , or VARCHAR
- Stable if the target date is a TIMESTAMPTZ

Syntax

```
DAYOFMONTH ( date )
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

Examples

```
=> SELECT DAYOFMONTH (TIMESTAMP 'sep 22, 2011 12:34');
 DAYOFMONTH
------------
         22
(1 row)
```

DAYOFWEEK

Returns the day of the week as an integer, where Sunday is day 1.

Behavior type

- Immutable if thetarget date is a TIMESTAMP , DATE , or VARCHAR
- Stable if the target date is a TIMESTAMPTZ

Syntax

```
DAYOFWEEK ( date )
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

Examples

```
=> SELECT DAYOFWEEK (TIMESTAMP 'sep 17, 2011 12:34');
 DAYOFWEEK
-----------
         7
(1 row)
```

DAYOFWEEK_ISO

Returns the ISO 8061 day of the week as an integer, where Monday is day 1.

Behavior type

- Immutable if thetarget date is a TIMESTAMP , DATE , or VARCHAR
- Stable if the target date is a TIMESTAMPTZ

Syntax

```
DAYOFWEEK_ISO ( date )
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

Examples

```
=> SELECT DAYOFWEEK_ISO(TIMESTAMP 'Sep 22, 2011 12:34');
 DAYOFWEEK_ISO
---------------
             4
(1 row)
```

The following example shows how to combine the DAYOFWEEK_ISO, WEEK_ISO, and YEAR_ISO functions to find the ISO day of the week, week, and year:

```
=> SELECT DAYOFWEEK_ISO('Jan 1, 2000'), WEEK_ISO('Jan 1, 2000'),YEAR_ISO('Jan1,2000');
 DAYOFWEEK_ISO | WEEK_ISO | YEAR_ISO
---------------+----------+----------
             6 |       52 |     1999
(1 row)
```

See also

- WEEK_ISO
- DAYOFWEEK_ISO
- http://en.wikipedia.org/wiki/ISO_8601

DAYOFYEAR

Returns the day of the year as an integer, where January 1 is day 1.

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , or VARCHAR
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
DAYOFYEAR ( date )
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

Examples

```
=> SELECT DAYOFYEAR (TIMESTAMP 'SEPT 22,2011 12:34');
 DAYOFYEAR
-----------
       265
(1 row)
```

DAYS

Returns the integer value of the specified date, where 1 AD is 1. If the date precedes 1 AD, DAYS returns a negative integer.

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , or VARCHAR
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
DAYS ( date )
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

Examples

```
=> SELECT DAYS (DATE '2011-01-22');
  DAYS
--------
 734159
(1 row)

=> SELECT DAYS (DATE 'March 15, 0044 BC');
  DAYS
--------
 -15997
(1 row)
```

EXTRACT

Retrieves sub-fields such as year or hour from date/time values and returns values of type NUMERIC . EXTRACT is intended for computational processing, rather than for formatting date/time values for display.

Behavior type

- Immutable if the specified date is a TIMESTAMP , DATE , or INTERVAL

- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
EXTRACT ( field FROM date )
```

Parameters

field

A constant value that specifies the sub-field to extract from *date* (see Field Values below).

date

The date to process, an expression that evaluates to one of the following data types:

- DATE (cast to TIMESTAMP)
- TIMESTAMP/TIMESTAMPTZ
- INTERVAL

Field values

CENTURY

The century number.

The first century starts at 0001-01-01 00:00:00 AD. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from –1 to 1.

DAY

The day (of the month) field (1–31).

DECADE

The year field divided by 10.

DOQ

The day within the current quarter. DOQ recognizes leap year days.

DOW

Zero-based day of the week, where Sunday=0.

> **Note**
> EXTRACT 's day of week numbering differs from the function TO_CHAR .

DOY

The day of the year (1–365/366)

EPOCH

Specifies to return one of the following:

- For DATE and TIMESTAMP values: the number of seconds before or since 1970-01-01 00:00:00-00 (if before, a negative number).
- For INTERVAL values, the total number of seconds in the interval.

HOUR

The hour field (0–23).

ISODOW

The ISO day of the week, an integer between 1 and 7 where Monday is 1.

ISOWEEK

The ISO week of the year, an integer between 1 and 53.

ISOYEAR

The ISO year.

MICROSECONDS

The seconds field, including fractional parts, multiplied by 1,000,000. This includes full seconds.

MILLENNIUM

The millennium number, where the first millennium is 1 and each millenium starts on 01-01- *y* 001 . For example, millennium 2 starts on 01-01-1001.

MILLISECONDS

The seconds field, including fractional parts, multiplied by 1000. This includes full seconds.

MINUTE

The minutes field (0 - 59).

MONTH

For TIMESTAMP values, the number of the month within the year (1 - 12) ; for interval values the number of months, modulo 12 (0 - 11).

QUARTER

The calendar quarter of the specified date as an integer, where the January-March quarter is 1, valid only for TIMESTAMP values.

SECOND

The seconds field, including fractional parts, 0–59, or 0-60 if the operating system implements leap seconds.

TIME ZONE

The time zone offset from UTC, in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC.

TIMEZONE_HOUR

The hour component of the time zone offset.

TIMEZONE_MINUTE

The minute component of the time zone offset.

WEEK

The number of the week of the calendar year that the day is in.

YEAR

The year field. There is no 0 AD , so subtract BC years from AD years accordingly.

Examples

Extract the day of the week and day in quarter from the current TIMESTAMP:

```
=> SELECT CURRENT_TIMESTAMP AS NOW;
              NOW
-------------------------------
 2016-05-03 11:36:08.829004-04
(1 row)
=> SELECT EXTRACT (DAY FROM CURRENT_TIMESTAMP);
 date_part
-----------
         3
(1 row)
=> SELECT EXTRACT (DOQ FROM CURRENT_TIMESTAMP);
 date_part
-----------
        33
(1 row)
```

Extract the timezone hour from the current time:

```
=> SELECT CURRENT_TIMESTAMP;
          ?column?
-------------------------------
 2016-05-03 11:36:08.829004-04
(1 row)

=>  SELECT EXTRACT(TIMEZONE_HOUR FROM CURRENT_TIMESTAMP);
 date_part
-----------
        -4
(1 row)
```

Extract the number of seconds since 01-01-1970 00:00:

```
=> SELECT EXTRACT(EPOCH FROM '2001-02-16 20:38:40-08'::TIMESTAMPTZ);
    date_part
------------------
 982384720.000000
(1 row)
```

Extract the number of seconds between 01-01-1970 00:00 and 5 days 3 hours before:

```
=> SELECT EXTRACT(EPOCH FROM -'5 days 3 hours'::INTERVAL);
  date_part
----------------
 -442800.000000
(1 row)
```

Convert the results from the last example to a TIMESTAMP:

```
=> SELECT 'EPOCH'::TIMESTAMPTZ -442800  * '1 second'::INTERVAL;
       ?column?
------------------------
 1969-12-26 16:00:00-05
(1 row)
```

GETDATE

Returns the current statement's start date and time as a TIMESTAMP value. This function is identical to SYSDATE .

GETDATE uses the date and time supplied by the operating system on the server to which you are connected, which is the same across all servers. Internally, GETDATE converts STATEMENT_TIMESTAMP from TIMESTAMPTZ to TIMESTAMP .

Behavior type

Stable

Syntax

```
GETDATE()
```

Examples

```
=> SELECT GETDATE();
        GETDATE
----------------------------
 2011-03-07 13:21:29.497742
(1 row)
```

See also

Date/time expressions

GETUTCDATE

Returns the current statement's start date and time as a TIMESTAMP value.

GETUTCDATE uses the date and time supplied by the operating system on the server to which you are connected, which is the same across all servers. Internally, GETUTCDATE converts STATEMENT_TIMESTAMP at TIME ZONE 'UTC'.

Behavior type

Stable

Syntax

```
GETUTCDATE()
```

Examples

```
=> SELECT GETUTCDATE();
       GETUTCDATE
----------------------------
 2011-03-07 20:20:26.193052
(1 row)
```

See also

- Date/time expressions

HOUR

Returns the hour portion of the specified date as an integer, where 0 is 00:00 to 00:59.

Behavior type

- Immutable if thespecified date is a TIMESTAMP
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
HOUR( date )
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ
- INTERVAL

Examples

```
=> SELECT HOUR (TIMESTAMP 'sep 22, 2011 12:34');
 HOUR
------
   12
(1 row)
=> SELECT HOUR (INTERVAL '35 12:34');
 HOUR
------
   12
(1 row)
=> SELECT HOUR ('12:34');
 HOUR
------
   12
(1 row)
```

ISFINITE

Tests for the special TIMESTAMP constant INFINITY and returns a value of type BOOLEAN.

Behavior type

Immutable

Syntax

```
ISFINITE ( timestamp )
```

Parameters

timestamp

Expression of type TIMESTAMP

Examples

```
SELECT ISFINITE(TIMESTAMP '2009-02-16 21:28:30');
 ISFINITE
----------
 t
(1 row)
SELECT ISFINITE(TIMESTAMP 'INFINITY');
 ISFINITE
----------
 f
(1 row)
```

JULIAN_DAY

Returns the integer value of the specified day according to the Julian calendar, where day 1 is the first day of the Julian period, January 1, 4713 BC (on the Gregorian calendar, November 24, 4714 BC).

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , or VARCHAR
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
JULIAN_DAY ( date )
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

Examples

```
=> SELECT JULIAN_DAY (DATE 'MARCH 15, 0044 BC');
 JULIAN_DAY
------------
    1705428
(1 row)

=> SELECT JULIAN_DAY (DATE '2001-01-01');
 JULIAN_DAY
------------
    2451911
(1 row)
```

LAST_DAY

Returns the last day of the month in the specified date.

Behavior type

- Immutable if the specified is a TIMESTAMP or DATE
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
LAST_DAY ( date )
```

Parameters

date

The date to process, one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMPTZ

Calculating first day of month

SQL does not support any function that returns the first day in the month of a given date. You must use other functions to work around this limitation. For example:

```
=> SELECT DATE ('2022/07/04') - DAYOFMONTH ('2022/07/04') +1;
 ?column?
------------
 2022-07-01
(1 row)

=> SELECT LAST_DAY('1929/06/06') - (SELECT DAY(LAST_DAY('1929/06/06'))-1);
 ?column?
------------
 1929-06-01
(1 row)
```

Examples

The following example returns the last day of February as 29 because 2016 is a leap year:

```
=> SELECT LAST_DAY('2016-02-28 23:30 PST') "Last Day";
 Last Day
------------
 2016-02-29
(1 row)
```

The following example returns the last day of February in a non-leap year:

```
> SELECT LAST_DAY('2017/02/03') "Last";
    Last
------------
 2017-02-28
(1 row)
```

The following example returns the last day of March, after converting the string value to the specified DATE type:

```
=> SELECT LAST_DAY('2003/03/15') "Last";
    Last
------------
 2012-03-31
(1 row)
```

LOCALTIME

Returns a value of type TIME that represents the start of the current transaction.

The return value does not change during the transaction. Thus, multiple calls to LOCALTIME within the same transaction return the same timestamp.

Behavior type

Stable

Syntax

```
LOCALTIME [ ( precision ) ]
```

Parameters

precision

Rounds the result to the specified number of fractional digits in the seconds field.

Examples

```
=> CREATE TABLE t1 (a int, b int);
CREATE TABLE

=> INSERT INTO t1 VALUES (1,2);
 OUTPUT
--------
      1
(1 row)

=> SELECT LOCALTIME time;
    time
-----------------
 15:03:14.595296
(1 row)

=> INSERT INTO t1 VALUES (3,4);
 OUTPUT
--------
      1
(1 row)

=> SELECT LOCALTIME;
    time
-----------------
 15:03:14.595296
(1 row)

=> COMMIT;
COMMIT
=> SELECT LOCALTIME;
    time
-----------------
 15:03:49.738032
(1 row)
```

LOCALTIMESTAMP

Returns a value of type TIMESTAMP/TIMESTAMPTZ that represents the start of the current transaction, and remains unchanged until the transaction is closed. Thus, multiple calls to LOCALTIMESTAMP within a given transaction return the same timestamp.

Behavior type

Stable

Syntax

```
LOCALTIMESTAMP [ ( precision ) ]
```

Parameters

precision

Rounds the result to the specified number of fractional digits in the seconds field.

Examples

```
=> CREATE TABLE t1 (a int, b int);
CREATE TABLE
=> INSERT INTO t1 VALUES (1,2);
 OUTPUT
--------
      1
(1 row)

=> SELECT LOCALTIMESTAMP(2) AS 'local timestamp';
    local timestamp
-------------------------
 2021-03-05 10:48:58.26
(1 row)

=> INSERT INTO t1 VALUES (3,4);
 OUTPUT
--------
      1
(1 row)

=> SELECT LOCALTIMESTAMP(2) AS 'local timestamp';
    local timestamp
-------------------------
 2021-03-05 10:48:58.26
(1 row)

=> COMMIT;
COMMIT
=> SELECT LOCALTIMESTAMP(2) AS 'local timestamp';
    local timestamp
-------------------------
 2021-03-05 10:50:08.99
(1 row)
```

MICROSECOND

Returns the microsecond portion of the specified date as an integer.

Behavior type

- Immutable if thespecified date is a TIMESTAMP , INTERVAL , or VARCHAR
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
MICROSECOND ( date )
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- TIMESTAMP
- TIMESTAMPTZ
- INTERVAL

Examples

```
=> SELECT MICROSECOND (TIMESTAMP 'Sep 22, 2011 12:34:01.123456');
 MICROSECOND
-------------
      123456
(1 row)
```

MIDNIGHT_SECONDS

Within the specified date, returns the number of seconds between midnight and the date's time portion.

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , or VARCHAR
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
MIDNIGHT_SECONDS ( date )
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

Examples

Get the number of seconds since midnight:

```
=> SELECT MIDNIGHT_SECONDS(CURRENT_TIMESTAMP);
 MIDNIGHT_SECONDS
------------------
            36480
(1 row)
```

Get the number of seconds between midnight and noon on March 3 2016:

```
=> SELECT MIDNIGHT_SECONDS('3-3-2016 12:00'::TIMESTAMP);
 MIDNIGHT_SECONDS
------------------
            43200
(1 row)
```

MINUTE

Returns the minute portion of the specified date as an integer.

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , VARCHAR or INTERVAL
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
MINUTE ( date )
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ
- INTERVAL

Examples

```
=> SELECT MINUTE('12:34:03.456789');
 MINUTE
--------
     34
(1 row)
=>SELECT MINUTE (TIMESTAMP 'sep 22, 2011 12:34');
 MINUTE
--------
     34
(1 row)
=> SELECT MINUTE(INTERVAL '35 12:34:03.456789');
 MINUTE
--------
     34
(1 row)
```

MONTH

Returns the month portion of the specified date as an integer.

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , VARCHAR or INTERVAL
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
MONTH ( date )
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ
- INTERVAL

Examples

In the following examples, Vertica returns the month portion of the specified string. For example, '6-9' represent September 6.

```
=> SELECT MONTH('6-9');
 MONTH
-------
     9
(1 row)
=> SELECT MONTH (TIMESTAMP 'sep 22, 2011 12:34');
 MONTH
-------
     9
(1 row)
=> SELECT MONTH(INTERVAL '2-35' year to month);
 MONTH
-------
    11
(1 row)
```

MONTHS_BETWEEN

Returns the number of months between two dates. MONTHS_BETWEEN can return an integer or a FLOAT:

- Integer: The day portions of *date1* and *date2* are the same, and neither date is the last day of the month. MONTHS_BETWEEN also returns an integer if both dates in *date1* and *date2* are the last days of their respective months. For example, MONTHS_BETWEEN calculates the difference between April 30 and March 31 as 1 month.
- FLOAT: The day portions of *date1* and *date2* are different and one or both dates are not the last day of their respective months. For example, the difference between April 2 and March 1 is 1.03225806451613 . To calculate month fractions, MONTHS_BETWEEN assumes all months contain 31 days.

MONTHS_BETWEEN disregards timestamp time portions.

Behavior type

- Immutable if both date arguments are of data type TIMESTAMP or DATE
- Stable if either date is a TIMESTAMPTZ

Syntax

```
MONTHS_BETWEEN ( date1 , date2 );
```

Parameters

date1

date2

Specify the dates to evaluate where *date1* and *date2* evaluate to one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMPTZ

If *date1* < *date2* , MONTHS_BETWEEN returns a negative value.

Examples

Return the number of months between April 7 2016 and January 7 2015:

```
=> SELECT MONTHS_BETWEEN ('04-07-16'::TIMESTAMP, '01-07-15'::TIMESTAMP);
 MONTHS_BETWEEN
----------------
             15
(1 row)
```

Return the number of months between March 31 2016 and February 28 2016 (MONTHS_BETWEEN assumes both months contain 31 days):

```
=> SELECT MONTHS_BETWEEN ('03-31-16'::TIMESTAMP, '02-28-16'::TIMESTAMP);
  MONTHS_BETWEEN
------------------
 1.09677419354839
(1 row)
```

Return the number of months between March 31 2016 and February 29 2016:

```
=> SELECT MONTHS_BETWEEN ('03-31-16'::TIMESTAMP, '02-29-16'::TIMESTAMP);
 MONTHS_BETWEEN
----------------
              1
(1 row)
```

NEW_TIME

Converts a timestamp value from one time zone to another and returns a TIMESTAMP.

Behavior type

Immutable

Syntax

```
NEW_TIME( 'timestamp' , 'timezone1' , 'timezone2')
```

Parameters

timestamp

The timestamp to convert, conforms to one of the following formats:

- TIMESTAMP / TIMESTAMPTZ
- DATE
- Character string that can be converted to a TIMESTAMP —for example, May 24, 2012 10:00 .

timezone1

`timezone2`

Specify the source and target timezones, one of the strings defined in /opt/vertica/share/timezonesets . For example:

- GMT : Greenwich Mean Time
- AST / ADT : Atlantic Standard/Daylight Time
- EST / EDT : Eastern Standard/Daylight Time
- CST / CDT : Central Standard/Daylight Time
- MST / MDT : Mountain Standard/Daylight Time
- PST / PDT : Pacific Standard/Daylight Time

Examples

Convert the specified time from Eastern Standard Time (EST) to Pacific Standard Time (PST):

```
=> SELECT NEW_TIME('05-24-12 13:48:00', 'EST', 'PST');
     NEW_TIME
---------------------
 2012-05-24 10:48:00
(1 row)
```

Convert 1:00 AM January 2012 from EST to PST:

```
=> SELECT NEW_TIME('01-01-12 01:00:00', 'EST', 'PST');
     NEW_TIME
---------------------
 2011-12-31 22:00:00
(1 row)
```

Convert the current time EST to PST:

```
=> SELECT NOW();
              NOW
-------------------------------
 2016-12-09 10:30:36.727307-05
(1 row)

=> SELECT NEW_TIME('NOW', 'EDT', 'CDT');
         NEW_TIME
----------------------------
 2016-12-09 09:30:36.727307
(1 row)
```

The following example returns the year 45 before the Common Era in Greenwich Mean Time and converts it to Newfoundland Standard Time:

```
=>  SELECT NEW_TIME('April 1, 45 BC', 'GMT', 'NST')::DATE;
   NEW_TIME
---------------
 0045-03-31 BC
(1 row)
```

NEXT_DAY

Returns the date of the first instance of a particular day of the week that follows the specified date.

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , or VARCHAR
- Stable if the specified date is a TIMESTAMPTZ

Syntax

```
NEXT_DAY( 'date', 'day-string')
```

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

day-string

The day of the week to process, a CHAR or VARCHAR string or character constant. Supply the full English name such as Tuesday, or any conventional abbreviation, such as Tue or Tues. *day-string* is not case sensitive and trailing spaces are ignored.

Examples

Get the date of the first Monday that follows April 29 2016:

```
=> SELECT NEXT_DAY('4-29-2016'::TIMESTAMP,'Monday') "NEXT DAY" ;
 NEXT DAY
------------
 2016-05-02
(1 row)
```

Get the first Tuesday that follows today:

```
SELECT NEXT_DAY(CURRENT_TIMESTAMP,'tues') "NEXT DAY" ;
 NEXT DAY
------------
 2016-05-03
(1 row)
```

NOW [date/time]

Returns a value of type TIMESTAMP WITH TIME ZONE representing the start of the current transaction. NOW is equivalent to CURRENT_TIMESTAMP except that it does not accept a precision parameter.

The return value does not change during the transaction. Thus, multiple calls to CURRENT_TIMESTAMP within the same transaction return the same timestamp.

Behavior type

Stable

Syntax

```
NOW()
```

Examples

```
=> CREATE TABLE t1 (a int, b int);
CREATE TABLE
=> INSERT INTO t1 VALUES (1,2);
 OUTPUT
--------
      1
(1 row)

=> SELECT NOW();
              NOW
-------------------------------
 2016-12-09 13:00:08.74685-05
(1 row)

=> INSERT INTO t1 VALUES (3,4);
 OUTPUT
--------
      1
(1 row)

=> SELECT NOW();
              NOW
-------------------------------
 2016-12-09 13:00:08.74685-05
(1 row)

=> COMMIT;
COMMIT
dbadmin=> SELECT NOW();
               NOW
---------------------------------
 2016-12-09 13:01:31.420624-05
(1 row)
```

OVERLAPS

Evaluates two time periods and returns true when they overlap, false otherwise.

Behavior type

- Stable when TIMESTAMP and TIMESTAMPTZ are both used, or when TIMESTAMPTZ is used with INTERVAL
- Immutable otherwise

Syntax

```
( start, end ) OVERLAPS ( start, end )
( start, interval) OVERLAPS ( start, interval )
```

Parameters

start

DATE , TIME , or TIMESTAMP / TIMESTAMPTZ value that specifies the beginning of a time period.

end

DATE , TIME , or TIMESTAMP / TIMESTAMPTZ value that specifies the end of a time period.

interval

Value that specifies the length of the time period.

Examples

Evaluate whether date ranges Feb 16 - Dec 21, 2016 and Oct 10 2008 - Oct 3 2016 overlap:

```
=> SELECT (DATE '2016-02-16', DATE '2016-12-21') OVERLAPS (DATE '2008-10-30', DATE '2016-10-30');
 overlaps
----------
 t
(1 row)
```

Evaluate whether date ranges Feb 16 - Dec 21, 2016 and Jan 01 - Oct 30 2008 - Oct 3, 2016 overlap:

```
=> SELECT (DATE '2016-02-16', DATE '2016-12-21') OVERLAPS (DATE '2008-01-30', DATE '2008-10-30');
 overlaps
----------
 f
(1 row)
```

Evaluate whether date range Feb 02 2016 + 1 week overlaps with date range Oct 16 2016 - 8 months:

```
=> SELECT (DATE '2016-02-16', INTERVAL '1 week') OVERLAPS (DATE '2016-10-16', INTERVAL '-8 months');
 overlaps
----------
 t
(1 row)
```

QUARTER

Returns calendar quarter of the specified date as an integer, where the January-March quarter is 1.

Syntax

```
QUARTER ( date )
```

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , or VARCHAR .
- Stable if the specified date is a TIMESTAMPTZ

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

Examples

```
=> SELECT QUARTER (TIMESTAMP 'sep 22, 2011 12:34');
 QUARTER
---------
       3
(1 row)
```

ROUND

Rounds the specified date or time. If you omit the precision argument, ROUND rounds to day (DD) precision.

Behavior type

- Immutable if thetarget date is a TIMESTAMP or DATE
- Stable if the target date is a TIMESTAMPTZ

Syntax

```
ROUND( rounding-target[, 'precision'] )
```

Parameters

rounding-target

An expression that evaluates to one of the following data types:

- DATE
- TIMESTAMP/TIMESTAMPTZ
- TIMESTAMPTZ

precision

A string constant that specifies precision for the rounded value, one of the following:

- **Century** : CC | SCC
- **Year** : SYYY | YYYY | YEAR | YYY | YY | Y
- **ISO Year** : IYYY | IYY | IY | I
- **Quarter** : Q
- **Month** : MONTH | MON | MM | RM
- **Same weekday as first day of year** : WW
- **Same weekday as first day of ISO year** : IW
- **Same weekday as first day of month** : W
- **Day** (default): DDD | DD | J
- **First weekday** : DAY | DY | D
- **Hour** : HH | HH12 | HH24
- **Minute** : MI
- **Second** : SS

> **Note**
> Hour, minute, and second rounding is not supported by DATE expressions.

Examples

Round to the nearest hour:

```
=> SELECT ROUND(CURRENT_TIMESTAMP, 'HH');
       ROUND
---------------------
 2016-04-28 15:00:00
(1 row)
```

Round to the nearest month:

```
=> SELECT ROUND('9-22-2011 12:34:00'::TIMESTAMP, 'MM');
       ROUND
---------------------
 2011-10-01 00:00:00
(1 row)
```

See also

TIMESTAMP_ROUND

SECOND

Returns the seconds portion of the specified date as an integer.

Syntax

```
SECOND ( date )
```

Behavior type

Immutable , except for TIMESTAMPTZ arguments where it is stable .

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- TIMESTAMP
- TIMESTAMPTZ
- INTERVAL

Examples

```
=> SELECT SECOND ('23:34:03.456789');
 SECOND
--------
      3
(1 row)
=> SELECT SECOND (TIMESTAMP 'sep 22, 2011 12:34');
 SECOND
--------
      0
(1 row)
=> SELECT SECOND (INTERVAL '35 12:34:03.456789');
 SECOND
--------
      3
(1 row)
```

STATEMENT_TIMESTAMP

Similar to TRANSACTION_TIMESTAMP , returns a value of type TIMESTAMP WITH TIME ZONE that represents the start of the current statement.

The return value does not change during statement execution. Thus, different stages of statement execution always have the same timestamp.

Behavior type

Stable

Syntax

```
STATEMENT_TIMESTAMP()
```

Examples

```
=> SELECT foo, bar FROM (SELECT STATEMENT_TIMESTAMP() AS foo)foo, (SELECT STATEMENT_TIMESTAMP() as bar)bar;
              foo               |               bar
--------------------------------+--------------------------------
 2016-12-07 14:55:51.543988-05 | 2016-12-07 14:55:51.543988-05
(1 row)
```

See also

- CLOCK_TIMESTAMP
- TRANSACTION_TIMESTAMP

SYSDATE

Returns the current statement's start date and time as a TIMESTAMP value. This function is identical to GETDATE .

SYSDATE uses the date and time supplied by the operating system on the server to which you are connected, which is the same across all servers. Internally, GETDATE converts STATEMENT_TIMESTAMP from TIMESTAMPTZ to TIMESTAMP .

Behavior type

Stable

Syntax

```
SYSDATE()
```

> **Note**
> You can call this function with no parentheses.

Examples

```
=> SELECT SYSDATE;
          sysdate
----------------------------
 2016-12-12 06:11:10.699642
(1 row)
```

See also

Date/time expressions

TIME_SLICE

Aggregates data by different fixed-time intervals and returns a rounded-up input TIMESTAMP value to a value that corresponds with the start or end of the time slice interval.

Given an input TIMESTAMP value such as 2000-10-28 00:00:01 , the start time of a 3-second time slice interval is 2000-10-28 00:00:00 , and the end time of the same time slice is 2000-10-28 00:00:03 .

Behavior type

Immutable

Syntax

```
TIME_SLICE( expression, slice-length [, 'time-unit' [, 'start-or-end' ] ] )
```

Parameters

expression

One of the following:

- Column of type TIMESTAMP
- String constant that can be parsed into a TIMESTAMP value. For example: '2004-10-19 10:23:54'

Vertica evaluates *expression* on each row.

slice-length

A positive integer that specifies the slice length.

time-unit

Time unit of the slice, one of the following:

- HOUR
- MINUTE
- SECOND (default)
- MILLISECOND
- MICROSECOND

start-or-end

Specifies whether the returned value corresponds to the start or end time with one of the following strings:

- START (default)
- END

> **Note**
> This parameter can be included only if you also supply a non-null *time-unit* argument.

Null argument handling

TIME_SLICE handles null arguments as follows:

- TIME_SLICE returns an error when any one of *slice-length* , *time-unit* , or *start-or-end* parameters is null.
- If *expression* is null and * *slice-length* *, * *time-unit* *, or * *start-or-end* * contain legal values, TIME_SLICE returns a NULL value instead of an error.

Usage

The following command returns the (default) start time of a 3-second time slice:

```
=> SELECT TIME_SLICE('2009-09-19 00:00:01', 3);
    TIME_SLICE
---------------------
 2009-09-19 00:00:00
(1 row)
```

The following command returns the end time of a 3-second time slice:

```
=> SELECT TIME_SLICE('2009-09-19 00:00:01', 3, 'SECOND', 'END');
    TIME_SLICE
---------------------
 2009-09-19 00:00:03
(1 row)
```

This command returns results in milliseconds, using a 3-second time slice:

```
=> SELECT TIME_SLICE('2009-09-19 00:00:01', 3, 'ms');
      TIME_SLICE
-------------------------
 2009-09-19 00:00:00.999

(1 row)
```

This command returns results in microseconds, using a 9-second time slice:

```
=> SELECT TIME_SLICE('2009-09-19 00:00:01', 3, 'us');
       TIME_SLICE
----------------------------
 2009-09-19 00:00:00.999999
(1 row)
```

The next example uses a 3-second interval with an input value of '00:00:01'. To focus specifically on seconds, the example omits date, though all values are implied as being part of the timestamp with a given input of '00:00:01' :

- '00:00: **00** ' is the start of the 3-second time slice
- '00:00: **03** ' is the end of the 3-second time slice.
- '00:00:03' is also the start of the *second* 3-second time slice. In time slice boundaries, the end value of a time slice does not belong to that time slice; it starts the next one.

When the time slice interval is not a factor of 60 seconds, such as a given slice length of 9 in the following example, the slice does not always start or end on 00 seconds:

```
=> SELECT TIME_SLICE('2009-02-14 20:13:01', 9);
    TIME_SLICE
---------------------
 2009-02-14 20:12:54
(1 row)
```

This is expected behavior, as the following properties are true for all time slices:

- Equal in length
- Consecutive (no gaps between them)
- Non-overlapping

To force the above example ('2009-02-14 20:13:01') to start at '2009-02-14 20:13:00', adjust the output timestamp values so that the remainder of 54 counts up to 60:

```
=> SELECT TIME_SLICE('2009-02-14 20:13:01', 9 )+'6 seconds'::INTERVAL AS time;
       time
---------------------
 2009-02-14 20:13:00
(1 row)
```

Alternatively, you could use a different slice length, which is divisible by 60, such as 5:

```
=> SELECT TIME_SLICE('2009-02-14 20:13:01', 5);
    TIME_SLICE
---------------------
 2009-02-14 20:13:00
(1 row)
```

A TIMESTAMPTZ value is implicitly cast to TIMESTAMP. For example, the following two statements have the same effect.

```
=> SELECT TIME_SLICE('2009-09-23 11:12:01'::timestamptz, 3);
    TIME_SLICE
---------------------
 2009-09-23 11:12:00
(1 row)


=> SELECT TIME_SLICE('2009-09-23 11:12:01'::timestamptz::timestamp, 3);

    TIME_SLICE
---------------------
 2009-09-23 11:12:00
(1 row)
```

Examples

You can use the SQL analytic functions FIRST_VALUE and LAST_VALUE to find the first/last price within each time slice group (set of rows belonging to the same time slice). This structure can be useful if you want to sample input data by choosing one row from each time slice group.

```
=> SELECT date_key, transaction_time, sales_dollar_amount,TIME_SLICE(DATE '2000-01-01' + date_key + transaction_time, 3),
FIRST_VALUE(sales_dollar_amount)
OVER (PARTITION BY TIME_SLICE(DATE '2000-01-01' + date_key + transaction_time, 3)
    ORDER BY DATE '2000-01-01' + date_key + transaction_time) AS first_value
FROM store.store_sales_fact
LIMIT 20;

 date_key | transaction_time | sales_dollar_amount |     time_slice      | first_value
----------+------------------+---------------------+---------------------+-------------
        1 | 00:41:16         |                 164 | 2000-01-02 00:41:15 |         164
        1 | 00:41:33         |                 310 | 2000-01-02 00:41:33 |         310
        1 | 15:32:51         |                 271 | 2000-01-02 15:32:51 |         271
        1 | 15:33:15         |                 419 | 2000-01-02 15:33:15 |         419
        1 | 15:33:44         |                 193 | 2000-01-02 15:33:42 |         193
        1 | 16:36:29         |                 466 | 2000-01-02 16:36:27 |         466
        1 | 16:36:44         |                 250 | 2000-01-02 16:36:42 |         250
        2 | 03:11:28         |                  39 | 2000-01-03 03:11:27 |          39
        3 | 03:55:15         |                 375 | 2000-01-04 03:55:15 |         375
        3 | 11:58:05         |                 369 | 2000-01-04 11:58:03 |         369
        3 | 11:58:24         |                 174 | 2000-01-04 11:58:24 |         174
        3 | 11:58:52         |                 449 | 2000-01-04 11:58:51 |         449
        3 | 19:01:21         |                 201 | 2000-01-04 19:01:21 |         201
        3 | 22:15:05         |                 156 | 2000-01-04 22:15:03 |         156
        4 | 13:36:57         |                -125 | 2000-01-05 13:36:57 |        -125
        4 | 13:37:24         |                -251 | 2000-01-05 13:37:24 |        -251
        4 | 13:37:54         |                 353 | 2000-01-05 13:37:54 |         353
        4 | 13:38:04         |                 426 | 2000-01-05 13:38:03 |         426
        4 | 13:38:31         |                 209 | 2000-01-05 13:38:30 |         209
        5 | 10:21:24         |                 488 | 2000-01-06 10:21:24 |         488
(20 rows)
```

TIME_SLICE rounds the transaction time to the 3-second slice length.

The following example uses the analytic (window) OVER clause to return the last trading price (the last row ordered by TickTime) in each 3-second time slice partition:

```
=> SELECT DISTINCT TIME_SLICE(TickTime, 3), LAST_VALUE(price)OVER (PARTITION BY TIME_SLICE(TickTime, 3)
ORDER BY TickTime ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING);
```

Note

If you omit the windowing clause from an analytic clause, LAST_VALUE defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW . Results can seem non-intuitive, because instead of returning the value from the bottom of the current partition, the function returns the bottom of the *window* , which continues to change along with the current input row that is being processed. For more information,

> see Time series analytics and SQL analytics .

In the next example, FIRST_VALUE is evaluated once for each input record and the data is sorted by ascending values. Use SELECT DISTINCT to remove the duplicates and return only one output record per TIME_SLICE :

```
=> SELECT DISTINCT TIME_SLICE(TickTime, 3), FIRST_VALUE(price)OVER (PARTITION BY TIME_SLICE(TickTime, 3)
ORDER BY TickTime ASC)
FROM tick_store;
    TIME_SLICE      | ?column?
--------------------+----------
 2009-09-21 00:00:06 |    20.00
 2009-09-21 00:00:09 |    30.00
 2009-09-21 00:00:00 |    10.00
(3 rows)
```

The information output by the above query can also return MIN , MAX , and AVG of the trading prices within each time slice.

```
=> SELECT DISTINCT TIME_SLICE(TickTime, 3),FIRST_VALUE(Price) OVER (PARTITION BY TIME_SLICE(TickTime, 3)
ORDER BY TickTime ASC),
  MIN(price) OVER (PARTITION BY TIME_SLICE(TickTime, 3)),
  MAX(price) OVER (PARTITION BY TIME_SLICE(TickTime, 3)),
  AVG(price) OVER (PARTITION BY TIME_SLICE(TickTime, 3))
FROM tick_store;
```

See also

- Aggregate functions
- FIRST_VALUE [analytic]
- LAST_VALUE [analytic]
- TIMESERIES clause
- TS_FIRST_VALUE
- TS_LAST_VALUE
- Using time zones with Vertica

TIMEOFDAY

Returns the wall-clock time as a text string. Function results advance during transactions.

Behavior type

Volatile

Syntax

```
TIMEOFDAY()
```

Examples

```
=> SELECT TIMEOFDAY();
              TIMEOFDAY
--------------------------------------
 Mon Dec 12 08:18:01.022710 2016 EST
(1 row)
```

TIMESTAMP_ROUND

Rounds the specified TIMESTAMP. If you omit the precision argument, TIMESTAMP_ROUND rounds to day (DD) precision.

Behavior type

- Immutable if thetarget date is a TIMESTAMP
- Stable if the target date is a TIMESTAMPTZ

Syntax

```
TIMESTAMP_ROUND ( rounding-target[, 'precision'] )
```

Parameters

rounding-target

An expression that evaluates to one of the following data types:

- TIMESTAMP/TIMESTAMPTZ
- TIMESTAMPTZ

precision

A string constant that specifies precision for the rounded value, one of the following:

- **Century** : CC | SCC
- **Year** : SYYY | YYYY | YEAR | YYY | YY | Y
- **ISO Year** : IYYY | IYY | IY | I
- **Quarter** : Q
- **Month** : MONTH | MON | MM | RM
- **Same weekday as first day of year** : WW
- **Same weekday as first day of ISO year** : IW
- **Same weekday as first day of month** : W
- **Day** (default): DDD | DD | J
- **First weekday** : DAY | DY | D
- **Hour** : HH | HH12 | HH24
- **Minute** : MI
- **Second** : SS

> **Note**
> Hour, minute, and second rounding is not supported by DATE expressions.

Examples

Round to the nearest hour:

```
=> SELECT TIMESTAMP_ROUND(CURRENT_TIMESTAMP, 'HH');
      ROUND
---------------------
 2016-04-28 15:00:00
(1 row)
```

Round to the nearest month:

```
=> SELECT TIMESTAMP_ROUND('9-22-2011 12:34:00'::TIMESTAMP, 'MM');
      ROUND
---------------------
 2011-10-01 00:00:00
(1 row)
```

See also

ROUND

TIMESTAMP_TRUNC

Truncates the specified TIMESTAMP. If you omit the precision argument, TIMESTAMP_TRUNC truncates to day (DD) precision.

Behavior type

- Immutable if thetarget date is a TIMESTAMP
- Stable if the target date is a TIMESTAMPTZ

Syntax

```
TIMESTAMP_TRUNC( trunc-target[, 'precision'] )
```

Parameters

trunc-target

An expression that evaluates to one of the following data types:

- TIMESTAMP/TIMESTAMPTZ
- TIMESTAMPTZ

precision

A string constant that specifies precision for the truncated value, one of the following:

- **Century** : CC | SCC
- **Year** : SYYY | YYYY | YEAR | YYY | YY | Y
- **ISO Year** : IYYY | IYY | IY | I
- **Quarter** : Q
- **Month** : MONTH | MON | MM | RM
- **Same weekday as first day of year** : WW
- **Same weekday as first day of ISO year** : IW
- **Same weekday as first day of month** : W
- **Day** : DDD | DD | J
- **First weekday** : DAY | DY | D
- **Hour** : HH | HH12 | HH24
- **Minute** : MI
- **Second** : SS

> **Note**
> Hour, minute, and second truncating is not supported by DATE expressions.

Examples

Truncate to the current hour:

```
=> SELECT TIMESTAMP_TRUNC(CURRENT_TIMESTAMP, 'HH');
  TIMESTAMP_TRUNC
---------------------
 2016-04-29 08:00:00
(1 row)
```

Truncate to the month:

```
=> SELECT TIMESTAMP_TRUNC('9-22-2011 12:34:00'::TIMESTAMP, 'MM');
  TIMESTAMP_TRUNC
---------------------
 2011-09-01 00:00:00
(1 row)
```

See also

TRUNC

TIMESTAMPADD

Adds the specified number of intervals to a TIMESTAMP or TIMESTAMPTZ value and returns a result of the same data type.

Behavior type

- Immutable if the input date is a TIMESTAMP
- Stable if the input date is a TIMESTAMPTZ

Syntax

```
TIMESTAMPADD ( datepart, count, start-date );
```

Parameters

datepart

Specifies the type of time intervals that TIMESTAMPADD adds to the specified start date. If *datepart* is an expression, it must be enclosed in parentheses:

```
TIMESTAMPADD((expression), interval, start;
```

datepart must evaluate to one of the following string literals, either quoted or unquoted:

- year | yy | yyyy
- quarter | qq | q
- month | mm | m
- day | dayofyear | dd | d | dy | y

- week | wk | ww
- hour | hh
- minute | mi | n
- second | ss | s
- millisecond | ms
- microsecond | mcs | us

count

Integer or integer expression that specifies the number of *datepart* intervals to add to *start-date* .

start-date

TIMESTAMP or TIMESTAMPTZ value.

Examples

Add two months to the current date:

```
=> SELECT CURRENT_TIMESTAMP AS Today;
          Today
-------------------------------
 2016-05-02 06:56:57.923045-04
(1 row)

=> SELECT TIMESTAMPADD (MONTH, 2, (CURRENT_TIMESTAMP)) AS TodayPlusTwoMonths;;
     TodayPlusTwoMonths
-------------------------------
 2016-07-02 06:56:57.923045-04
(1 row)
```

Add 14 days to the beginning of the current month:

```
=> SELECT TIMESTAMPADD (DD, 14, (SELECT TRUNC((CURRENT_TIMESTAMP), 'MM')));
   timestampadd
---------------------
 2016-05-15 00:00:00
(1 row)
```

TIMESTAMPDIFF

Returns the time span between two TIMESTAMP or TIMESTAMPTZ values, in the intervals specified. TIMESTAMPDIFF excludes the start date in its calculation.

Behavior type

- Immutable if start and end dates are TIMESTAMP
- Stable if start and end dates are TIMESTAMPTZ

Syntax

```
TIMESTAMPDIFF ( datepart, start, end );
```

Parameters

datepart

Specifies the type of date or time intervals that TIMESTAMPDIFF returns. If *datepart* is an expression, it must be enclosed in parentheses:

```
TIMESTAMPDIFF((expression), start, end );
```

datepart must evaluate to one of the following string literals, either quoted or unquoted:

- year | yy | yyyy
- quarter | qq | q
- month | mm | m
- day | dayofyear | dd | d | dy | y
- week | wk | ww
- hour | hh
- minute | mi | n
- second | ss | s

- millisecond | ms
- microsecond | mcs | us

start **,** ***end***

Specify the start and end dates, where *start* and *end* evaluate to one of the following data types:

- TIMESTAMP/TIMESTAMPTZ
- TIMESTAMPTZ

If *end* < *start* , TIMESTAMPDIFF returns a negative value.

Date part intervals

TIMESTAMPDIFF uses the *datepart* argument to calculate the number of intervals between two dates, rather than the actual amount of time between them. For detailed information, see DATEDIFF .

Examples

```
=> SELECT TIMESTAMPDIFF (YEAR,'1-1-2006 12:34:00', '1-1-2008 12:34:00');
 timestampdiff
---------------
             2
(1 row)
```

See also

DATEDIFF

TRANSACTION_TIMESTAMP

Returns a value of type `TIME WITH TIMEZONE` that represents the start of the current transaction.

The return value does not change during the transaction. Thus, multiple calls to TRANSACTION_TIMESTAMP within the same transaction return the same timestamp.

TRANSACTION_TIMESTAMP is equivalent to CURRENT_TIMESTAMP , except it does not accept a precision parameter.

Behavior type

Stable

Syntax

```
TRANSACTION_TIMESTAMP()
```

Examples

```
=> SELECT foo, bar FROM (SELECT TRANSACTION_TIMESTAMP() AS foo)foo, (SELECT TRANSACTION_TIMESTAMP() as bar)bar;
              foo              |              bar
-------------------------------+-------------------------------
 2016-12-12 08:18:00.988528-05 | 2016-12-12 08:18:00.988528-05
(1 row)
```

See also

- CLOCK_TIMESTAMP
- STATEMENT_TIMESTAMP

TRUNC

Truncates the specified date or time. If you omit the precision argument, TRUNC truncates to day (DD) precision.

Behavior type

- Immutable if thetarget date is a TIMESTAMP or DATE
- Stable if the target date is a TIMESTAMPTZ

Syntax

```
TRUNC( trunc-target[, 'precision'] )
```

Parameters

***** ***trunc-target*** *****

An expression that evaluates to one of the following data types:

- DATE

- TIMESTAMP/TIMESTAMPTZ
- TIMESTAMPTZ

precision

A string constant that specifies precision for the truncated value, one of the following:

- **Century** : CC | SCC
- **Year** : SYYY | YYYY | YEAR | YYY | YY | Y
- **ISO Year** : IYYY | IYY | IY | I
- **Quarter** : Q
- **Month** : MONTH | MON | MM | RM
- **Same weekday as first day of year** : WW
- **Same weekday as first day of ISO year** : IW
- **Same weekday as first day of month** : W
- **Day** (default): DDD | DD | J
- **First weekday** : DAY | DY | D
- **Hour** : HH | HH12 | HH24
- **Minute** : MI
- **Second** : SS

> **Note**
> Hour, minute, and second truncating is not supported by DATE expressions.

Examples

Truncate to the current hour:

```
=> => SELECT TRUNC(CURRENT_TIMESTAMP, 'HH');
       TRUNC
---------------------
 2016-04-29 10:00:00
(1 row)
```

Truncate to the month:

```
=> SELECT TRUNC('9-22-2011 12:34:00'::TIMESTAMP, 'MM');
   TIMESTAMP_TRUNC
---------------------
 2011-09-01 00:00:00
(1 row)
```

See also

TIMESTAMP_TRUNC

WEEK

Returns the week of the year for the specified date as an integer, where the first week begins on the first Sunday on or preceding January 1.

Syntax

```
WEEK ( date )
```

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , or VARCHAR
- Stable if the specified date is a TIMESTAMPTZ

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

Examples

January 2 is on Saturday, so WEEK returns 1:

```
=> SELECT WEEK ('1-2-2016'::DATE);
 WEEK
------
    1
(1 row)
```

January 3 is the second Sunday in 2016, so WEEK returns 2:

```
=> SELECT WEEK ('1-3-2016'::DATE);
 WEEK
------
    2
(1 row)
```

WEEK_ISO

Returns the week of the year for the specified date as an integer, where the first week starts on Monday and contains January 4. This function conforms with the ISO 8061 standard.

Syntax

```
WEEK_ISO ( date )
```

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , or VARCHAR
- Stable if the specified date is a TIMESTAMPTZ

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

Examples

The first week of 2016 begins on Monday January 4:

```
=> SELECT WEEK_ISO ('1-4-2016'::DATE);
 WEEK_ISO
----------
        1
(1 row)
```

January 3 2016 returns week 53 of the previous year (2015):

```
=> SELECT WEEK_ISO ('1-3-2016'::DATE);
 WEEK_ISO
----------
       53
(1 row)
```

In 2015, January 4 is on Sunday, so the first week of 2015 begins on the preceding Monday (December 29 2014):

```
=> SELECT WEEK_ISO ('12-29-2014'::DATE);
 WEEK_ISO
----------
        1
(1 row)
```

YEAR

Returns an integer that represents the year portion of the specified date.

Syntax

```
YEAR( date )
```

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , VARCHAR , or INTERVAL
- Stable if the specified date is a TIMESTAMPTZ

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ
- INTERVAL

Examples

```
=> SELECT YEAR(CURRENT_DATE::DATE);
 YEAR
------
 2016
(1 row)
```

See also

YEAR_ISO

YEAR_ISO

Returns an integer that represents the year portion of the specified date. The return value is based on the ISO 8061 standard.

The first week of the ISO year is the week that contains January 4.

Syntax

```
YEAR_ISO ( date )
```

Behavior type

- Immutable if thespecified date is a TIMESTAMP , DATE , or VARCHAR
- Stable if the specified date is a TIMESTAMPTZ

Parameters

date

The date to process, one of the following data types:

- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ

Examples

```
> SELECT YEAR_ISO(CURRENT_DATE::DATE);
 YEAR_ISO
----------
    2016
(1 row)
```

See also

YEAR

IP address functions

IP functions perform conversion, calculation, and manipulation operations on IP, network, and subnet addresses.

In this section

- INET_ATON
- INET_NTOA
- V6_ATON
- V6_NTOA
- V6_SUBNETA
- V6_SUBNETN
- V6_TYPE

INET_ATON

Converts a string that contains a dotted-quad representation of an IPv4 network address to an INTEGER. It trims any surrounding white space from the string. This function returns NULL if the string is NULL or contains anything other than a quad dotted IPv4 address.

Behavior type

Immutable

Syntax

```
INET_ATON ( expression )
```

Arguments

expression

the string to convert.

Examples

```
=> SELECT INET_ATON('209.207.224.40');
 inet_aton
------------
 3520061480
(1 row)

=> SELECT INET_ATON('1.2.3.4');
 inet_aton
-----------
  16909060
(1 row)

=> SELECT TO_HEX(INET_ATON('1.2.3.4'));
 to_hex
---------
 1020304
(1 row)
```

See also

- INET_NTOA
- V6_ATON
- V6_NTOA
- V6_SUBNETA
- V6_SUBNETN
- V6_TYPE

INET_NTOA

Converts an INTEGER value into a VARCHAR dotted-quad representation of an IPv4 network address. INET_NTOA returns NULL if the integer value is NULL, negative, or is greater than 2^{32} (4294967295).

Behavior type

Immutable

Syntax

```
INET_NTOA ( expression )
```

Arguments

expression

The integer network address to convert.

Examples

```
=> SELECT INET_NTOA(16909060);
 inet_ntoa
-----------
 1.2.3.4
(1 row)

=> SELECT INET_NTOA(03021962);
 inet_ntoa
-------------
 0.46.28.138
(1 row)
```

See also

- INET_ATON
- V6_ATON
- V6_NTOA
- V6_SUBNETA
- V6_SUBNETN
- V6_TYPE

V6_ATON

Converts a string containing a colon-delimited IPv6 network address into a VARBINARY string. Any spaces around the IPv6 address are trimmed. This function returns NULL if the input value is NULL or it cannot be parsed as an IPv6 address. This function relies on the Linux function inet_pton .

Behavior type

Immutable

Syntax

```
V6_ATON ( expression )
```

Arguments

expression

(VARCHAR) the string containing an IPv6 address to convert.

Examples

```
=> SELECT V6_ATON('2001:DB8::8:800:200C:417A');
                    v6_aton
-----------------------------------------------------
  \001\015\270\000\000\000\000\000\010\010\000 \014Az
(1 row)

=> SELECT V6_ATON('1.2.3.4');
              v6_aton
---------------------------------------------------------------------
 \000\000\000\000\000\000\000\000\000\000\377\377\001\002\003\004
(1 row)
SELECT TO_HEX(V6_ATON('2001:DB8::8:800:200C:417A'));
              to_hex
-----------------------------------
 20010db8000000000080800200c417a
(1 row)

=> SELECT V6_ATON('::1.2.3.4');
              v6_aton
---------------------------------------------------------------------
 \000\000\000\000\000\000\000\000\000\000\000\000\001\002\003\004
(1 row)
```

See also

- V6_NTOA
- V6_SUBNETA
- V6_SUBNETN
- V6_TYPE
- INET_ATON
- INET_NTOA

V6_NTOA

Converts an IPv6 address represented as varbinary to a character string.

Behavior type

Immutable

Syntax

```
V6_NTOA ( expression )
```

Arguments

expression

(VARBINARY) is the binary string to convert.

Notes

The following syntax converts an IPv6 address represented as VARBINARY B to a string A.

V6_NTOA right-pads B to 16 bytes with zeros, if necessary, and calls the Linux function inet_ntop .

```
=> V6_NTOA(VARBINARY B) -> VARCHAR A
```

If B is NULL or longer than 16 bytes, the result is NULL.

Vertica automatically converts the form '::ffff:1.2.3.4' to '1.2.3.4'.

Examples

```
=> SELECT V6_NTOA(' \001\015\270\000\000\000\000\000\010\010\000 \014Az');
        v6_ntoa
---------------------------
 2001:db8::8:800:200c:417a
(1 row)

=> SELECT V6_NTOA(V6_ATON('1.2.3.4'));
 v6_ntoa
---------
 1.2.3.4
(1 row)

=> SELECT V6_NTOA(V6_ATON('::1.2.3.4'));
  v6_ntoa
-----------
 ::1.2.3.4
(1 row)
```

See also

- V6_ATON

V6_SUBNETA

Returns a VARCHAR containing a subnet address in CIDR (Classless Inter-Domain Routing) format from a binary or alphanumeric IPv6 address. Returns NULL if either parameter is NULL, the address cannot be parsed as an IPv6 address, or the subnet value is outside the range of 0 to 128.

Behavior type

Immutable

Syntax

```
V6_SUBNETA ( address, subnet)
```

Arguments

address

VARBINARY or VARCHAR containing the IPv6 address.

subnet

The size of the subnet in bits as an INTEGER. This value must be greater than zero and less than or equal to 128.

Examples

```
=> SELECT V6_SUBNETA(V6_ATON('2001:db8::8:800:200c:417a'), 28);
 v6_subneta
---------------
 2001:db0::/28
(1 row)
```

See also

- V6_ATON
- V6_NTOA
- V6_SUBNETN
- V6_TYPE
- INET_ATON
- INET_NTOA

V6_SUBNETN

Calculates a subnet address in CIDR (Classless Inter-Domain Routing) format from a varbinary or alphanumeric IPv6 address.

Behavior type

Immutable

Syntax

```
V6_SUBNETN ( address, subnet-size)
```

Arguments

address

The IPv6 address as a VARBINARY or VARCHAR. The format you pass in determines the date type of the output. If you pass in a VARBINARY address, V6_SUBNETN returns a VARBINARY value. If you pass in a VARCHAR value, it returns a VARCHAR.

subnet-size

The size of the subnet as an INTEGER.

Notes

The following syntax masks a BINARY IPv6 address B so that the N left-most bits of S form a subnet address, while the remaining right-most bits are cleared.

V6_SUBNETN right-pads B to 16 bytes with zeros, if necessary and masks B , preserving its N-bit subnet prefix.

```
=> V6_SUBNETN(VARBINARY B, INT8 N) -> VARBINARY(16) S
```

If B is NULL or longer than 16 bytes, or if N is not between 0 and 128 inclusive, the result is NULL.

S = [B]/N in Classless Inter-Domain Routing notation (CIDR notation).

The following syntax masks an alphanumeric IPv6 address A so that the N leftmost bits form a subnet address, while the remaining rightmost bits are cleared.

```
=> V6_SUBNETN(VARCHAR A, INT8 N) -> V6_SUBNETN(V6_ATON(A), N) -> VARBINARY(16) S
```

Examples

This example returns VARBINARY, after using V6_ATON to convert the VARCHAR string to VARBINARY:

```
=> SELECT V6_SUBNETN(V6_ATON('2001:db8::8:800:200c:417a'), 28);
                    v6_subnetn
-----------------------------------------------------------------
 \001\015\260\000\000\000\000\000\000\000\000\000\000\000\000\000
```

See also

- V6_ATON
- V6_NTOA
- V6_TYPE
- V6_SUBNETA
- INET_ATON
- INET_NTOA

V6_TYPE

Returns an INTEGER value that classifies the type of the network address passed to it as defined in IETF RFC 4291 section 2.4 . For example, If you pass this function the string 127.0.0.1 , it returns 2 which indicates the address is a loopback address. This function accepts both IPv4 and IPv6 addresses.

Behavior type

Immutable

Syntax

```
V6_TYPE ( address)
```

Arguments

address

A VARBINARY or VARCHAR containing an IPv6 or IPv4 address to describe.

Returns

The values returned by this function are:

Return Value	Address Type	Description
0	GLOBAL	Global unicast addresses
1	LINKLOCAL	Link-Local unicast (and private-use) addresses
2	LOOPBACK	Loopback addresses
3	UNSPECIFIED	Unspecifiedaddresses
4	MULTICAST	Multicastaddresses

The return value is based on the following table of IP address ranges:

Address Family	CIDR	Type
IPv4	0.0.0.0/8	UNSPECIFIED
	10.0.0.0/8	LINKLOCAL
	127.0.0.0/8	LOOPBACK
	169.254.0.0/16	LINKLOCAL
	172.16.0.0/12	LINKLOCAL
	192.168.0.0/16	LINKLOCAL
	224.0.0.0/4	MULTICAST
	All other addresses	GLOBAL
IPv6	::0/128	UNSPECIFIED
	::1/128	LOOPBACK

	fe80::/10	LINKLOCAL
	ff00::/8	MULTICAST
	All other addresses	GLOBAL

This function returns NULL if you pass it a NULL value or an invalid address.

Examples

```
=> SELECT V6_TYPE(V6_ATON('192.168.2.10'));
 v6_type
---------
       1
(1 row)

=> SELECT V6_TYPE(V6_ATON('2001:db8::8:800:200c:417a'));
 v6_type
---------
       0
(1 row)
```

See also

- V6_ATON
- V6_NTOA
- INET_ATON
- INET_NTOA
- V6_SUBNETN
- V6_SUBNETA
- IP Version 6 Addressing Architecture
- IPv4 Global Unicast Address Assignments

Sequence functions

The sequence functions provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

In this section

- CURRVAL
- NEXTVAL

CURRVAL

Returns the last value across all nodes that was set by NEXTVAL on this sequence in the current session. If NEXTVAL was never called on this sequence since its creation, Vertica returns an error.

Syntax

```
CURRVAL ('[[database.]schema.]sequence-name')
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

sequence-name

The target sequence

Privileges

- SELECT privilege on sequence
- USAGE privilege on sequence schema

Restrictions

You cannot invoke CURRVAL in a SELECT statement, in the following contexts:

- WHERE clause

- GROUP BY clause
- ORDER BY clause
- DISTINCT clause
- UNION
- Subquery

You also cannot invoke CURRVAL to act on a sequence in:

- UPDATE or DELETE subqueries
- Views

Examples

See Creating and using named sequences .

See also

NEXTVAL

NEXTVAL

Returns the next value in a sequence. Call NEXTVAL after creating a sequence to initialize the sequence with its default value. Thereafter, call NEXTVAL to increment the sequence value for ascending sequences, or decrement its value for descending sequences.

Syntax

```
NEXTVAL ('[[database.]schema.]sequence')
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

sequence

Identifies the target sequence.

Privileges

- SELECT privilege on sequence
- USAGE privilege on sequence schema

Restrictions

You cannot invoke NEXTVAL in a SELECT statement, in the following contexts:

- WHERE clause
- GROUP BY clause
- ORDER BY clause
- DISTINCT clause
- UNION
- Subquery

You also cannot invoke NEXTVAL to act on a sequence in:

- UPDATE or DELETE subqueries
- Views

You can use subqueries to work around some of these restrictions. For example, to use sequences with a DISTINCT clause:

```
=> SELECT t.col1, shift_allocation_seq.NEXTVAL FROM (
   SELECT DISTINCT col1 FROM av_temp1) t;
```

Examples

See Creating and using named sequences .

See also

CURRVAL

String functions

String functions perform conversion, extraction, or manipulation operations on strings, or return information about strings.

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types CHAR, VARCHAR, BINARY, and VARBINARY.

Unless otherwise noted, all of the functions listed in this section work on all four data types. As opposed to some other SQL implementations, Vertica keeps CHAR strings unpadded internally, padding them only on final output. So converting a CHAR(3) 'ab' to VARCHAR(5) results in a VARCHAR of length 2, not one with length 3 including a trailing space.

Some of the functions described here also work on data of non-string types by converting that data to a string representation first. Some functions work only on character strings, while others work only on binary strings. Many work for both. BINARY and VARBINARY functions ignore multibyte UTF-8 character boundaries.

Non-binary character string functions handle normalized multibyte UTF-8 characters, as specified by the Unicode Consortium. Unless otherwise specified, those character string functions for which it matters can optionally specify whether VARCHAR arguments should be interpreted as octet (byte) sequences, or as (locale-aware) sequences of UTF-8 characters. This is accomplished by adding "USING OCTETS" or "USING CHARACTERS" (default) as a parameter to the function.

Some character string functions are stable because in general UTF-8 case-conversion, searching and sorting can be locale dependent. Thus, LOWER is stable, while LOWERB is immutable. The USING OCTETS clause converts these functions into their "B" forms, so they become immutable. If the locale is set to collation=binary, which is the default, all string functions—except CHAR_LENGTH/CHARACTER_LENGTH, LENGTH, SUBSTR, and OVERLAY—are converted to their "B" forms and so are immutable.

BINARY implicitly converts to VARBINARY, so functions that take VARBINARY arguments work with BINARY.

For other functions that operate on strings (but not VARBINARY), see Regular expression functions.

In this section

- ASCII
- BIT_LENGTH
- BITCOUNT
- BITSTRING_TO_BINARY
- BTRIM
- CHARACTER_LENGTH
- CHR
- COLLATION
- CONCAT
- DECODE
- EDIT_DISTANCE
- GREATEST
- GREATESTB
- HEX_TO_BINARY
- HEX_TO_INTEGER
- INITCAP
- INITCAPB
- INSERT
- INSTR
- INSTRB
- ISUTF8
- JARO_DISTANCE
- JARO_WINKLER_DISTANCE
- LEAST
- LEASTB
- LEFT
- LENGTH
- LOWER
- LOWERB
- LPAD
- LTRIM
- MAKEUTF8
- MD5
- OCTET_LENGTH
- OVERLAY
- OVERLAYB
- POSITION
- POSITIONB
- QUOTE_IDENT
- QUOTE_LITERAL

- QUOTE_NULLABLE
- REPEAT
- REPLACE
- RIGHT
- RPAD
- RTRIM
- SHA1
- SHA224
- SHA256
- SHA384
- SHA512
- SOUNDEX
- SOUNDEX_MATCHES
- SPACE
- SPLIT_PART
- SPLIT_PARTB
- STRPOS
- STRPOSB
- SUBSTR
- SUBSTRB
- SUBSTRING
- TRANSLATE
- TRIM
- UPPER
- UPPERB

ASCII

Converts the first character of a VARCHAR datatype to an INTEGER. This function is the opposite of the CHR function.

ASCII operates on UTF-8 characters and single-byte ASCII characters. It returns the same results for the ASCII subset of UTF-8.

Behavior type

Immutable

Syntax

```
ASCII ( expression )
```

Arguments

expression

VARCHAR (string) to convert.

Examples

This example returns employee last names that begin with L. The ASCII equivalent of L is 76:

```
=> SELECT employee_last_name FROM employee_dimension
    WHERE ASCII(SUBSTR(employee_last_name, 1, 1)) = 76
     LIMIT 5;
 employee_last_name
--------------------
 Lewis
 Lewis
 Lampert
 Lampert
 Li
(5 rows)
```

BIT_LENGTH

Returns the length of the string expression in bits (bytes * 8) as an INTEGER. BIT_LENGTH applies to the contents of VARCHAR and VARBINARY fields.

Behavior type

Immutable

Syntax

```
BIT_LENGTH ( expression )
```

Arguments

expression

(CHAR or VARCHAR or BINARY or VARBINARY) is the string to convert.

Examples

Expression	Result
SELECT BIT_LENGTH('abc'::varbinary);	24
SELECT BIT_LENGTH('abc'::binary);	8
SELECT BIT_LENGTH(''::varbinary);	0
SELECT BIT_LENGTH(''::binary);	8
SELECT BIT_LENGTH(null::varbinary);	
SELECT BIT_LENGTH(null::binary);	
SELECT BIT_LENGTH(VARCHAR 'abc');	24
SELECT BIT_LENGTH(CHAR 'abc');	24
SELECT BIT_LENGTH(CHAR(6) 'abc');	48
SELECT BIT_LENGTH(VARCHAR(6) 'abc');	24
SELECT BIT_LENGTH(BINARY(6) 'abc');	48
SELECT BIT_LENGTH(BINARY 'abc');	24
SELECT BIT_LENGTH(VARBINARY 'abc');	24
SELECT BIT_LENGTH(VARBINARY(6) 'abc');	24

See also

- CHARACTER_LENGTH
- LENGTH
- OCTET_LENGTH

BITCOUNT

Returns the number of one-bits (sometimes referred to as set-bits) in the given VARBINARY value. This is also referred to as the population count.

Behavior type

Immutable

Syntax

```
BITCOUNT ( expression )
```

Arguments

expression

(BINARY or VARBINARY) is the string to return.

Examples

```
=> SELECT BITCOUNT(HEX_TO_BINARY('0x10'));
 BITCOUNT
----------
        1
(1 row)
=> SELECT BITCOUNT(HEX_TO_BINARY('0xF0'));
 BITCOUNT
----------
        4
(1 row)
=> SELECT BITCOUNT(HEX_TO_BINARY('0xAB'));
 BITCOUNT
----------
        5
(1 row)
```

BITSTRING_TO_BINARY

Translates the given VARCHAR bitstring representation into a VARBINARY value. This function is the inverse of TO_BITSTRING .

Behavior type

Immutable

Syntax

```
BITSTRING_TO_BINARY ( expression )
```

Arguments

expression

The VARCHAR string to process.

Examples

If there are an odd number of characters in the hex value, the first character is treated as the low nibble of the first (furthest to the left) byte.

```
=> SELECT BITSTRING_TO_BINARY('0110000101100010');
 BITSTRING_TO_BINARY
---------------------
 ab
(1 row)
```

BTRIM

Removes the longest string consisting only of specified characters from the start and end of a string.

Behavior type

Immutable

Syntax

```
BTRIM ( expression [ , characters-to-remove ] )
```

Arguments

expression

(CHAR or VARCHAR) is the string to modify

characters-to-remove

(CHAR or VARCHAR) specifies the characters to remove. The default is the space character.

Examples

```
=> SELECT BTRIM('xyxtrimyyx', 'xy');
 BTRIM
-------
 trim
(1 row)
```

See also

- LTRIM

- RTRIM
- TRIM

CHARACTER_LENGTH

The CHARACTER_LENGTH() function:

- Returns the string length in UTF-8 characters for CHAR and VARCHAR columns
- Returns the string length in bytes (octets) for BINARY and VARBINARY columns
- Strips the padding from CHAR expressions but not from VARCHAR expressions
- Is identical to LENGTH() for CHAR and VARCHAR. For binary types, CHARACTER_LENGTH() is identical to OCTET_LENGTH() .

Behavior type

Immutable if USING OCTETS , stable otherwise.

Syntax

```
[ CHAR_LENGTH | CHARACTER_LENGTH ] ( expression ... [ USING { CHARACTERS | OCTETS } ] )
```

Arguments

expression

(CHAR or VARCHAR) is the string to measure

USING CHARACTERS | OCTETS

Determines whether the character length is expressed in characters (the default) or octets.

Examples

```
=> SELECT CHAR_LENGTH('1234  '::CHAR(10) USING OCTETS);
 octet_length
--------------
            4
(1 row)

=> SELECT CHAR_LENGTH('1234  '::VARCHAR(10));
 char_length
-------------
           6
(1 row)

=> SELECT CHAR_LENGTH(NULL::CHAR(10)) IS NULL;
 ?column?
----------
 t
(1 row)
```

See also

- BIT_LENGTH

CHR

Converts the first character of an INTEGER datatype to a VARCHAR.

Behavior type

Immutable

Syntax

```
CHR ( expression )
```

Arguments

expression

(INTEGER) is the string to convert and is masked to a single character.

Notes

- CHR is the opposite of the ASCII function.
- CHR operates on UTF-8 characters, not only on single-byte ASCII characters. It continues to get the same results for the ASCII subset of UTF-8.

Examples

This example returns the VARCHAR datatype of the CHR expressions 65 and 97 from the employee table:

```
=> SELECT CHR(65), CHR(97) FROM employee;
 CHR | CHR
-----+-----
 A   | a
 A   | a
 A   | a
 A   | a
 A   | a
 A   | a
 A   | a
 A   | a
 A   | a
 A   | a
 A   | a
 A   | a
(12 rows)
```

COLLATION

Applies a collation to two or more strings. Use COLLATION with ORDER BY , GROUP BY , and equality clauses.

Syntax

```
COLLATION ( 'expression' [ , 'locale_or_collation_name' ] )
```

Arguments

' *expression* '

Any expression that evaluates to a column name or to two or more values of type CHAR or VARCHAR .

' *locale_or_collation_name* '

The ICU (International Components for Unicode) locale or collation name to use when collating the string. If you omit this parameter, COLLATION uses the collation associated with the session locale.

To determine the current session locale, enter the vsql meta-command \locale :

```
=> \locale
en_US@collation=binary
```

To set the locale and collation, use \locale as follows:

```
=> \locale en_US@collation=binary
INFO 2567:  Canonical locale: 'en_US'
Standard collation: 'LEN_KBINARY'
English (United States)
```

Locales

The locale used for COLLATION can be one of the following:

- The default locale
- A session locale
- A locale that you specify when you call COLLATION . If you specify the locale, Vertica applies the collation associated with that locale to the data. COLLATION does not modify the collation for any other columns in the table.

For a list of valid ICU locales, go to Locale Explorer (ICU) .

Binary and non-binary collations

The Vertica default locale is en_US@collation=binary , which uses *binary collation* . Binary collation compares binary representations of strings. Binary collation is fast, but it can result in a sort order where K precedes c because the binary representation of K is lower than c .

For non-binary collation, Vertica transforms the data according to the rules of the locale or the specified collation, and then applies the sorting rules. Suppose the locale collation is non-binary and you request a GROUP BY on string data. In this case,Vertica calls COLLATION , whether or not you specify the function in your query.

For information about collation naming, see Collator Naming Scheme .

Examples

Collating GROUP BY results

The following examples are based on a Premium_Customer table that contains the following data:

```
=> SELECT * FROM Premium_Customer;
 ID | LName  | FName
----+--------+---------
  1 | Mc Coy | Bob
  2 | Mc Coy | Janice
  3 | McCoy  | Jody
  4 | McCoy  | Peter
  5 | McCoy  | Brendon
  6 | Mccoy  | Cameron
  7 | Mccoy  | Lisa
```

The first statement shows how COLLATION applies the collation for the EN_US locale to the LName column for the locale EN_US . Vertica sorts the GROUP BY output as follows:

- Last names with spaces
- Last names where "coy" starts with a lowercase letter
- Last names where "Coy" starts with an uppercase letter

```
=> SELECT * FROM Premium_Customer ORDER BY COLLATION(LName, 'EN_US'), FName;
 ID | LName  | FName
----+--------+---------
  1 | Mc Coy | Bob
  2 | Mc Coy | Janice
  6 | Mccoy  | Cameron
  7 | Mccoy  | Lisa
  5 | McCoy  | Brendon
  3 | McCoy  | Jody
  4 | McCoy  | Peter
```

The next statement shows how COLLATION collates the LName column for the locale LEN_AS :

- LEN indicates the language (L) is English (EN).
- AS (Alternate Shifted) instructs COLLATION that lowercase letters come before uppercase (shifted) letters.

In the results, the last names in which "coy" starts with a lowercase letter precede the last names where "Coy" starts with an uppercase letter.

```
=> SELECT * FROM Premium_Customer ORDER BY COLLATION(LName, 'LEN_AS'), FName;
 ID | LName  | FName
----+--------+---------
  6 | Mccoy  | Cameron
  7 | Mccoy  | Lisa
  1 | Mc Coy | Bob
  5 | McCoy  | Brendon
  2 | Mc Coy | Janice
  3 | McCoy  | Jody
  4 | McCoy  | Peter
```

Comparing strings with an equality clause

In the following query, COLLATION removes spaces and punctuation when comparing two strings in English. It then determines whether the two strings still have the same value after the punctuation has been removed:

```
=> SELECT COLLATION ('U.S.A', 'LEN_AS') = COLLATION('USA', 'LEN_AS');
?column?
----------
t
```

Sorting strings in non-english languages

The following table contains data that uses the German character eszett, ß:

```
=> SELECT * FROM t1;
    a      | b | c
-----------+---+----
 ßstringß   | 1 | 10
 SSstringSS | 2 | 20
 random1    | 3 | 30
 random1    | 4 | 40
 random2    | 5 | 50
```

When you specify the collation LDE_S1 :

- LDE indicates the language (L) is German (DE).
- S1 indicates the strength (S) of 1 (primary). This value indicates that the collation does not need to consider accents and case.

The query returns the data in the following order:

```
=> SELECT a FROM t1 ORDER BY COLLATION(a, 'LDE_S1'));
    a
------------
 random1
 random1
 random2
 SSstringSS
 ßstringß
```

CONCAT

Concatenates two strings and returns a varchar data type. If either argument is null, concat returns null.

Syntax

```
CONCAT ('string-expression1, string-expression2)
```

Behavior type

Immutable

Arguments

string-expression1* , *string-expression2

The values to concatenate, any data type that can be cast to a string value.

Examples

The following examples use a sample table named alphabet with two varchar columns:

```
=> CREATE TABLE alphabet (letter1 varchar(2), letter2 varchar(2));
CREATE TABLE
=> COPY alphabet FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> A|B
>> C|D
>> \.
=> SELECT * FROM alphabet;
 letter1 | letter2
---------+---------
 C       | D
 A       | B
(2 rows)
```

Concatenate the contents of the first column with a character string:

```
=> SELECT CONCAT(letter1, ' is a letter') FROM alphabet;
   CONCAT
---------------
 A is a letter
 C is a letter
(2 rows)
```

Concatenate the output of two nested CONCAT functions:

```
=> SELECT CONCAT(CONCAT(letter1, ' and '), CONCAT(letter2, ' are both letters')) FROM alphabet;
         CONCAT
--------------------------
 C and D are both letters
 A and B are both letters
(2 rows)
```

Concatenate a date and string:

```
=> SELECT current_date today;
  today
------------
 2021-12-10
(1 row)

=> SELECT CONCAT('2021-12-31'::date - current_date, ' days until end of year 2021');
            CONCAT
--------------------------------
 21 days until end of year 2021
(1 row)
```

DECODE

Compares * expression *to each search value one by one. If * expression *is equal to a search, the function returns the corresponding result. If no match is found, the function returns default. If default is omitted, the function returns null.

DECODE is similar to the IF-THEN-ELSE and CASE expressions:

```
CASE expression
[WHEN search THEN result]
[WHEN search THEN result]
...
[ELSE default];
```

The arguments can have any data type supported by Vertica. The result types of individual results are promoted to the least common type that can be used to represent all of them. This leads to a character string type, an exact numeric type, an approximate numeric type, or a DATETIME type, where all the various result arguments must be of the same type grouping.

Behavior type

Immutable

Syntax

```
DECODE ( expression, search, result [ , search, result ]...[, default ] )
```

Arguments

expression

The value to compare.

search

The value compared against *expression.*

result

The value returned, if * expression *is equal to search.

default

Optional. If no matches are found, DECODE returns default. If default is omitted, then DECODE returns NULL (if no matches are found).

Examples

The following example converts numeric values in the weight column from the product_dimension table to descriptive values in the output.

```
=> SELECT product_description, DECODE(weight,
     2, 'Light',
    50, 'Medium',
    71, 'Heavy',
    99, 'Call for help',
       'N/A')
 FROM product_dimension
 WHERE category_description = 'Food'
 AND department_description = 'Canned Goods'
 AND sku_number BETWEEN 'SKU-#49750' AND 'SKU-#49999'
 LIMIT 15;
      product_description       |     case
--------------------------------+---------------
 Brand #499 canned corn          | N/A
 Brand #49900 fruit cocktail     | Medium
 Brand #49837 canned tomatoes    | Heavy
 Brand #49782 canned peaches     | N/A
 Brand #49805 chicken noodle soup  | N/A
 Brand #49944 canned chicken broth | N/A
 Brand #49819 canned chili       | N/A
 Brand #49848 baked beans        | N/A
 Brand #49989 minestrone soup    | N/A
 Brand #49778 canned peaches     | N/A
 Brand #49770 canned peaches     | N/A
 Brand #4977 fruit cocktail      | N/A
 Brand #49933 canned olives      | N/A
 Brand #49750 canned olives      | Call for help
 Brand #49777 canned tomatoes    | N/A
(15 rows)
```

EDIT_DISTANCE

Calculates and returns the Levenshtein distance between two strings. The return value indicates the minimum number of single-character edits—insertions, deletions, or substitutions—that are required to change one string into the other. Compare to Jaro distance and Jaro-Winkler distance .

Behavior type

Immutable

Syntax

```
EDIT_DISTANCE ( string-expression1, string-expression2 )
```

Arguments

string-expression1* , *string-expression2

The two VARCHAR expressions to compare.

Examples

The Levenshtein distance between kitten and knitting is 3:

```
=> SELECT EDIT_DISTANCE ('kitten', 'knitting');
 EDIT_DISTANCE
---------------
             3
(1 row)
```

EDIT_DISTANCE calculates that no fewer than three changes are required to transform kitten to knitting :

1. kitten → knitten (insert n after k)
2. knitten → knittin (substitute i for e)

3. knittin → knitting (append g)

GREATEST

Returns the largest value in a list of expressions of any data type. All data types in the list must be the same or compatible . A NULL value in any one of the expressions returns NULL. Results can vary, depending on the locale's collation setting.

Behavior type

Stable

Syntax

```
GREATEST ( { * | expression[,...] } )
```

Arguments

*** | *expression* [,...]**

The expressions to evaluate, one of the following:

- * (asterisk)

  Evaluates all columns in the queried table.
- *expression*

  An expression of any data type. Functions that are included in *expression* must be deterministic.

Examples

GREATEST returns 10 as the largest value in the list:

```
=> SELECT GREATEST(7,5,10);
 GREATEST
----------
       10
(1 row)
```

If you put quotes around the integer expressions, GREATEST compares the values as strings and returns '7' as the greatest value:

```
=> SELECT GREATEST('7', '5', '10');
 GREATEST
----------
 7
(1 row)
```

The next example returns FLOAT 1.5 as the greatest because the integer is implicitly cast to float:

```
=> SELECT GREATEST(1, 1.5);
 GREATEST
----------
      1.5
(1 row)
```

GREATEST queries all columns in a view based on the VMart table product_dimension , and returns the largest value in each row:

```
=> CREATE VIEW query1 AS SELECT shelf_width, shelf_height, shelf_depth FROM product_dimension;
CREATE VIEW
=> SELECT shelf_width, shelf_height, shelf_depth, greatest(*) FROM query1 WHERE shelf_width = 1;
 shelf_width | shelf_height | shelf_depth | greatest
-------------+--------------+-------------+----------
           1 |            3 |           1 |        3
           1 |            3 |           3 |        3
           1 |            5 |           4 |        5
           1 |            2 |           2 |        2
           1 |            1 |           3 |        3
           1 |            2 |           2 |        2
           1 |            2 |           3 |        3
           1 |            1 |           5 |        5
           1 |            1 |           4 |        4
           1 |            5 |           3 |        5
           1 |            4 |           2 |        4
           1 |            4 |           5 |        5
           1 |            5 |           3 |        5
           1 |            2 |           5 |        5
           1 |            4 |           2 |        4
           1 |            4 |           4 |        4
           1 |            1 |           2 |        2
           1 |            4 |           3 |        4
...
```

See also

LEAST

GREATESTB

Returns the largest value in a list of expressions of any data type, using binary ordering. All data types in the list must be the same or compatible . A NULL value in any one of the expressions returns NULL. Results can vary, depending on the locale's collation setting.

Behavior type

Immutable

Syntax

```
GREATEST ( { * | expression[,...] } )
```

Arguments

*** | *expression* [,...]**

The expressions to evaluate, one of the following:

- * (asterisk)

  Evaluates all columns in the queried table.
- *expression*

  An expression of any data type. Functions that are included in *expression* must be deterministic.

Examples

The following command selects straße as the greatest in the series of inputs:

```
=> SELECT GREATESTB('straße', 'strasse');
 GREATESTB
-----------
 straße
(1 row)
```

GREATESTB returns 10 as the largest value in the list:

```
=> SELECT GREATESTB(7,5,10);
 GREATESTB
-----------
        10
(1 row)
```

If you put quotes around the integer expressions, GREATESTB compares the values as strings and returns '7' as the greatest value:

```
=> SELECT GREATESTB('7', '5', '10');
 GREATESTB
-----------
 7
(1 row)
```

The next example returns FLOAT 1.5 as the greatest because the integer is implicitly cast to float:

```
=> SELECT GREATESTB(1, 1.5);
 GREATESTB
-----------
       1.5
(1 row)
```

GREATESTB queries all columns in a view based on the VMart table product_dimension , and returns the largest value in each row:

```
=> CREATE VIEW query1 AS SELECT shelf_width, shelf_height, shelf_depth FROM product_dimension;
CREATE VIEW
=> SELECT shelf_width, shelf_height, shelf_depth, greatestb(*) FROM query1 WHERE shelf_width = 1;
 shelf_width | shelf_height | shelf_depth | greatestb
-------------+--------------+-------------+-----------
           1 |            3 |           1 |         3
           1 |            3 |           3 |         3
           1 |            5 |           4 |         5
           1 |            2 |           2 |         2
           1 |            1 |           3 |         3
           1 |            2 |           2 |         2
           1 |            2 |           3 |         3
           1 |            1 |           5 |         5
           1 |            1 |           4 |         4
           1 |            5 |           3 |         5
           1 |            4 |           2 |         4
           1 |            4 |           5 |         5
           1 |            5 |           3 |         5
           1 |            2 |           5 |         5
           1 |            4 |           2 |         4
...
```

See also

LEASTB

HEX_TO_BINARY

Translates the given VARCHAR hexadecimal representation into a VARBINARY value.

Behavior type

Immutable

Syntax

```
HEX_TO_BINARY ( [ 0x ] expression )
```

Arguments

expression

(BINARY or VARBINARY) String to translate.

0x

Optional prefix.

Notes

VARBINARY HEX_TO_BINARY(VARCHAR) converts data from character type in hexadecimal format to binary type. This function is the inverse of TO_HEX .

```
HEX_TO_BINARY(TO_HEX(x)) = x)
TO_HEX(HEX_TO_BINARY(x)) = x)
```

If there are an odd number of characters in the hexadecimal value, the first character is treated as the low nibble of the first (furthest to the left) byte.

Examples

If the given string begins with "0x" the prefix is ignored. For example:

```
=> SELECT HEX_TO_BINARY('0x6162') AS hex1, HEX_TO_BINARY('6162') AS hex2;
 hex1 | hex2
------+------
 ab   | ab
(1 row)
```

If an invalid hex value is given, Vertica returns an “invalid binary representation" error; for example:

```
=> SELECT HEX_TO_BINARY('0xffgf');
ERROR:  invalid hex string "0xffgf"
```

See also

- TO_HEX

HEX_TO_INTEGER

Translates the given VARCHAR hexadecimal representation into an INTEGER value.

Vertica completes this conversion as follows:

- Adds the 0x prefix if it is not specified in the input
- Casts the VARCHAR string to a NUMERIC
- Casts the NUMERIC to an INTEGER

Behavior type

Immutable

Syntax

```
HEX_TO_INTEGER ( [ 0x ] expression )
```

Arguments

expression

VARCHAR is the string to translate.

0x

Is the optional prefix.

Examples

You can enter the string with or without the Ox prefix. For example:

```
=> SELECT HEX_TO_INTEGER ('0aedc')
       AS hex1,HEX_TO_INTEGER ('aedc') AS hex2;
 hex1  | hex2
-------+-------
 44764 | 44764
(1 row)
```

If you pass the function an invalid hex value, Vertica returns an invalid input syntax error; for example:

```
=> SELECT HEX_TO_INTEGER ('0xffgf');
ERROR 3691:  Invalid input syntax for numeric: "0xffgf"
```

See also

- TO_HEX
- HEX_TO_BINARY

INITCAP

Capitalizes first letter of each alphanumeric word and puts the rest in lowercase.

Behavior type

Immutable

Syntax

INITCAP (*expression*)

Arguments

expression

(VARCHAR) is the string to format.

Notes

- Depends on collation setting of the locale.
- INITCAP is restricted to 32750 octet inputs, since it is possible for the UTF-8 representation of result to double in size.

Examples

Expression	Result
SELECT INITCAP('high speed database');	High Speed Database
SELECT INITCAP('LINUX TUTORIAL');	Linux Tutorial
SELECT INITCAP('abc DEF 123aVC 124Btd,lAsT');	Abc Def 123Avc 124Btd,Last
SELECT INITCAP('');	
SELECT INITCAP(null);	

INITCAPB

Capitalizes first letter of each alphanumeric word and puts the rest in lowercase. Multibyte characters are not converted and are skipped.

Behavior type

Immutable

Syntax

INITCAPB (*expression*)

Arguments

expression

(VARCHAR) is the string to format.

Notes

Depends on collation setting of the locale.

Examples

Expression	Result
SELECT INITCAPB('étudiant');	éTudiant
SELECT INITCAPB('high speed database');	High Speed Database
SELECT INITCAPB('LINUX TUTORIAL');	Linux Tutorial
SELECT INITCAPB('abc DEF 123aVC 124Btd,lAsT');	Abc Def 123Avc 124Btd,Last
SELECT INITCAPB('');	
SELECT INITCAPB(null);	

INSERT

Inserts a character string into a specified location in another character string.

Syntax

```
INSERT( 'string1', n, m, 'string2' )
```

Behavior type

Immutable

Arguments

string1

(VARCHAR) Is the string in which to insert the new string.

n

A character of type INTEGER that represents the starting point for the insertion within* string1 *. You specify the number of characters from the first character in string1 as the starting point for the insertion. For example, to insert characters before "c", in the string "abcdef," enter 3.

m

A character of type INTEGER that represents the number of characters in* string1 *(if any)* *that should be replaced by the insertion. For example,if you want the insertion to replace the letters "cd" in the string "abcdef, " enter 2.

string2

(VARCHAR) Is the string to be inserted.

Examples

The following example changes the string Warehouse to Storehouse using the INSERT function:

```
=> SELECT INSERT ('Warehouse',1,3,'Stor');
   INSERT
------------
 Storehouse
(1 row)
```

INSTR

Searches * string *for * substring *and returns an integer indicating the position of the character in * string *that is the first character of this *occurrence* . The return value is based on the character position of the identified character.

Behavior type

Immutable

Syntax

```
INSTR ( string , substring [, position [, occurrence ] ] )
```

Arguments

string

(CHAR or VARCHAR, or BINARY or VARBINARY) Text expression to search.

substring

(CHAR or VARCHAR, or BINARY or VARBINARY) String to search for.

position

Nonzero integer indicating the character of string where Vertica begins the search. If position is negative, then Vertica counts backward from the end of string and then searches backward from the resulting position. The first character of string occupies the default position 1, and position cannot be 0.

occurrence

Integer indicating which occurrence of string Vertica searches. The value of occurrence must be positive (greater than 0), and the default is 1.

Notes

Both *position* and *occurrence* must be of types that can resolve to an integer. The default values of both parameters are 1, meaning Vertica begins searching at the first character of string for the first occurrence of substring. The return value is relative to the beginning of string, regardless of the value of position, and is expressed in characters.

If the search is unsuccessful (that is, if substring does not appear * occurrence *times after the *position* character of *string,* the return value is 0.

Examples

The first example searches forward in string ‘abc’ for substring ‘b’. The search returns the position in ‘abc’ where ‘b’ occurs, or position 2. Because no position parameters are given, the default search starts at ‘a’, position 1.

```
=> SELECT INSTR('abc', 'b');
 INSTR
-------
     2
(1 row)
```

The following three examples use character position to search backward to find the position of a substring.

> **Note**
> Although it might seem intuitive that the function returns a negative integer, the position of *n* occurrence is read left to right in the sting, even though the search happens in reverse (from the end—or right side—of the string).

In the first example, the function counts backward one character from the end of the string, starting with character 'c'. The function then searches backward for the first occurrence of 'a', which it finds it in the first position in the search string.

```
=> SELECT INSTR('abc', 'a', -1);
 INSTR
-------
     1
(1 row)
```

In the second example, the function counts backward one byte from the end of the string, starting with character 'c'. The function then searches backward for the first occurrence of 'a', which it finds it in the first position in the search string.

```
=> SELECT INSTR(VARBINARY 'abc', VARBINARY 'a', -1);
 INSTR
-------
     1
(1 row)
```

In the third example, the function counts backward one character from the end of the string, starting with character 'b', and searches backward for substring 'bc', which it finds in the second position of the search string.

```
=> SELECT INSTR('abcb', 'bc', -1);
 INSTR
-------
     2
(1 row)
```

In the fourth example, the function counts backward one character from the end of the string, starting with character 'b', and searches backward for substring 'bcef', which it does not find. The result is 0.

```
=> SELECT INSTR('abcb', 'bcef', -1);
INSTR
-------
     0
(1 row)
```

In the fifth example, the function counts backward one byte from the end of the string, starting with character 'b', and searches backward for substring 'bcef', which it does not find. The result is 0.

```
=> SELECT INSTR(VARBINARY 'abcb', VARBINARY 'bcef', -1);
INSTR
-------
     0
(1 row)
```

Multibyte characters are treated as a single character:

```
=> SELECT INSTR('aébc', 'b');
 INSTR
-------
     3
(1 row)
```

Use INSTRB to treat multibyte characters as binary:

```
=> SELECT INSTRB('aébc', 'b');
  INSTRB
--------
      4
(1 row)
```

INSTRB

Searches *string* for *substring* and returns an integer indicating the octet position within string that is the first *occurrence* . The return value is based on the octet position of the identified byte.

Behavior type

Immutable

Syntax

```
INSTRB ( string , substring [, position [, occurrence ] ] )
```

Arguments

string

Is the text expression to search.

substring

Is the string to search for.

position

Is a nonzero integer indicating the character of string where Vertica begins the search. If position is negative, then Vertica counts backward from the end of string and then searches backward from the resulting position. The first byte of string occupies the default position 1, and position cannot be 0.

occurrence

Is an integer indicating which occurrence of string Vertica searches. The value of occurrence must be positive (greater than 0), and the default is 1.

Notes

Both *position* and *occurrence* must be of types that can resolve to an integer. The default values of both parameters are 1, meaning Vertica begins searching at the first byte of string for the first occurrence of substring. The return value is relative to the beginning of string, regardless of the value of position, and is expressed in octets.

If the search is unsuccessful (that is, if substring does not appear * occurrence *times after the * position *character of * string, *then the return value is 0.

Examples

```
=> SELECT INSTRB('straße', 'ß');
 INSTRB
--------
      5
(1 row)
```

See also

- INSTR

ISUTF8

Tests whether a string is a valid UTF-8 string. Returns true if the string conforms to UTF-8 standards, and false otherwise. This function is useful to test strings for UTF-8 compliance before passing them to one of the regular expression functions, such as REGEXP_LIKE , which expect UTF-8 characters by default.

ISUTF8 checks for invalid UTF8 byte sequences, according to UTF-8 rules:

- invalid bytes
- an unexpected continuation byte
- a start byte not followed by enough continuation bytes
- an Overload Encoding

The presence of an invalid UTF-8 byte sequence results in a return value of false.

To coerce a string to UTF-8, use MAKEUTF8 .

Syntax

```
ISUTF8( string );
```

Arguments

string

The string to test for UTF-8 compliance.

Examples

```
=> SELECT ISUTF8(E'\xC2\xBF'); -- UTF-8 INVERTED QUESTION MARK ISUTF8
--------
 t
(1 row)

=> SELECT ISUTF8(E'\xC2\xC0'); -- UNDEFINED UTF-8 CHARACTER
 ISUTF8
--------
 f
(1 row)
```

JARO_DISTANCE

Calculates and returns the Jaro similarity , an edit distance between two sequences. It is useful for queries designed for short strings, such as finding similar names. Also see Jaro-Winkler distance , which adds a prefix scale favoring strings that match in the beginning, and edit distance , which returns the Levenshtein distance between two strings.

Behavior type

Immutable

Syntax

```
JARO_DISTANCE (string-expression1, string-expression2)
```

Arguments

string-expression1 , string-expression2

The two VARCHAR expressions to compare. Neither can be NULL.

Example

Return only the names with a Jaro distance from 'rode' that is greater than 0.6:

```
=> SELECT name FROM names WHERE JARO_DISTANCE('rode', name) > 0.6;
name
---------
fred
frieda
rodgers
rogers
(4 rows)
```

JARO_WINKLER_DISTANCE

Calculates and returns the Jaro-Winkler similarity , an edit distance between two sequences. It is useful for queries designed for short strings, such as finding similar names. It is a variant of the Jaro distance metric, to which it adds a prefix scale giving more favorable ratings for strings that match from the beginning. See also edit distance , which returns the Levenshtein distance between two strings.

Behavior type

Immutable

Syntax

```
JARO_WINKLER_DISTANCE (string-expression1 , string-expression2 [ USING PARAMETERS prefix_scale=scale, prefix_length=length])
```

Arguments

string-expression1 , string-expression2

The two VARCHAR expressions to compare. Neither can be NULL.

Parameters

scale

A FLOAT specifying the scale value by which to weight the importance of matching prefixes. Optional.

default = 0.1

length

An non-negative INT representing the maximum matching prefix length. Optional.

default = 4

Examples

Return only the names with a Jaro-Winkler distance from 'rode' that is greater than 0.6:

```
=> SELECT name FROM names WHERE JARO_WINKLER_DISTANCE('rode', name) > 0.6;
name
---------
fred
frieda
rodgers
rogers
(4 rows)
```

The Jaro-Winkler distance between 'help' and 'hello' given a prefix_scale of 0.1 and prefix_length of 0 is 0.783333333333333:

```
=> select JARO_WINKLER_DISTANCE('help', 'hello' USING PARAMETERS prefix_scale=0.1, prefix_length=0);
jaro_winkler_distance
----------------------
0.783333333333333
(1 row)
```

LEAST

Returns the smallest value in a list of expressions of any data type. All data types in the list must be the same or compatible . A NULL value in any one of the expressions returns NULL. Results can vary, depending on the locale's collation setting.

Behavior type

Stable

Syntax

```
LEAST ( { * | expression[,...] } )
```

Arguments

*** | *expression* [,...]**

The expressions to evaluate, one of the following:

- * (asterisk)
  Evaluates all columns in the queried table.
- *expression*
  An expression of any data type. Functions that are included in *expression* must be deterministic.

Examples

LEASTB returns 5 as the smallest value in the list:

```
=> SELECT LEASTB(7, 5, 10);
 LEASTB
--------
      5
(1 row)
```

If you put quotes around the integer expressions, LEASTB compares the values as strings and returns '10' as the smallest value:

```
=> SELECT LEASTB('7', '5', '10');
 LEASTB
--------
 10
(1 row)
```

LEAST returns 1.5, as INTEGER 2 is implicitly cast to FLOAT:

```
=> SELECT LEAST(2, 1.5);
 LEAST
-------
   1.5
(1 row)
```

LEAST queries all columns in a view based on the VMart table product_dimension , and returns the smallest value in each row:

```
=> CREATE VIEW query1 AS SELECT shelf_width, shelf_height, shelf_depth FROM product_dimension;
CREATE VIEW
=> SELECT shelf_height, shelf_width, shelf_depth, least(*) FROM query1 WHERE shelf_height = 5;
 shelf_height | shelf_width | shelf_depth | least
--------------+-------------+-------------+-------
            5 |           3 |           4 |     3
            5 |           4 |           3 |     3
            5 |           1 |           4 |     1
            5 |           4 |           1 |     1
            5 |           2 |           4 |     2
            5 |           2 |           3 |     2
            5 |           1 |           3 |     1
            5 |           1 |           3 |     1
            5 |           5 |           1 |     1
            5 |           2 |           4 |     2
            5 |           4 |           5 |     4
            5 |           2 |           4 |     2
            5 |           4 |           4 |     4
            5 |           3 |           4 |     3
...
```

See also

GREATEST

LEASTB

Returns the smallest value in a list of expressions of any data type, using binary ordering. All data types in the list must be the same or compatible . A NULL value in any one of the expressions returns NULL. Results can vary, depending on the locale's collation setting.

Behavior type

Immutable

Syntax

```
LEASTB ( { * | expression[,...] } )
```

Arguments

*** | *expression* [,...]**

The expressions to evaluate, one of the following:

- * (asterisk)

  Evaluates all columns in the queried table.
- *expression*

An expression of any data type. Functions that are included in *expression* must be deterministic.

Examples

The following command selects strasse as the smallest value in the list:

```
=> SELECT LEASTB('straße', 'strasse');
 LEASTB
---------
 strasse
(1 row)
```

LEASTB returns 5 as the smallest value in the list:

```
=> SELECT LEAST(7, 5, 10);
 LEAST
-------
     5
(1 row)
```

If you put quotes around the integer expressions, LEAST compares the values as strings and returns '10' as the smallest value:

```
=> SELECT LEASTB('7', '5', '10');
 LEAST
-------
 10
(1 row)
```

The next example returns 1.5, as INTEGER 2 is implicitly cast to FLOAT:

```
=> SELECT LEASTB(2, 1.5);
 LEASTB
--------
   1.5
(1 row)
```

LEASTB queries all columns in a view based on the VMart table product_dimension , and returns the smallest value in each row:

```
=> CREATE VIEW query1 AS SELECT shelf_width, shelf_height, shelf_depth FROM product_dimension;
CREATE VIEW
=> SELECT shelf_height, shelf_width, shelf_depth, leastb(*) FROM query1 WHERE shelf_height = 5;
 shelf_height | shelf_width | shelf_depth | leastb
--------------+-------------+-------------+--------
            5 |           3 |           4 |      3
            5 |           4 |           3 |      3
            5 |           1 |           4 |      1
            5 |           4 |           1 |      1
            5 |           2 |           4 |      2
            5 |           2 |           3 |      2
            5 |           1 |           3 |      1
            5 |           1 |           3 |      1
            5 |           5 |           1 |      1
            5 |           2 |           4 |      2
            5 |           4 |           5 |      4
            5 |           2 |           4 |      2
            5 |           4 |           4 |      4
            5 |           3 |           4 |      3
            5 |           5 |           4 |      4
            5 |           5 |           1 |      1
            5 |           3 |           1 |      1
...
```

See also

GREATESTB

LEFT

Returns the specified characters from the left side of a string.

Behavior type

Immutable

Syntax

```
LEFT ( string-expr, length )
```

Arguments

string-expr

The string expression to return.

length

An integer value that specifies how many characters to return.

Examples

```
=> SELECT LEFT('vertica', 3);
 LEFT
------
 ver
(1 row)
```

```
 SELECT DISTINCT(
   LEFT (customer_name, 4)) FnameTruncated
   FROM customer_dimension ORDER BY FnameTruncated LIMIT 10;
 FnameTruncated
----------------
 Alex
 Amer
 Amy
 Anna
 Barb
 Ben
 Bett
 Bria
 Carl
 Crai
(10 rows)
```

See also

SUBSTR

LENGTH

Returns the length of a string. The behavior of LENGTH varies according to the input data type:

- CHAR and VARCHAR: Identical to CHARACTER_LENGTH , returns the string length in UTF-8 characters, .
- CHAR: Strips padding.
- BINARY and VARBINARY: Identical to OCTET_LENGTH , returns the string length in bytes (octets).

Behavior type

Immutable

Syntax

```
LENGTH ( expression )
```

Arguments

expression

String to evaluate, one of the following: CHAR, VARCHAR, BINARY or VARBINARY.

Examples

Statement	Returns
SELECT LENGTH('1234 '::CHAR(10));	4

SELECT LENGTH('1234 '::VARCHAR(10));	6
SELECT LENGTH('1234 '::BINARY(10));	10
SELECT LENGTH('1234 '::VARBINARY(10));	6
SELECT LENGTH(NULL::CHAR(10)) IS NULL;	t

See also

BIT_LENGTH

LOWER

Takes a string value and returns a VARCHAR value converted to lowercase.

Behavior type

stable

Syntax

```
LOWER ( expression )
```

Arguments

expression

CHAR or VARCHAR string to convert, where the string width is ≤ 65000 octets.

Important

In practice, *expression* should not exceed 32,500 octets. LOWER does not use the locale's collation setting—for example, collation=binary —to identify its encoding; rather, it treats the input argument as a UTF-8 encoded string. The UTF-8 representation of the input value might be double its original width. As a result, LOWER returns an error if the input value exceeds 32,500 octets.

Note also that if *expression* is a table column, LOWER calculates its size from the column's defined width, and not from the column data. If the column width is greater than VARCHAR(32500), Vertica returns an error.

Examples

```
=> SELECT LOWER('AbCdEfG');
 LOWER
---------
 abcdefg
(1 row)

=> SELECT LOWER('The Bat In The Hat');
      LOWER
--------------------
 the bat in the hat
(1 row)

=> SELECT LOWER('ÉTUDIANT');
 LOWER
----------
 étudiant
(1 row)
```

LOWERB

Returns a character string with each ASCII character converted to lowercase. Multi-byte characters are skipped and not converted.

Behavior type

Immutable

Syntax

```
LOWERB ( expression )
```

Arguments

expression

CHAR or VARCHAR string to convert

Examples

In the following example, the multi-byte UTF-8 character É is not converted to lowercase:

```
=> SELECT LOWERB('ÉTUDIANT');
  LOWERB
----------
 Étudiant
(1 row)

=> SELECT LOWER('ÉTUDIANT');
  LOWER
----------
 étudiant
(1 row)

=> SELECT LOWERB('AbCdEfG');
 LOWERB
---------
 abcdefg
(1 row)

=> SELECT LOWERB('The Vertica Database');
       LOWERB
----------------------
 the vertica database
(1 row)
```

LPAD

Returns a VARCHAR value representing a string of a specific length filled on the left with specific characters.

Behavior type

Immutable

Syntax

```
LPAD ( expression , length [ , fill ] )
```

Arguments

expression

(CHAR OR VARCHAR) specifies the string to fill

length

(INTEGER) specifies the number of characters to return

fill

(CHAR OR VARCHAR) specifies the repeating string of characters with which to fill the output string. The default is the space character.

Examples

```
=> SELECT LPAD('database', 15, 'xzy');
     LPAD
-----------------
 xzyxzyxdatabase
(1 row)
```

If the string is already longer than the specified length it is truncated on the right:

```
=> SELECT LPAD('establishment', 10, 'abc');
   LPAD
------------
 establishm
(1 row)
```

LTRIM

Returns a VARCHAR value representing a string with leading blanks removed from the left side (beginning).

Behavior type

Immutable

Syntax

```
LTRIM ( expression [ , characters ] )
```

Arguments

expression

(CHAR or VARCHAR) is the string to trim

characters

(CHAR or VARCHAR) specifies the characters to remove from the left side of *expression* . The default is the space character.

Examples

```
=> SELECT LTRIM('zzzyyyyyyyxxxxxxxxtrim', 'xyz');
 LTRIM
-------
 trim
(1 row)
```

See also

- BTRIM
- RTRIM
- TRIM

MAKEUTF8

Coerces a string to UTF-8 by removing or replacing non-UTF-8 characters.

MAKEUTF8 flags invalid UTF-8 characters byte by byte. For example, the byte sequence 0xE0 0x7F 0x80 is an invalid three-byte UTF-8 sequence, but the middle byte, 0x7F , is a valid one-byte UTF-8 character. In this example, 0x7F is preserved and the other two bytes are removed or replaced.

Syntax

```
MAKEUTF8( string-expression [USING PARAMETERS param=value] );
```

Arguments

string-expression

The string expression to evaluate for non-UTF-8 characters

Parameters

replacement_string

Specifies the VARCHAR(16) string that MAKEUTF8 uses to replace each non-UTF-8 character that it finds in *string-expression* . If this parameter is omitted, non-UTF-8 characters are removed. For example, the following SQL specifies to replace all non-UTF characters in the name column with the string ^ :

```
=> SELECT MAKEUTF8(name USING PARAMETERS replacement_string='^') FROM people;
```

MD5

Calculates the MD5 hash of string, returning the result as a VARCHAR string in hexadecimal.

Behavior type

Immutable

Syntax

```
MD5 ( string )
```

Arguments

string

Is the argument string.

Examples

```
=> SELECT MD5('123');
               MD5
----------------------------------
 202cb962ac59075b964b07152d234b70
(1 row)

=> SELECT MD5('Vertica'::bytea);
               MD5
----------------------------------
 fc45b815747d8236f9f6fdb9c2c3f676
(1 row)
```

See also

- SHA1
- SHA224
- SHA256
- SHA384
- SHA512

OCTET_LENGTH

Takes one argument as an input and returns the string length in octets for all string types.

Behavior type

Immutable

Syntax

```
OCTET_LENGTH ( expression )
```

Arguments

expression

(CHAR or VARCHAR or BINARY or VARBINARY) is the string to measure.

Notes

- If the data type of *expression* is a CHAR, VARCHAR or VARBINARY, the result is the same as the actual length of *expression* in octets. For CHAR, the length does not include any trailing spaces.
- If the data type of *expression* is BINARY, the result is the same as the fixed-length of *expression* .
- If the value of *expression* is NULL, the result is NULL.

Examples

Expression	Result
SELECT OCTET_LENGTH(CHAR(10) '1234 ');	4
SELECT OCTET_LENGTH(CHAR(10) '1234');	4
SELECT OCTET_LENGTH(CHAR(10) ' 1234');	6
SELECT OCTET_LENGTH(VARCHAR(10) '1234 ');	6
SELECT OCTET_LENGTH(VARCHAR(10) '1234 ');	5
SELECT OCTET_LENGTH(VARCHAR(10) '1234');	4
SELECT OCTET_LENGTH(VARCHAR(10) ' 1234');	7
SELECT OCTET_LENGTH('abc'::VARBINARY);	3
SELECT OCTET_LENGTH(VARBINARY 'abc');	3

SELECT OCTET_LENGTH(VARBINARY 'abc ');	5
SELECT OCTET_LENGTH(BINARY(6) 'abc');	6
SELECT OCTET_LENGTH(VARBINARY '');	0
SELECT OCTET_LENGTH(''::BINARY);	1
SELECT OCTET_LENGTH(null::VARBINARY);	
SELECT OCTET_LENGTH(null::BINARY);	

See also

- BIT_LENGTH
- CHARACTER_LENGTH
- LENGTH

OVERLAY

Replaces part of a string with another string and returns the new string value as a VARCHAR.

Behavior type

Immutable if using OCTETS, Stable otherwise

Syntax

```
OVERLAY ( input-string PLACING replace-string FROM position [ FOR extent ] [ USING { CHARACTERS | OCTETS } ] )
```

Arguments

input-string

The string to process, of type CHAR or VARCHAR.

replace-string

The string to replace the specified substring of *input-string* , of type CHAR or VARCHAR.

position

Integer ≥1 that specifies the first character or octet of *input-string* to overlay *replace-string* .

extent

Integer that specifies how many characters or octets of *input-string* to overlay with *replace-string* . If omitted, OVERLAY uses the length of *replace-string* .

For example, compare the following calls to OVERLAY:

- OVERLAY omits FOR clause. The number of characters replaced in the input string equals the number of characters in replacement string ABC :

```
dbadmin=> SELECT OVERLAY ('123456789' PLACING 'ABC' FROM 5);
 overlay
-----------
 1234ABC89
(1 row)
```

- OVERLAY includes a FOR clause that specifies to replace four characters in the input string with the replacement string. The replacement string is three characters long, so OVERLAY returns a string that is one character shorter than the input string:

```
=> SELECT OVERLAY ('123456789' PLACING 'ABC' FROM 5 FOR 4);
 overlay
----------
 1234ABC9
(1 row)
```

- OVERLAY includes a FOR clause that specifies to replace -2 characters in the input string with the replacement string. The function returns a string that is two characters longer than the input string:

```
=> SELECT OVERLAY ('123456789' PLACING 'ABC' FROM 5 FOR -2);
   overlay
----------------
 1234ABC3456789
(1 row)
```

USING CHARACTERS | OCTETS

Specifies whether OVERLAY uses characters (default) or octets.

> **Note**
> If you specify USING OCTETS , Vertica calls the OVERLAYB function.

Examples

```
=> SELECT OVERLAY('123456789' PLACING 'xxx' FROM 2);
 overlay
-----------
 1xxx56789
(1 row)

=> SELECT OVERLAY('123456789' PLACING 'XXX' FROM 2 USING OCTETS);
 overlayb
-----------
 1XXX56789
(1 row)

=> SELECT OVERLAY('123456789' PLACING 'xxx' FROM 2 FOR 4);
 overlay
----------
 1xxx6789
(1 row)

=> SELECT OVERLAY('123456789' PLACING 'xxx' FROM 2 FOR 5);
 overlay
---------
 1xxx789
(1 row)

=> SELECT OVERLAY('123456789' PLACING 'xxx' FROM 2 FOR 6);
 overlay
---------
 1xxx89
(1 row)
```

OVERLAYB

Replaces part of a string with another string and returns the new string as an octet value.

The OVERLAYB function treats the multibyte character string as a string of octets (bytes) and use octet numbers as incoming and outgoing position specifiers and lengths. The strings themselves are type VARCHAR, but they treated as if each byte was a separate character.

Behavior type

Immutable

Syntax

```
OVERLAYB ( input-string, replace-string, position [, extent ] )
```

Arguments

input-string

The string to process, of type CHAR or VARCHAR.

replace-string

The string to replace the specified substring of *input-string* , of type CHAR or VARCHAR.

position

Integer ≥1 that specifies the first octet of* input-string * to overlay *replace-string* .

extent

Integer that specifies how many octets of *input-string* to overlay with *replace-string* . If omitted, OVERLAY uses the length of *replace-string* .

Examples

```
=> SELECT OVERLAYB('123456789', 'ééé', 2);
 OVERLAYB
----------
 1ééé89
(1 row)
=> SELECT OVERLAYB('123456789', 'ßßß', 2);
 OVERLAYB
----------
 1ßßß89
(1 row)
=> SELECT OVERLAYB('123456789', 'xxx', 2);
 OVERLAYB
-----------
 1xxx56789
(1 row)
=> SELECT OVERLAYB('123456789', 'xxx', 2, 4);
 OVERLAYB
----------
 1xxx6789
(1 row)
=> SELECT OVERLAYB('123456789', 'xxx', 2, 5);
 OVERLAYB
----------
 1xxx789
(1 row)
=> SELECT OVERLAYB('123456789', 'xxx', 2, 6);
 OVERLAYB
----------
 1xxx89
(1 row)
```

POSITION

Returns an INTEGER value representing the character location of a specified substring with a string (counting from one).

Behavior type

Immutable

Syntax 1

```
POSITION ( substring IN string [ USING { CHARACTERS | OCTETS } ] )
```

Arguments

substring

(CHAR or VARCHAR) is the substring to locate

string

(CHAR or VARCHAR) is the string in which to locate the substring

USING CHARACTERS | OCTETS

Determines whether the position is reported by using characters (the default) or octets.

Syntax 2

```
POSITION ( substring IN string )
```

Arguments

substring

(VARBINARY) is the substring to locate

string

(VARBINARY) is the string in which to locate the substring

Notes

- When the string and substring are CHAR or VARCHAR, the return value is based on either the character or octet position of the substring.
- When the string and substring are VARBINARY, the return value is always based on the octet position of the substring.
- The string and substring must be consistent. Do not mix VARBINARY with CHAR or VARCHAR.
- POSITION is similar to STRPOS although POSITION allows finding by characters and by octet.
- If the string is not found, the return value is zero.

Examples

```
=> SELECT POSITION('é' IN 'étudiant' USING CHARACTERS);
 position
----------
        1
(1 row)
=> SELECT POSITION('ß' IN 'straße' USING OCTETS);
 positionb
-----------
         5
(1 row)

=> SELECT POSITION('c' IN 'abcd' USING CHARACTERS);
 position
----------
        3
(1 row)

=> SELECT POSITION(VARBINARY '456' IN VARBINARY '123456789');
 position
----------
        4
(1 row)

SELECT POSITION('n' in 'León') as 'default',
       POSITIONB('León', 'n') as 'POSITIONB',
       POSITION('n' in 'León' USING CHARACTERS) as 'pos_chars',
       POSITION('n' in 'León' USING OCTETS) as 'pos_oct',INSTR('León','n'),
       INSTRB('León','n'), REGEXP_INSTR('León','n');
 default | POSITIONB | pos_chars | pos_oct | INSTR | INSTRB | REGEXP_INSTR
---------+-----------+-----------+---------+-------+--------+--------------
       4 |         5 |         4 |       5 |     4 |      5 |            4
(1 row)
```

POSITIONB

Returns an INTEGER value representing the octet location of a specified substring with a string (counting from one).

Behavior type

Immutable

Syntax

```
POSITIONB ( string, substring )
```

Arguments

string

(CHAR or VARCHAR) is the string in which to locate the substring

substring

(CHAR or VARCHAR) is the substring to locate

Examples

```
=> SELECT POSITIONB('straße', 'ße');
 POSITIONB
-----------
         5
(1 row)


=> SELECT POSITIONB('étudiant', 'é');
 POSITIONB
-----------
         1
(1 row)
```

QUOTE_IDENT

Returns the specified string argument in the format required to use the string as an identifier in an SQL statement. Quotes are added as needed—for example, if the string contains non-identifier characters or is an SQL or Vertica-reserved keyword:

- 1time
- Next week
- SELECT

Embedded double quotes are doubled.

> **Note**
>
> - SQL identifiers such as table and column names are stored as created, and references to them are resolved using case-insensitive compares. Thus, you do not need to double-quote mixed-case identifiers.
> - Vertica quotes all reserved keywords, even if unused.

Behavior type

Immutable

Syntax

```
QUOTE_IDENT( 'string' )
```

Arguments

string

String to quote

Examples

Quoted identifiers are case-insensitive, and Vertica does not supply the quotes:

```
=> SELECT QUOTE_IDENT('VErtIcA');
 QUOTE_IDENT
-------------
 VErtIcA
(1 row)

=> SELECT QUOTE_IDENT('Vertica database');
    QUOTE_IDENT
--------------------
 "Vertica database"
(1 row)
```

Embedded double quotes are doubled:

```
=> SELECT QUOTE_IDENT('Vertica "!" database');
      QUOTE_IDENT
--------------------------
 "Vertica ""!"" database"
(1 row)
```

The following example uses the SQL keyword SELECT, so results are double quoted:

```
=> SELECT QUOTE_IDENT('select');
 QUOTE_IDENT
-------------
 "select"
(1 row)
```

See also

- QUOTE_LITERAL
- QUOTE_NULLABLE

QUOTE_LITERAL

Returns the given string suitably quoted for use as a string literal in a SQL statement string. Embedded single quotes and backslashes are doubled. As per the SQL standard, the function recognizes two consecutive single quotes within a string literal as a single quote character.

Behavior type

Immutable

Syntax

```
QUOTE_LITERAL ( string )
```

Arguments

string-expression

Argument that resolves to one or more strings to format as string literals.

Examples

In the following example, the first query returns no first name for Cher or Sting; the second query uses QUOTE_LITERAL, which sets off string values with single quotes, including empty strings. In this case, fname for Sting is set to an empty string (''), while fname for Cher is empty, indicating that it is set to null value:

```
=> SELECT * FROM lead_vocalists ORDER BY lname ASC;
 fname  |  lname  |                    band
--------+---------+-----------------------------------------------
        | Cher    | ["Sonny and Cher"]
 Mick   | Jagger  | ["Rolling Stones"]
 Diana  | Ross    | ["Supremes"]
 Grace  | Slick   | ["Jefferson Airplane","Jefferson Starship"]
        | Sting   | ["Police"]
 Stevie | Winwood | ["Spencer Davis Group","Traffic","Blind Faith"]
(6 rows)

=> SELECT QUOTE_LITERAL (fname) "First Name", QUOTE_NULLABLE (lname) "Last Name", band FROM lead_vocalists ORDER BY lname ASC;
 First Name | Last Name |                    band
------------+-----------+-----------------------------------------------
            | 'Cher'    | ["Sonny and Cher"]
 'Mick'     | 'Jagger'  | ["Rolling Stones"]
 'Diana'    | 'Ross'    | ["Supremes"]
 'Grace'    | 'Slick'   | ["Jefferson Airplane","Jefferson Starship"]
 ''         | 'Sting'   | ["Police"]
 'Stevie'   | 'Winwood' | ["Spencer Davis Group","Traffic","Blind Faith"]
(6 rows)
```

See also

- Character string literals
- QUOTE_NULLABLE

QUOTE_NULLABLE

Returns the given string suitably quoted for use as a string literal in an SQL statement string; or if the argument is null, returns the unquoted string NULL . Embedded single-quotes and backslashes are properly doubled.

Behavior type

Immutable

Syntax

```
QUOTE_NULLABLE ( string-expression )
```

Arguments

string-expression

Argument that resolves to one or more strings to format as string literals. If *string-expression* resolves to null value, QUOTE_NULLABLE returns NULL .

Examples

The following examples use the table lead_vocalists , where the first names (fname) for Cher and Sting are set to NULL and an empty string, respectively

```
=> SELECT * from lead_vocalists ORDER BY lname DESC;
 fname  |  lname  |                    band
--------+---------+---------------------------------------------
 Stevie | Winwood | ["Spencer Davis Group","Traffic","Blind Faith"]
        | Sting   | ["Police"]
 Grace  | Slick   | ["Jefferson Airplane","Jefferson Starship"]
 Diana  | Ross    | ["Supremes"]
 Mick   | Jagger  | ["Rolling Stones"]
        | Cher    | ["Sonny and Cher"]
(6 rows)

=> SELECT * FROM lead_vocalists WHERE fname IS NULL;
 fname | lname |       band
-------+-------+--------------------
       | Cher  | ["Sonny and Cher"]
(1 row)

=> SELECT * FROM lead_vocalists WHERE fname = '';
 fname | lname |    band
-------+-------+------------
       | Sting | ["Police"]
(1 row)
```

The following query uses QUOTE_NULLABLE. Like QUOTE_LITERAL , QUOTE_NULLABLE sets off string values with single quotes, including empty strings. Unlike QUOTE_LITERAL, QUOTE_NULLABLE outputs NULL for null values:

```
=> SELECT QUOTE_NULLABLE (fname) "First Name", QUOTE_NULLABLE (lname) "Last Name", band FROM lead_vocalists ORDER BY fname DESC;
 First Name | Last Name |                    band
------------+-----------+---------------------------------------------
 NULL       | 'Cher'    | ["Sonny and Cher"]
 'Stevie'   | 'Winwood' | ["Spencer Davis Group","Traffic","Blind Faith"]
 'Mick'     | 'Jagger'  | ["Rolling Stones"]
 'Grace'    | 'Slick'   | ["Jefferson Airplane","Jefferson Starship"]
 'Diana'    | 'Ross'    | ["Supremes"]
 ''         | 'Sting'   | ["Police"]
(6 rows)
```

See also

Character string literals

REPEAT

Replicates a string the specified number of times and concatenates the replicated values as a single string. The return value takes on the data type of the string argument. Return values for non-LONG data types and LONG data types can be up to 65000 and 32000000 bytes in length, respectively. If the length of *string* * *count* exceeds these limits, Vertica silently truncates the results.

Behavior type

Immutable

Syntax

```
REPEAT ( 'string', count )
```

Arguments

string

The string to repeat, one of the following:

- CHAR
- VARCHAR
- BINARY
- VARBINARY
- LONG VARCHAR
- LONG VARBINARY

count

An integer expression that specifies how many times to repeat *string* .

Examples

The following example repeats vmart three times:

```
=> SELECT REPEAT ('vmart', 3);
   REPEAT
-----------------
 vmartvmartvmart
(1 row)
```

REPLACE

Replaces all occurrences of characters in a string with another set of characters.

Behavior type

Immutable

Syntax

```
REPLACE ('string', 'target', 'replacement' )
```

Arguments

string

The string to modify.

target

The characters in *string* to replace.

replacement

The characters to replace *target* .

Examples

```
=> SELECT REPLACE('Documentation%20Library', '%20', ' ');
       REPLACE
-----------------------
 Documentation Library
(1 row)

=> SELECT REPLACE('This &amp; That', '&amp;', 'and');
   REPLACE
---------------
 This and That
(1 row)

=> SELECT REPLACE('straße', 'ß', 'ss');
 REPLACE
---------
 strasse
(1 row)
```

RIGHT

Returns the specified characters from the right side of a string.

Behavior type

Immutable

Syntax

```
RIGHT ( string-expr, length )
```

Arguments

string-expr

The string expression to return.

length

An integer value that specifies how many characters to return.

Examples

The following query returns the last three characters of the string 'vertica':

```
=> SELECT RIGHT('vertica', 3);
 RIGHT
-------
 ica
(1 row)
```

The following query queries date column date_ordered from table store.store_orders_fact . It coerces the dates to strings and extracts the last five characters from each string. It then returns all distinct strings:

```
SELECT DISTINCT(
  RIGHT(date_ordered::varchar, 5)) MonthDays
  FROM store.store_orders_fact ORDER BY MonthDays;
 MonthDays
-----------
 01-01
 01-02
 01-03
 01-04
 01-05
 01-06
 01-07
 01-08
 01-09
 01-10
 02-01
 02-02
 02-03
 ...
 11-08
 11-09
 11-10
 12-01
 12-02
 12-03
 12-04
 12-05
 12-06
 12-07
 12-08
 12-09
 12-10
(120 rows)
```

See also

SUBSTR

RPAD

Returns a VARCHAR value representing a string of a specific length filled on the right with specific characters.

Behavior type

Immutable

Syntax

```
RPAD ( expression , length [ , fill ] )
```

Arguments

expression

(CHAR OR VARCHAR) specifies the string to fill

length

(INTEGER) specifies the number of characters to return

fill

(CHAR OR VARCHAR) specifies the repeating string of characters with which to fill the output string. The default is the space character.

Examples

```
=> SELECT RPAD('database', 15, 'xzy');
     RPAD
-----------------
 databasexzyxzyx
(1 row)
```

If the string is already longer than the specified length it is truncated on the right:

```
=> SELECT RPAD('database', 6, 'xzy');
  RPAD
--------
 databa
(1 row)
```

RTRIM

Returns a VARCHAR value representing a string with trailing blanks removed from the right side (end).

Behavior type

Immutable

Syntax

```
RTRIM ( expression [ , characters ] )
```

Arguments

expression

(CHAR or VARCHAR) is the string to trim

characters

(CHAR or VARCHAR) specifies the characters to remove from the right side of *expression* . The default is the space character.

Examples

```
=> SELECT RTRIM('trimzzzyyyyyyyxxxxxxx', 'xyz');
 RTRIM
-------
 trim
(1 row)
```

See also

- BTRIM
- LTRIM
- TRIM

SHA1

Uses the US Secure Hash Algorithm 1 to calculate the SHA1 hash of string. Returns the result as a VARCHAR string in hexadecimal.

Behavior type

Immutable

Syntax

```
SHA1 ( string )
```

Arguments

string

The VARCHAR or VARBINARY string to be calculated.

Examples

The following examples calculate the SHA1 hash of the provided strings:

```
=> SELECT SHA1('123');
                 SHA1
------------------------------------------
 40bd001563085fc35165329ea1ff5c5ecbdbbeef
(1 row)
```

```
=> SELECT SHA1('Vertica'::bytea);
                 SHA1
------------------------------------------
 ee2cff8d3444995c6c301546c4fc5ee152d77c11
(1 row)
```

See also

- MD5
- SHA224
- SHA256
- SHA384
- SHA512

SHA224

Uses the US Secure Hash Algorithm 2 to calculate the SHA224 hash of string. Returns the result as a VARCHAR string in hexadecimal.

Behavior type

Immutable

Syntax

```
SHA224 ( string )
```

Arguments

string

The VARCHAR or VARBINARY string to be calculated.

Examples

The following examples calculate the SHA224 hash of the provided strings:

```
=> SELECT SHA224('abc');
                   SHA224
----------------------------------------------------------
78d8045d684abd2eece923758f3cd781489df3a48e1278982466017f
(1 row)
```

```
=> SELECT SHA224('Vertica'::bytea);
                   SHA224
----------------------------------------------------------
 135ac268f64ff3124aeeebc3cc0af0a29fd600a3be8e29ed97e45e25
(1 row)
```

```
=> SELECT sha224(''::varbinary) = 'd14a028c2a3a2bc9476102bb288234c415a2b01f828ea62ac5b3e42f' AS "TRUE";
 TRUE
------
t
(1 row)
```

See also

- MD5
- SHA1()
- SHA256()
- SHA384()
- SHA512()

SHA256

Uses the US Secure Hash Algorithm 2 to calculate the SHA256 hash of string. Returns the result as a VARCHAR string in hexadecimal.

Behavior type

Immutable

Syntax

```
SHA256 ( string )
```

Arguments

string

The VARCHAR or VARBINARY string to be calculated.

Examples

The following examples calculate the SHA256 hash of the provided strings:

```
=> SELECT SHA256('abc');
                              SHA256
------------------------------------------------------------------
 a665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27ae3
(1 row)
```

```
=> SELECT SHA256('Vertica'::bytea);
                              SHA256
------------------------------------------------------------------
 9981b0b7df9f5be06e9e1a7f4ae2336a7868d9ab522b9a6ca6a87cd9ed95ba53
(1 row)
```

```
=> SELECT sha256('') = 'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855' AS "TRUE";
 TRUE
------
t
(1 row)
```

See also

- MD5
- SHA1
- SHA224
- SHA384
- SHA512

SHA384

Uses the US Secure Hash Algorithm 2 to calculate the SHA384 hash of string. Returns the result as a VARCHAR string in hexadecimal.

Behavior type

Immutable

Syntax

```
SHA384 ( string )
```

Arguments

string

The VARCHAR or VARBINARY string to be calculated.

Examples

The following examples calculate the SHA384 hash of the provided strings:

```
=> SELECT SHA384('123');
                                  SHA384
-------------------------------------------------------------------------------------------
 9a0a82f0c0cf31470d7affede3406cc9aa8410671520b727044eda15b4c25532a9b5cd8aaf9cec4919d76255b6bfb00f
(1 row)
```

```
=> SELECT SHA384('Vertica'::bytea);
                                  SHA384
-------------------------------------------------------------------------------------------
 3431a717dc3289862bbd636a064d26980b47ebe4684b800cff4756f0c24985866ef97763eafd548fedb0ce28722c96bb
(1 row)
```

See also

- MD5
- SHA1
- SHA224
- SHA256
- SHA512

SHA512

Uses the US Secure Hash Algorithm 2 to calculate the SHA512 hash of string. Returns the result as a VARCHAR string in hexadecimal.

Behavior type

Immutable

Syntax

```
SHA512 ( string )
```

Arguments

string

The VARCHAR or VARBINARY string to be calculated.

Examples

The following examples calculate the SHA512 hash of the provided strings:

```
=> SELECT SHA512('123');
                                                  SHA512
-----------------------------------------------------------------------------------------------------------------------------
 3c9909afec25354d551dae21590bb26e38d53f2173b8d3dc3eee4c047e7ab1c1eb8b85103e3be7ba613b31bb5c9c36214dc9f14a42fd7a2fdb84856bca5c44c2


(1 row)
```

```
=> SELECT SHA512('Vertica'::bytea);
                                                  SHA512
-----------------------------------------------------------------------------------------------------------------------------
 c4ee2b2d17759226a3897c9c30d7c6df1145c4582849bb5191ee140bce05b83d3d869890cc3619b534fea6f97ff28a739d8b568a5ade66e756b3243ef97d3f00
(1 row)
```

See also

- MD5
- SHA1
- SHA224
- SHA256
- SHA384

SOUNDEX

Takes a VARCHAR argument and returns a four-character code that enables comparison of that argument with other SOUNDEX-encoded strings that are spelled differently in English, but are phonetically similar. SOUNDEX implements an algorithm that was developed by Robert C. Russell and Margaret King Odell, and is described in The Art of Computer Programming, Vol. 3 .

Behavior type

Immutable

Syntax

```
SOUNDEX ( string-expression )
```

Arguments

string-expression

The VARCHAR expression to encode.

Soundex encoding algorithm

Vertica uses the following Soundex encoding algorithm, which complies with most SQL implementations:

1. Save the first letter. Map all occurrences of a, e, i, o, u, y, h, w to zero (0).
2. Replace all consonants (include the first letter) with digits:
   - b, f, p, v → 1
   - c, g, j, k, q, s, x, z → 2
   - d, t → 3
   - l → 4
   - m, n → 5
   - r → 6
3. Replace all adjacent same digits with one digit, and then remove all zero (0) digits
4. If the saved letter's digit is the same as the resulting first digit, remove the digit (keep the letter).
5. Append 3 zeros if result contains less than 3 digits. Remove all except first letter and 3 digits after it.

> **Note**
> Encoding ignores all non-alphabetic characters—for example, the apostrophe in O'Connor.

Examples

Find last names in the employee_dimension table that are phonetically similar to Lee :

```
SELECT employee_last_name, employee_first_name, employee_state
   FROM public.employee_dimension
   WHERE SOUNDEX(employee_last_name) = SOUNDEX('Lee')
   ORDER BY employee_state, employee_last_name, employee_first_name;
Lea             | James             | AZ
Li              | Sam               | AZ
Lee             | Darlene           | CA
Lee             | Juanita           | CA
Li              | Amy               | CA
Li              | Barbara           | CA
Li              | Ben               | CA
...
```

See also

SOUNDEX_MATCHES

SOUNDEX_MATCHES

Compares the Soundex encodings of two strings. The function then returns an integer that indicates the number of matching characters, in the same order. The return value is 0 to 4 inclusive, where 0 indicates no match, and 4 an exact match.

For details on how Vertica implements Soundex encoding, see Soundex Encoding Algorithm .

Behavior type

Immutable

Syntax

```
SOUNDEX_MATCHES ( string-expression1, string-expression2 )
```

Arguments

string-expression1 , ***string-expression2***

The two VARCHAR expressions to encode and compare.

Examples

Find how well the Soundex encodings of two strings match:

- Compare the Soundex encodings of Lewis and Li :

```
> SELECT SOUNDEX_MATCHES('Lewis', 'Li');
 SOUNDEX_MATCHES
-----------------
               3
(1 row)
```

- Compare the Soundex encodings of Lee and Li :

```
=> SELECT SOUNDEX_MATCHES('Lee', 'Li');
 SOUNDEX_MATCHES
-----------------
               4
(1 row)
```

Find last names in the employee_dimension table whose Soundex encodings match at least 3 characters in the encoding for Lewis :

```
=> SELECT DISTINCT(employee_last_name)
    FROM public.employee_dimension
    WHERE SOUNDEX_MATCHES (employee_last_name, 'Lewis' ) >= 3 ORDER BY employee_last_name;
 employee_last_name
--------------------
 Lea
 Lee
 Leigh
 Lewis
 Li
 Reyes
(6 rows)
```

See also

SOUNDEX

SPACE

Returns the specified number of blank spaces, typically for insertion into a character string.

Behavior type

Immutable

Syntax

```
SPACE(n)
```

Arguments

n

An integer argument that specifies how many spaces to insert.

Examples

The following example concatenates strings x and y with 10 spaces inserted between them:

```
=> SELECT 'x' || SPACE(10) || 'y' AS Ten_spaces;
  Ten_spaces
--------------
 x          y
(1 row)
```

SPLIT_PART

Splits string on the delimiter and returns the string at the location of the beginning of the specified field (counting from 1).

Behavior type

Immutable

Syntax

```
SPLIT_PART ( string , delimiter , field )
```

Arguments

string

Argument string

delimiter

Delimiter

field

(INTEGER) Number of the part to return

Notes

Use this with the character form of the subfield.

Examples

The specified integer of 2 returns the second string, or def .

```
=> SELECT SPLIT_PART('abc~@~def~@~ghi', '~@~', 2);
 SPLIT_PART
------------
 def
(1 row)
```

In the next example, specify 3, which returns the third string, or 789 .

```
=> SELECT SPLIT_PART('123~|~456~|~789', '~|~', 3);
 SPLIT_PART
------------
 789
(1 row)
```

The tildes are for readability only. Omitting them returns the same results:

```
=> SELECT SPLIT_PART('123|456|789', '|', 3);
 SPLIT_PART
------------
 789
(1 row)
```

See what happens if you specify an integer that exceeds the number of strings: The result is not null, it is an empty string.

```
=> SELECT SPLIT_PART('123|456|789', '|', 4);
 SPLIT_PART
------------

(1 row)

=> SELECT SPLIT_PART('123|456|789', '|', 4) IS NULL;
 ?column?
----------
 f
(1 row)
```

If SPLIT_PART had returned NULL, LENGTH would have returned 0.

```
=> SELECT LENGTH (SPLIT_PART('123|456|789', '|', 4));
 LENGTH
--------
      0
(1 row)
```

If the locale of your database is BINARY, SPLIT_PART calls SPLIT_PARTB:

```
=> SHOW LOCALE;
  name  |              setting
--------+---------------------------------------
 locale | en_US@collation=binary (LEN_KBINARY)
(1 row)
=> SELECT SPLIT_PART('123456789', '5', 1);
 split_partb
-------------
 1234
(1 row)

=> SET LOCALE TO 'en_US@collation=standard';
INFO 2567:  Canonical locale: 'en_US@collation=standard'
Standard collation: 'LEN'
English (United States, collation=standard)
SET

=> SELECT SPLIT_PART('123456789', '5', 1);
 split_part
------------
 1234
(1 row)
```

See also

- SPLIT_PARTB

SPLIT_PARTB

Divides an input string on a delimiter character and returns the Nth segment, counting from 1. The VARCHAR arguments are treated as octets rather than UTF-8 characters.

Behavior type

Immutable

Syntax

```
SPLIT_PARTB ( string, delimiter, part-number)
```

Arguments

string

VARCHAR, the string to split.

delimiter

VARCHAR, the delimiter between segments.

part-number

INTEGER, the part number to return. The first part is 1, not 0.

Examples

The following example returns the third part of its input:

```
=> SELECT SPLIT_PARTB('straße~@~café~@~soupçon', '~@~', 3);
 SPLIT_PARTB
-------------
 soupçon
(1 row)
```

The tildes are for readability only. Omitting them returns the same results:

```
=> SELECT SPLIT_PARTB('straße @ café @ soupçon', '@', 3);
 SPLIT_PARTB
-------------
  soupçon
(1 row)
```

If the requested part number is greater than the number of parts, the function returns an empty string:

```
=> SELECT SPLIT_PARTB('straße @ café @ soupçon', '@', 4);
 SPLIT_PARTB
-------------

(1 row)

=> SELECT SPLIT_PARTB('straße @ café @ soupçon', '@', 4) IS NULL;
 ?column?
----------
 f
(1 row)
```

If the locale of your database is BINARY, SPLIT_PART calls SPLIT_PARTB:

```
=> SHOW LOCALE;
  name  |               setting
--------+--------------------------------------
 locale | en_US@collation=binary (LEN_KBINARY)
(1 row)
=> SELECT SPLIT_PART('123456789', '5', 1);
 split_partb
-------------
 1234
(1 row)

=> SET LOCALE TO 'en_US@collation=standard';
INFO 2567:  Canonical locale: 'en_US@collation=standard'
Standard collation: 'LEN'
English (United States, collation=standard)
SET

=> SELECT SPLIT_PART('123456789', '5', 1);
 split_part
------------
 1234
(1 row)
```

See also

- SPLIT_PART

STRPOS

Returns an INTEGER value that represents the location of a specified substring within a string (counting from one). If the substring is not found, STRPOS returns 0.

STRPOS is similar to POSITION ; however, POSITION allows finding by characters and by octet.

Behavior type

Immutable

Syntax

```
STRPOS ( string-expression , substring )
```

Arguments

string-expression

The string in which to locate *substring*

substring

The substring to locate in *string-expression*

Examples

```
=> SELECT ship_type, shipping_key, strpos (ship_type, 'DAY') FROM shipping_dimension WHERE strpos > 0 ORDER BY ship_type, shipping_key;
        ship_type          | shipping_key | strpos
---------------------------------+--------------+--------
 NEXT DAY                       |            1 |      6
 NEXT DAY                       |           13 |      6
 NEXT DAY                       |           19 |      6
 NEXT DAY                       |           22 |      6
 NEXT DAY                       |           26 |      6
 NEXT DAY                       |           30 |      6
 NEXT DAY                       |           34 |      6
 NEXT DAY                       |           38 |      6
 NEXT DAY                       |           45 |      6
 NEXT DAY                       |           51 |      6
 NEXT DAY                       |           67 |      6
 NEXT DAY                       |           69 |      6
 NEXT DAY                       |           80 |      6
 NEXT DAY                       |           90 |      6
 NEXT DAY                       |           96 |      6
 NEXT DAY                       |           98 |      6
 TWO DAY                        |            9 |      5
 TWO DAY                        |           21 |      5
 TWO DAY                        |           28 |      5
 TWO DAY                        |           32 |      5
 TWO DAY                        |           40 |      5
 TWO DAY                        |           43 |      5
 TWO DAY                        |           49 |      5
 TWO DAY                        |           50 |      5
 TWO DAY                        |           52 |      5
 TWO DAY                        |           53 |      5
 TWO DAY                        |           61 |      5
 TWO DAY                        |           73 |      5
 TWO DAY                        |           81 |      5
 TWO DAY                        |           83 |      5
 TWO DAY                        |           84 |      5
 TWO DAY                        |           85 |      5
 TWO DAY                        |           94 |      5
 TWO DAY                        |          100 |      5
(34 rows)
```

STRPOSB

Returns an INTEGER value representing the location of a specified substring within a string, counting from one, where each octet in the string is counted (as opposed to characters).

Behavior type

Immutable

Syntax

```
STRPOSB ( string , substring )
```

Arguments

string

(CHAR or VARCHAR) is the string in which to locate the substring

substring

(CHAR or VARCHAR) is the substring to locate

Notes

STRPOSB is identical to POSITIONB except for the order of the arguments.

Examples

```
=> SELECT STRPOSB('straße', 'e');
 STRPOSB
---------
       7
(1 row)


=> SELECT STRPOSB('étudiant', 'tud');
 STRPOSB
---------
       3
(1 row)
```

SUBSTR

Returns VARCHAR or VARBINARY value representing a substring of a specified string.

Behavior type

Immutable

Syntax

```
SUBSTR ( string , position [ , extent ] )
```

Arguments

string

(CHAR/VARCHAR or BINARY/VARBINARY) is the string from which to extract a substring. If null, Vertica returns no results.

position

(INTEGER or DOUBLE PRECISION) is the starting position of the substring (counting from one by characters). If 0 or negative, Vertica returns no results.

extent

(INTEGER or DOUBLE PRECISION) is the length of the substring to extract (in characters). The default is the end of the string.

Notes

SUBSTR truncates DOUBLE PRECISION input values.

Examples

```
=> SELECT SUBSTR('abc'::binary(3),1);
 substr
--------
 abc
(1 row)

=> SELECT SUBSTR('123456789', 3, 2);
 substr
--------
 34
(1 row)

=> SELECT SUBSTR('123456789', 3);
 substr
---------
 3456789
(1 row)

=> SELECT SUBSTR(TO_BITSTRING(HEX_TO_BINARY('0x10')), 2, 2);
 substr
--------
 00
(1 row)

=> SELECT SUBSTR(TO_HEX(10010), 2, 2);
 substr
--------
 71
(1 row)
```

SUBSTRB

Returns an octet value representing the substring of a specified string.

Behavior type

Immutable

Syntax

```
SUBSTRB ( string , position [ , extent ] )
```

Arguments

string

(CHAR/VARCHAR) is the string from which to extract a substring.

position

(INTEGER or DOUBLE PRECISION) is the starting position of the substring (counting from one in octets).

extent

(INTEGER or DOUBLE PRECISION) is the length of the substring to extract (in octets). The default is the end of the string

Notes

- This function treats the multibyte character string as a string of octets (bytes) and uses octet numbers as incoming and outgoing position specifiers and lengths. The strings themselves are type VARCHAR, but they treated as if each octet were a separate character.
- SUBSTRB truncates DOUBLE PRECISION input values.

Examples

```
=> SELECT SUBSTRB('soupçon', 5);
 SUBSTRB
---------
 çon
(1 row)

=> SELECT SUBSTRB('soupçon', 5, 2);
 SUBSTRB
---------
 ç
(1 row)
```

Vertica returns the following error message if you use BINARY/VARBINARY:

```
=>SELECT SUBSTRB('abc'::binary(3),1);
ERROR: function substrb(binary, int) does not exist, or permission is denied for substrb(binary, int)
HINT: No function matches the given name and argument types. You may need to add explicit type casts.
```

SUBSTRING

Returns a value representing a substring of the specified string at the given position, given a value, a position, and an optional length. SUBSTRING truncates DOUBLE PRECISION input values.

Behavior type

Immutable if USING OCTETS, stable otherwise.

Syntax

```
SUBSTRING ( string, position[, length ]
   [USING {CHARACTERS | OCTETS } ] )
SUBSTRING ( string FROM position [ FOR length ]
   [USING { CHARACTERS | OCTETS } ] )
```

Arguments

string

(CHAR/VARCHAR or BINARY/VARBINARY) is the string from which to extract a substring

position

(INTEGER or DOUBLE PRECISION) is the starting position of the substring (counting from one by either characters or octets). (The default is characters.) If position is greater than the length of the given value, an empty value is returned.

length

(INTEGER or DOUBLE PRECISION) is the length of the substring to extract in either characters or octets. (The default is characters.) The default is the end of the string.If a length is given the result is at most that many bytes. The maximum length is the length of the given value less the given position. If no length is given or if the given length is greater than the maximum length then the length is set to the maximum length.

USING CHARACTERS | OCTETS

Determines whether the value is expressed in characters (the default) or octets.

Examples

```
=> SELECT SUBSTRING('abc'::binary(3),1);
 substring
-----------
 abc
(1 row)

=> SELECT SUBSTRING('soupçon', 5, 2 USING CHARACTERS);
 substring
-----------
 ço
(1 row)

=> SELECT SUBSTRING('soupçon', 5, 2 USING OCTETS);
 substring
-----------
 ç
(1 row)
```

If you use a negative position, then the functions starts at a non-existent position. In this example, that means counting eight characters starting at position -4. So the function starts at the empty position -4 and counts five characters, including a position for zero which is also empty. This returns three characters.

```
=> SELECT SUBSTRING('1234567890', -4, 8);
 substring
 -----------
 123
(1 row)
```

TRANSLATE

Replaces individual characters in *string_to_replace* with other characters.

Behavior type

Immutable

Syntax

```
TRANSLATE ( string_to_replace , from_string , to_string );
```

Arguments

string_to_replace

String to be translated.

from_string

Contains characters that should be replaced in string_to_replace.

to_string

Any character in string_to_replace that matches a character in from_string is replaced by the corresponding character in to_string.

Examples

```
=> SELECT TRANSLATE('straße', 'ß', 'ss');
 TRANSLATE
-----------
 strase
(1 row)
```

TRIM

Combines the BTRIM, LTRIM, and RTRIM functions into a single function.

Behavior type

Immutable

Syntax

```
TRIM ( [ [ LEADING | TRAILING | BOTH ] [ characters ] FROM ] expression )
```

Arguments

LEADING

Removes the specified characters from the left side of the string

TRAILING

Removes the specified characters from the right side of the string

BOTH

Removes the specified characters from both sides of the string (default)

characters

(CHAR or VARCHAR) specifies the characters to remove from *expression* . The default is the space character.

expression

(CHAR or VARCHAR) is the string to trim

Examples

```
=> SELECT '-' || TRIM(LEADING 'x' FROM 'xxdatabasexx') || '-';
   ?column?
--------------
 -databasexx-
(1 row)

=> SELECT '-' || TRIM(TRAILING 'x' FROM 'xxdatabasexx') || '-';
   ?column?
--------------
 -xxdatabase-
(1 row)

=> SELECT '-' || TRIM(BOTH 'x' FROM 'xxdatabasexx') || '-';
  ?column?
------------
 -database-
(1 row)

=> SELECT '-' || TRIM('x' FROM 'xxdatabasexx') || '-';
  ?column?
------------
 -database-
(1 row)

=> SELECT '-' || TRIM(LEADING FROM '  database  ') || '-';
   ?column?
--------------
 -database  -
(1 row)

=> SELECT '-' || TRIM('  database  ') || '-';  ?column?
------------
 -database-
(1 row)
```

See also

- BTRIM
- LTRIM
- RTRIM

UPPER

Returns a VARCHAR value containing the argument converted to uppercase letters.

Starting in Release 5.1, this function treats the string argument as a UTF-8 encoded string, rather than depending on the collation setting of the locale (for example, collation=binary) to identify the encoding.

Behavior type

stable

Syntax

```
UPPER ( expression )
```

Arguments

expression

CHAR or VARCHAR containing the string to convert

Notes

UPPER is restricted to 32500 octet inputs, since it is possible for the UTF-8 representation of result to double in size.

Examples

```
=> SELECT UPPER('AbCdEfG');
 UPPER
---------
 ABCDEFG
(1 row)
=> SELECT UPPER('étudiant');
 UPPER
---------
 ÉTUDIANT
(1 row)
```

UPPERB

Returns a character string with each ASCII character converted to uppercase. Multibyte characters are not converted and are skipped.

Behavior type

Immutable

Syntax

```
UPPERB ( expression )
```

Arguments

expression

(CHAR or VARCHAR) is the string to convert

Examples

In the following example, the multibyte UTF-8 character é is not converted to uppercase:

```
=> SELECT UPPERB('étudiant');
 UPPERB
---------
 éTUDIANT
(1 row)

=> SELECT UPPERB('AbCdEfG');
 UPPERB
---------
 ABCDEFG
(1 row)

=> SELECT UPPERB('The Vertica Database');
      UPPERB
----------------------
 THE VERTICA DATABASE
(1 row)
```

URI functions

The functions in this section follow the RFC 3986 standard for percent-encoding a Universal Resource Identifier (URI).

In this section

- URI_PERCENT_DECODE
- URI_PERCENT_ENCODE

URI_PERCENT_DECODE

Decodes a percent-encoded Universal Resource Identifier (URI) according to the RFC 3986 standard.

Syntax

```
URI_PERCENT_DECODE (expression)
```

Behavior type

Immutable

Parameters

expression

(VARCHAR) is the string to convert.

Examples

The following example invokes uri_percent_decode on the Websites column of the URI table and returns a decoded URI:

```
=> SELECT URI_PERCENT_DECODE(Websites) from URI;
            URI_PERCENT_DECODE
-----------------------------------------------
 http://www.faqs.org/rfcs/rfc3986.html x xj%a%
(1 row)
```

The following example returns the original URI in the Websites column and its decoded version:

```
=> SELECT Websites, URI_PERCENT_DECODE (Websites) from URI;

                  Websites                  |            URI_PERCENT_DECODE
------------------------------------------------------+-----------------------------------------------
 http://www.faqs.org/rfcs/rfc3986.html+x%20x%6a%a% | http://www.faqs.org/rfcs/rfc3986.html x xj%a%

(1 row)
```

URI_PERCENT_ENCODE

Encodes a Universal Resource Identifier (URI) according to the RFC 3986 standard for percent encoding. For compatibility with older encoders, this function converts + to space; space is converted to %20 .

Syntax

```
URI_PERCENT_ENCODE (expression)
```

Behavior type

Immutable

Parameters

expression

(VARCHAR) is the string to convert.

Examples

The following example shows how the uri_percent_encode function is invoked on a the Websites column of the URI table and returns an encoded URI:

```
=> SELECT URI_PERCENT_ENCODE(Websites) from URI;
            URI_PERCENT_ENCODE
-------------------------------------------
 http%3A%2F%2Fexample.com%2F%3F%3D11%2F15
(1 row)
```

The following example returns the original URI in the Websites column and it's encoded form:

```
=> SELECT Websites, URI_PERCENT_ENCODE(Websites) from URI;          Websites          |          URI_PERCENT_ENCODE
----------------------------+-------------------------------------------
 http://example.com/?=11/15 | http%3A%2F%2Fexample.com%2F%3F%3D11%2F15
(1 row)
```

UUID functions

Currently, Vertica provides one function to support UUID data types, UUID_GENERATE .

In this section

- UUID_GENERATE

UUID_GENERATE

Returns a new universally unique identifier (UUID) that is generated based on high-quality randomness from /dev/urandom .

Behavior type

Volatile

Syntax

```
UUID_GENERATE()
```

Examples

```
=> CREATE TABLE Customers(
    cust_id UUID DEFAULT UUID_GENERATE(),
  lname VARCHAR(36),
  fname VARCHAR(24));
CREATE TABLE
=> INSERT INTO Customers VALUES (DEFAULT, 'Kearney', 'Thomas');
 OUTPUT
--------
      1
(1 row)

=> COPY Customers (lname, fname) FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> Pham|Duc
>> Garcia|Mary
>> \.
=> SELECT * FROM Customers;
               cust_id               |  lname  | fname
-------------------------------------+---------+--------
 03fe0794-ac5d-42d4-8246-54f7ec81ed0c | Pham    | Duc
 6950313d-c77e-4c11-a86e-0a54aa3ec114 | Kearney | Thomas
 9c9653ce-c2e4-4441-b0f7-0137b54cc28c | Garcia  | Mary
(3 rows)
```

Database Designer functions

Database Designer functions perform the following operations, generally performed in the following order:

1. Create a design .
2. Set design properties .
3. Populate a design .
4. Create design and deployment scripts .
5. Get design data .
6. Clean up .

Important

You can also use meta-function DESIGNER_SINGLE_RUN , which encapsulates all of these steps with a single call. The meta-function iterates over all queries within a specified timespan, and returns with a design ready for deployment.

For detailed information, see Workflow for running Database Designer programmatically . For information on required privileges, see Privileges for running Database Designer functions

> **Caution**
> Before running Database Designer functions on an existing schema, back up the current design by calling EXPORT_CATALOG .

Create a design

DESIGNER_CREATE_DESIGN directs Database Designer to create a design.

Set design properties

The following functions let you specify design properties:

- DESIGNER_SET_DESIGN_TYPE : Specifies whether the design is comprehensive or incremental.
- DESIGNER_DESIGN_PROJECTION_ENCODINGS : Analyzes encoding in the specified projections and creates a script that implements encoding recommendations.
- DESIGNER_SET_DESIGN_KSAFETY : Sets the K-safety value for a comprehensive design.
- DESIGNER_SET_OPTIMIZATION_OBJECTIVE : Specifies whether the design optimizes for query or load performance.
- DESIGNER_SET_PROPOSE_UNSEGMENTED_PROJECTIONS : Enables inclusion of unsegmented projections in the design.

Populate a design

The following functions let you add tables and queries to your Database Designer design:

- DESIGNER_ADD_DESIGN_TABLES : Adds the specified tables to a design.
- DESIGNER_ADD_DESIGN_QUERY , DESIGNER_ADD_DESIGN_QUERIES , DESIGNER_ADD_DESIGN_QUERIES_FROM_RESULTS : Adds queries to the design and weights them.

Create design and deployment scripts

The following functions populate the Database Designer workspace and create design and deployment scripts. You can also analyze statistics, deploy the design automatically, and drop the workspace after the deployment:

- DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY : Populates the design and creates design and deployment scripts.
- DESIGNER_WAIT_FOR_DESIGN : Waits for a currently running design to complete.

Reset a design

DESIGNER_RESET_DESIGN discards all the run-specific information of the previous Database Designer build or deployment of the specified design but retains its configuration.

Get design data

The following functions display information about projections and scripts that the Database Designer created:

- DESIGNER_OUTPUT_ALL_DESIGN_PROJECTIONS : Sends to standard output DDL statements that define design projections.
- DESIGNER_OUTPUT_DEPLOYMENT_SCRIPT : Sends to standard output a design's deployment script.

Cleanup

The following functions cancel any running Database Designer operation or drop a Database Designer design and all its contents:

- DESIGNER_CANCEL_POPULATE_DESIGN : Cancels population or deployment operation for the specified design if it is currently running.
- DESIGNER_DROP_DESIGN : Removes the schema associated with the specified design and all its contents.
- DESIGNER_DROP_ALL_DESIGNS : Removes all Database Designer-related schemas associated with the current user.

In this section

- DESIGNER_ADD_DESIGN_QUERIES
- DESIGNER_ADD_DESIGN_QUERIES_FROM_RESULTS
- DESIGNER_ADD_DESIGN_QUERY
- DESIGNER_ADD_DESIGN_TABLES
- DESIGNER_CANCEL_POPULATE_DESIGN
- DESIGNER_CREATE_DESIGN
- DESIGNER_DESIGN_PROJECTION_ENCODINGS
- DESIGNER_DROP_ALL_DESIGNS
- DESIGNER_DROP_DESIGN
- DESIGNER_OUTPUT_ALL_DESIGN_PROJECTIONS
- DESIGNER_OUTPUT_DEPLOYMENT_SCRIPT

- DESIGNER_RESET_DESIGN
- DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY
- DESIGNER_SET_DESIGN_KSAFETY
- DESIGNER_SET_DESIGN_TYPE
- DESIGNER_SET_OPTIMIZATION_OBJECTIVE
- DESIGNER_SET_PROPOSE_UNSEGMENTED_PROJECTIONS
- DESIGNER_SINGLE_RUN
- DESIGNER_WAIT_FOR_DESIGN

DESIGNER_ADD_DESIGN_QUERIES

Reads and evaluates queries from an input file, and adds the queries that it accepts to the specified design. All accepted queries are assigned a weight of 1.

The following requirements apply:

- All queried tables must previously be added to the design with DESIGNER_ADD_DESIGN_TABLES .
- If the design type is incremental, the Database Designer reads only the first 100 queries in the input file, and ignores all queries beyond that number.

All accepted queries are added to the system table DESIGN_QUERIES .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_ADD_DESIGN_QUERIES ( 'design-name', 'queries-file' [, return-results] )
```

Parameters

design-name

Name of the target design.

queries-file

Absolute path and name of the file that contains the queries to evaluate, on the local file system of the node where the session is connected, or another file system or object store that Vertica supports.

return-results

Boolean, optionally specifies whether to return results of the add operation to standard output. If set to true, Database Designer returns the following results:

- Number of accepted queries
- Number of queries referencing non-design tables
- Number of unsupported queries
- Number of illegal queries

Privileges

Non-superuser: design creator with all privileges required to execute the queries in *input-file* .

Errors

Database Designer returns an error in the following cases:

- The query contains illegal syntax.
- The query references:
  - External or system tables only
  - Local temporary or other non-design tables
- DELETE or UPDATE query has one or more subqueries.
- INSERT query does not include a SELECT clause.
- Database Designer cannot optimize the query.

Examples

The following example adds queries from vmart_queries.sql to the VMART_DESIGN design. This file contains nine queries. The statement includes a third argument of true, so Database Designer returns results of the add operation:

```
=> SELECT DESIGNER_ADD_DESIGN_QUERIES ('VMART_DESIGN', '/tmp/examples/vmart_queries.sql', 'true');
...
 DESIGNER_ADD_DESIGN_QUERIES
-----------------------------------------------------
 Number of accepted queries                  =9
 Number of queries referencing non-design tables =0
 Number of unsupported queries               =0
 Number of illegal queries                   =0
(1 row)
```

See also

Running Database Designer programmatically

DESIGNER_ADD_DESIGN_QUERIES_FROM_RESULTS

Executes the specified query and evaluates results in the following columns:

- QUERY_TEXT (required): Text of potential design queries.
- QUERY_WEIGHT (optional): The weight assigned to each query that indicates its importance relative to other queries, a real number >0 and ≤ 1. Database Designer uses this setting when creating the design to prioritize the query. If DESIGNER_ADD_DESIGN_QUERIES_FROM_RESULTS returns any results that omit this value, Database Designer sets their weight to 1.

After evaluating the queries in QUERY_TEXT , DESIGNER_ADD_DESIGN_QUERIES_FROM_RESULTS adds all accepted queries to the design. An unlimited number of queries can be added to the design.

Before you add queries to a design, you must add the queried tables with DESIGNER_ADD_DESIGN_TABLES .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_ADD_DESIGN_QUERIES_FROM_RESULTS ( 'design-name', 'query' )
```

Parameters

design-name

Name of the target design.

query

A valid SQL query whose results contain columns named QUERY_TEXT and, optionally, QUERY_WEIGHT .

Privileges

Non-superuser: design creator with all privileges required to execute the specified query, and all queries returned by this function

Errors

Database Designer returns an error in the following cases:

- The query contains illegal syntax.
- The query references:
  - External or system tables only
  - Local temporary or other non-design tables
- DELETE or UPDATE query has one or more subqueries.
- INSERT query does not include a SELECT clause.
- Database Designer cannot optimize the query.

Examples

The following example queries the system table QUERY_REQUESTS for all long-running queries (> 1 million microseconds) and adds them to the VMART_DESIGN design. The query returns no information on query weights, so all queries are assigned a weight of 1:

```
=> SELECT DESIGNER_ADD_DESIGN_QUERIES_FROM_RESULTS ('VMART_DESIGN',
  'SELECT request as query_text FROM query_requests where request_duration_ms > 1000000 AND request_type =
  ''QUERY'';');
```

See also

Running Database Designer programmatically

DESIGNER_ADD_DESIGN_QUERY

Reads and parses the specified query, and if accepted, adds it to the design. Before you add queries to a design, you must add the queried tables with DESIGNER_ADD_DESIGN_TABLES .

All accepted queries are added to the system table DESIGN_QUERIES .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_ADD_DESIGN_QUERY ( 'design-name', 'design-query' [, query-weight] )
```

Parameters

design-name

Name of the target design.

design-query

Executable SQL query.

query-weight

Optionally assigns a weight to each query that indicates its importance relative to other queries, a real number >0 and ≤ 1. Database Designer uses this setting to prioritize queries in the design .

If you omit this parameter, Database Designer assigns a weight of 1.

Privileges

Non-superuser: design creator with all privileges required to execute the specified query

Errors

Database Designer returns an error in the following cases:

- The query contains illegal syntax.
- The query references:
  - External or system tables only
  - Local temporary or other non-design tables
- DELETE or UPDATE query has one or more subqueries.
- INSERT query does not include a SELECT clause.
- Database Designer cannot optimize the query.

Examples

The following example adds the specified query to the VMART_DESIGN design and assigns that query a weight of 0.5:

```
=> SELECT DESIGNER_ADD_DESIGN_QUERY (
  'VMART_DESIGN',
  'SELECT customer_name, customer_type FROM customer_dimension ORDER BY customer_name ASC;', 0.5
  );
```

See also

Running Database Designer programmatically

DESIGNER_ADD_DESIGN_TABLES

Adds the specified tables to a design. You must run DESIGNER_ADD_DESIGN_TABLES before adding design queries to the design. If no tables are added to the design, Vertica does not accept design queries.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_ADD_DESIGN_TABLES ( 'design-name', '[ table-spec[,...] ]' [, 'analyze-statistics'] )
```

Parameters

design-name

Name of the Database Designer design.

***table-spec* [,...]**

One or more comma-delimited arguments that specify which tables to add to the design, where each *table-spec* argument can specify tables as follows:

- [*schema* .] *table*
  Add *table* to the design.
- *schema* .*
  Add all tables in *schema* .

If set to an empty string, Vertica adds all tables in the database to which the user has access.

analyze-statistics

Boolean that optionally specifies whether to run ANALYZE_STATISTICS after adding the specified tables to the design, by default set to false .

Accurate statistics help Database Designer optimize compression and query performance. Updating statistics takes time and resources.

Privileges

Non-superuser: design creator with USAGE privilege on the design table schema and owner of the design table

Examples

The following example adds to design VMART_DESIGN all tables from schemas online_sales and store , and analyzes statistics for those tables:

```
=> SELECT DESIGNER_ADD_DESIGN_TABLES('VMART_DESIGN', 'online_sales.*, store.*','true');
 DESIGNER_ADD_DESIGN_TABLES
----------------------------
                          7
(1 row)
```

See also

Running Database Designer programmatically

DESIGNER_CANCEL_POPULATE_DESIGN

Cancels population or deployment operation for the specified design if it is currently running. When you cancel a deployment, the Database Designer cancels the projection refresh operation. It does not roll back projections that it already deployed and refreshed.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_CANCEL_POPULATE_DESIGN ( 'design-name' )
```

Parameters

design-name

Name of the design operation to cancel.

Privileges

Non-superuser: design creator

Examples

The following example cancels a currently running design for VMART_DESIGN and then drops the design:

```
=> SELECT DESIGNER_CANCEL_POPULATE_DESIGN ('VMART_DESIGN');
=> SELECT DESIGNER_DROP_DESIGN ('VMART_DESIGN', 'true');
```

See also

Running Database Designer programmatically

DESIGNER_CREATE_DESIGN

Creates a design with the specified name.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_CREATE_DESIGN ( 'design-name' )
```

Parameters

design-name

Name of the design to create, can contain only alphanumeric and underscore (_) characters.

Two users cannot have designs with the same name at the same time.

Privileges

- Superuser
- DBDUSER with WRITE privileges on storage location of *design-name* .

Database Designer system views

If any of the following V_MONITOR tables do not already exist from previous designs, DESIGNER_CREATE_DESIGN creates them:

- DESIGNS
- DESIGN_TABLES
- DEPLOYMENT_PROJECTIONS
- DEPLOYMENT_PROJECTION_STATEMENTS
- DESIGN_QUERIES
- OUTPUT_DEPLOYMENT_STATUS
- OUTPUT_EVENT_HISTORY

Examples

The following example creates the design VMART_DESIGN :

```
=> SELECT DESIGNER_CREATE_DESIGN('VMART_DESIGN');
 DESIGNER_CREATE_DESIGN
------------------------
                      0
(1 row)
```

See also

Running Database Designer programmatically

DESIGNER_DESIGN_PROJECTION_ENCODINGS

Analyzes encoding in the specified projections, creates a script to implement encoding recommendations, and optionally deploys the recommendations.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_DESIGN_PROJECTION_ENCODINGS ( '[ proj-spec[,... ] ]', '[destination]' [, 'deploy'] [, 'reanalyze-encodings'] )
```

Parameters

***proj-spec* [,...]**

One or more comma-delimited projections to add to the design. Each projection can be specified in one of the following ways:

- [[*schema* .] *table* .] *projection*
  Specifies to analyze *projection* .
- *schema* .*
  Specifies to analyze all projections in the named schema.
- [*schema* .] *table*
  Specifiesto analyze all projections of the named table.

If set to an empty string, Vertica analyzes all projections in the database to which the user has access.
For example, the following statement specifies to analyze all projections in schema private , and send the results to the file encodings.sql :

```
=> SELECT DESIGNER_DESIGN_PROJECTION_ENCODINGS ('mydb.private.*','encodings.sql');
```

destination

Specifies where to send output, one of the following:

- Empty string ('') writes the script to standard output.
- Pathname of a SQL output file. If you specify a file that does not exist, the function creates one. If you specify only a file name, Vertica creates it in the catalog directory. If the file already exists, the function silently overwrites its contents.

deploy

Boolean that specifies whether to deploy encoding changes.

Default: false

reanalyze-encodings

Boolean that specifies whether DESIGNER_DESIGN_PROJECTION_ENCODINGS analyzes encodings in a projection where all columns are already encoded:

- false : Analyzes no columns and generates no recommendations if all columns are encoded.
- true : Ignores existing encodings and generates recommendations.

Default: false

Privileges

Superuser, or DBDUSER with the following privileges:

- OWNER of all projections to analyze
- USAGE privilege on the schema for the specified projections

Examples

The following example requests that Database Designer analyze encodings of the table online_sales.call_center_dimension :

- The second parameter *destination* is set to an empty string, so the script is sent to standard output (shown truncated below).
- The last two parameters *deploy* and *reanalyze-encodings* are omitted, so Database Designer does not execute the script or reanalyze existing encodings:

```
=> SELECT DESIGNER_DESIGN_PROJECTION_ENCODINGS ('online_sales.call_center_dimension','');

            DESIGNER_DESIGN_PROJECTION_ENCODINGS
-------------------------------------------------------------------

CREATE PROJECTION call_center_dimension_DBD_1_seg_EncodingDesign /*+createtype(D)*/
(
 call_center_key ENCODING COMMONDELTA_COMP,
 cc_closed_date,
 cc_open_date,
 cc_name ENCODING ZSTD_HIGH_COMP,
 cc_class ENCODING ZSTD_HIGH_COMP,
 cc_employees,
 cc_hours ENCODING ZSTD_HIGH_COMP,
 cc_manager ENCODING ZSTD_HIGH_COMP,
 cc_address ENCODING ZSTD_HIGH_COMP,
 cc_city ENCODING ZSTD_COMP,
 cc_state ENCODING ZSTD_FAST_COMP,
 cc_region ENCODING ZSTD_HIGH_COMP
)
AS
 SELECT call_center_dimension.call_center_key,
        call_center_dimension.cc_closed_date,
        call_center_dimension.cc_open_date,
        call_center_dimension.cc_name,
        call_center_dimension.cc_class,
        call_center_dimension.cc_employees,
        call_center_dimension.cc_hours,
        call_center_dimension.cc_manager,
        call_center_dimension.cc_address,
        call_center_dimension.cc_city,
        call_center_dimension.cc_state,
        call_center_dimension.cc_region
 FROM online_sales.call_center_dimension
 ORDER BY call_center_dimension.call_center_key
SEGMENTED BY hash(call_center_dimension.call_center_key) ALL NODES KSAFE 1;

select refresh('online_sales.call_center_dimension');

select make_ahm_now();

DROP PROJECTION online_sales.call_center_dimension CASCADE;

ALTER PROJECTION online_sales.call_center_dimension_DBD_1_seg_EncodingDesign RENAME TO call_center_dimension;
(1 row)
```

See also

Running Database Designer programmatically

DESIGNER_DROP_ALL_DESIGNS

Removes all Database Designer-related schemas associated with the current user. Use this function to remove database objects after one or more Database Designer sessions complete execution.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_DROP_ALL_DESIGNS()
```

Parameters

None.

Privileges

Non-superuser: design creator

Examples

The following example removes all schema and their contents associated with the current user. DESIGNER_DROP_ALL_DESIGNS returns the number of designs dropped:

```
=> SELECT DESIGNER_DROP_ALL_DESIGNS();
 DESIGNER_DROP_ALL_DESIGNS
---------------------------
                         2
(1 row)
```

See also

- DESIGNER_CANCEL_POPULATE_DESIGN
- DESIGNER_DROP_DESIGN

DESIGNER_DROP_DESIGN

Removes the schema associated with the specified design and all its contents. Use DESIGNER_DROP_DESIGN after a Database Designer design or deployment completes successfully. You must also use it to drop a design before creating another one under the same name.

To drop all designs that you created, use DESIGNER_DROP_ALL_DESIGNS .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_DROP_DESIGN ( 'design-name' [, force-drop ] )
```

Parameters

design-name

Name of the design to drop.

force-drop

Boolean that overrides any dependencies that otherwise prevent Vertica from executing this function—for example, the design is in use or is currently being deployed. If you omit this parameter, Vertica sets it to false .

Privileges

Non-superuser: design creator

Examples

The following example deletes the Database Designer design VMART_DESIGN and all its contents:

```
=> SELECT DESIGNER_DROP_DESIGN ('VMART_DESIGN');
```

See also

Running Database Designer programmatically

DESIGNER_OUTPUT_ALL_DESIGN_PROJECTIONS

Displays the DDL statements that define the design projections to standard output.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_OUTPUT_ALL_DESIGN_PROJECTIONS ( 'design-name' )
```

Parameters

design-name

Name of the target design.

Privileges

Superuseror DBDUSER

Examples

The following example returns the design projection DDL statements for vmart_design :

```
=> SELECT DESIGNER_OUTPUT_ALL_DESIGN_PROJECTIONS('vmart_design');
CREATE PROJECTION customer_dimension_DBD_1_rep_VMART_DESIGN /*+createtype(D)*/
(
 customer_key ENCODING DELTAVAL,
 customer_type ENCODING AUTO,
 customer_name ENCODING AUTO,
 customer_gender ENCODING REL,
 title ENCODING AUTO,
 household_id ENCODING DELTAVAL,
 customer_address ENCODING AUTO,
 customer_city ENCODING AUTO,
 customer_state ENCODING AUTO,
 customer_region ENCODING AUTO,
 marital_status ENCODING AUTO,
 customer_age ENCODING DELTAVAL,
 number_of_children ENCODING BLOCKDICT_COMP,
 annual_income ENCODING DELTARANGE_COMP,
 occupation ENCODING AUTO,
 largest_bill_amount ENCODING DELTAVAL,
 store_membership_card ENCODING BLOCKDICT_COMP,
 customer_since ENCODING DELTAVAL,
 deal_stage ENCODING AUTO,
 deal_size ENCODING DELTARANGE_COMP,
 last_deal_update ENCODING DELTARANGE_COMP
)
AS
 SELECT customer_key,
        customer_type,
        customer_name,
        customer_gender,
        title,
        household_id,
        customer_address,
        customer_city,
        customer_state,
        customer_region,
        marital_status,
        customer_age,
        number_of_children,
        annual_income,
        occupation,
        largest_bill_amount,
        store_membership_card,
        customer_since,
        deal_stage,
        deal_size,
        last_deal_update
 FROM public.customer_dimension
 ORDER BY customer_gender,
          annual_income
UNSEGMENTED ALL NODES;
CREATE PROJECTION product_dimension_DBD_2_rep_VMART_DESIGN /*+createtype(D)*/
(
...
```

See also

DESIGNER_OUTPUT_DEPLOYMENT_SCRIPT

DESIGNER_OUTPUT_DEPLOYMENT_SCRIPT

Displays the deployment script for the specified design to standard output. If the design is already deployed, Vertica ignores this function.

To output only the CREATE PROJECTION commands in a design script, use DESIGNER_OUTPUT_ALL_DESIGN_PROJECTIONS .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_OUTPUT_DEPLOYMENT_SCRIPT ( 'design-name' )
```

Parameters

design-name

Name of the target design.

Privileges

Non-superuser: design creator

Examples

The following example displays the deployment script for VMART_DESIGN :

```
=> SELECT DESIGNER_OUTPUT_DEPLOYMENT_SCRIPT('VMART_DESIGN');
CREATE PROJECTION customer_dimension_DBD_1_rep_VMART_DESIGN /*+createtype(D)*/
...
CREATE PROJECTION product_dimension_DBD_2_rep_VMART_DESIGN /*+createtype(D)*/
...
select refresh('public.customer_dimension,
                public.product_dimension,
                public.promotion.dimension,
                public.date_dimension');
select make_ahm_now();
DROP PROJECTION public.customer_dimension_super CASCADE;
DROP PROJECTION public.product_dimension_super CASCADE;
...
```

See also

DESIGNER_OUTPUT_ALL_DESIGN_PROJECTIONS

DESIGNER_RESET_DESIGN

Discards all run-specific information of the previous Database Designer build or deployment of the specified design but keeps its configuration. You can make changes to the design as needed, for example, by changing parameters or adding additional tables and/or queries, before running the design again.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_RESET_DESIGN ( 'design-name' )
```

Parameters

design-name

Name of the design to reset.

Privileges

Non-superuser: design creator

Examples

The following example resets the Database Designer design VMART_DESIGN:

```
=> SELECT DESIGNER_RESET_DESIGN ('VMART_DESIGN');
```

DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY

Populates the design and creates the design and deployment scripts. DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY can also analyze statistics, deploy the design, and drop the workspace after the deployment.

The files output by this function have the permissions 666 or rw-rw-rw-, which allows any Linux user on the node to read or write to them. It is highly recommended that you keep the files in a secure directory.

> **Caution**
> DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY does not create a backup copy of the current design before deploying the new design. Before running this function, back up the existing schema design with EXPORT_CATALOG .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY (
  'design-name',
  'output-design-file',
  'output-deployment-file'
  [ , 'analyze-statistics']
  [ , 'deploy']
  [ , 'drop-design-workspace']
  [ , 'continue-after-error']
  )
```

Parameters

design-name

Name of the design to populate and deploy.

output-design-filename

Absolute path and name of the file to contain DDL statements that create design projections, on the local file system of the node where the session is connected, or another file system or object store that Vertica supports.

output-deployment-filename

Absolute path and name of the file to contain the deployment script, on the local file system of the node where the session is connected, or another file system or object store that Vertica supports.

analyze-statistics

Specifies whether to collect or refresh statistics for the tables before populating the design. If set to true, Vertica Invokes ANALYZE_STATISTICS . Accurate statistics help Database Designer optimize compression and query performance. However, updating statistics requires time and resources.

Default: false

deploy

Specifies whether to deploy the Database Designer design using the deployment script created by this function.

Default: true

drop-design-workspace

Specifies whether to drop the design workspace after the design is deployed.

Default: true

continue-after-error

Specifies whether DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY continues to run after an error occurs. By default, an error causes this function to terminate.

Default: false

Privileges

Non-superuser: design creator with WRITE privileges on storage locations of design and deployment scripts

Requirements

Before calling this function, you must:

- Create a design, a logical schema with tables.

- Associate tables with the design.
- Load queries to the design.
- Set design properties (K-safety level, mode, and policy).

Examples

The following example creates projections for and deploys the VMART_DESIGN design, and analyzes statistics about the design tables.

```
=> SELECT DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY (
  'VMART_DESIGN',
  '/tmp/examples/vmart_design_files/design_projections.sql',
  '/tmp/examples/vmart_design_files/design_deploy.sql',
  'true',
  'true',
  'false',
  'false'
  );
```

See also

Running Database Designer programmatically

DESIGNER_SET_DESIGN_KSAFETY

Sets K-safety for a comprehensive design and stores the K-safety value in the DESIGNS table. Database Designer ignores this function for incremental designs.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_SET_DESIGN_KSAFETY ( 'design-name' [, k-level ] )
```

Parameters

design-name

Name of the design for which you want to set the K-safety value, type VARCHAR.

k-level

An integer between 0 and 2 that specifies the level of K-safety for the target design. This value must be compatible with the number of nodes in the database cluster:

- *k-level* = 0 : ≥ 1 nodes
- *k-level* = 1 : ≥ 3 nodes
- *k-level* = 2 : ≥ 5 nodes

If you omit this parameter, Vertica sets K-safety for this design to 0 or 1, according to the number of nodes: 1 if the cluster contains ≥ 3 nodes, otherwise 0.

If you are a DBADMIN user and *k-level* differs from system K-safety, Vertica changes system K-safety as follows:

- If *k-level* is less than system K-safety, Vertica changes system K-safety to the lower level after the design is deployed.
- If *k-level* is greater than system K-safety and is valid for the database cluster, Vertica creates the required number of buddy projections for the tables in this design. If the design applies to all database tables, or all tables in the database have the required number of buddy projections, Database Designer changes system K-safety to *k-level* .
  If the design excludes some database tables and the number of their buddy projections is less than *k-level* , Database Designer leaves system K-safety unchanged. Instead, it returns a warning and indicates which tables need new buddy projections in order to adjust system K-safety.

If you are a DBDUSER, Vertica ignores this parameter.

Privileges

Non-superuser: design creator

Examples

The following example set K-safety for the VMART_DESIGN design to 1:

```
=> SELECT DESIGNER_SET_DESIGN_KSAFETY('VMART_DESIGN', 1);
```

See also

Running Database Designer programmatically

DESIGNER_SET_DESIGN_TYPE

Specifies whether Database Designer creates a comprehensive or incremental design. DESIGNER_SET_DESIGN_TYPE stores the design mode in the DESIGNS table.

Important

If you do not explicitly set a design mode with this function, Database Designer creates a comprehensive design.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_SET_DESIGN_TYPE ( 'design-name', 'mode' )
```

Parameters

design-name

Name of the target design.

mode

Name of the mode that Database Designer should use when designing the database, one of the following:

- COMPREHENSIVE : Creates an initial or replacement design for all tables in the specified schemas. You typically create a comprehensive design for a new database.
- INCREMENTAL : Modifies an existing design with additional projection that are optimized for new or modified queries.

> **Note**
> Incremental designs always inherit the K-safety value of the database.

For more information, see Design Types .

Privileges

Non-superuser: design creator

Examples

The following examples show the two design mode options for the VMART_DESIGN design:

```
=> SELECT DESIGNER_SET_DESIGN_TYPE(
   'VMART_DESIGN',
   'COMPREHENSIVE');
DESIGNER_SET_DESIGN_TYPE
--------------------------
                        0
(1 row)
=> SELECT DESIGNER_SET_DESIGN_TYPE(
   'VMART_DESIGN',
   'INCREMENTAL');
 DESIGNER_SET_DESIGN_TYPE
--------------------------
                        0
(1 row)
```

See also

Running Database Designer programmatically

DESIGNER_SET_OPTIMIZATION_OBJECTIVE

Valid only for comprehensive database designs, specifies the optimization objective Database Designer uses. Database Designer ignores this function for incremental designs.

DESIGNER_SET_OPTIMIZATION_OBJECTIVE stores the optimization objective in the DESIGNS table.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_SET_OPTIMIZATION_OBJECTIVE ( 'design-name', 'policy' )
```

Parameters

design-name

Name of the target design.

policy

Specifies the design's optimization policy, one of the following:

- QUERY : Optimize for query performance. This can result in a larger database storage footprint because additional projections might be created.
- LOAD : Optimize for load performance so database size is minimized. This can result in slower query performance.
- BALANCED : Balance the design between query performance and database size.

Privileges

Non-superuser: design creator

Examples

The following example sets the optimization objective option for the VMART_DESIGN design: to QUERY :

```
=> SELECT DESIGNER_SET_OPTIMIZATION_OBJECTIVE(  'VMART_DESIGN', 'QUERY');
 DESIGNER_SET_OPTIMIZATION_OBJECTIVE
-------------------------------------
                                   0
(1 row)
```

See also

Running Database Designer programmatically

DESIGNER_SET_PROPOSE_UNSEGMENTED_PROJECTIONS

Specifies whether a design can include unsegmented projections . Vertica ignores this function on a one-node cluster, where all projections must be unsegmented.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_SET_PROPOSE_UNSEGMENTED_PROJECTIONS ( 'design-name', unsegmented )
```

Parameters

design-name

Name of the target design.

unsegmented

Boolean that specifies whether Database Designer can propose unsegmented projections for tables in this design. When you create a design, the propose_unsegmented_projections value in system table DESIGNS for this design is set to true. If DESIGNER_SET_PROPOSE_UNSEGMENTED_PROJECTIONS sets this value to false, Database Designer proposes only segmented projections.

Privileges

Non-superuser: design creator

Examples

The following example specifies that Database Designer can propose only segmented projections for tables in the design VMART_DESIGN :

```
=> SELECT DESIGNER_SET_PROPOSE_UNSEGMENTED_PROJECTIONS('VMART_DESIGN', false);
```

See also

Running Database Designer programmatically

DESIGNER_SINGLE_RUN

Evaluates all queries that completed execution within the specified timespan, and returns with a design that is ready for deployment. This design includes projections that are recommended for optimizing the evaluated queries. Unless you redirect output, DESIGNER_SINGLE_RUN returns the design to stdout.

> **Tip**
> Before running DESIGNER_SINGLE_RUN, collect statistics on the queried data by calling ANALYZE_STATISTICS and ANALYZE_STATISTICS_PARTITION .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_SINGLE_RUN ('interval')
```

interval

Specifies an interval of time that precedes the meta-function call. Database Designer evaluates all queries that ran to completion over the specified interval.

Privileges

Superuser or DBUSER

Examples

```
---------------------------------------------------------------------------
-- SSBM dataset test
---------------------------------------------------------------------------
-- create ssbm schema
\! $TARGET/bin/vsql -f 'sql/SSBM/SSBM_schema.sql' > /dev/null 2>&1
\! $TARGET/bin/vsql -f 'sql/SSBM/SSBM_constraints.sql' > /dev/null 2>&1
\! $TARGET/bin/vsql -f 'sql/SSBM/SSBM_funcdeps.sql' > /dev/null 2>&1

-- run these queries
\! $TARGET/bin/vsql -f 'sql/SSBM/SSBM_queries.sql' > /dev/null 2>&1
-- Run single API
select designer_single_run('1 minute');

...

designer_single_run
----------------------------------------------------------------------------
```

```
 p_mfgr ENCODING AUTO,
 p_category ENCODING AUTO,
 p_brand1 ENCODING AUTO,
 p_color ENCODING AUTO,
 p_type ENCODING AUTO,
 p_size ENCODING AUTO,
 p_container ENCODING AUTO
)
AS
 SELECT p_partkey,
        p_name,
        p_mfgr,
        p_category,
        p_brand1,
        p_color,
        p_type,
        p_size,
        p_container
 FROM public.part
 ORDER BY p_partkey
UNSEGMENTED ALL NODES;

CREATE PROJECTION public.supplier_DBD_2_rep_SingleDesign /*+createtype(D)*/
(
 s_suppkey ENCODING AUTO,
 s_name ENCODING AUTO,
 s_address ENCODING AUTO,
 s_city ENCODING AUTO,
 s_nation ENCODING AUTO,
 s_region ENCODING AUTO,
 s_phone ENCODING AUTO
)
AS
 SELECT s_suppkey,
        s_name,
        s_address,
        s_city,
        s_nation,
        s_region,
        s_phone
 FROM public.supplier
 ORDER BY s_suppkey
UNSEGMENTED ALL NODES;

CREATE PROJECTION public.customer_DBD_3_rep_SingleDesign /*+createtype(D)*/
(
 c_custkey ENCODING AUTO,
 c_name ENCODING AUTO,
 c_address ENCODING AUTO,
 c_city ENCODING AUTO,
 c_nation ENCODING AUTO,
 c_region ENCODING AUTO,
 c_phone ENCODING AUTO,
 c_mktsegment ENCODING AUTO
)
AS
 SELECT c_custkey,
        c_name,
        c_address,
        c_city,
        c_nation,
```

```
       c_region,
       c_phone,
       c_mktsegment
 FROM public.customer
 ORDER BY c_custkey
UNSEGMENTED ALL NODES;

CREATE PROJECTION public.dwdate_DBD_4_rep_SingleDesign /*+createtype(D)*/
(
 d_datekey ENCODING AUTO,
 d_date ENCODING AUTO,
 d_dayofweek ENCODING AUTO,
 d_month ENCODING AUTO,
 d_year ENCODING AUTO,
 d_yearmonthnum ENCODING AUTO,
 d_yearmonth ENCODING AUTO,
 d_daynuminweek ENCODING AUTO,
 d_daynuminmonth ENCODING AUTO,
 d_daynuminyear ENCODING AUTO,
 d_monthnuminyear ENCODING AUTO,
 d_weeknuminyear ENCODING AUTO,
 d_sellingseason ENCODING AUTO,
 d_lastdayinweekfl ENCODING AUTO,
 d_lastdayinmonthfl ENCODING AUTO,
 d_holidayfl ENCODING AUTO,
 d_weekdayfl ENCODING AUTO
)
AS
 SELECT d_datekey,
       d_date,
       d_dayofweek,
       d_month,
       d_year,
       d_yearmonthnum,
       d_yearmonth,
       d_daynuminweek,
       d_daynuminmonth,
       d_daynuminyear,
       d_monthnuminyear,
       d_weeknuminyear,
       d_sellingseason,
       d_lastdayinweekfl,
       d_lastdayinmonthfl,
       d_holidayfl,
       d_weekdayfl
 FROM public.dwdate
 ORDER BY d_datekey
UNSEGMENTED ALL NODES;

CREATE PROJECTION public.lineorder_DBD_5_rep_SingleDesign /*+createtype(D)*/
(
 lo_orderkey ENCODING AUTO,
 lo_linenumber ENCODING AUTO,
 lo_custkey ENCODING AUTO,
 lo_partkey ENCODING AUTO,
 lo_suppkey ENCODING AUTO,
 lo_orderdate ENCODING AUTO,
 lo_orderpriority ENCODING AUTO,
 lo_shippriority ENCODING AUTO,
 lo_quantity ENCODING AUTO,
 lo_extendedprice ENCODING AUTO,
```

```
 lo_ordertotalprice ENCODING AUTO,
 lo_discount ENCODING AUTO,
 lo_revenue ENCODING AUTO,
 lo_supplycost ENCODING AUTO,
 lo_tax ENCODING AUTO,
 lo_commitdate ENCODING AUTO,
 lo_shipmode ENCODING AUTO
)
AS
 SELECT lo_orderkey,
        lo_linenumber,
        lo_custkey,
        lo_partkey,
        lo_suppkey,
        lo_orderdate,
        lo_orderpriority,
        lo_shippriority,
        lo_quantity,
        lo_extendedprice,
        lo_ordertotalprice,
        lo_discount,
        lo_revenue,
        lo_supplycost,
        lo_tax,
        lo_commitdate,
        lo_shipmode
 FROM public.lineorder
 ORDER BY lo_suppkey
UNSEGMENTED ALL NODES;

(1 row)
```

DESIGNER_WAIT_FOR_DESIGN

Waits for completion of operations that are populating and deploying the design. Ctrl+C cancels this operation and returns control to the user.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DESIGNER_WAIT_FOR_DESIGN ( 'design-name' )
```

Parameters

design-name

Name of the running design.

Privileges

Superuser, or DBDUSER with USAGE privilege on the design schema

Examples

The following example requests to wait for the currently running design of VMART_DESIGN to complete:

```
=> SELECT DESIGNER_WAIT_FOR_DESIGN ('VMART_DESIGN');
```

See also

- DESIGNER_CANCEL_POPULATE_DESIGN
- DESIGNER_DROP_ALL_DESIGNS
- DESIGNER_DROP_DESIGN

Directed queries functions

The following meta-functions let you batch export query plans as directed queries from one Vertica database, and import those directed queries to another database.

In this section

- EXPORT_DIRECTED_QUERIES
- IMPORT_DIRECTED_QUERIES
- SAVE_PLANS

EXPORT_DIRECTED_QUERIES

Generates SQL for creating directed queries from a set of input queries.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
EXPORT_DIRECTED_QUERIES('input-file', '[output-file]')
```

Arguments

input-file

A SQL file that contains one or more input queries. See Input Format below for details on format requirements.

output-file

Specifies where to write the generated SQL for creating directed queries. If *output-file* already exists, EXPORT_DIRECTED_QUERIES returns with an error. If you supply an empty string, Vertica writes the SQL to standard output. See Output Format below for details.

Privileges

Superuser

Input format

The input file that you supply to EXPORT_DIRECTED_QUERIES contains one or more input queries. For each input query, you can optionally specify two fields that are used in the generated directed query:

- DirQueryName provides the directed query's unique identifier, a string that conforms to conventions described in Identifiers .
- DirQueryComment specifies a quote-delimited string, up to 128 characters.

You format each input query as follows:

```
--DirQueryName=query-name
--DirQueryComment='comment'
input-query
```

Output format

EXPORT_DIRECTED_QUERIES generates SQL for creating directed queries, and writes the SQL to the specified file or to standard output. In both cases, output conforms to the following format:

```
/* Query: directed-query-name */
/* Comment: directed-query-comment */
SAVE QUERY input-query;
CREATE DIRECTED QUERY CUSTOM 'directed-query-name'
COMMENT 'directed-query-comment'
OPTVER 'vertica-release-num'
PSDATE 'timestamp'
annotated-query
```

If a given input query omits DirQueryName and DirQueryComment fields, EXPORT_DIRECTED_QUERIES automatically generates the following output:

- /* Query: Autoname: *timestamp* . *n* */ , where *n* is a zero-based integer index that ensures uniqueness among auto-generated names with the same timestamp.
- /* Comment: Optimizer-generated directed query */

Error handling

If any errors or warnings occur during EXPORT_DIRECTED_QUERIES execution, it returns with a message like this one:

```
1 queries successfully exported.
1 warning message was generated.
Queries exported to /home/dbadmin/outputQueries.
See error report, /home/dbadmin/outputQueries.err for details.
```

EXPORT_DIRECTED_QUERIES writes all errors and warnings to a file that it creates on the same path as the output file, and uses the output file's base name.

For example:

```
-------------------------------------------------------------------------------------------------
WARNING: Name field not supplied. Using auto-generated name: 'Autoname:2016-04-25 15:03:32.115317.0'
Input Query: SELECT employee_dimension.employee_first_name, employee_dimension.employee_last_name, employee_dimension.job_title FROM
public.employee_dimension WHERE (employee_dimension.employee_city = 'Boston'::varchar(6)) ORDER BY employee_dimension.job_title;
END WARNING
```

Examples

See Exporting directed queries .

See also

- Batch query plan export
- IMPORT_DIRECTED_QUERIES

IMPORT_DIRECTED_QUERIES

Imports to the database catalog directed queries from a SQL file that was generated by EXPORT_DIRECTED_QUERIES . If no directed queries are specified, Vertica lists all directed queries in the SQL file.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
IMPORT_DIRECTED_QUERIES( 'export-file'[, 'directed-query-name'[,...] ] )
```

Arguments

export-file

A SQL file generated by EXPORT_DIRECTED_QUERIES . When you run this file, Vertica creates the specified directed queries in the current database catalog.

directed-query-name

The name of a directed query that is defined in *export-file* . You can specify multiple comma-delimited directed query names.

If you omit this parameter, Vertica lists the names of all directed queries in *export-file* .

Privileges

Superuser

Examples

See Importing directed queries .

See also

Batch query plan export

SAVE_PLANS

Creates optimizer-generated directed queries from the most frequently executed queries, up to the maximum specified. You can also limit the scope of SAVE_PLANS to queries only issued after a specified date.

As SAVE_PLANS iterates over past queries, it tests them against various restrictions. In general, directed queries support only SELECT statements as input. Within this broad requirement, input queries are subject to other restrictions . After qualifying all candidate input queries, SAVE_PLANS operates as follows:

1. Calls CREATE DIRECTED QUERY OPTIMIZER on all qualified input queries, which creates a directed query for each unique input query.
2. Saves metadata on the new set of directed queries to the system table DIRECTED_QUERIES , where all directed queries of that set share the same integer identifier.

All directed queries created by SAVE_PLANS are initially inactive. You can activate them individually; you can also use SAVE_PLANS_VERSION

identifiers to activate , deactivate , and drop one or more sets of directed queries.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SAVE_PLANS( query-budget [, since-date] [, drop-old-plans[, 'comment']] )
```

Arguments

***query-budget** ***

Maximum number of input queries to save as directed queries, an integer between 1 and 100, inclusive.

since-date

The earliest timestamp of input queries to save as directed queries.

drop-old-plans

Boolean, specifies whether to drop all directed queries generated by earlier SAVE_PLANS invocations. Only directed queries that were generated by the current Vertica version are dropped; directed queries generated by earlier Vertica versions are untouched. To drop older directed queries, use DROP DIRECTED QUERY .

comment

String comment that is attached to all plans saved with this function call.

Privileges

Superuser

Generated metadata

For each set of directed queries that SAVE_PLANS creates, Vertica updates the system table DIRECTED_QUERIES with metadata on each directed query in the set:

Column name	SAVE_PLANS-generated data
QUERY_NAME	Concatenated from the following strings: `save_plans_query-label_query-number_save-plans-version` where: • *query-label* is a LABEL hint embedded in the input query associated with this directed query. If theinput query contains no label, then this string is set to nolabel . • *query-number* is an integer in a continuous sequence between 0 and *budget-query* , which uniquely identifies this directed query from others in the same SAVE_PLANS-generated set. • *[save-plans-version](/sql-reference/system-tables/v-catalog-schema/directed-queries.html#SAVE_PLANS_VERSION)* identifies the set of directed queries to which this directed query belongs.
SAVE_PLANS_VERSION	Identifies a set of directed queries that were generated by the same call to SAVE_PLANS. All directed queries of the set share the same SAVE_PLANS_VERSION integer, which increments by 1 the previous highest SAVE_PLANS_VERSION setting. Use this identifier to activate , deactivate , and drop a set of directed queries.
USERNAME	User who invoked SAVE_PLANS to create this set of directed queries.
SINCE_DATE	The *since-date* timestamp supplied to SAVE_PLANS, which specified the earliest timestamp of input queries to evaluate as directed query candidates.
DIGEST	Hash of saved query plan data, used by the optimizer to map identical input queries to the same active directed query.

Examples

See Bulk-Creation of Directed Queries .

Error-handling functions

Error-handling functions take a string and return the string when the query is executed.

In this section

- THROW_ERROR

THROW_ERROR

Returns a user-defined error message.

In a multi-node cluster, race conditions might cause the order of error messages to differ.

Behavior type

Immutable

Syntax

```
THROW_ERROR ( message )
```

Parameters

message

The VARCHAR string to return.

Examples

Return an error message when a CASE statement is met:

```
=> CREATE TABLE pitcher_err (some_text varchar);
CREATE TABLE
=> COPY pitcher_err FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> big foo value
>> bigger foo other value
>> bar another foo value
>> \.
=> SELECT (CASE WHEN true THEN THROW_ERROR('Failure!!!') ELSE some_text END) FROM pitcher_err;
ERROR 7137:  USER GENERATED ERROR: Failure!!!
```

Return an error message when a CASE statement using REGEXP_LIKE is met:

```
=> SELECT (CASE WHEN REGEXP_LIKE(some_text, 'other') THEN THROW_ERROR('Failure at "' || some_text || '"') END) FROM pitcher_err;
ERROR 4566:  USER GENERATED ERROR: Failure at "bar another foo value"
```

Flex functions

This section contains helper functions for use in working with flex tables and flexible columns for complex types. You can use these functions with flex tables, their associated *flex_table* _keys tables and *flex_table* _view views, and flexible columns in external tables. These functions do not apply to other tables.

For more information about flex tables, see Flex tables . For more information about flexible columns for complex types, see Flexible complex types .

Flex functions allow you to manage and query flex tables. You can also use the map functions to query flexible complex-type columns in non-flex tables.

In this section

- Flex data functions
- Flex extractor functions
- Flex map functions

Flex data functions

The flex table data helper functions supply information you need to directly query data in flex tables. After you compute keys and create views from the raw data, you can use field names directly in queries instead of using map functions to extract data. The fata functions are:

- COMPUTE_FLEXTABLE_KEYS : Computes map keys from the map data in a flex table and populates a keys table with the results. Use this function before building a view.
- BUILD_FLEXTABLE_VIEW : Uses the keys in a table to create a view definition for the source table. Use this function after computing flex table keys.
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW : Performs both of the preceding functions in one call.

- MATERIALIZE_FLEXTABLE_COLUMNS : Materializes a specified number of columns.
- RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW : Replaces the flextable_data_keys table and the flextable_data_view view, linking both the keys table and the view to the parent flex table.

Flex table dependencies

Each flex table has two dependent objects, a keys table and a view. While both objects are dependent on their parent table, you can drop either object independently. Dropping the parent table removes both dependents, without a CASCADE option.

Associating flex tables and views

The helper functions automatically use the dependent table and view if they are internally linked with the parent table. You create both when you create the flex table. You can drop either the keys table or the view and re-create objects of the same name. However, if you do so, the new objects are not internally linked with the parent flex table.

In this case, you can restore the internal links of these objects to the parent table. To do so, drop the keys table and the view before calling the RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW function. Calling this function re-creates the keys table and view.

The remaining helper functions perform the tasks described in this section.

In this section

- BUILD_FLEXTABLE_VIEW
- COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW
- MATERIALIZE_FLEXTABLE_COLUMNS
- RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW

BUILD_FLEXTABLE_VIEW

Creates, or re-creates, a view for a default or user-defined keys table, ignoring any empty keys.

> **Note**
> If the length of a key exceeds 65,000, Vertica truncates the key.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
BUILD_FLEXTABLE_VIEW ('[[database.]schema.]flex-table'
   [ [,'view-name'] [,'user-keys-table'] ])
```

Arguments

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

flex-table

The flex table name. By default, this function builds or rebuilds a view for the input table with the current contents of the associated flex_table_keys table.

view-name

A custom view name. Use this option to build a new view for *flex-table* with the name you specify.

user-keys-table

Name of a keys table from which to create the view. Use this option if you created a custom keys table from the flex table map data, rather than from the default flex_table_keys table. The function builds a view from the keys in user_keys , rather than from flex_table_keys .

Examples

The following examples show how to call BUILD_FLEXTABLE_VIEW with 1, 2, or 3 arguments.

To create, or re-create, a default view:

1. Call the function with an input flex table:

```
=> SELECT BUILD_FLEXTABLE_VIEW('darkdata');
                build_flextable_view
-----------------------------------------------------------
 The view public.darkdata_view is ready for querying
(1 row)
```

The function creates a view with the default name (darkdata_view) from the darkdata _keys table.

2. Query a key name from the new or updated view:

```
=> SELECT "user.id" FROM darkdata_view;
 user.id
-----------
 340857907
 727774963
 390498773
 288187825
 164464905
 125434448
 601328899
 352494946
(12 rows)
```

To create, or re-create, a view with a custom name:

1. Call the function with two arguments, an input flex table, darkdata , and the name of the view to create, dd_view :

```
=> SELECT BUILD_FLEXTABLE_VIEW('darkdata', 'dd_view');
          build_flextable_view
-----------------------------------------------
 The view public.dd_view is ready for querying
(1 row)
```

2. Query a key name (user.lang) from the new or updated view (dd_view):

```
=> SELECT "user.lang" FROM dd_view;
 user.lang
-----------
 tr
 en
 es
 en
 en
 it
 es
 en
(12 rows)
```

To create a view from a custom keys table with BUILD_FLEXTABLE_VIEW , the custom table must have the same schema and table definition as the default table (darkdata_keys). Create a custom keys table, using any of these three approaches:

- Create a columnar table with all keys from the default keys table for a flex table (darkdata_keys):

```
=> CREATE TABLE new_darkdata_keys AS SELECT * FROMdarkdata_keys;
CREATE TABLE
```

- Create a columnar table without content (LIMIT 0) from the default keys table for a flex table (darkdata_keys):

```
=> CREATE TABLE new_darkdata_keys AS SELECT * FROM darkdata_keys LIMIT 0;
CREATE TABLE
kdb=> SELECT * FROM new_darkdata_keys;
 key_name | frequency | data_type_guess
----------+-----------+-----------------
(0 rows)
```

- Create a columnar table without content (LIMIT 0) from the default keys table, and insert two values (' user.lang ', ' user.name ') into the key_name column:

```
=> CREATE TABLE dd_keys AS SELECT * FROM darkdata_keys limit 0;
CREATE TABLE
=> INSERT INTO dd_keys (key_name) values ('user.lang');
 OUTPUT
--------
      1
(1 row)
=> INSERT INTO dd_keys (key_name) values ('user.name');
 OUTPUT
--------
      1
(1 row)
=> SELECT * FROM dd_keys;
 key_name  | frequency | data_type_guess
-----------+-----------+-----------------
 user.lang |           |
 user.name |           |
(2 rows)
```

After creating a custom keys table, call BUILD_FLEXTABLE_VIEW with all arguments (an input flex table, the new view name, the custom keys table):

```
=> SELECT BUILD_FLEXTABLE_VIEW('darkdata', 'dd_view', 'dd_keys');
           build_flextable_view
-----------------------------------------------
 The view public.dd_view is ready for querying
(1 row)
```

Query the new view:

```
=> SELECT * FROM dd_view;
```

See also

- COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW
- MATERIALIZE_FLEXTABLE_COLUMNS
- RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW

COMPUTE_FLEXTABLE_KEYS

Computes the virtual columns (keys and values) from flex table VMap data. Use this function to compute keys without creating an associated table view. To also build a view, use COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
COMPUTE_FLEXTABLE_KEYS ('[[database.]schema.]flex-table')
```

Arguments

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

flex-table

Name of the flex table.

Output

The function stores its results in a table named *flex-table* _keys . The table has the following columns:

Column	Description
KEY_NAME	The name of the virtual column (key). Keys larger than 65,000 bytes are truncated.

FREQUENCY	The number of times the key occurs in the VMap.
DATA_TYPE_GUESS	Estimate of the data type for the key based on the non-null values found in the VMap. The function determines the type of each non-string value, depending on the length of the key, and whether the key includes nested maps. If the EnableBetterFlexTypeGuessing configuration parameter is 0 (OFF), this function instead treats all flex table keys as string types ([LONG] VARCHAR or [LONG] VARBINARY).

COMPUTE_FLEXTABLE_KEYS sets the column width for keys to the length of the largest value for each key multiplied by the FlexTableDataTypeGuessMultiplier factor.

Examples

In the following example, JSON data with consistent fields has been loaded into a flex table. Had the data been more varied, you would see different numbers of occurrences in the keys table:

```
=> SELECT COMPUTE_FLEXTABLE_KEYS('reviews_flex');
          COMPUTE_FLEXTABLE_KEYS
-----------------------------------------------
 Please see public.reviews_flex_keys for updated keys
(1 row)

SELECT * FROM reviews_flex_keys;
  key_name   | frequency | data_type_guess
-------------+-----------+-----------------
 user_id     |      1000 | Varchar(44)
 useful      |      1000 | Integer
 text        |      1000 | Varchar(9878)
 stars       |      1000 | Numeric(5,2)
 review_id   |      1000 | Varchar(44)
 funny       |      1000 | Integer
 date        |      1000 | Timestamp
 cool        |      1000 | Integer
 business_id |      1000 | Varchar(44)
(9 rows)
```

See also

- BUILD_FLEXTABLE_VIEW
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW
- MATERIALIZE_FLEXTABLE_COLUMNS
- RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW

COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW

Combines the functionality of BUILD_FLEXTABLE_VIEW and COMPUTE_FLEXTABLE_KEYS to compute virtual columns (keys) from the VMap data of a flex table and construct a view. Creating a view with this function ignores empty keys. If you do not need to perform both operations together, use one of the single-operation functions instead.

> **Note**
> If the length of a key exceeds 65,000, Vertica truncates the key.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW ('flex-table')
```

Arguments

flex-table

Name of a flex table

Examples

This example shows how to call the function for the darkdata flex table.

```
=> SELECT COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW('darkdata');
             compute_flextable_keys_and_build_view
-------------------------------------------------------------------------
 Please see public.darkdata_keys for updated keys
The view public.darkdata_view is ready for querying
(1 row)
```

See also

- BUILD_FLEXTABLE_VIEW
- COMPUTE_FLEXTABLE_KEYS
- MATERIALIZE_FLEXTABLE_COLUMNS
- RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW

MATERIALIZE_FLEXTABLE_COLUMNS

Materializes virtual columns listed as *key_names* in the *flextable_keys* table you compute using either COMPUTE_FLEXTABLE_KEYS or COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW .

> **Note**
> Each column that you materialize with this function counts against the data storage limit of your license. To check your Vertica license compliance, call the AUDIT() or AUDIT_FLEX() functions.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
MATERIALIZE_FLEXTABLE_COLUMNS ('[[database.]schema.]flex-table' [, n-columns [, keys-table-name] ])
```

Arguments

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

flex-table

The name of the flex table with columns to materialize. The function:

- Skips any columns already materialized
- Ignores any empty keys

n-columns

The number of columns to materialize, up to 9800. The function attempts to materialize the number of columns from the keys table, skipping any columns already materialized. It orders the materialized results by frequency, descending. If not specified, the default is a maximum of 50 columns.

keys-table-name

The name of a keys from which to materialize columns. The function:

- Materializes *n-columns* columns from the keys table
- Skips any columns already materialized
- Orders the materialized results by frequency, descending

Examples

The following example shows how to call MATERIALIZE_FLEXTABLE_COLUMNS to materialize columns. First, load a sample file of tweets (tweets_10000.json) into the flex table twitter_r . After loading data and computing keys for the sample flex table, call MATERIALIZE_FLEXTABLE_COLUMNS to materialize the first four columns:

```
=> COPY twitter_r FROM '/home/release/KData/tweets_10000.json' parser fjsonparser();
 Rows Loaded
-------------
       10000
(1 row)

=> SELECT compute_flextable_keys ('twitter_r');
              compute_flextable_keys
-----------------------------------------------------
 Please see public.twitter_r_keys for updated keys
(1 row)

=> SELECT MATERIALIZE_FLEXTABLE_COLUMNS('twitter_r', 4);
    MATERIALIZE_FLEXTABLE_COLUMNS
-------------------------------------------------------------------------------
 The following columns were added to the table public.twitter_r:
       contributors
       entities.hashtags
       entities.urls
For more details, run the following query:
SELECT * FROM v_catalog.materialize_flextable_columns_results WHERE table_schema = 'public' and table_name = 'twitter_r';

(1 row)
```

The last message in the example recommends querying the MATERIALIZE_FLEXTABLE_COLUMNS_RESULTS system table for the results of materializing the columns, as shown:

```
=> SELECT * FROM v_catalog.materialize_flextable_columns_results WHERE table_schema = 'public' and table_name = 'twitter_r';
table_id          | table_schema | table_name |       creation_time           |     key_name      | status |    message
------------------+--------------+------------+-------------------------------+-------------------+--------+---------------------
 45035996273733172 | public       | twitter_r  | 2013-11-20 17:00:27.945484-05| contributors      | ADDED  | Added successfully
 45035996273733172 | public       | twitter_r  | 2013-11-20 17:00:27.94551-05 | entities.hashtags | ADDED  | Added successfully
 45035996273733172 | public       | twitter_r  | 2013-11-20 17:00:27.945519-05| entities.urls     | ADDED  | Added successfully
 45035996273733172 | public       | twitter_r  | 2013-11-20 17:00:27.945532-05| created_at        | EXISTS | Column of same name already
(4 rows)
```

See also

- BUILD_FLEXTABLE_VIEW
- COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW
- RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW

RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW

Restores the keys table and the view. The function also links the keys table with its associated flex table, in cases where either table is dropped. The function also indicates whether it restored one or both objects.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW ('flex-table')
```

Arguments

flex-table

Name of a flex table

Examples

This example shows how to invoke this function with an existing flex table, restoring both the keys table and view:

```
=> SELECT RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW('darkdata');
                RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW
-------------------------------------------------------------------------------
The keys table public.darkdata_keys was restored successfully.
The view public.darkdata_view was restored successfully.
(1 row)
```

This example illustrates that the function restored darkdata_view , but that darkdata_keys did not need restoring:

```
=> SELECT RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW('darkdata');
                RESTORE_FLEXTABLE_DEFAULT_KEYS_TABLE_AND_VIEW
-------------------------------------------------------------------------------
 The keys table public.darkdata_keys already exists and is linked to darkdata.
 The view public.darkdata_view was restored successfully.
(1 row)
```

After restoring the keys table, there is no content. To populate the flex keys, call the COMPUTE_FLEXTABLE_KEYS function.

```
=> SELECT * FROM darkdata_keys;
 key_name | frequency | data_type_guess
----------+-----------+-----------------
(0 rows)
```

See also

- BUILD_FLEXTABLE_VIEW
- COMPUTE_FLEXTABLE_KEYS
- COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW
- MATERIALIZE_FLEXTABLE_COLUMNS

Flex extractor functions

The flex extractor scalar functions process polystructured data. Each function accepts input data that is any of:

- Existing database content
- A table
- Returned from an expression
- Entered directly

These functions do not parse data from an external file source. All functions return a single VMap value. The extractor functions can return data with NULL-specified columns.

In this section

- MAPDELIMITEDEXTRACTOR
- MAPJSONEXTRACTOR
- MAPREGEXEXTRACTOR

MAPDELIMITEDEXTRACTOR

Extracts data with a delimiter character and other optional arguments, returning a single VMap value.

Syntax

```
MAPDELIMITEDEXTRACTOR (record-value [ USING PARAMETERS param=value[,...] ])
```

Arguments

record-value

String containing a JSON or delimited format record on which to apply the expression.

Parameters

delimiter

Single delimiter character.

Default: |

header_names

Delimiter-separated list of column header names.

Default: ucol *n* , where *n* is the column offset number, starting with 0 for the first column.

trim

Boolean, trim white space from header names and field values.

Default: true

treat_empty_val_as_null

Boolean, set empty fields to NULL rather than an empty string ('').

Default: true

Examples

These examples use a short set of delimited data:

```
Name|CITY|New city|State|zip
Tom|BOSTON|boston|MA|01
Eric|Burlington|BURLINGTON|MA|02
Jamie|cambridge|CAMBRIDGE|MA|08
```

To begin, save this data as delim.dat .

1. Create a flex table, dflex :

```
=> CREATE FLEX TABLE dflex();
CREATE TABLE
```

2. Use COPY to load the delim.dat file. Use the flex tables fdelimitedparser with the header='false' option:

```
=> COPY dflex FROM '/home/release/kmm/flextables/delim.dat' parser fdelimitedparser(header='false');
 Rows Loaded
-------------
           4
(1 row)
```

3. Create a columnar table, dtab , with an identity id column, a delim column, and a vmap column to hold a VMap:

```
=> CREATE TABLE dtab (id IDENTITY(1,1), delim varchar(128), vmap long varbinary(512));
CREATE TABLE
```

4. Use COPY to load the delim.dat file into the dtab table. MAPDELIMITEDEXTRACTOR uses the header_names parameter to specify a header row for the sample data, along with delimiter '!' :

```
=> COPY dtab(delim, vmap AS MAPDELIMITEDEXTRACTOR (delim
   USING PARAMETERS header_names='Name|CITY|New City|State|Zip')) FROM '/home/dbadmin/data/delim.dat'
DELIMITER '!';

 Rows Loaded
-------------
           4
(1 row)
```

5. Use MAPTOSTRING for the flex table dflex to view the __raw__ column contents. Notice the default header names in use (ucol0 – ucol4), since you specified header='false' when you loaded the flex table:

```
=> SELECT MAPTOSTRING(__raw__) FROM dflex limit 10;
                                maptostring
-------------------------------------------------------------------------------------
 {
   "ucol0" : "Jamie",
   "ucol1" : "cambridge",
   "ucol2" : "CAMBRIDGE",
   "ucol3" : "MA",
   "ucol4" : "08"
}

 {
   "ucol0" : "Name",
   "ucol1" : "CITY",
   "ucol2" : "New city",
   "ucol3" : "State",
   "ucol4" : "zip"
}

 {
   "ucol0" : "Tom",
   "ucol1" : "BOSTON",
   "ucol2" : "boston",
   "ucol3" : "MA",
   "ucol4" : "01"
}

 {
   "ucol0" : "Eric",
   "ucol1" : "Burlington",
   "ucol2" : "BURLINGTON",
   "ucol3" : "MA",
   "ucol4" : "02"
}

(4 rows)
```

6. Use MAPTOSTRING again, this time with the dtab table's vmap column. Compare the results of this output to those for the flex table. Note that MAPTOSTRING returns the header_name parameter values you specified when you loaded the data:

```
=> SELECT MAPTOSTRING(vmap) FROM dtab;
                                                maptostring
-----------------------------------------------------------------------------------------------------------------------
 {
   "CITY" : "CITY",
   "Name" : "Name",
   "New City" : "New city",
   "State" : "State",
   "Zip" : "zip"
}

 {
   "CITY" : "BOSTON",
   "Name" : "Tom",
   "New City" : "boston",
   "State" : "MA",
   "Zip" : "02121"
}

 {
   "CITY" : "Burlington",
   "Name" : "Eric",
   "New City" : "BURLINGTON",
   "State" : "MA",
   "Zip" : "02482"
}

 {
   "CITY" : "cambridge",
   "Name" : "Jamie",
   "New City" : "CAMBRIDGE",
   "State" : "MA",
   "Zip" : "02811"
}

(4 rows)
```

7. Query the delim column to view the contents differently:

```
=> SELECT delim FROM dtab;
                delim
-------------------------------------
 Name|CITY|New city|State|zip
 Tom|BOSTON|boston|MA|02121
 Eric|Burlington|BURLINGTON|MA|02482
 Jamie|cambridge|CAMBRIDGE|MA|02811
(4 rows)
```

See also

- MAPJSONEXTRACTOR
- MAPREGEXEXTRACTOR

MAPJSONEXTRACTOR

Extracts content of repeated JSON data objects,, including nested maps, or data with an outer list of JSON elements. You can set one or more optional parameters to control the extraction process.

> **Note**
> Empty input does not generate warnings or errors.

Syntax

```
MAPJSONEXTRACTOR (record-value [ USING PARAMETERS param=value[,...] ])
```

Arguments

record-value

String containing a JSON or delimited format record on which to apply the expression.

Parameters

flatten_maps

Boolean, flatten sub-maps within the JSON data, separating map **levels** with a period (.).

Default: true

flatten_arrays

Boolean, convert lists to sub-maps with integer keys. Lists are not flattened by default.

Default value: false

reject_on_duplicate

Boolean, ignore duplicate records (false), or reject duplicates (true). In either case, loading is unaffected.

Default: false

reject_on_empty_key

Boolean, reject any row that contains a key without a value.

Default: false

omit_empty_keys

Boolean, omit any key from the load data without a value.

Default: false

start_point

Name of a key in the JSON load data at which to begin parsing. The parser ignores all data before the start_point value. The parser processes data after the first instance, and up to the second, ignoring any remaining data.

Default: none

Examples

These examples use the following sample JSON data:

```
{ "id": "5001", "type": "None" }
{ "id": "5002", "type": "Glazed" }
{ "id": "5005", "type": "Sugar" }
{ "id": "5007", "type": "Powdered Sugar" }
{ "id": "5004", "type": "Maple" }
```

Save this example data as bake_single.json , and load that file.

1. Create a flex table, flexjson :

```
=> CREATE FLEX TABLE flexjson();
CREATE TABLE
```

2. Use COPY to load the bake_single.json file with the fjsonparser parser:

```
=> COPY flexjson FROM '/home/dbadmin/data/bake_single.json' parser fjsonparser();
 Rows Loaded
-------------
           5
(1 row)
```

3. Create a columnar table, coljson , with an IDENTITY column (id), a json column, and a column to hold a VMap, called vmap :

```
=> CREATE TABLE coljson(id IDENTITY(1,1), json varchar(128), vmap long varbinary(10000));
CREATE TABLE
```

4. Use COPY to load the bake_single.json file into the coljson table, using MAPJSONEXTRACTOR:

```
=> COPY coljson (json, vmap AS MapJSONExtractor(json)) FROM '/home/dbadmin/data/bake_single.json';
 Rows Loaded
-------------
           5
(1 row)
```

5. Use the MAPTOSTRING function for the flex table flexjson to output the __raw__ column contents as strings:

```
=> SELECT MAPTOSTRING(__raw__) FROM flexjson limit 5;
                    maptostring
-----------------------------------------------------------
 {
   "id" : "5001",
   "type" : "None"
}

 {
   "id" : "5002",
   "type" : "Glazed"
}

 {
   "id" : "5005",
   "type" : "Sugar"
}

 {
   "id" : "5007",
   "type" : "Powdered Sugar"
}

 {
   "id" : "5004",
   "type" : "Maple"
}

(5 rows)
```

6. Use MAPTOSTRING again, this time with the coljson table's vmap column and compare the results. The element order differs:

```
=> SELECT MAPTOSTRING(vmap) FROM coljson limit 5;
                    maptostring
-----------------------------------------------------------
 {
   "id" : "5001",
   "type" : "None"
}

 {
   "id" : "5002",
   "type" : "Glazed"
}

 {
   "id" : "5004",
   "type" : "Maple"
}

 {
   "id" : "5005",
   "type" : "Sugar"
}

 {
   "id" : "5007",
   "type" : "Powdered Sugar"
}

(5 rows)
```

See also

- MAPDELIMITEDEXTRACTOR
- MAPREGEXEXTRACTOR

MAPREGEXEXTRACTOR

Extracts data with a regular expression and returns results as a VMap.

Syntax

```
MAPREGEXEXTRACTOR (record-value [ USING PARAMETERS param=value[,...] ])
```

Arguments

record-value

String containing a JSON or delimited format record on which to apply the regular expression.

Parameters

pattern

Regular expression used to extract the desired data.

Default: Empty string ('')

use_jit

Boolean, use just-in-time compiling when parsing the regular expression.

Default: false

record_terminator

Character used to separate input records.

Default: \n

logline_column

Destination column containing the full string that the regular expression matched.

Default: Empty string ('')

Examples

These examples use the following regular expression, which searches for information that includes the timestamp , date , thread_name , and thread_id strings.

> **Caution**
> For display purposes, this sample regular expression adds new line characters to split long lines of text. To use this expression in a query, first copy and edit the example to remove any new line characters.

This example expression loads any thread_id hex value, regardless of whether it has a 0x prefix, (<thread_id>(?:0x)?[0-9a-f]+) .

```
'^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
(?<thread_name>[A-Za-z ]+):(?<thread_id>(?:0x)?[0-9a-f]+)
?(?<transaction_id>[0-9a-f])?(?:[(?<component>\w+)]
\<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @[?(?<enode>\w+)]?: )
?(?<text>.*)'
```

The following examples may include newline characters for display purposes.

1. Create a flex table, flogs :

```
=> CREATE FLEX TABLE flogs();
CREATE TABLE
```

2. Use COPY to load a sample log file (vertica.log), using the flex table fregexparser . Note that this example includes added line characters for displaying long text lines.

```
=> COPY flogs FROM '/home/dbadmin/tempdat/vertica.log' PARSER FREGEXPARSER(pattern='
^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+) (?<thread_name>[A-Za-z ]+):
(?<thread_id>(?:0x)?[0-9a-f])-?(?<transaction_id>[0-9a-f])?(?:[(?<component>\w+)]
\<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @[?(?<enode>\w+)]?: )?(?<text>.*)');
Rows Loaded
-------------
81399
(1 row)
```

3. Use to return the results from calling MAPREGEXEXTRACTOR with a regular expression. The output returns the results of the function in string format.

```
=> SELECT MAPTOSTRING(MapregexExtractor(E'2014-04-02 04:02:51.011
TM Moveout:0x2aab9000f860-a0000000002067 [Txn] <INFO>
Begin Txn: a0000000002067 \'Moveout: Tuple Mover\" using PARAMETERS
pattern=
'(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
(?<thread_name>[A-Za-z ]+):(?<thread_id>(?:0x)?[0-9a-f]+)
(?(?<transaction_id>[0-9a-f])?(?:[(?<component>\w+)]
<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @[?(?<enode>\w+)]?: )
?(?<text>.*)'

)) FROM flogs where __identity__=13;
maptostring
-------------------------------------------------------------------------------------------
{
"component" : "Txn",
"level" : "INFO",
"text" : "Begin Txn: a0000000002067 'Moveout: Tuple Mover'",
"thread_id" : "0x2aab9000f860",
"thread_name" : "TM Moveout",
"time" : "2014-04-02 04:02:51.011",
"transaction_id" : "a0000000002067"
}
(1 row)
```

See also

- MAPDELIMITEDEXTRACTOR
- MAPJSONEXTRACTOR

Flex map functions

The flex map functions let you extract and manipulate nested map data.

The first argument of all flex map functions (except EMPTYMAP and MAPAGGREGATE) takes a VMap. The VMap can originate from the __raw__ column in a flex table or be returned from a map or extraction function.

All map functions (except EMPTYMAP and MAPAGGREGATE) accept either a LONG VARBINARY or a LONG VARCHAR map argument.

In the following example, the outer MAPLOOKUP function operates on the VMap data returned from the inner MAPLOOKUP function:

```
=> MAPLOOKUP(MAPLOOKUP(ret_map, 'batch'), 'scripts')
```

You can use flex map functions exclusively with:

- Flex tables
- Their associated _keys tables and _view views
- Flexible complex-type columns

In this section

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPPUT
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

EMPTYMAP

Constructs a new VMap with one row but without keys or data. Use this transform function to populate a map without using a flex parser. Instead, you use either from SQL queries or from map data present elsewhere in the database.

Syntax

```
EMPTYMAP()
```

Examples

Create an Empty Map

```
=> SELECT EMPTYMAP();
                        emptymap
-------------------------------------------------------------------
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
(1 row)
```

Create an Empty Map from an Existing Flex Table

If you create an empty map from an existing flex table, the new map has the same number of rows as the table from which it was created.

This example shows the result if you create an empty map from the darkdata table, which has 12 rows of JSON data:

```
=> SELECT EMPTYMAP() FROM darkdata;
                        emptymap
-------------------------------------------------------------------
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
 \001\000\000\000\004\000\000\000\000\000\000\000\000\000\000\000
(12 rows)
```

See also

- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPAGGREGATE

Returns a LONG VARBINARY VMap with key and value pairs supplied from two VARCHAR input columns. This function requires an OVER clause.

Syntax

```
MAPAGGREGATE (keys-column1, values-column2 [USING PARAMETERS param=value[,...]])
```

Arguments

keys-column

Table column with the keys for the key/value pairs of the returned VMap data. Keys with a NULL value are excluded. If there are duplicate keys, the duplicate key and value that appear first in the query result are used, while the other duplicates are omitted.

values-column

Table column with the values for the key/value pairs of the returned VMap data.

Parameters

max_vmap_length

Maximum length in bytes for the VMap result, an integer between 1-32000000 inclusive.

Default: 130000

on_overflow

Overflow behavior for cases when the VMap result is larger than the max_vmap_length . The value must be one of the following strings:

- 'ERROR': Returns an error when overflow occurs.
- 'TRUNCATE': Stops aggregating key/value pairs if the result exceeds max_vmap_length . The query executes, but the resulting VMap does not have all key/value pairs. When the provided max_vmap_length is not large enough to store an empty VMap, the result returned is NULL. Note that you need to specify order criteria in the OVER clause to get consistent results.
- 'RETURN_NULL': Return NULL if overflow occurs.

Default: 'ERROR'

Examples

The following examples use this input table:

```
=> SELECT * FROM inventory;
  product     | stock
--------------+--------
  Planes      | 100
  Trains      | 50
  Automobiles | 200
(3 rows)
```

Call MAPAGGREGATE as follows to return the raw_map data of the resulting VMap:

```
=> SELECT raw_map FROM (SELECT MAPAGGREGATE(product, stock) OVER(ORDER BY product) FROM inventory) inventory;
raw_map
-----------------------------------------------------------------------------------------------------------
\001\000\000\000\030\000\000\000\003\000\000\000\020\000\000\000\023\000\000\000\026\000\000\00020010050\003
\000\000\000\020\000\000\000\033\000\000\000!\000\000\000AutomobilesPlanesTrains
(1 row)
```

To transform the returned raw_map data into string representation, use MAPAGGREGATE with MAPTOSTRING :

```
=> SELECT MAPTOSTRING(raw_map) FROM (SELECT MAPAGGREGATE(product, stock) OVER(ORDER BY product) FROM
inventory) inventory;
MAPTOSTRING
-------------------------------------------------------------
{
  "Automobiles": "200",
  "Planes": "100",
  "Trains": "50"
}
(1 row)
```

If you run the above query with on_overflow left as default and a max_vmap_length less than the returned VMap size, the function returns with an error message indicating the need to increase VMap length:

```
=> SELECT MAPTOSTRING(raw_map) FROM (SELECT MAPAGGREGATE(product, stock USING PARAMETERS max_vmap_length=60)
OVER(ORDER BY product) FROM inventory) inventory;
-----------------------------------------------------------------------------------------------------
ERROR 5861:  Error calling processPartition() in User Function MapAggregate at [/data/jenkins/workspace
/RE-PrimaryBuilds/RE-Build-Master_2/server/udx/supported/flextable/Dict.cpp:1324], error code: 0, message:
Exception while finalizing map aggregation: Output VMap length is too small [60]. HINT: Set the parameter
max_vmap_length=71 and retry your query
```

Switching the value of on_overflow allows you to alter how MAPAGGREGATE behaves in the case of overflow. For example, changing on_overflow to 'RETURN_NULL' causes the above query to execute and return NULL:

```
SELECT raw_map IS NULL FROM (SELECT MAPAGGREGATE(product, stock USING PARAMETERS max_vmap_length=60,
on_overflow='RETURN_NULL') OVER(ORDER BY product) FROM inventory) inventory;
?column?
----------
t
(1 row)
```

If on_overflow is set to 'TRUNCATE', the resulting VMap has enough space for two of the key/value pairs, but must cut the third:

```
SELECT raw_map IS NULL FROM (SELECT MAPAGGREGATE(product, stock USING PARAMETERS max_vmap_length=60,
on_overflow='TRUNCATE') OVER(ORDER BY product) FROM inventory) inventory;
MAPTOSTRING
-----------------------------------------------
{
  "Automobiles": "200",
  "Planes": "100"
}
(1 row)
```

See also

- EMPTYMAP
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPCONTAINSKEY

Determines whether a VMap contains a virtual column (key). This scalar function returns true (t), if the virtual column exists, or false (f) if it does not. Determining that a key exists before calling maplookup() lets you distinguish between NULL returns. The maplookup() function uses for both a non-existent key and an existing key with a NULL value.

Syntax

```
MAPCONTAINSKEY (VMap-data, 'virtual-column-name')
```

Arguments

VMap-data

Any VMap data. The VMap can exist as:

- The __raw__ column of a flex table
- Data returned from a map function such as MAPLOOKUP
- Other database content

virtual-column-name

Name of the key to check.

Examples

This example shows how to use the mapcontainskey() functions with maplookup() . View the results returned from both functions. Check whether the empty fields that maplookup() returns indicate a NULL value for the row (**t**) or no value (f):

You can use mapcontainskey( ) to determine that a key exists before calling maplookup(). The maplookup() function uses both NULL returns and existing keys with NULL values to indicate a non-existent key.

```
=> SELECT MAPLOOKUP(__raw__, 'user.location'), MAPCONTAINSKEY(__raw__, 'user.location')
FROM darkdata ORDER BY 1;
 maplookup | mapcontainskey
-----------+----------------
           | t
           | t
           | t
           | t
 Chile     | t
 Narnia    | t
 Uptown..  | t
 chicago   | t
           | f
           | f
           | f
           | f

(12 rows)
```

See also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPCONTAINSVALUE

Determines whether a VMap contains a specific value. Use this scalar function to return true (t) if the value exists, or false (f) if it does not.

Syntax

```
MAPCONTAINSVALUE (VMap-data, 'virtual-column-value')
```

Arguments

VMap-data

Any VMap data. The VMap can exist as:

- The __raw__ column of a flex table
- Data returned from a map function such as MAPLOOKUP
- Other database content

virtual-column-value

Value to confirm.

Examples

This example shows how to use mapcontainsvalue() to determine whether or not a virtual column contains a particular value. Create a flex table (ftest), and populate it with some virtual columns and values. Name both virtual columns one :

```
=> CREATE FLEX TABLE ftest();
CREATE TABLE
=> copy ftest from stdin parser fjsonparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"one":1, "two":2}
>> {"one":"one","2":"2"}
>> \.
```

Call mapcontainsvalue() on the ftest map data. The query returns false (f) for the first virtual column, and true (t) for the second , which contains the value one :

```
=> SELECT MAPCONTAINSVALUE(__raw__, 'one') FROM ftest;
mapcontainsvalue
------------------
f
t
(2 rows)
```

See also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPITEMS

Returns information about items in a VMap. Use this transform function with one or more optional arguments to access polystructured values within the VMap data. This function requires an over()` clause.

Syntax

```
MAPITEMS ( VMap-data [, passthrough-arg[,...] ])
```

Arguments

VMap-data

Any VMap data. The VMap can exist as:

- The __raw__ column of a flex table
- Data returned from a map function such as MAPLOOKUP
- Other database content

max_key_length

In a __raw__ column, determines the maximum length of keys that the function can return. Keys that are longer than *max_key_length* cause the query to fail. Defaults to the smaller of VMap column length and 65K.

max_value_length

In a __raw__ column, determines the maximum length of values the function can return. Values that are larger than *max_value_length* cause the query to fail. Defaults to the smaller of VMap column length and 65K.

passthrough-arg

One or more arguments indicating keys within the map data in VMap-data .

Examples

The following examples illustrate using MAPITEMS() with the over(PARTITION BEST) clause.

This example determines the number of virtual columns in the map data using a flex table, labeled darkmountain . Query using the count() function to return the number of virtual columns in the map data:

```
=> SELECT COUNT(keys) FROM (SELECT MAPITEMS(darkmountain.__raw__) OVER(PARTITION BEST) FROM
 darkmountain) AS a;
 count
-------
    19
(1 row)
```

The next example determines what items exist in the map data:

```
=> SELECT * FROM (SELECT MAPITEMS(darkmountain.__raw__) OVER(PARTITION BEST) FROM darkmountain) AS a;
    keys     |    values
-------------+---------------
 hike_safety | 50.6
 name        | Mt Washington
 type        | mountain
 height      | 17000
 hike_safety | 12.2
 name        | Denali
 type        | mountain
 height      | 29029
 hike_safety | 34.1
 name        | Everest
 type        | mountain
 height      | 14000
 hike_safety | 22.8
 name        | Kilimanjaro
 type        | mountain
 height      | 29029
 hike_safety | 15.4
 name        | Mt St Helens
 type        | volcano
(19 rows)
```

The following example shows how to restrict the length of returned values to 100000:

```
=> SELECT LENGTH(keys), LENGTH(values) FROM (SELECT MAPITEMS(__raw__ USING PARAMETERS max_value_length=100000) OVER() FROM t1)
x;
 LENGTH | LENGTH
--------+--------
      9 | 98899
(1 row)
```

Directly Query a Key Value in a VMap

Review the following JSON input file, simple.json . In particular, notice the array called three_Array , and its four values:

```
{
 "one": "one",
 "two": 2,
 "three_Array":
 [
   "three_One",
   "three_Two",
   3,
   "three_Four"
 ],
 "four": 4,
 "five_Map":
 {
   "five_One": 51,
   "five_Two": "Fifty-two",
   "five_Three": "fifty three",
   "five_Four": 54,
   "five_Five": "5 x 5"
 },
 "six": 6
}
```

1. Create a flex table, mapper:

```
=> CREATE FLEX TABLE mapper();
CREATE TABLE
```

Load simple.json into the flex table mapper:

```
=> COPY mapper FROM '/home/dbadmin/data/simple.json' parser fjsonparser (flatten_arrays=false,
flatten_maps=false);
 Rows Loaded
-------------
           1
(1 row)
```

Call MAPKEYS on the flex table's __raw__ column to see the flex table's keys, but not the key submaps. The return values indicate three_Array as one of the virtual columns:

```
=> SELECT MAPKEYS(__raw__) OVER() FROM mapper;
    keys
-------------
 five_Map
 four
 one
 six
 three_Array
 two
(6 rows)
```

Call mapitems on flex table mapper with three_Array as a pass-through argument to the function. The call returns these array values:

```
=> SELECT __identity__, MAPITEMS(three_Array) OVER(PARTITION BY __identity__) FROM mapper;
 __identity__ | keys |   values
--------------+------+------------
            1 | 0    | three_One
            1 | 1    | three_Two
            1 | 2    | 3
            1 | 3    | three_Four
(4 rows)
```

See also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE

- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPKEYS

Returns the virtual columns (and values) present in any VMap data. This transform function requires an OVER(PARTITION BEST) clause.

Syntax

```
MAPKEYS (VMap-data)
```

Arguments

VMap-data

Any VMap data. The VMap can exist as:

- The __raw__ column of a flex table
- Data returned from a map function such as MAPLOOKUP
- Other database content

max_key_length

In a __raw__ column, specifies the maximum length of keys that the function can return. Keys that are longer than *max_key_length* cause the query to fail. Defaults to the smaller of VMap column length and 65K.

Examples

Determine Number of Virtual Columns in Map Data

This example shows how to create a query, using an over(PARTITION BEST) clause with a flex table, darkdata to find the number of virtual column in the map data. The table is populated with JSON tweet data.

```
=> SELECT COUNT(keys) FROM (SELECT MAPKEYS(darkdata.__raw__) OVER(PARTITION BEST) FROM darkdata) AS a;
 count
-------
   550
(1 row)
```

Query Ordered List of All Virtual Columns in the Map

This example shows a snippet of the return data when you query an ordered list of all virtual columns in the map data:

```
=> SELECT * FROM (SELECT MAPKEYS(darkdata.__raw__) OVER(PARTITION BEST) FROM darkdata) AS a;
    keys
-------------------------------------
 contributors
 coordinates
 created_ at
 delete.status.id
 delete.status.id_str
 delete.status.user_id
 delete.status.user_id_str
 entities.hashtags
 entities.media
 entities.urls
 entities.user_mentions
 favorited
 geo
 id
.
.
.
 user.statuses_count
 user.time_zone
 user.url
 user.utc_offset
 user.verified
(125 rows)
```

Specify the Maximum Length of Keys that MAPKEYS Can Return

```
=> SELECT MAPKEYS(__raw__ USING PARAMETERS max_key_length=100000) OVER() FROM mapper;
    keys
-------------
 five_Map
 four
 one
 six
 three_Array
 two
(6 rows)
```

See also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPKEYSINFO

Returns virtual column information from a given map. This transform function requires an OVER(PARTITION BEST) clause.

Syntax

```
MAPKEYSINFO (VMap-data)
```

Arguments

VMap-data

Any VMap data. The VMap can exist as:

- The __raw__ column of a flex table
- Data returned from a map function such as MAPLOOKUP
- Other database content

max_key_length

In a __raw__ column, determines the maximum length of keys that the function can return. Keys that are longer than *max_key_length* cause the query to fail. Defaults to the smaller of VMap column length and 65K.

Returns

This function is a superset of the MAPKEYS () function. It returns the following information about each virtual column:

Column	Description
keys	The virtual column names in the raw data.
length	The data length of the key name, which can differ from the actual string length.
type_oid	The OID type into which the value should be converted. Currently, the type is always 116 for a LONG VARCHAR , or 199 for a nested map that is stored as a LONG VARBINARY .
row_num	The number of rows in which the key was found.
field_num	The field number in which the key exists.

Examples

This example shows a snippet of the return data you receive if you query an ordered list of all virtual columns in the map data:

```
=> SELECT * FROM (SELECT MAPKEYSINFO(darkdata.__raw__) OVER(PARTITION BEST) FROM darkdata) AS a;
                    keys                     | length | type_oid | row_num | field_num
-------------------------------------------------------------+--------+----------+---------+-----------
 contributors                                |      0 |      116 |       1 |         0
 coordinates                                 |      0 |      116 |       1 |         1
 created_at                                  |     30 |      116 |       1 |         2
 entities.hashtags                           |     93 |      199 |       1 |         3
 entities.media                              |    772 |      199 |       1 |         4
 entities.urls                               |     16 |      199 |       1 |         5
 entities.user_mentions                      |     16 |      199 |       1 |         6
 favorited                                   |      1 |      116 |       1 |         7
 geo                                         |      0 |      116 |       1 |         8
 id                                          |     18 |      116 |       1 |         9
 id_str                                      |     18 |      116 |       1 |        10
.
.
.
 delete.status.id                            |     18 |      116 |      11 |         0
 delete.status.id_str                        |     18 |      116 |      11 |         1
 delete.status.user_id                       |      9 |      116 |      11 |         2
 delete.status.user_id_str                   |      9 |      116 |      11 |         3
 delete.status.id                            |     18 |      116 |      12 |         0
 delete.status.id_str                        |     18 |      116 |      12 |         1
 delete.status.user_id                       |      9 |      116 |      12 |         2
 delete.status.user_id_str                   |      9 |      116 |      12 |         3
(550 rows)
```

Specify the Maximum Length of Keys that MAPKEYSINFO Can Return

```
=> SELECT MAPKEYSINFO(__raw__ USING PARAMETERS max_key_length=100000) OVER() FROM mapper;
   keys
-------------
 five_Map
 four
 one
 six
 three_Array
 two
(6 rows)
```

See also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPLOOKUP

Returns single-key values from VMAP data. This scalar function returns a LONG VARCHAR , with values, or NULL if the virtual column does not have a value.

Using maplookup is case insensitive to virtual column names. To avoid loading same-name values, set the fjsonparser parser reject_on_duplicate parameter to true when data loading.

You can control the behavior for non-scalar values in a VMAP (like arrays), when loading data with the fjsonparser or favroparser parsers and its flatten-arrays argument. See JSON data and the FJSONPARSER reference.

For information about using maplookup() to access nested JSON data, see Querying nested data .

Syntax

```
MAPLOOKUP ( VMap-data, 'virtual-column-name' [USING PARAMETERS [case_sensitive={false | true}] [, buffer_size=n] ] )
```

Parameters

VMap-data

Any VMap data. The VMap can exist as:

- The __raw__ column of a flex table
- Data returned from a map function such as MAPLOOKUP
- Other database content

virtual-column-name

The name of the virtual column whose values this function returns.

buffer_size

[Optional parameter] Specifies the maximum length (in bytes) of each value returned for *virtual-column-name* . To return all values for *virtual-column-name* , specify a buffer_size equal to or greater than (=>) the number of bytes for any returned value. Any returned values greater in length than buffer_size are rejected.

Default: 0 (No limit on buffer_size)

case_sensitive

[Optional parameter]

Specifies whether to return values for *virtual-column-name* if keys with different cases exist.
Example:

```
(... USING PARAMETERS case_sensitive=true)
```

Default: false

Examples

This example returns the values of one virtual column, user.location :

```
=> SELECT MAPLOOKUP(__raw__, 'user.location') FROM darkdata ORDER BY 1;
 maplookup
-----------
 Chile
 Nesnia
 Uptown
 .
 .
 chicago
(12 rows)
```

Using maplookup buffer_size

Use the buffer_size= parameter to indicate the maximum length of any value that maplookup returns for the virtual column you specify. If none of the returned key values can be greater than n bytes, use this parameter to allocate n bytes as the buffer_size .

For the next example, save this JSON data to a file, simple_name.json :

```
{
 "name": "sierra",
 "age": "63",
 "eyes": "brown",
 "weapon": "doggie"
}
{
 "name": "janis",
 "age": "10",
 "eyes": "blue",
 "weapon": "humor"
}
{
 "name": "ben",
 "age": "43",
 "eyes": "blue",
 "weapon": "sword"
}
{
 "name": "jen",
 "age": "38",
 "eyes": "green",
 "weapon": "shopping"
}
```

1. Create a flex table, logs .
2. Load the simple_name.json data into logs , using the fjsonparser . Specify the flatten_arrays option as True :
   ```
   => COPY logs FROM '/home/dbadmin/data/simple_name.json'
    PARSER fjsonparser(flatten_arrays=True);
   ```
3. Use maplookup with buffer_size=0 for the logs table name key. This query returns all of the values:

```
=> SELECT MAPLOOKUP(__raw__, 'name' USING PARAMETERS buffer_size=0) FROM logs;
 MapLookup
-----------
 sierra
 ben
 janis
 jen
(4 rows)
```

4. Next, call maplookup() three times, specifying the buffer_size parameter as 3 , 5 , and 6 , respectively. Now, maplookup() returns values with a byte length less than or equal to (<=) buffer_size :

```
=> SELECT MAPLOOKUP(__raw__, 'name' USING PARAMETERS buffer_size=3) FROM logs;
 MapLookup
-----------

 ben

 jen
(4 rows)
=> SELECT MAPLOOKUP(__raw__, 'name' USING PARAMETERS buffer_size=5) FROM logs;
 MapLookup
-----------

 janis
 jen
 ben
(4 rows)
=> SELECT MAPLOOKUP(__raw__, 'name' USING PARAMETERS buffer_size=6) FROM logs;
 MapLookup
-----------
 sierra
 janis
 jen
 ben
(4 rows)
```

Disambiguate Empty Output Rows

This example shows how to interpret empty rows. Using maplookup without first checking whether a key exists can be ambiguous. When you review the following output, 12 empty rows, you cannot determine whether a user.location key has:

- A non-NULL value
- A NULL value
- No value

```
=> SELECT MAPLOOKUP(__raw__, 'user.location') FROM darkdata;
 maplookup
-----------









(12 rows)
```

To disambiguate empty output rows, use the mapcontainskey() function in conjunction with maplookup() . When maplookup returns an empty field, the corresponding value from mapcontainskey indicates t for a NULL or other value, or f for no value.

The following example output using both functions lists rows with NULL or a name value as t , and rows with no value as f :

```
=> SELECT MAPLOOKUP(__raw__, 'user.location'), MAPCONTAINSKEY(__raw__, 'user.location')
FROM darkdata ORDER BY 1;
 maplookup | mapcontainskey
-----------+----------------
           | t
           | t
           | t
           | t
 Chile     | t
 Nesnia    | t
 Uptown    | t
 chicago   | t
           | f >>>>>>>>>>No value
           | f >>>>>>>>>>No value
           | f >>>>>>>>>>No value
           | f >>>>>>>>>>No value
(12 rows)
```

Check for Case-Sensitive Virtual Columns

You can use maplookup() with the case_sensitive parameter to return results when key names with different cases exist.

1. Save the following sample content as a JSON file. This example saves the file as repeated_key_name.json :

```
{
  "test": "lower1"
}
{
  "TEST": "upper1"
}
{
  "TEst": "half1"
}
{
  "test": "lower2",
  "TEst": "half2"
}
{
  "TEST": "upper2",
  "TEst": "half3"
}
{
  "test": "lower3",
  "TEST": "upper3"
}
{
  "TEst": "half4",
  "test": "lower4",
  "TEST": "upper4"
}
{
  "TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttest
TesttestTesttestTesttestTesttest":"1",
    "TesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttestTesttest
TesttestTesttestTesttestTesttestTest12345":"2"
}
```

2. Create a flex table, dupe , and load the JSON file:

```
=> CREATE FLEX TABLE dupe();
CREATE TABLE
dbt=> COPY dupe FROM '/home/release/KData/repeated_key_name.json' parser fjsonparser();
 Rows Loaded
-------------
           8
(1 row)
```

See also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPSIZE
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPPUT

Accepts a VMap and one or more key/value pairs and returns a new VMap with the key/value pairs added. Keys must be set using the auxiliary function SetMapKeys() , and can only be constant strings. If the VMap has any of the new input keys, then the original values are replaced by the new ones.

Syntax

```
MAPPUT (VMap-data, value[,...] USING PARAMETERS keys=SetMapKeys('key'[,...])
```

Arguments

VMap-data

Any VMap data. The VMap can exist as:

- The __raw__ column of a flex table
- Data returned from a map function such as MAPLOOKUP .
- Other database content

***value* [,...]**

One or more values to add to the VMap specified in VMap-data .

Parameters

keys

The result of SetMapKeys() . SetMapKeys() takes one or more constant string arguments.

The following example shows how to create a flex table and use COPY to enter some basic JSON data. After creating a second flex table, insert the new VMap results from mapput() , with additional key/value pairs.

1. Create sample table:
```
=> CREATE FLEX TABLE vmapdata1();
CREATE TABLE
```
2. Load sample JSON data from STDIN:
```
=> COPY vmapdata1 FROM stdin parser fjsonparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"aaa": 1, "bbb": 2, "ccc": 3}
>> \.
```
3. Create another flex table and use the function to insert data into it: => CREATE FLEX TABLE vmapdata2(); => INSERT INTO vmapdata2 SELECT MAPPUT(__raw__, '7','8','9' using parameters keys=SetMapKeys('xxx','yyy','zzz')) from vmapdata1;
4. View the difference between the original and the new flex tables:

```
=> SELECT MAPTOSTRING(__raw__) FROM vmapdata1;
                maptostring
-----------------------------------------------------------
{
  "aaa" : "1",
  "bbb" : "2",
  "ccc" : "3"
}
(1 row)

=> SELECT MAPTOSTRING(__raw__) from vmapdata2;
                maptostring
-----------------------------------------------------------
{
  "mapput" : {
     "aaa" : "1",
     "bbb" : "2",
     "ccc" : "3",
     "xxx" : "7",
     "yyy" : "8",
     "zzz" : "9"
  }
}
```

See also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPVALUES
- MAPVERSION

MAPSIZE

Returns the number of virtual columns present in any VMap data. Use this scalar function to determine the size of keys.

Syntax

```
MAPSIZE (VMap-data)
```

Arguments

VMap-data

Any VMap data. The VMap can exist as:

- The __raw__ column of a flex table
- Data returned from a map function such as MAPLOOKUP
- Other database content

Examples

This example shows the returned sizes from the number of keys in the flex table darkmountain :

```
=> SELECT MAPSIZE(__raw__) FROM darkmountain;
 mapsize
---------
       3
       4
       4
       4
       4
(5 rows)
```

See also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPTOSTRING
- MAPVALUES
- MAPVERSION

MAPTOSTRING

Recursively builds a string representation of VMap data, including nested JSON maps. Use this transform function to display the VMap contents in a LONG VARCHAR format. You can use MAPTOSTRING to see how map data is nested before querying virtual columns with MAPVALUES .

Syntax

```
MAPTOSTRING ( VMap-data [ USING PARAMETERS param=value ] )
```

Arguments

VMap-data

Any VMap data. The VMap can exist as:

- The __raw__ column of a flex table
- Data returned from a map function such as MAPLOOKUP
- Other database content

Parameters

canonical_json

Boolean, whether to produce canonical JSON format, using the first instance of any duplicate keys in the map data. If false, the function returns duplicate keys and their values.

Default: true

Examples

The following example uses this table definition and sample data:

```
=> CREATE FLEX TABLE darkdata();
CREATE TABLE

=> COPY darkdata FROM stdin parser fjsonparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"aaa": 1, "aaa": 2, "AAA": 3, "bbb": "aaa\"bbb"}
>> \.
```

Calling MAPTOSTRING with the default value of canonical_json returns only the first instance of the duplicate key:

```
=> SELECT MAPTOSTRING (__raw__) FROM darkdata;
                    maptostring
-----------------------------------------------------------------
{
   "AAA" : "3",
   "aaa" : "1",
   "bbb" : "aaa\"bbb"
}
(1 row)
```

With canonical_json set to false, the function returns all of the keys, including duplicates:

```
=> SELECT MAPTOSTRING(__raw__ using parameters canonical_json=false) FROM darkdata;
                     maptostring
-----------------------------------------------------------------
{
       "aaa":  "1",
       "aaa":  "2",
       "AAA":  "3",
       "bbb":  "aaa"bbb"
}
(1 row)
```

See also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPVALUES
- MAPVERSION

MAPVALUES

Returns a string representation of the top-level values from a VMap. This transform function requires an OVER() clause.

Syntax

```
MAPVALUES (VMap-data)
```

Arguments

VMap-data

Any VMap data. The VMap can exist as:

- The __raw__ column of a flex table
- Data returned from a map function such as MAPLOOKUP
- Other database content

max_value_length

In a __raw__ column, specifies the maximum length of values the function can return. Values that are larger than *max_value_length* cause the query to fail. Defaults to the smaller of VMap column length and 65K.

Examples

The following example shows how to query a darkmountain flex table, using an over() clause (in this case, the over(PARTITION BEST) clause) with mapvalues() .

```
=> SELECT * FROM (SELECT MAPVALUES(darkmountain.__raw__) OVER(PARTITION BEST) FROM darkmountain) AS a;
   values
---------------
 29029
 34.1
 Everest
 mountain
 29029
 15.4
 Mt St Helens
 volcano
 17000
 12.2
 Denali
 mountain
 14000
 22.8
 Kilimanjaro
 mountain
 50.6
 Mt Washington
 mountain
(19 rows)
```

Specify the Maximum Length of Values that MAPVALUES Can Return

```
=> SELECT MAPVALUES(__raw__ USING PARAMETERS max_value_length=100000) OVER() FROM mapper;
   keys
-------------
 five_Map
 four
 one
 six
 three_Array
 two
(6 rows)
```

See also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVERSION

MAPVERSION

Returns the version or invalidity of any map data. This scalar function returns the map version (such as 1) or -1 , if the map data is invalid.

Syntax

```
MAPVERSION (VMap-data)
```

Arguments

VMap-data

Any VMap data. The VMap can exist as:

- The __raw__ column of a flex table

- Data returned from a map function such as MAPLOOKUP
- Other database content

Examples

The following example shows how to use mapversion() with the darkmountain flex table, returning mapversion 1 for the flex table map data:

```
=> SELECT MAPVERSION(__raw__) FROM darkmountain;
 mapversion
------------
          1
          1
          1
          1
          1
(5 rows)
```

See also

- EMPTYMAP
- MAPAGGREGATE
- MAPCONTAINSKEY
- MAPCONTAINSVALUE
- MAPITEMS
- MAPKEYS
- MAPKEYSINFO
- MAPLOOKUP
- MAPSIZE
- MAPTOSTRING
- MAPVALUES

Formatting functions

Formatting functions provide a powerful tool set for converting various data types (DATE/TIME, INTEGER, FLOATING POINT) to formatted strings and for converting from formatted strings to specific data types.

In this section

- Template patterns for date/time formatting
- Template patterns for numeric formatting
- TO_BITSTRING
- TO_CHAR
- TO_DATE
- TO_HEX
- TO_NUMBER
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ

Template patterns for date/time formatting

In an output template string (for TO_CHAR), certain patterns are recognized and replaced with appropriately formatted data from the value to format. Any text that is not a template pattern is copied verbatim. Similarly, in an input template string (for anything other than TO_CHAR), template patterns identify the parts of the input data string to look at and the values to find there.

> **Note**
> Vertica uses the ISO 8601:2004 style for date/time fields in Vertica log files. For example:
>
> ```
> 2020-03-25 05:04:22.372 Init Session:0x7f8fcefec700-a000000013dcd4 [Txn] <INFO> Begin Txn: a000000013dcd4 'read role info'
> ```

Pattern	Description

HH	Hour of day (00-23)
HH12	Hour of day (01-12)
HH24	Hour of day (00-23)
MI	Minute (00-59)
SS	Second (00-59)
MS	Millisecond (000-999)
US	Microsecond (000000-999999)
SSSS	Seconds past midnight (0-86399)
AM A.M. PM P.M.	Meridian indicator (uppercase)
am a.m. pm p.m.	Meridian indicator (lowercase)
Y YYY	Year (4 and more digits) with comma
YYYY	Year (4 and more digits)
YYY	Last 3 digits of year
YY	Last 2 digits of year
Y	Last digit of year
IYYY	ISO year (4 and more digits)
IYY	Last 3 digits of ISO year
IY	Last 2 digits of ISO year
I	Last digits of ISO year
BC B.C. AD A.D.	Era indicator (uppercase)
bc b.c. ad a.d.	Era indicator (lowercase)
MONTH	Full uppercase month name (blank-padded to 9 chars)
Month	Full mixed-case month name (blank-padded to 9 chars)
month	Full lowercase month name (blank-padded to 9 chars)
MON	Abbreviated uppercase month name (3 chars)
Mon	Abbreviated mixed-case month name (3 chars)
mon	Abbreviated lowercase month name (3 chars)
MM	Month number (01-12)
DAY	Full uppercase day name (blank-padded to 9 chars)

Day	Full mixed-case day name (blank-padded to 9 chars)
day	full lowercase day name (blank-padded to 9 chars)
DY	Abbreviated uppercase day name (3 chars)
Dy	Abbreviated mixed-case day name (3 chars)
dy	Abbreviated lowercase day name (3 chars)
DDD	Day of year (001-366)
DD	Day of month (01-31) for TIMESTAMP **Note** For INTERVAL, DD is day of year (001-366) because day of month is undefined.
D	Day of week (1-7; Sunday is 1)
W	Week of month (1-5) (The first week starts on the first day of the month.)
WW	Week number of year (1-53) (The first week starts on the first day of the year.)
IW	ISO week number of year (The first Thursday of the new year is in week 1.)
CC	Century (2 digits)
J	Julian Day (days since January 1, 4712 BC)
Q	Quarter
RM	Month in Roman numerals (I-XII; I=January) (uppercase)
rm	Month in Roman numerals (i-xii; i=January) (lowercase)
TZ	Time-zone name (uppercase)
tz	Time-zone name (lowercase)

Template pattern modifiers

Certain modifiers can be applied to any date/time template pattern to alter its behavior. For example, FMMonth is the Month pattern with the FM modifier.

Modifier	**Description**
AM	Time is before 12:00
AT	Ignored
JULIAN, JD, J	Next field is Julian Day

FM prefix	Fill mode (suppress padding blanks and zeros) For example: FMMonth **Note:** The FM modifier suppresses leading zeros and trailing blanks that would otherwise be added to make the output of a pattern fixed width.
FX prefix	Fixed format global option For example : FX Month DD Day
ON	Ignored
PM	Time is on or after 12:00
T	Next field is time
TH suffix	Uppercase ordinal number suffix For example: DDTH
th suffix	Lowercase ordinal number suffix For example: DDth
TM prefix	Translation mode (print localized day and month names based on lc_messages). For example: TMMonth

Examples

Use TO_TIMESTAMP to convert an expression using the pattern 'YYY MON' :

```
=> SELECT TO_TIMESTAMP('2017 JUN', 'YYYY MON');
   TO_TIMESTAMP
---------------------
 2017-06-01 00:00:00
(1 row)
```

Use TO_DATE to convert an expression using the pattern 'YYY-MMDD' :

```
=> SELECT TO_DATE('2017-1231', 'YYYY-MMDD');
 TO_DATE
------------
 2017-12-31
(1 row)
```

Template patterns for numeric formatting

Pattern	Description
9	Value with the specified number of digits
0	Value with leading zeros
.	Decimal point
,	Group (thousand) separator
PR	Negative value in angle brackets

S	Sign anchored to number (uses locale)
L	Currency symbol (uses locale)
D	Decimal point (uses locale)
G	Group separator (uses locale)
MI	Minus sign in specified position (if number < 0)
PL	Plus sign in specified position (if number > 0)
SG	Plus/minus sign in specified position
RN	Roman numeral (input between 1 and 3999)
TH/th	Ordinal number suffix
V	Shift specified number of digits
EEEE	Scientific notation (not implemented yet)

Usage

- A sign formatted using SG, PL, or MI is not anchored to the number. For example:

```
=> SELECT to_char(-12, 'S9999'), to_char(-12, 'MI9999');
 to_char | to_char
---------+---------
  -12    | -  12
(1 row)
```

  - TO_CHAR(-12, 'S9999') produces ' -12'
  - TO_CHAR(-12, 'MI9999') produces '- 12'
- 9 results in a value with the same number of digits as there are 9s. If a digit is not available it outputs a space.
- TH does not convert values less than zero and does not convert fractional numbers.
- V effectively multiplies the input values by 10^ *n* , where *n* is the number of digits following V. TO_CHAR does not support the use of V combined with a decimal point—for example: 99.9V99 .

TO_BITSTRING

Returns a VARCHAR that represents the given VARBINARY value in bitstring format. This function is the inverse of BITSTRING_TO_BINARY .

Behavior type

Immutable

Syntax

```
TO_BITSTRING ( expression )
```

Arguments

expression

The VARCHAR string to process.

Examples

```
=> SELECT TO_BITSTRING('ab'::BINARY(2));
  to_bitstring
------------------
 0110000101100010
(1 row)

=> SELECT TO_BITSTRING(HEX_TO_BINARY('0x10'));
 to_bitstring
--------------
00010000
(1 row)

=> SELECT TO_BITSTRING(HEX_TO_BINARY('0xF0'));
 to_bitstring
--------------
11110000
(1 row)
```

See also

BITCOUNT

TO_CHAR

Converts date/time and numeric values into text strings.

Behavior type

Stable

Syntax

```
TO_CHAR ( expression [, pattern ] )
```

Parameters

expression

Specifies the value to convert, one of the following data types:

- DOUBLE PRECISION
- INTEGER
- INTERVAL
- TIME/TIMETZ
- TIMESTAMP/TIMESTAMPTZ

The following restrictions apply:

- TO_CHAR does not support binary data types BINARY and VARBINARY
- TO_CHAR does not support the use of V combined with a decimal point—for example, 99.9V99

pattern

A CHAR or VARCHAR that specifies an output pattern string. See Template patterns for date/time formatting .

Notes

- Vertica pads TO_CHAR output with a leading space, so positive and negative values have the same length. To suppress padding, use the FM prefix .
- TO_CHAR accepts TIME and TIMETZ data types as inputs if you explicitly cast TIME to TIMESTAMP and TIMETZ to TIMESTAMPTZ.

```
=> SELECT TO_CHAR(TIME '14:34:06.4','HH12:MI am'), TO_CHAR(TIMETZ '14:34:06.4+6','HH12:MI am');
 TO_CHAR  | TO_CHAR
----------+----------
 02:34 pm | 04:34 am
(1 row)
```

- You can extract the timezone hour from TIMETZ:

```
=> SELECT EXTRACT(timezone_hour FROM TIMETZ '10:30+13:30');
 date_part
-----------
        13
(1 row)
```

- Ordinary text is allowed in TO_CHAR templates and is output literally. You can put a substring in double quotes to force it to be interpreted as literal text even if it contains pattern key words. In the following example, YYYY is replaced by the year data, but the Y in Year is not:

```
=> SELECT to_char(CURRENT_TIMESTAMP, '"Hello Year " YYYY');
    to_char
------------------
 Hello Year  2021
(1 row)
```

- TO_CHAR uses different day-of-the-week numbering (see the D template pattern) than EXTRACT .
- Given an INTERVAL type, TO_CHAR formats HH and HH12 as hours in a single day, while HH24 can output hours exceeding a single day—for example, >24 .
- To include a double quote (") character in output, precede it with a double backslash (\\ ). This is necessary because the backslash already has a special meaning in a string constant. For example: '\\"YYYY Month\\"'
- When rounding, the last digit of the rounded representation is selected to be even if the number is exactly half way between the two.

Examples

TO_CHAR expression and pattern argument	**Output**
CURRENT_TIMESTAMP, 'Day, DD HH12:MI:SS'	Tuesday , 06 05:39:18
CURRENT_TIMESTAMP, 'FMDay, FMDD HH12:MI:SS'	Tuesday, 6 05:39:18
TIMETZ '14:34:06.4+6','HH12:MI am'	04:34 am
-0.1, '99.99'	-.10
-0.1, 'FM9.99'	-.1
0.1, '0.9'	0.1
12, '9990999.9'	`0012.0`
12, 'FM9990999.9'	0012.
485, '999'	485
-485, '999'	-485
485, '9 9 9'	4 8 5
1485, '9,999'	1,485
1485, '9G999'	1 485
148.5, '999.999'	148.500
148.5, 'FM999.999'	148.5
148.5, 'FM999.990'	148.500
148.5, '999D999'	148,500
3148.5, '9G999D999'	3 148,500
-485, '999S'	485-
-485, '999MI'	485-
485, '999MI'	485

485, 'FM999MI'	485
485, 'PL999'	+485
485, 'SG999'	+485
-485, 'SG999'	-485
-485, '9SG99'	4-85
-485, '999PR'	<485>
485, 'L999'	DM 485
485, 'RN'	`CDLXXXV`
485, 'FMRN'	CDLXXXV
5.2, 'FMRN'	V
482, '999th'	482nd
485, '"Good number:"999'	Good number: 485
485.8, '"Pre:"999" Post:" .999'	Pre: 485 Post: .800
12, '99V999'	12000
12.4, '99V999'	12400
12.45, '99V9'	125
-1234.567	-1234.567
'1999-12-25'::DATE	1999-12-25
'1999-12-25 11:31'::TIMESTAMP	1999-12-25 11:31:00
'1999-12-25 11:31 EST'::TIMESTAMPTZ	1999-12-25 11:31:00-05
'3 days 1000.333 secs'::INTERVAL	3 days 00:16:40.333

See also

DATE_PART

TO_DATE

Converts a string value to a DATE type.

Behavior type

Stable

Syntax

```
TO_DATE ( expression , pattern )
```

Parameters

expression

Specifies the string value to convert, either CHAR or VARCHAR .

pattern

A CHAR or VARCHAR that specifies an output pattern string. See:

- Template patterns for date/time formatting
- Template patterns for numeric formatting

Input value considerations

TO_DATE requires a CHAR or VARCHAR expression. For other input types, use TO_CHAR to perform an explicit cast to a CHAR or VARCHAR before using this function.

Notes

- To use a double quote character in the output, precede it with a double backslash. This is necessary because the backslash already has a special meaning in a string constant. For example: '\\"YYYY Month\\"'
- TO_TIMESTAMP , TO_TIMESTAMP_TZ , and TO_DATE skip multiple blank spaces in the input string if the FX option is not used. FX must be specified as the first item in the template. For example:
  - TO_TIMESTAMP('2000 JUN', 'YYYY MON') is correct.
  - TO_TIMESTAMP('2000 JUN', 'FXYYYY MON') returns an error, because TO_TIMESTAMP expects one space only.
- The YYYY conversion from string to TIMESTAMP or DATE has a restriction if you use a year with more than four digits. You must use a non-digit character or template after YYYY , otherwise the year is always interpreted as four digits. For example, given the following arguments, TO_DATE interprets the five-digit year 20000 as a four-digit year:

```
=> SELECT TO_DATE('200001131','YYYYMMDD');
 TO_DATE
------------
 2000-01-13
(1 row)
```

  Instead, use a non-digit separator after the year. For example:

```
=> SELECT TO_DATE('20000-1131', 'YYYY-MMDD');
  TO_DATE
-------------
 20000-12-01
(1 row)
```

- In conversions from string to TIMESTAMP or DATE , the CC field is ignored if there is a YYY, YYYY or Y,YYY field. If CC is used with YY or Y, then the year is computed as (CC–1)*100+YY.

Examples

```
=> SELECT TO_DATE('13 Feb 2000', 'DD Mon YYYY');
 to_date
------------
 2000-02-13
(1 row)
```

See also

Date/time functions

TO_HEX

Returns a VARCHAR or VARBINARY representing the hexadecimal equivalent of a number. This function is the inverse of HEX_TO_BINARY .

Behavior type

Immutable

Syntax

```
TO_HEX ( number )
```

Arguments

number

An INTEGER or VARBINARY value to convert to hexadecimal. If you supply a VARBINARY argument, the function's return value is not preceded by 0x .

Examples

```
=> SELECT TO_HEX(123456789);
 TO_HEX
---------
 75bcd15
(1 row)
```

For VARBINARY inputs, the returned value is not preceded by 0x . For example:

```
=> SELECT TO_HEX('ab'::binary(2));
 TO_HEX
--------
 6162
(1 row)
```

TO_NUMBER

Converts a string value to DOUBLE PRECISION.

Behavior type

Stable

Syntax

```
TO_NUMBER ( expression, [ pattern ] )
```

Parameters

expression

Specifies the string value to convert, either CHAR or VARCHAR.

pattern

A string value, either CHAR or VARCHAR, that specifies an output pattern string using one of the supported Template patterns for numeric formatting . If you omit this parameter, TO_NUMBER returns a floating point.

Notes

To use a double quote character in the output, precede it with a double backslash. This is necessary because the backslash already has a special meaning in a string constant. For example: '\\"YYYY Month\\"'

> **Note**
> To convert a date string to a numeric value, use the appropriate date/time function , such as EXTRACT .

Examples

```
=> SELECT TO_NUMBER('MCML', 'rn');
TO_NUMBER
-----------
1950
(1 row)
```

It the pattern parameter is omitted, the function returns a floating point. For example:

```
=> SELECT TO_NUMBER('-123.456e-01');
 TO_NUMBER
-----------
 -12.3456
```

TO_TIMESTAMP

Converts a string value or a UNIX/POSIX epoch value to a TIMESTAMP type.

Behavior type

Stable

Syntax

```
TO_TIMESTAMP ( { expression, pattern } | unix-epoch )
```

Parameters

expression

Specifies the string value to convert, of type CHAR or VARCHAR.

pattern

A CHAR or VARCHAR that specifies an output pattern string. See:

- Template patterns for date/time formatting
- Template patterns for numeric formatting

unix-epoch

DOUBLE PRECISION value that specifies some number of seconds elapsed since midnight UTC of January 1, 1970, excluding leap seconds. INTEGER values are implicitly cast to DOUBLE PRECISION.

Notes

- Millisecond (MS) and microsecond (US) values in a conversion from string to TIMESTAMP are used as part of the seconds after the decimal point. For example TO_TIMESTAMP('12:3', 'SS:MS') is not 3 milliseconds, but 300, because the conversion counts it as 12 + 0.3 seconds. This means for the format SS:MS , the input values 12:3 , 12:30 , and 12:300 specify the same number of milliseconds. To get three milliseconds, use 12:003 , which the conversion counts as 12 + 0.003 = 12.003 seconds.
  Here is a more complex example: TO_TIMESTAMP('15:12:02.020.001230', 'HH:MI:SS.MS.US') is 15 hours, 12 minutes, and 2 seconds + 20 milliseconds + 1230 microseconds = 2.021230 seconds.
- To use a double quote character in the output, precede it with a double backslash. This is necessary because the backslash already has a special meaning in a string constant. For example: '\\"YYYY Month\\"'
- TO_TIMESTAMP , TO_TIMESTAMP_TZ , and TO_DATE skip multiple blank spaces in the input string if the FX option is not used. FX must be specified as the first item in the template. For example:
  - TO_TIMESTAMP('2000 JUN', 'YYYY MON') is correct.
  - TO_TIMESTAMP('2000 JUN', 'FXYYYY MON') returns an error, because TO_TIMESTAMP expects one space only.
- The YYYY conversion from string to TIMESTAMP or DATE has a restriction if you use a year with more than four digits. You must use a non-digit character or template after YYYY , otherwise the year is always interpreted as four digits. For example, given the following arguments, TO_DATE interprets the five-digit year 20000 as a four-digit year:

```
=> SELECT TO_DATE('200001131','YYYYMMDD');
 TO_DATE
------------
 2000-01-13
(1 row)
```

  Instead, use a non-digit separator after the year. For example:

```
=> SELECT TO_DATE('20000-1131', 'YYYY-MMDD');
  TO_DATE
-------------
 20000-12-01
(1 row)
```

- In conversions from string to TIMESTAMP or DATE , the CC field is ignored if there is a YYY, YYYY or Y,YYY field. If CC is used with YY or Y, then the year is computed as (CC–1)*100+YY.

Examples

```
=> SELECT TO_TIMESTAMP('13 Feb 2009', 'DD Mon YYYY');
   TO_TIMESTAMP
---------------------
 1200-02-13 00:00:00
(1 row)
=> SELECT TO_TIMESTAMP(200120400);
   TO_TIMESTAMP
---------------------
 1976-05-05 01:00:00
(1 row)
```

See also

Date/time functions

TO_TIMESTAMP_TZ

Converts a string value or a UNIX/POSIX epoch value to a TIMESTAMP WITH TIME ZONE type.

Behavior type

Immutable if single argument form, Stable otherwise.

Syntax

```
TO_TIMESTAMP_TZ ( { expression, pattern } | unix-epoch )
```

Parameters

expression

Specifies the string value to convert, of type CHAR or VARCHAR.

pattern

A CHAR or VARCHAR that specifies an output pattern string. See:

- Template patterns for date/time formatting
- Template patterns for numeric formatting

unix-epoch

A DOUBLE PRECISION value that specifies some number of seconds elapsed since midnight UTC of January 1, 1970, excluding leap seconds. INTEGER values are implicitly cast to DOUBLE PRECISION.

Notes

- Millisecond (MS) and microsecond (US) values in a conversion from string to TIMESTAMP are used as part of the seconds after the decimal point. For example TO_TIMESTAMP('12:3', 'SS:MS') is not 3 milliseconds, but 300, because the conversion counts it as 12 + 0.3 seconds. This means for the format SS:MS , the input values 12:3 , 12:30 , and 12:300 specify the same number of milliseconds. To get three milliseconds, use 12:003 , which the conversion counts as 12 + 0.003 = 12.003 seconds.
  Here is a more complex example: TO_TIMESTAMP('15:12:02.020.001230', 'HH:MI:SS.MS.US') is 15 hours, 12 minutes, and 2 seconds + 20 milliseconds + 1230 microseconds = 2.021230 seconds.
- To use a double quote character in the output, precede it with a double backslash. This is necessary because the backslash already has a special meaning in a string constant. For example: '\\"YYYY Month\\"'
- TO_TIMESTAMP , TO_TIMESTAMP_TZ , and TO_DATE skip multiple blank spaces in the input string if the FX option is not used. FX must be specified as the first item in the template. For example:
  - TO_TIMESTAMP('2000 JUN', 'YYYY MON') is correct.
  - TO_TIMESTAMP('2000 JUN', 'FXYYYY MON') returns an error, because TO_TIMESTAMP expects one space only.
- The YYYY conversion from string to TIMESTAMP or DATE has a restriction if you use a year with more than four digits. You must use a non-digit character or template after YYYY , otherwise the year is always interpreted as four digits. For example, given the following arguments, TO_DATE interprets the five-digit year 20000 as a four-digit year:

  ```
  => SELECT TO_DATE('200001131','YYYYMMDD');
   TO_DATE
  ------------
   2000-01-13
  (1 row)
  ```

  Instead, use a non-digit separator after the year. For example:

  ```
  => SELECT TO_DATE('20000-1131', 'YYYY-MMDD');
    TO_DATE
  -------------
   20000-12-01
  (1 row)
  ```

- In conversions from string to TIMESTAMP or DATE , the CC field is ignored if there is a YYY, YYYY or Y,YYY field. If CC is used with YY or Y, then the year is computed as (CC–1)*100+YY.

Examples

```
=> SELECT TO_TIMESTAMP_TZ('13 Feb 2009', 'DD Mon YYY');
    TO_TIMESTAMP_TZ
------------------------
 1200-02-13 00:00:00-05
(1 row)

=> SELECT TO_TIMESTAMP_TZ(200120400);
    TO_TIMESTAMP_TZ
------------------------
 1976-05-05 01:00:00-04
(1 row)
```

See also

Date/time functions

Geospatial functions

Geospatial functions manipulate complex two-dimensional spatial objects and store them in a database according to the Open Geospatial Consortium (OGC) standards.

Function naming conventions

The geospatial functions use the following naming conventions:

- Most ST_ *function-name* functions are compliant with the latest OGC standard OGC SFA-SQL version 1.2.1 (reference. number is OGC 06-104r4, date: 2010-08-04). Currently, some ST_ *function-name* functions may not support all data types. Each function page contains details about the supported data types.

  > **Note**
  > Some functions, such as ST_GeomFromText, are based on previous versions of the standard.

- The STV_ *function-name* functions are unique to Vertica and not compliant with OGC standards. Each function page explains its functionality in detail.

Verifying spatial objects validity

Many spatial functions do not validate their parameters. If you pass an invalid spatial object to an ST_ or STV_ function, the function might return an error or produce incorrect results.

To avoid this issue, Vertica recommends that you first run ST_IsValid on all spatial objects to validate the parameters. If your object is not valid, run STV_IsValidReason to get information about the location of the invalidity.

In this section

- ST_Area
- ST_AsBinary
- ST_AsText
- ST_Boundary
- ST_Buffer
- ST_Centroid
- ST_Contains
- ST_ConvexHull
- ST_Crosses
- ST_Difference
- ST_Disjoint
- ST_Distance
- ST_Envelope
- ST_Equals
- ST_GeographyFromText
- ST_GeographyFromWKB
- ST_GeoHash
- ST_GeometryN
- ST_GeometryType
- ST_GeomFromGeoHash
- ST_GeomFromGeoJSON
- ST_GeomFromText
- ST_GeomFromWKB
- ST_Intersection
- ST_Intersects
- ST_IsEmpty
- ST_IsSimple
- ST_IsValid
- ST_Length
- ST_NumGeometries
- ST_NumPoints
- ST_Overlaps
- ST_PointFromGeoHash
- ST_PointN

- ST_Relate
- ST_SRID
- ST_SymDifference
- ST_Touches
- ST_Transform
- ST_Union
- ST_Within
- ST_X
- ST_XMax
- ST_XMin
- ST_Y
- ST_YMax
- ST_YMin
- STV_AsGeoJSON
- STV_Create_Index
- STV_Describe_Index
- STV_Drop_Index
- STV_DWithin
- STV_Export2Shapefile
- STV_Extent
- STV_ForceLHR
- STV_Geography
- STV_GeographyPoint
- STV_Geometry
- STV_GeometryPoint
- STV_GetExportShapefileDirectory
- STV_Intersect scalar function
- STV_Intersect transform function
- STV_IsValidReason
- STV_LineStringPoint
- STV_MemSize
- STV_NN
- STV_PolygonPoint
- STV_Refresh_Index
- STV_Rename_Index
- STV_Reverse
- STV_SetExportShapefileDirectory
- STV_ShpCreateTable
- STV_ShpSource and STV_ShpParser

ST_Area

Calculates the area of a spatial object.

The units are:

- GEOMETRY objects: spatial reference system identifier (SRID) units
- GEOGRAPHY objects: square meters

Behavior type

Immutable

Syntax

```
ST_Area( g )
```

Arguments

g

Spatial object for which you want to calculate the area, type GEOMETRY or GEOGRAPHY

Returns

FLOAT

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes
Polygon	Yes	Yes
Multipolygon	Yes	Yes
GeometryCollection	Yes	No

Examples

The following examples show how to use ST_Area.

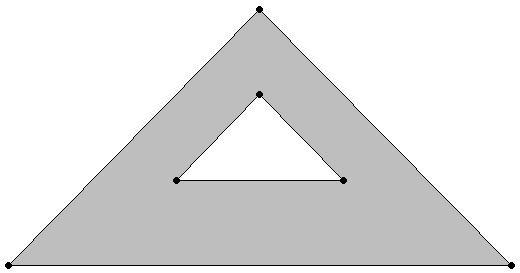
Calculate the area of a polygon:

```
=> SELECT ST_Area(ST_GeomFromText('POLYGON((0 0,1 0,1 1,0 1,0 0))'));
 ST_Area
---------
       1
(1 row)
```

Calculate the area of a multipolygon:

```
=> SELECT ST_Area(ST_GeomFromText('MultiPolygon(((0 0,1 0,1 1,0 1,0 0)),
   ((2 2,2 3,4 6,3 3,2 2)))'));
 ST_Area
---------
     3
(1 row)
```

Suppose the polygon has a hole, as in the following figure.



Calculate the area, excluding the area of the hole:

```
=> SELECT ST_Area(ST_GeomFromText('POLYGON((2 2,5 5,8 2,2 2),
   (4 3,5 4,6 3,4 3))'));
 ST_Area
---------
       8
(1 row)
```

Calculate the area of a geometry collection:

```
=> SELECT ST_Area(ST_GeomFromText('GEOMETRYCOLLECTION(POLYGON((20.5 20.45,
  20.51 20.52,20.69 20.32,20.5 20.45)),POLYGON((10 20,30 40,25 50,10 20)))'));
 ST_Area
----------
 150.0073
(1 row)
```

Calculate the area of a geography object:

```
=> SELECT ST_Area(ST_GeographyFromText('POLYGON((20.5 20.45,20.51 20.52,
  20.69 20.32,20.5 20.45))'));
    ST_Area
------------------
 84627437.116037
(1 row)
```

ST_AsBinary

Creates the Well-Known Binary (WKB) representation of a spatial object. Use this function when you need to convert an object to binary form for porting spatial data to or from other applications.

The Open Geospatial Consortium (OGC) defines the format of a WKB representation in the Simple Feature Access Part 1 - Common Architecture specification.

Behavior type

Immutable

Syntax

```
ST_AsBinary( g )
```

Arguments

g

Spatial object for which you want the WKB, type GEOMETRY or GEOGRAPHY

Returns

LONG VARBINARY

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	Yes	No	No

Examples

The following example shows how to use ST_AsBinary.

Retrieve WKB and WKT representations:

```
=> CREATE TABLE locations (id INTEGER, name VARCHAR(100), geom1 GEOMETRY(800), geom2 GEOGRAPHY);
CREATE TABLE
=> COPY locations
    (id, geom1x FILLER LONG VARCHAR(800), geom1 AS ST_GeomFromText(geom1x), geom2x FILLER LONG VARCHAR (800),
     geom2 AS ST_GeographyFromText(geom2x))
   FROM stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1|POINT(2 3)|
>> 2|LINESTRING(2 4,1 5)|
>> 3||POLYGON((-70.96 43.27,-70.67 42.95,-66.90 44.74,-67.81 46.08,-67.81 47.20,-69.22 47.43,-71.09 45.25,-70.96 43.27))
>> \.
=> SELECT id, ST_AsText(geom1),ST_AsText(geom2) FROM locations ORDER BY id ASC;
 id |       ST_AsText       |                ST_AsText
----+-----------------------+-------------------------------------------
  1 | POINT (2 3)           |
  2 | LINESTRING (2 4, 1 5) |
  3 |                       | POLYGON ((-70.96 43.27, -70.67 42.95, -66.9 44.74, -67.81 46.08, -67.81 47.2, -69.22 47.43, -71.09 45.25, -70.96 43.27))
=> SELECT id, ST_AsBinary(geom1),ST_AsBinary(geom2) FROM locations ORDER BY id ASC;
.
.
.
(3 rows)
```

Calculate the length of a WKB using the Vertica SQL function LENGTH :

```
=> SELECT LENGTH(ST_AsBinary(St_GeomFromText('POLYGON ((-1 2, 0 3, 1 2,
                                          0 1, -1 2))')));
 LENGTH
--------
     93
(1 row)
```

See also

ST_AsText

ST_AsText

Creates the Well-Known Text (WKT) representation of a spatial object. Use this function when you need to specify a spatial object in ASCII form.

The Open Geospatial Consortium (OGC) defines the format of a WKT string in the Simple Feature Access Part 1 - Common Architecture specification.

Behavior type

Immutable

Syntax

```
ST_AsText( g )
```

Arguments

g

Spatial object for which you want the WKT string, type GEOMETRY or GEOGRAPHY

Returns

LONG VARCHAR

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes

Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	Yes	No	No

Examples

The following example shows how to use ST_AsText.

Retrieve WKB and WKT representations:

```
=> CREATE TABLE locations (id INTEGER, name VARCHAR(100), geom1 GEOMETRY(800),
   geom2 GEOGRAPHY);
CREATE TABLE
=> COPY locations
    (id, geom1x FILLER LONG VARCHAR(800), geom1 AS ST_GeomFromText(geom1x), geom2x FILLER LONG VARCHAR (800),
     geom2 AS ST_GeographyFromText(geom2x))
    FROM stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1|POINT(2 3)|
>> 2|LINESTRING(2 4,1 5)|
>> 3||POLYGON((-70.96 43.27,-70.67 42.95,-66.90 44.74,-67.81 46.08,-67.81 47.20,-69.22 47.43,-71.09 45.25,-70.96 43.27))
>> \.
=> SELECT id, ST_AsText(geom1),ST_AsText(geom2) FROM locations ORDER BY id ASC;
 id |       ST_AsText       |                ST_AsText
----+-----------------------+-----------------------------------------
  1 | POINT (2 3)           |
  2 | LINESTRING (2 4, 1 5) |
  3 |                       | POLYGON ((-70.96 43.27, -70.67 42.95, -66.9 44.74, -67.81 46.08, -67.81 47.2, -69.22 47.43, -71.09 45.25, -70.96 43.27))
(3 rows)
```

Calculate the length of a WKT using the Vertica SQL function LENGTH :

```
=> SELECT LENGTH(ST_AsText(St_GeomFromText('POLYGON ((-1 2, 0 3, 1 2,
                                          0 1, -1 2))')));
 LENGTH
--------
     37
(1 row)
```

See also

- ST_AsBinary

ST_Boundary

Calculates the boundary of the specified GEOMETRY object. An object's boundary is the set of points that define the limit of the object.

For a linestring, the boundary is the start and end points. For a polygon, the boundary is a linestring that begins and ends at the same point.

Behavior type

Immutable

Syntax

```
ST_Boundary( g )
```

Arguments

g

Spatial object for which you want the boundary, type GEOMETRY

Returns

GEOMETRY

Supported data types

Data Type	GEOMETRY
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	No

Examples

The following examples show how to use ST_Boundary.

Returns a linestring that represents the boundary:

```
=> SELECT ST_AsText(ST_Boundary(ST_GeomFromText('POLYGON((-1 -1,2 2,
  0 1,-1 -1))')));
 ST_AsText
--------------
LINESTRING(-1 -1, 2 2, 0 1, -1 -1)
(1 row)
```

Returns a multilinestring that contains the boundaries of both polygons:

```
=> SELECT ST_AsText(ST_Boundary(ST_GeomFromText('POLYGON((2 2,5 5,8 2,2 2),
  (4 3,5 4,6 3,4 3))')));
                          ST_AsText
-------------------------------------------------------------------
 MULTILINESTRING ((2 2, 5 5, 8 2, 2 2), (4 3, 5 4, 6 3, 4 3))
(1 row)
```

The boundary of a linestring is its start and end points:

```
=> SELECT ST_AsText(ST_Boundary(ST_GeomFromText(
  'LINESTRING(1 1,2 2,3 3,4 4)')));
      ST_AsText
-----------------------
 MULTIPOINT (1 1, 4 4)
(1 row)
```

A closed linestring has no boundary because it has no start and end points:

```
=> SELECT ST_AsText(ST_Boundary(ST_GeomFromText(
  'LINESTRING(1 1,2 2,3 3,4 4,1 1)')));
   ST_AsText
------------------
 MULTIPOINT EMPTY
(1 row)
```

ST_Buffer

Creates a GEOMETRY object greater than or equal to a specified distance from the boundary of a spatial object. The distance is measured in Cartesian coordinate units. ST_Buffer does not accept a distance size greater than +1e15 or less than –1e15.

Behavior type

Immutable

Syntax

```
ST_Buffer( g, d )
```

Arguments

g

Spatial object for which you want to calculate the buffer, type GEOMETRY

d

Distance from the object in Cartesian coordinate units, type FLOAT

Returns

GEOMETRY

Supported data types

Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

Usage tips

- If you specify a positive distance, ST_Buffer returns a polygon that represents the points within or equal to the distance outside the object. If you specify a negative distance, ST_Buffer returns a polygon that represents the points within or equal to the distance inside the object.
- For points, multipoints, linestrings, and multilinestrings, if you specify a negative distance, ST_Buffer returns an empty polygon.
- The Vertica Place version of ST_Buffer returns the buffer as a polygon, so the buffer object has corners at its vertices. It does not contain rounded corners.

Examples

The following example shows how to use ST_Buffer.

Returns a GEOMETRY object:

```
=> SELECT ST_AsText(ST_Buffer(ST_GeomFromText('POLYGON((0 1,1 4,4 3,0 1))'),1));
                                ST_AsText
-------------------------------------------------------------------------------
POLYGON ((-0.188847498856 -0.159920845081, -1.12155598386 0.649012935089, 0.290814745534 4.76344136152,
 0.814758063466 5.02541302048, 4.95372324225 3.68665254814, 5.04124517538 2.45512549204, -0.188847498856 -0.159920845081))
(1 row)
```

ST_Centroid

Calculates the geometric center—the centroid—of a spatial object. If points or linestrings or both are present in a geometry with polygons, only the polygons contribute to the calculation of the centroid. Similarly, if points are present with linestrings, the points do not contribute to the calculation of the centroid.

To calculate the centroid of a GEOGRAPHY object, see the examples for STV_Geometry and STV_Geography .

Behavior type

Immutable

Syntax

```
ST_Centroid( g )
```

Arguments

g

Spatial object for which you want to calculate the centroid, type GEOMETRY

Returns

GEOMETRY (POINT only)

Supported data types

Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

Examples

The following examples show how to use ST_Centroid.

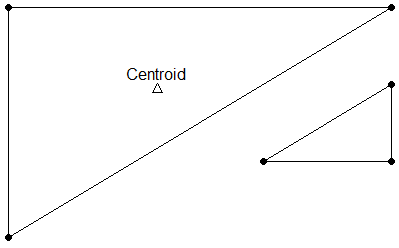
Calculate the centroid for a polygon:

```
=> SELECT ST_AsText(ST_Centroid(ST_GeomFromText('POLYGON((-1 -1,2 2,-1 2,
  -1 -1))')));
 ST_AsText
------------
 POINT (-0 1)
(1 row)
```

Calculate the centroid for a multipolygon:

```
=> SELECT ST_AsText(ST_Centroid(ST_GeomFromText('MULTIPOLYGON(((1 0,2 1,2 0,
  1 0)),((-1 -1,2 2,-1 2,-1 -1)))')));
              ST_AsText
---------------------------------------
 POINT (0.166666666667 0.933333333333)
(1 row)
```

This figure shows the centroid for the multipolygon.



ST_Contains

Determines if a spatial object is entirely inside another spatial object without existing only on its boundary. Both arguments must be the same spatial data type. Either specify two GEOMETRY objects or two GEOGRAPHY objects.

If an object such as a point or linestring only exists along a spatial object's boundary, then ST_Contains returns false. The interior of a linestring is all the points on the linestring except the start and end points.

ST_Contains(*g1* , *g2*) is functionally equivalent to ST_Within(*g2* , *g1*) .

GEOGRAPHY Polygons with a vertex or border on the International Date Line (IDL) or the North or South pole are not supported.

Behavior type

Immutable

Syntax

```
ST_Contains( g1, g2
             [USING PARAMETERS spheroid={true | false}] )
```

Arguments

g1

Spatial object, type GEOMETRY or GEOGRAPHY

g2

Spatial object, type GEOMETRY or GEOGRAPHY

Parameters

spheroid = {true | false}

(Optional) BOOLEAN that specifies whether to use a perfect sphere or WGS84.
Default : False

Returns

BOOLEAN

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	No	No
Linestring	Yes	Yes	No
Multilinestring	Yes	No	No
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	No
GeometryCollection	Yes	No	No

Compatible GEOGRAPHY pairs:

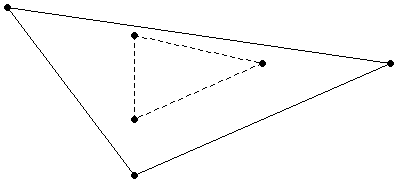
Data Type	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point-Point	Yes	No
Linestring-Point	Yes	No
Polygon-Point	Yes	Yes
Multipolygon-Point	Yes	No

Examples

The following examples show how to use ST_Contains.

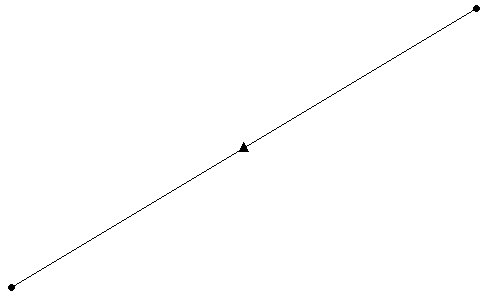
The first polygon does not completely contain the second polygon:

```
=> SELECT ST_Contains(ST_GeomFromText('POLYGON((0 2,1 1,0 -1,0 2))'),
   ST_GeomFromText('POLYGON((-1 3,2 1,0 -3,-1 3))'));
 ST_Contains
-------------
 f
(1 row)
```



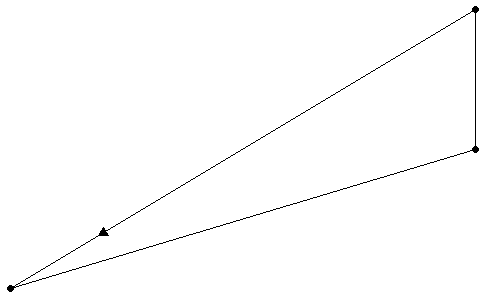
If a point is on a linestring, but not on an end point:

```
=> SELECT ST_Contains(ST_GeomFromText('LINESTRING(20 20,30 30)'),
   ST_GeomFromText('POINT(25 25)'));
 ST_Contains
-------------
 t
(1 row)
```



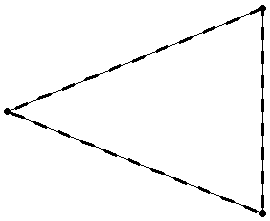
If a point is on the boundary of a polygon:

```
=> SELECT ST_Contains(ST_GeographyFromText('POLYGON((20 20,30 30,30 25,20 20))'),
   ST_GeographyFromText('POINT(20 20)'));
 ST_Contains
-------------
 f
(1 row)
```



Two spatially equivalent polygons:

```
=> SELECT ST_Contains (ST_GeomFromText('POLYGON((-1 2, 0 3, 0 1, -1 2))'),
   ST_GeomFromText('POLYGON((0 3, -1 2, 0 1, 0 3))'));
 ST_Contains
-------------
 t
(1 row)
```



See also

- ST_Overlaps
- ST_Within

ST_ConvexHull

Calculates the smallest convex GEOMETRY object that contains a GEOMETRY object.

Behavior type

Immutable

Syntax

```
ST_ConvexHull( g )
```

Arguments

g

Spatial object for which you want the convex hull, type GEOMETRY

Returns

GEOMETRY

Supported data types

Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

Examples

The following examples show how to use ST_ConvexHull.

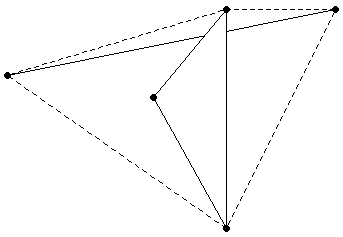
For a pair of points in a geometry collection:

```
=> SELECT ST_AsText(ST_ConvexHull(ST_GeomFromText('GEOMETRYCOLLECTION(
   POINT(1 1),POINT(0 0))')));
      ST_AsText
-----------------------
 LINESTRING (1 1, 0 0)
(1 row)
```

For a geometry collection:

```
=> SELECT ST_AsText(ST_ConvexHull(ST_GeomFromText('GEOMETRYCOLLECTION(
  LINESTRING(2.5 3,-2 1.5), POLYGON((0 1,1 3,1 -2,0 1)))')));
              ST_AsText
-----------------------------------------------
 POLYGON ((1 -2, -2 1.5, 1 3, 2.5 3, 1 -2))
(1 row)
```

The solid lines represent the original geometry collection and the dashed lines represent the convex hull.



ST_Crosses

Determines if one GEOMETRY object spatially crosses another GEOMETRY object. If two objects touch only at a border, ST_Crosses returns FALSE.

Two objects spatially cross when both of the following are true:

- The two objects have some, but not all, interior points in common.
- The dimension of the result of their intersection is less than the maximum dimension of the two objects.

Behavior type

Immutable

Syntax

```
ST_Crosses( g1, g2 )
```

Arguments

g1

Spatial object, type GEOMETRY

g2

Spatial object, type GEOMETRY

Returns

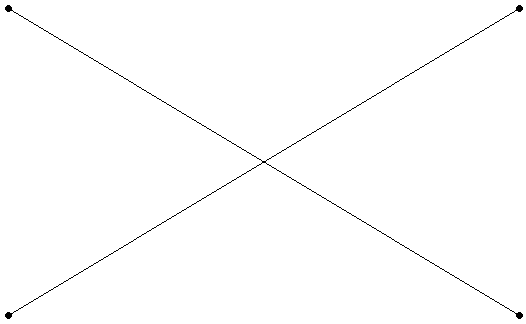
BOOLEAN

Supported data types

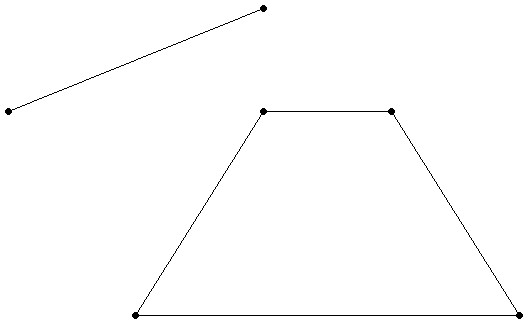
Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

Examples

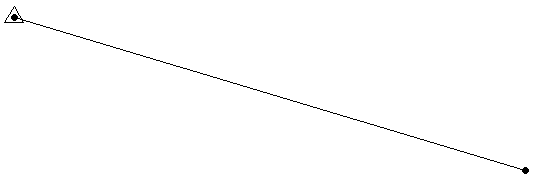
The following examples show how to use ST_Crosses.



```
=> SELECT ST_Crosses(ST_GeomFromText('LINESTRING(-1 3,1 4)'),
   ST_GeomFromText('LINESTRING(-1 4,1 3)'));
 ST_Crosses
------------
 t
(1 row)
```



```
=> SELECT ST_Crosses(ST_GeomFromText('LINESTRING(-1 1,1 2)'),
   ST_GeomFromText('POLYGON((1 1,0 -1,3 -1,2 1,1 1))'));
 ST_Crosses
------------
 f
(1 row)
```



```
=> SELECT ST_Crosses(ST_GeomFromText('POINT(-1 4)'),
   ST_GeomFromText('LINESTRING(-1 4,1 3)'));
ST_ Crosses
------------
 f
(1 row)
```

ST_Difference

Calculates the part of a spatial object that does not intersect with another spatial object.

Behavior type

Immutable

Syntax

```
ST_Difference( g1, g2 )
```

Arguments

g1

Spatial object, type GEOMETRY

g2

Spatial object, type GEOMETRY

Returns

GEOMETRY

Supported data types

Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

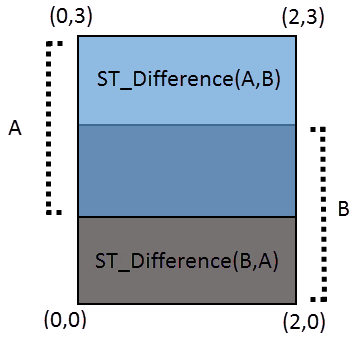
Examples

The following examples show how to use ST_Difference.

Two overlapping linestrings:

```
=> SELECT ST_AsText(ST_Difference(ST_GeomFromText('LINESTRING(0 0,0 2)'),
   ST_GeomFromText('LINESTRING(0 1,0 2)')));
      ST_AsText
----------------------
 LINESTRING (0 0, 0 1)
(1 row)
=> SELECT ST_AsText(ST_Difference(ST_GeomFromText('LINESTRING(0 0,0 3)'),
   ST_GeomFromText('LINESTRING(0 1,0 2)')));
               ST_AsText
-----------------------------------------
 MULTILINESTRING ((0 0, 0 1), (0 2, 0 3))
(1 row)
```

Two overlapping polygons:




```
=> SELECT ST_AsText(ST_Difference(ST_GeomFromText('POLYGON((0 1,0 3,2 3,2 1,0 1))'),
   ST_GeomFromText('POLYGON((0 0,0 2,2 2,2 0,0 0))')));
         ST_AsText
-------------------------------------
 POLYGON ((0 2, 0 3, 2 3, 2 2, 0 2))
(1 row)
```

Two non-intersecting polygons:

```
=> SELECT ST_AsText(ST_Difference(ST_GeomFromText('POLYGON((1 1,1 3,3 3,3 1,
  1 1))'),ST_GeomFromText('POLYGON((1 5,1 7,-1 7,-1 5,1 5))')));
          ST_AsText
--------------------------------------
 POLYGON ((1 1, 1 3, 3 3, 3 1, 1 1))
(1 row)
```

ST_Disjoint

Determines if two GEOMETRY objects do not intersect or touch.

If ST_Disjoint returns TRUE for a pair of GEOMETRY objects, ST_Intersects returns FALSE for the same two objects.

GEOGRAPHY Polygons with a vertex or border on the International Date Line (IDL) or the North or South pole are not supported.

Behavior type

Immutable

Syntax

```
ST_Disjoint( g1, g2
                [USING PARAMETERS spheroid={true | false}] )
```

Arguments

g1

Spatial object, type GEOMETRY

g2

Spatial object, type GEOMETRY

Parameters

spheroid = {true | false}

(Optional) BOOLEAN that specifies whether to use a perfect sphere or WGS84.
Default : False

Returns

BOOLEAN

Supported data types

Data Type	**GEOMETRY**	**GEOGRAPHY (WGS84)**
Point	Yes	Yes
Multipoint	Yes	No
Linestring	Yes	No
Multilinestring	Yes	No
Polygon	Yes	Yes
Multipolygon	Yes	No
GeometryCollection	Yes	No

Compatible GEOGRAPHY pairs:

Data Type
GEOGRAPHY (WGS84)

Point-Point
No

Linestring-Point
No

Polygon-Point

Yes

Multipolygon-Point

No

Examples

The following examples show how to use ST_Disjoint.

Two non-intersecting or touching polygons:

```
=> SELECT ST_Disjoint (ST_GeomFromText('POLYGON((-1 2,0 3,0 1,-1 2))'),
   ST_GeomFromText('POLYGON((1 0, 1 1, 2 2, 1 0))'));
 ST_Disjoint
-------------
 t
(1 row)
```

Two intersecting linestrings:

```
=> SELECT ST_Disjoint(ST_GeomFromText('LINESTRING(-1 2,0 3)'),
   ST_GeomFromText('LINESTRING(0 2,-1 3)'));
 ST_Disjoint
-------------
 f
(1 row)
```

Two polygons touching at a single point:

```
=> SELECT ST_Disjoint (ST_GeomFromText('POLYGON((-1 2, 0 3, 0 1, -1 2))'),
   ST_GeomFromText('POLYGON((0 2, 1 1, 1 2, 0 2))'));
 ST_Disjoint
--------------
 f
(1 row)
```

See also

- ST_Intersects

ST_Distance

Calculates the shortest distance between two spatial objects. For GEOMETRY objects, the distance is measured in Cartesian coordinate units. For GEOGRAPHY objects, the distance is measured in meters.

Parameters g1 and g2 must be both GEOMETRY objects or both GEOGRAPHY objects.

Behavior type

Immutable

Syntax

```
ST_Distance( g1, g2
    [USING PARAMETERS spheroid={ true | false } ] )
```

Arguments

g1

Spatial object, type GEOMETRY or GEOGRAPHY

g2

Spatial object, type GEOMETRY or GEOGRAPHY

Parameters

spheroid = { true | false }

(Optional) BOOLEAN that specifies whether to use a perfect sphere or WGS84.
Default : False

Returns

FLOAT

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	No
Multipolygon	Yes	Yes	No
GeometryCollection	Yes	No	No

Compatible GEOGRAPHY pairs:

Data Type	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point-Point	Yes	Yes
Linestring-Point	Yes	Yes
Multilinestring-Point	Yes	Yes
Polygon-Point	Yes	No
Multipoint-Point	Yes	Yes
Multipoint-Multilinestring	Yes	No
Multipolygon-Point	Yes	No

Recommendations

Vertica recommends pruning invalid data before using ST_Distance. Invalid geography values could return non-guaranteed results.

Examples

The following examples show how to use ST_Distance.

Distance between two polygons:

```
=> SELECT ST_Distance(ST_GeomFromText('POLYGON((-1 -1,2 2,0 1,-1 -1))'),
                      ST_GeomFromText('POLYGON((5 2,7 4,5 5,5 2))'));
 ST_Distance
-------------
           3
(1 row)
```

Distance between a point and a linestring in meters:

```
=> SELECT ST_Distance(ST_GeographyFromText('POINT(31.75 31.25)'),
    ST_GeographyFromText('LINESTRING(32 32,32 35,40.5 35,32 35,32 32)'));
   ST_Distance
-------------------
 86690.3950562969
(1 row)
```

ST_Envelope

Calculates the minimum bounding rectangle that contains the specified GEOMETRY object.

Behavior type

Immutable

Syntax

```
ST_Envelope( g )
```

Arguments

g

Spatial object for which you want to find the minimum bounding rectangle, type GEOMETRY

Returns

GEOMETRY

Supported data types

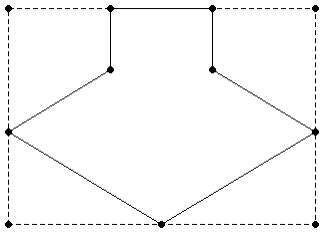
Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

Examples

The following example shows how to use ST_Envelope.

Returns the minimum bounding rectangle:

```
=> SELECT ST_AsText(ST_Envelope(ST_GeomFromText('POLYGON((0 0,1 1,1 2,2 2,
  2 1,3 0,1.5 -1.5,0 0))')));
              ST_AsText
-------------------------------------------
 POLYGON ((0 -1.5, 3 -1.5, 3 2, 0 2, 0 -1.5))
(1 row)
```



ST_Equals

Determines if two spatial objects are spatially equivalent. The coordinates of the two objects and their WKT/WKB representations must match exactly for ST_Equals to return TRUE.

The order of the points do not matter in determining spatial equivalence:

- LINESTRING(1 2, 4 3) equals LINESTRING(4 3, 1 2).
- POLYGON ((0 0, 1 1, 1 2, 2 2, 2 1, 3 0, 1.5 -1.5, 0 0)) equals POLYGON((1 1 , 1 2, 2 2, 2 1, 3 0, 1.5 -1.5, 0 0, 1 1)).

- MULTILINESTRING((1 2, 4 3),(0 0, -1 -4)) equals MULTILINESTRING((0 0, -1 -4),(1 2, 4 3)).

Coordinates are stored as FLOAT types. Thus, rounding errors are expected when importing Well-Known Text (WKT) values because the limitations of floating-point number representation.

g1 and g2 must both be GEOMETRY objects or both be GEOGRAPHY objects. Also, g1 and g2 cannot both be of type GeometryCollection.

Behavior type

Immutable

Syntax

```
ST_Equals( g1, g2 )
```

Arguments

g1

Spatial object to compare to g2 , type GEOMETRY or GEOGRAPHY

g2

Spatial object to compare to g1 , type GEOMETRY or GEOGRAPHY

Returns

BOOLEAN

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	No	No	No

Examples

The following examples show how to use ST_Equals.

Two linestrings:

```
=> SELECT ST_Equals (ST_GeomFromText('LINESTRING(-1 2, 0 3)'),
   ST_GeomFromText('LINESTRING(0 3, -1 2)'));
 ST_Equals
--------------
 t
(1 row)
```

Two polygons:

```
=> SELECT ST_Equals (ST_GeographyFromText('POLYGON((43.22 42.21,40.3 39.88,
   42.1 50.03,43.22 42.21))'),ST_GeographyFromText('POLYGON((43.22 42.21,
   40.3 39.88,42.1 50.31,43.22 42.21))'));
 ST_Equals
--------------
 f
(1 row)
```

ST_GeographyFromText

Converts a Well-Known Text (WKT) string into its corresponding GEOGRAPHY object. Use this function to convert a WKT string into the format expected by the Vertica Place functions.

A GEOGRAPHY object is a spatial object with coordinates (longitude, latitude) defined on the surface of the earth. Coordinates are expressed in degrees (longitude, latitude) from reference planes dividing the earth.

The maximum size of a GEOGRAPHY object is 10 MB. If you pass a WKT to ST_GeographyFromText, the result is a spatial object whose size is greater than 10 MB, ST_GeographyFromText returns an error.

The Open Geospatial Consortium (OGC) defines the format of a WKT string in Section 7 in the Simple Feature Access Part 1 - Common Architecture specification.

Behavior type

Immutable

Syntax

```
ST_GeographyFromText( wkt [ USING PARAMETERS ignore_errors={'y'|'n'} ] )
```

Arguments

wkt

Well-Known Text (WKT) string of a GEOGRAPHY object, type LONG VARCHAR

ignore_errors

(Optional) ST_GeographyFromText returns the following, based on the parameters supplied:

- NULL—If wkt is invalid and ignore_errors = 'y' .
- Error—If wkt is invalid and ignore_errors = 'n' or is unspecified.

Returns

GEOGRAPHY

Supported data types

Data Type	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes
Polygon	Yes	Yes
Multipolygon	Yes	Yes
GeometryCollection	No	No

Examples

The following example shows how to use ST_GeographyFromText.

Convert WKT into a GEOGRAPHY object:

```
=> CREATE TABLE wkt_ex (g GEOGRAPHY);
 CREATE TABLE
=> INSERT INTO wkt_ex VALUES(ST_GeographyFromText('POLYGON((1 2,3 4,2 3,1 2))'));
 OUTPUT
--------
      1
(1 row)
```

ST_GeographyFromWKB

Converts a Well-Known Binary (WKB) value into its corresponding GEOGRAPHY object. Use this function to convert a WKB into the format expected by Vertica Place functions.

A GEOGRAPHY object is a spatial object defined on the surface of the earth. Coordinates are expressed in degrees (longitude, latitude) from reference planes dividing the earth. All calculations are in meters.

The maximum size of a GEOGRAPHY object is 10 MB. If you pass a WKB to ST_GeographyFromWKB that results in a spatial object whose size is greater than 10 MB, ST_GeographyFromWKB returns an error.

The Open Geospatial Consortium (OGC) defines the format of a WKB representation in Section 8 in the Simple Feature Access Part 1 - Common Architecture specification.

Behavior type

Immutable

Syntax

```
ST_GeographyFromWKB( wkb [ USING PARAMETERS ignore_errors={'y'|'n'} ] )
```

Arguments

wkb

Well-Known Binary (WKB) value of a GEOGRAPHY object, type LONG VARBINARY

ignore_errors

(Optional) ST_GeographyFromWKB returns the following, based on the parameters supplied:

- NULL—If wkb is invalid and ignore_errors = 'y' .
- Error—If wkb is invalid and ignore_errors = 'n' or is unspecified.

Returns

GEOGRAPHY

Supported data types

Data Type	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes
Polygon	Yes	Yes
Multipolygon	Yes	Yes
GeometryCollection	No	No

Examples

The following example shows how to use ST_GeographyFromWKB.

Convert WKB into a GEOGRAPHY object:

```
=> CREATE TABLE wkb_ex (g GEOGRAPHY);
CREATE TABLE
=> INSERT INTO wkb_ex VALUES(ST_GeographyFromWKB(X'0103000000010000000 ... );
 OUTPUT
--------
      1
(1 row)
```

ST_GeoHash

Returns a GeoHash in the shape of the specified geometry.

Behavior type

Immutable

Syntax

```
ST_GeoHash( SpatialObject [ USING PARAMETERS numchars=n] )
```

Arguments

Spatial object

A GEOMETRY or GEOGRAPHY spatial object. Inputs must be in polar coordinates (-180 <= x <= 180 and -90 <= y <= 90) for all points inside the given geometry.

n

Specifies the length, in characters, of the returned GeoHash.

Returns

GEOHASH

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	Yes	No	No

Examples

The following examples show how to use ST_PointFromGeoHash.

Generate a full precision GeoHash for the specified geometry:

```
=> SELECT ST_GeoHash(ST_GeographyFromText('POINT(3.14 -1.34)'));
ST_GeoHash
----------------------
kpf0rkn3zmcswks75010
(1 row)
```

Generate a GeoHash based on the first five characters of the specified geometry:

```
=>  select ST_GeoHash(ST_GeographyFromText('POINT(3.14 -1.34)')USING PARAMETERS numchars=5);
ST_GeoHash
------------
kpf0r
(1 row)
```

ST_GeometryN

Returns the *n* th geometry within a geometry object.

If *n* is out of range of the index, then NULL is returned.

Behavior type

Immutable

Syntax

```
ST_GeometryN( g , n )
```

Arguments

g

Spatial object of type GEOMETRY.

n

The geometry's index number, 1-based.

Returns

GEOMETRY

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	No	No	No

Examples

The following examples show how to use ST_GeometryN.

Return the second geometry in a multipolygon:

```
=> CREATE TABLE multipolygon_geom (gid int, geom GEOMETRY(1000));
CREATE TABLE
=> COPY multipolygon_geom(gid, gx FILLER LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>9|MULTIPOLYGON(((2 6, 2 9, 6 9, 7 7, 4 6, 2 6)),((0 0, 0 5, 1 0, 0 0)),((0 2, 2 5, 4 5, 0 2)))
>>\.
=> SELECT gid, ST_AsText(ST_GeometryN(geom, 2)) FROM multipolygon_geom;
 gid |           ST_AsText
-----+--------------------------------
   9 | POLYGON ((0 0, 0 5, 1 0, 0 0))
(1 row)
```

Return all the geometries within a multipolygon:

```
=> CREATE TABLE multipolygon_geom (gid int, geom GEOMETRY(1000));
CREATE TABLE
=> COPY multipolygon_geom(gid, gx FILLER LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>9|MULTIPOLYGON(((2 6, 2 9, 6 9, 7 7, 4 6, 2 6)),((0 0, 0 5, 1 0, 0 0)),((0 2, 2 5, 4 5, 0 2)))
>>\.
=> CREATE TABLE series_numbers (numbs int);
CREATE TABLE
=> COPY series_numbers FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1
>> 2
>> 3
>> 4
>> 5
>> \.
=> SELECT numbs, ST_AsText(ST_GeometryN(geom, numbs))
   FROM multipolygon_geom, series_numbers
   WHERE ST_AsText(ST_GeometryN(geom, numbs)) IS NOT NULL
   ORDER BY numbs ASC;
 numbs |              ST_AsText
-------+---------------------------------------
     1 | POLYGON ((2 6, 2 9, 6 9, 7 7, 4 6, 2 6))
     2 | POLYGON ((0 0, 0 5, 1 0, 0 0))
     3 | POLYGON ((0 2, 2 5, 4 5, 0 2))
(3 rows)
```

See also

ST_NumGeometries

ST_GeometryType

Determines the class of a spatial object.

Behavior type

Immutable

Syntax

```
ST_GeometryType( g )
```

Arguments

g

Spatial object for which you want the class, type GEOMETRY or GEOGRAPHY

Returns

VARCHAR

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes
Polygon	Yes	Yes

Multipolygon	Yes	Yes
GeometryCollection	Yes	No

Examples

The following example shows how to use ST_GeometryType.

Returns spatial class:

```
=> SELECT ST_GeometryType(ST_GeomFromText('GEOMETRYCOLLECTION(LINESTRING(1 1,
  2 2), POLYGON((1 3,4 5,2 2,1 3)))'));
   ST_GeometryType
-----------------------
 ST_GeometryCollection
(1 row)
```

ST_GeomFromGeoHash

Returns a polygon in the shape of the specified GeoHash.

Behavior type

Immutable

Syntax

```
ST_GeomFromGeoHash(GeoHash)
```

Arguments

GeoHash

A valid GeoHash string of arbitrary length.

Returns

GEOGRAPHY

Examples

The following examples show how to use ST_GeomFromGeoHash.

Converts a GeoHash string to a Geography object and back to a GeoHash

```
=>  SELECT ST_GeoHash(ST_GeomFromGeoHash('vert1c9'));
ST_GeoHash
--------------------
vert1c9
(1 row)
```

Returns a polygon of the specified GeoHash and uses ST_AsText to convert the polygon, rectangle map tile, into Well-Known Text:

```
=>  SELECT ST_AsText(ST_GeomFromGeoHash('drt3jj9n4dpcbcdef'));
ST_AsText
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
POLYGON ((-71.1459699298 42.3945346513, -71.1459699297 42.3945346513, -71.1459699297 42.3945346513, -71.1459699298 42.3945346513, -71.1459699298 42.3945346513))
(1 row)
```

Returns multiple polygons and their areas for the specified GeoHashes. The polygon for the high level GeoHash (1234) has a significant area, while the low level GeoHash (1234567890bcdefhjkmn) has an area of zero.

```
=>  SELECT ST_Area(short) short_area, ST_AsText(short) short_WKT, ST_Area(long) long_area, ST_AsText(long) long_WKT from (SELECT
ST_GeomFromGeoHash('1234') short, ST_GeomFromGeoHash('1234567890bcdefhjkmn') long) as foo;
-[ RECORD 1 ]----------------------------------------------------------------------------------------------------------------------------------------------------------------
short_area | 24609762.8991076
short_WKT  | POLYGON ((-122.34375 -88.2421875, -121.9921875 -88.2421875, -121.9921875 -88.06640625, -122.34375 -88.06640625, -122.34375 -
88.2421875))
long_area  | 0
long_WKT   | POLYGON ((-122.196077187 -88.2297377551, -122.196077187 -88.2297377551, -122.196077187 -88.2297377551, -122.196077187 -
88.2297377551, -122.196077187 -88.2297377551))
```

ST_GeomFromGeoJSON

Converts the geometry portion of a GeoJSON record in the standard format into a GEOMETRY object. This function returns an error when you provide a GeoJSON Feature or FeatureCollection object.

Behavior type

Immutable

Syntax

```
ST_GeomFromGeoJSON( geojson [, srid] [ USING PARAMETERS param=value[,...] ] );
```

Arguments

geojson

String containing a GeoJSON GEOMETRY object, type LONG VARCHAR.

Vertica accepts the following GeoJSON key values:

- type
- coordinates
- geometries

Other key values are ignored.

srid

Spatial reference system identifier (SRID) of the GEOMETRY object, type INTEGER.
The SRID is stored in the GEOMETRY object, but does not influence the results of spatial computations.

This argument is optional when not performing operations.

Parameters

ignore_3d

(Optional) Boolean, whether to silently remove 3D and higher-dimensional data from the returned GEOMETRY object or return an error, based on the following values:

- true: Removes 3D and higher-dimensional data from the returned GEOMETRY object.
- false (default): Returns an error when the GeoJSON contains 3D or higher-dimensional data.

ignore_errors

(Optional) Boolean, whether to ignore errors on invalid GeoJSON objects or return an error, based on the following values:

- true: Ignores errors during GeoJSON parsing and returns NULL.
- false (default): Returns an error if GeoJSON parsing fails.

> **Note**
> The ignore_errors setting takes precedence over the ignore_3d setting. For example, if ignore_errors is set to true and ignore_3d is set to false, the function returns NULL if a GeoJSON object contains 3D and higher-dimensional data.

Returns

GEOMETRY

Supported data types

- Point
- Multipoint
- Linestring

- Multilinestring
- Polygon
- Multipolygon
- GeometryCollection

Examples

The following example shows how to use ST_GeomFromGeoJSON.

Validating a single record

The following example validates a ST_GeomFromGeoJSON statement with ST_IsValid . The statement includes the SRID 4326 to indicate that the point data type represents latitude and longitude coordinates, and sets ignore_3d to true to ignore the last value that represents the altitude:

```
=> SELECT ST_IsValid(ST_GeomFromGeoJSON('{"type":"Point","coordinates":[35.3606, 138.7274, 29032]}', 4326 USING PARAMETERS ignore_3d=true));
 ST_IsValid
------------
 t
(1 row)
```

Loading data into a table

The following example processes GeoJSON types from STDIN and stores them in a GEOMETRY data type table column:

1. Create a table named polygons that stores GEOMETRY spatial types:
   ```
   => CREATE TABLE polygons(geom GEOMETRY(1000));
   CREATE TABLE
   ```
2. Use COPY to read supported GEOMETRY data types from STDIN and store them in an object named geom :
   ```
   => COPY polygons(geojson filler VARCHAR(1000), geom as ST_GeomFromGeoJSON(geojson)) FROM STDIN;
   Enter data to be copied followed by a newline.
   End with a backslash and a period on a line by itself.
   >> { "type": "Polygon", "coordinates": [ [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ] ] }
   >> { "type": "Point", "coordinates": [1, 2] }
   >> { "type": "Polygon", "coordinates": [ [ [1, 3], [3, 2], [1, 1], [3, 0], [1, 0], [1, 3] ] ] }
   >> \.
   ```
3. Query the polygons table. The following example uses ST_AsText to return the geom object in its Well-known text (WKT) representation, and uses ST_IsValid to validate each object:
   ```
   => SELECT ST_AsText(geom), ST_IsValid(geom) FROM polygons;
                 ST_AsText                 | ST_IsValid
   ----------------------------------------+------------
    POINT (1 2)                            | t
    POLYGON ((1 3, 3 2, 1 1, 3 0, 1 0, 1 3))     | f
    POLYGON ((100 0, 101 0, 101 1, 100 1, 100 0)) | t
   (3 rows)
   ```

ST_GeomFromText

Converts a Well-Known Text (WKT) string into its corresponding GEOMETRY object. Use this function to convert a WKT string into the format expected by the Vertica Place functions.

A GEOMETRY object is a spatial object defined by the coordinates of a plane. Coordinates are expressed as points on a Cartesian plane (*x* , *y*). SRID values of 0 to 2^{32-1} are valid. SRID values outside of this range will generate an error.

The maximum size of a GEOMETRY object is 10 MB. If you pass a WKT to ST_GeomFromText and the result is a spatial object whose size is greater than 10 MB, ST_GeomFromText returns an error.

The Open Geospatial Consortium (OGC) defines the format of a WKT representation. See section 7 in the Simple Feature Access Part 1 - Common Architecture specification.

Behavior type

Immutable

Syntax

```
ST_GeomFromText( wkt [, srid] [ USING PARAMETERS ignore_errors={'y'|'n'} ])
```

Arguments

wkt

Well-Known Text (WKT) string of a GEOMETRY object, type LONG VARCHAR.

srid

(Optional when not performing operations)

Spatial reference system identifier (SRID) of the GEOMETRY object, type INTEGER.
The SRID is stored in the GEOMETRY object, but does not influence the results of spatial computations.

ignore_errors

(Optional) ST_GeomFromText returns the following, based on parameters supplied:

- NULL—If wkt is invalid and ignore_errors = 'y' .
- Error—If wkt is invalid and ignore_errors = 'n' or is unspecified.

Returns

GEOMETRY

Supported data types

Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	No

Examples

The following example shows how to use ST_GeomFromText.

Convert WKT into a GEOMETRY object:

```
=> SELECT ST_Area(ST_GeomFromText('POLYGON((1 1,2 3,3 5,0 5,1 -2,0 0,1 1))'));
 ST_Area
---------
       6
(1 row)
```

ST_GeomFromWKB

Converts the Well-Known Binary (WKB) value to its corresponding GEOMETRY object. Use this function to convert a WKB into the format expected by many of the Vertica Place functions.

A GEOMETRY object is a spatial object with coordinates (*x* , *y*) defined in the Cartesian plane.

The maximum size of a GEOMETRY object is 10 MB. If you pass a WKB to ST_GeomFromWKB and the result is a spatial object whose size is greater than 10 MB, ST_GeomFromWKB returns an error.

The Open Geospatial Consortium (OGC) defines the format of a WKB representation in section 8 in the Simple Feature Access Part 1 - Common Architecture specification.

Behavior type

Immutable

Syntax

```
ST_GeomFromWKB( wkb[, srid] [ USING PARAMETERS ignore_errors={'y'|'n'} ])
```

Arguments

wkb

Well-Known Binary (WKB) value of a GEOMETRY object, type LONG VARBINARY

srid

(Optional) Spatial reference system identifier (SRID) of the GEOMETRY object, type INTEGER.

The SRID is stored in the GEOMETRY object, but does not influence the results of spatial computations.

ignore_errors

(Optional) ST_GeomFromWKB returns the following, based on the parameters supplied:

- NULL—If wkb is invalid and ignore_errors = 'y' .
- Error—If wkb is invalid and ignore_errors = 'n' or is unspecified.

Returns

GEOMETRY

Supported data types

Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

Examples

The following example shows how to use ST_GeomFromWKB.

Convert GEOMETRY into WKT:

```
=> CREATE TABLE t(g GEOMETRY);
CREATE TABLE
=> INSERT INTO t VALUES(
ST_GeomFromWKB(X'0103000000010000000400000000000000000000000000000000000000000000000000000000000000f
            03f0000000000000000f64ae1c7022db54400000000000000f03f0000000000000000000000000000000000'));
 OUTPUT
--------
      1
(1 row)
=> SELECT ST_AsText(g) from t;
            ST_AsText
-----------------------------------
 POLYGON ((0 0, 1 0, 1e+23 1, 0 0))
(1 row)
```

ST_Intersection

Calculates the set of points shared by two GEOMETRY objects.

Behavior type

Immutable

Syntax

```
ST_Intersection( g1, g2 )
```

Arguments

g1

Spatial object, type GEOMETRY

g2

Spatial object, type GEOMETRY

Returns

GEOMETRY

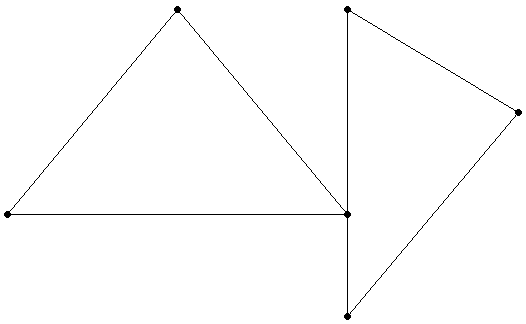
Supported data types

Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

Examples

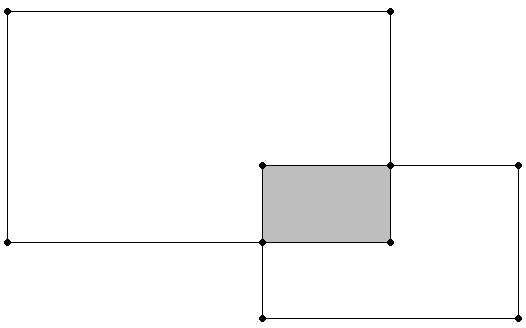
The following examples show how to use ST_Intersection.

Two polygons intersect at a single point:



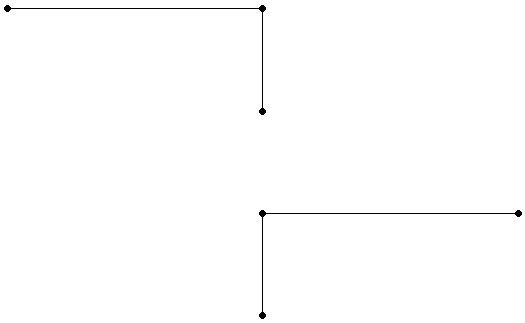
```
=> SELECT ST_AsText(ST_Intersection(ST_GeomFromText('POLYGON((0 2,1 1,0 -1,
   0 2))'),ST_GeomFromText('POLYGON((-1 2,0 0,-2 0,-1 2))')));
    ST_AsText
-----------------
 POINT(0 0)
(1 row)
```

Two polygons:



```
=> SELECT ST_AsText(ST_Intersection(ST_GeomFromText('POLYGON((1 2,1 5,4 5,
   4 2,1 2))'), ST_GeomFromText('POLYGON((3 1,3 3,5 3,5 1,3 1))')));
 ST_AsText
-------------------
POLYGON ((4 3, 4 2, 3 2, 3 3, 4 3))
(1 row)
```

Two non-intersecting linestrings:



```
=> SELECT ST_AsText(ST_Intersection(ST_GeomFromText('LINESTRING(1 1,1 3,3 3)'),
   ST_GeomFromText('LINESTRING(1 5,1 7,-1 7)')));
      ST_AsText
--------------------------
 GEOMETRYCOLLECTION EMPTY
(1 row)
```

ST_Intersects

Determines if two GEOMETRY or GEOGRAPHY objects intersect or touch at a single point. If ST_Disjoint returns TRUE, ST_Intersects returns FALSE for the same GEOMETRY or GEOGRAPHY objects.

GEOGRAPHY Polygons with a vertex or border on the International Date Line (IDL) or the North or South pole are not supported.

Behavior type

Immutable

Syntax

```
ST_Intersects( g1, g2
               [USING PARAMETERS bbox={true | false}, spheroid={true | false}])
```

Arguments

g1

Spatial object, type GEOMETRY

g2

Spatial object, type GEOMETRY

Parameters

bbox = {true | false}

Boolean. Intersects the bounding box of g1 and g2 .

Default: False

spheroid = {true | false}

(Optional) BOOLEAN that specifies whether to use a perfect sphere or WGS84.
Default : False

Returns

BOOLEAN

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (WGS84)

Point	Yes	Yes
Multipoint	Yes	No
Linestring	Yes	No
Multilinestring	Yes	No
Polygon	Yes	Yes
Multipolygon	Yes	No
GeometryCollection	Yes	No

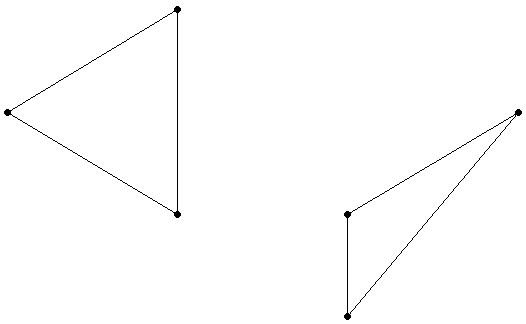
Compatible GEOGRAPHY pairs:

Data Type	**GEOGRAPHY (WGS84)**
Point-Point	No
Linestring-Point	No
Polygon-Point	Yes
Multipolygon-Point	No

Examples

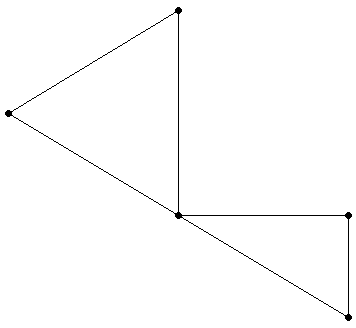
The following examples show how to use ST_Intersects.

Two polygons do not intersect or touch:



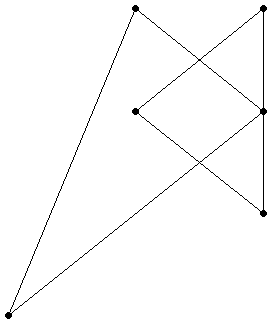
```
=> SELECT ST_Intersects (ST_GeomFromText('POLYGON((-1 2,0 3,0 1,-1 2))'),
   ST_GeomFromText('POLYGON((1 0,1 1,2 2,1 0))'));
 ST_Intersects
---------------
 f
(1 row)
```

Two polygons touch at a single point:



```
=> SELECT ST_Intersects (ST_GeomFromText('POLYGON((-1 2,0 3,0 1,-1 2))'),
   ST_GeomFromText('POLYGON((1 0,1 1,0 1,1 0))'));
 ST_Intersects
---------------
 t
(1 row)
```

Two polygons intersect:



```
=> SELECT ST_Intersects (ST_GeomFromText('POLYGON((-1 2, 0 3, 0 1, -1 2))'),
   ST_GeomFromText('POLYGON((0 2, -1 3, -2 0, 0 2))'));
 ST_Intersects
---------------
 t
(1 row)
```

See also

ST_Disjoint

ST_IsEmpty

Determines if a spatial object represents the empty set. An empty object has no dimension.

Behavior type

Immutable

Syntax

```
ST_IsEmpty( g )
```

Arguments

g

Spatial object, type GEOMETRY or GEOGRAPHY

Returns

BOOLEAN

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	Yes	No	No

Examples

The following example shows how to use ST_IsEmpty.

An empty polygon:

```
=> SELECT ST_IsEmpty(ST_GeomFromText('GeometryCollection EMPTY'));
 ST_IsEmpty
------------
 t
(1 row)
```

ST_IsSimple

Determines if a spatial object does not intersect itself or touch its own boundary at any point.

Behavior type

Immutable

Syntax

```
ST_IsSimple( g )
```

Arguments

g

Spatial object, type GEOMETRY or GEOGRAPHY

Returns

BOOLEAN

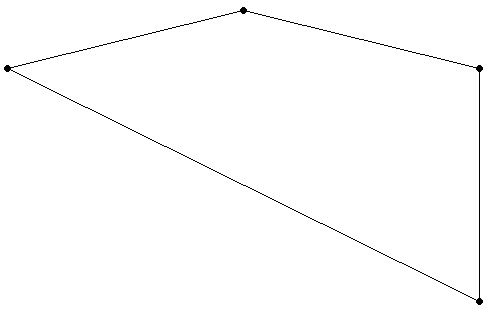
Supported data types

Data Type	**GEOMETRY**	**GEOGRAPHY (Perfect Sphere)**
Point	Yes	Yes
Multipoint	Yes	No
Linestring	Yes	Yes
Multilinestring	Yes	No
Polygon	Yes	Yes
Multipolygon	Yes	No
GeometryCollection	No	No

Examples

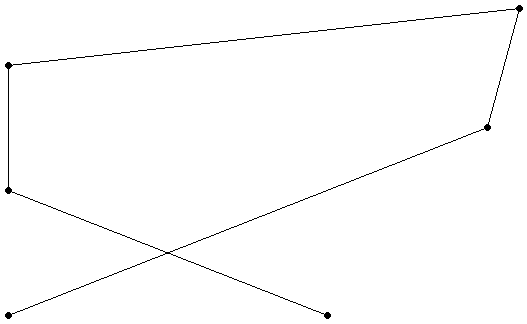
The following examples show how to use ST_IsSimple.

Polygon does not intersect itself:



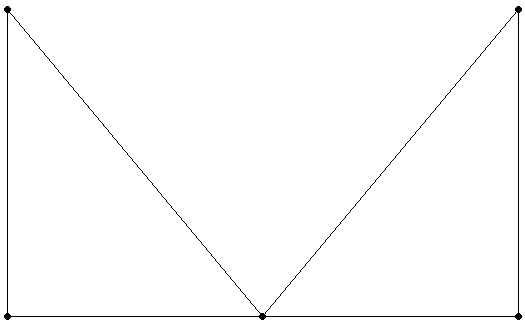
```
=> SELECT ST_IsSimple(ST_GeomFromText('POLYGON((-1 2,0 3,1 2,1 -2,-1 2))'));
 ST_IsSimple
-------------
 t
(1 row)
```

Linestring intersects itself.:



```
=> SELECT ST_IsSimple(ST_GeographyFromText('LINESTRING(10 10,25 25,26 34.5,
   10 30,10 20,20 10)'));
 St_IsSimple
-------------
 f
(1 row)
```

Linestring touches its interior at one or more locations:



```
=> SELECT ST_IsSimple(ST_GeomFromText('LINESTRING(0 0,0 1,1 0,2 1,2 0,0 0)'));
 ST_IsSimple
-------------
 f
(1 row)
```

ST_IsValid

Determines if a spatial object is well formed or valid. If the object is valid, ST_IsValid returns TRUE; otherwise, it returns FALSE. Use STV_IsValidReason to identify the location of the invalidity.

Spatial validity applies only to polygons and multipolygons. A polygon or multipolygon is valid if all of the following are true:

- The polygon is closed; its start point is the same as its end point.
- Its boundary is a set of linestrings.
- The boundary does not touch or cross itself.
- Any polygons in the interior do not touch the boundary of the exterior polygon except at a vertex.

The Open Geospatial Consortium (OGC) defines the validity of a polygon in section 6.1.11.1 of the Simple Feature Access Part 1 - Common Architecture specification.

If you are not sure if a polygon is valid, run ST_IsValid first. If you pass an invalid spatial object to a Vertica Place function, the function fails or returns incorrect results.

Behavior type

Immutable

Syntax

```
ST_IsValid( g )
```

Arguments

g

Geospatial object to test for validity, value of type GEOMETRY or GEOGRAPHY (WGS84).

Returns

BOOLEAN

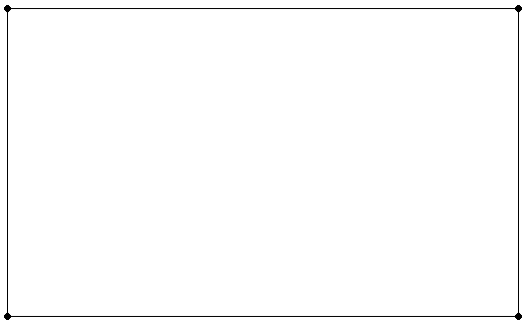
Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	No	No
Multipoint	Yes	No	No
Linestring	Yes	No	No
Multilinestring	Yes	No	No
Polygon	Yes	No	Yes
Multipolygon	Yes	No	No
GeometryCollection	Yes	No	No

Examples

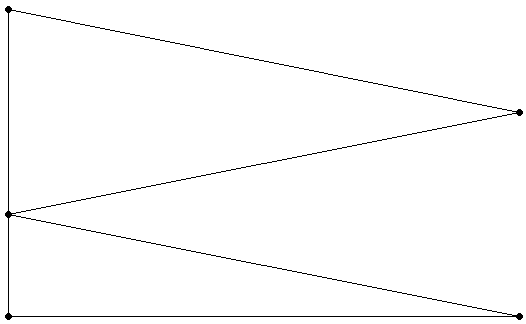
The following examples show how to use ST_IsValid.

Valid polygon:



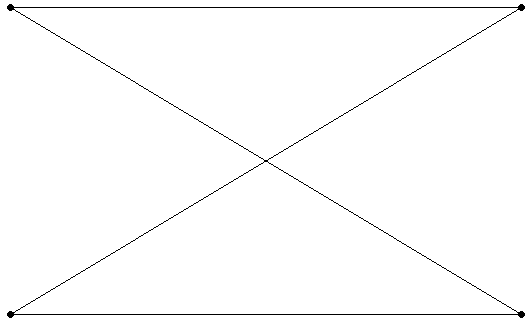
```
=> SELECT ST_IsValid(ST_GeomFromText('POLYGON((1 1,1 3,3 3,3 1,1 1))'));
 ST_IsValid
------------
 t
(1 row)
```

Invalid polygon:



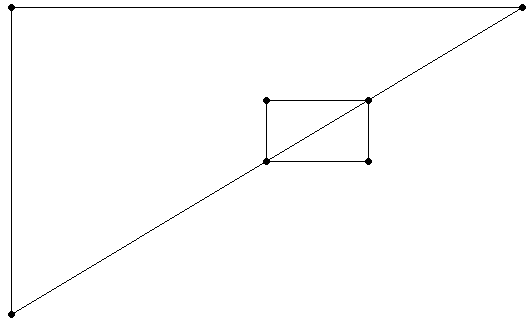
```
=> SELECT ST_IsValid(ST_GeomFromText('POLYGON((1 3,3 2,1 1,3 0,1 0,1 3))'));
 ST_IsValid
------------
 f
(1 row)
```

Invalid polygon:



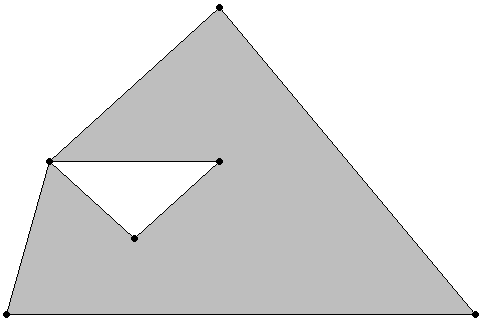
```
=> SELECT ST_IsValid(ST_GeomFromText('POLYGON((0 0,2 2,0 2,2 0,0 0))'));
 ST_IsValid
------------
 f
(1 row)
```

Invalid multipolygon:.



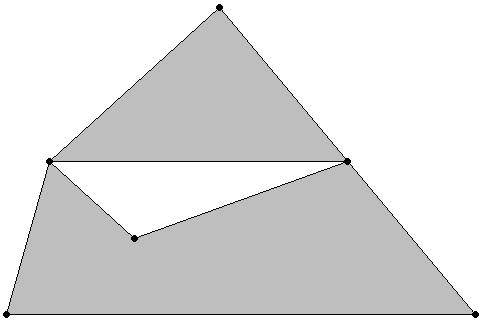
```
=> SELECT ST_IsValid(ST_GeomFromText('MULTIPOLYGON(((0 0, 0 1, 1 1, 0 0)),
   ((0.5 0.5, 0.7 0.5, 0.7 0.7, 0.5 0.7, 0.5 0.5)))'));
 ST_IsValid
------------
 f
(1 row)
```

Valid polygon with hole:



```
=> SELECT ST_IsValid(ST_GeomFromText('POLYGON((1 1,3 3,6 -1,0.5 -1,1 1),
   (1 1,3 1,2 0,1 1))'));
 ST_IsValid
------------
 t
(1 row)
```

Invalid polygon with hole:



```
=> SELECT ST_IsValid(ST_GeomFromText('POLYGON((1 1,3 3,6 -1,0.5 -1,1 1),
  (1 1,4.5 1,2 0,1 1))'));
 ST_IsValid
------------
 f
(1 row)
```

ST_Length

Calculates the length of a spatial object. For GEOMETRY objects, the length is measured in Cartesian coordinate units. For GEOGRAPHY objects, the length is measured in meters.

Calculates the length as follows:

- The length of a point or multipoint object is 0.
- The length of a linestring is the sum of the lengths of each line segment The length of a line segment is the distance from the start point to the end point.
- The length of a polygon is the sum of the lengths of the exterior boundary and any interior boundaries.
- The length of a multilinestring, multipolygon, or geometrycollection is the sum of the lengths of all the objects it contains.

> **Note**
> ST_Length does not calculate the length of WKTs or WKBs. To calculate the lengths of those objects, use the Vertica LENGTH SQL function with ST_AsBinary or ST_AsText.

Behavior type

Immutable

Syntax

```
ST_Length( g )
```

Arguments

g

Spatial object for which you want to calculate the length, type GEOMETRY or GEOGRAPHY

Returns

FLOAT

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes

Polygon	Yes	Yes
Multipolygon	Yes	Yes
GeometryCollection	Yes	No

Examples

The following examples show how to use ST_Length.

Returns length in Cartesian coordinate units:

```
=> SELECT ST_Length(ST_GeomFromText('LINESTRING(-1 -1,2 2,4 5,6 7)'));
   ST_Length
------------------
 10.6766190873295
(1 row)
```

Returns length in meters:

```
=> SELECT ST_Length(ST_GeographyFromText('LINESTRING(-56.12 38.26,-57.51 39.78,
  -56.37 45.24)'));
   ST_Length
------------------
 821580.025733461
(1 row)
```

ST_NumGeometries

Returns the number of geometries contained within a spatial object. Single GEOMETRY or GEOGRAPHY objects return 1 and empty objects return NULL.

Behavior type

Immutable

Syntax

```
ST_NumGeometries( g )
```

Arguments

g

Spatial object of type GEOMETRY or GEOGRAPHY

Returns

INTEGER

Supported data types

Data Type	**GEOMETRY**	**GEOGRAPHY (Perfect Sphere)**	**GEOGRAPHY (WGS84)**
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	No	No	No

Examples

The following example shows how to use ST_NumGeometries.

Return the number of geometries:

```
=> SELECT ST_NumGeometries(ST_GeomFromText('MULTILINESTRING ((1 5, 2 4, 5 3, 6 6), (3 5, 3 7))'));
 ST_NumGeometries
------------------
                2
(1 row)
```

See also

ST_GeometryN

ST_NumPoints

Calculates the number of vertices of a spatial object, empty objects return NULL.

The first and last vertex of polygons and multipolygons are counted separately.

Behavior type

Immutable

Syntax

```
ST_NumPoints( g )
```

Arguments

g

Spatial object for which you want to count the vertices, type GEOMETRY or GEOGRAPHY

Returns

INTEGER

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	No	No	No

Examples

The following examples show how to use ST_NumPoints.

Returns the number of vertices in a linestring:

```
=> SELECT ST_NumPoints(ST_GeomFromText('LINESTRING(1.33 1.56,2.31 3.4,2.78 5.82,
   3.76 3.9,4.11 3.27,5.85 4.34,6.9 4.231,7.61 5.77)'));
 ST_NumPoints
--------------
            8
(1 row)
```

Use ST_Boundary and ST_NumPoints to return the number of vertices of a polygon:

```
=> SELECT ST_NumPoints(ST_Boundary(ST_GeomFromText('POLYGON((1 2,1 4,
  2 5,3 6,4 6,5 5,4 4,3 3,1 2))')));
 ST_NumPoints
--------------
            9
(1 row)
```

ST_Overlaps

Determines if a GEOMETRY object shares space with another GEOMETRY object, but is not completely contained within that object. They must overlap at their interiors. If two objects touch at a single point or intersect only along a boundary, they do not overlap. Both parameters must have the same dimension; otherwise, ST_Overlaps returns FALSE.

Behavior type

Immutable

Syntax

```
ST_Overlaps ( g1, g2 )
```

Arguments

g1

Spatial object, type GEOMETRY

g2

Spatial object, type GEOMETRY

Returns

BOOLEAN

Supported data types

Data Type

GEOMETRY

Point

Yes

Multipoint

Yes

Linestring

Yes

Multilinestring

Yes

Polygon

Yes

Multipolygon

Yes

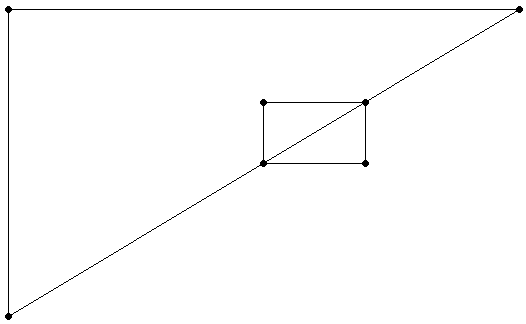
GeometryCollection

Yes

Examples

The following examples show how to use ST_Overlaps.

Polygon_1 overlaps but does not completely contain Polygon_2:



```
=> SELECT ST_Overlaps(ST_GeomFromText('POLYGON((0 0, 0 1, 1 1, 0 0))'),
   ST_GeomFromText('POLYGON((0.5 0.5, 0.7 0.5, 0.7 0.7, 0.5 0.7, 0.5 0.5))'));
 ST_Overlaps
-------------
 t
(1 row)
```

Two objects with different dimensions:

```
=> SELECT ST_Overlaps(ST_GeomFromText('LINESTRING(2 2,4 4)'),
   ST_GeomFromText('POINT(3 3)'));
 ST_Overlaps
-------------
 f
(1 row)
```

ST_PointFromGeoHash

Returns the center point of the specified GeoHash.

Behavior type

Immutable

Syntax

```
ST_PointFromGeoHash(GeoHash)
```

Arguments

GeoHash

A valid GeoHash string of arbitrary length.

Returns

GEOGRAPHY POINT

Examples

The following examples show how to use ST_PointFromGeoHash.

Returns the geography point of a high-level GeoHash and uses ST_AsText to convert that point into Well-Known Text:

```
=>  SELECT ST_AsText(ST_PointFromGeoHash('dr'));
ST_AsText
-------------------------
POINT (-73.125 42.1875)
(1 row)
```

Returns the geography point of a detailed GeoHash and uses ST_AsText to convert that point into Well-Known Text:

```
=>  SELECT ST_AsText(ST_PointFromGeoHash('1234567890bcdefhjkmn'));
ST_AsText
----------------------------------------
POINT (-122.196077187 -88.2297377551)
(1 row)
```

ST_PointN

Finds the *n* th point of a spatial object. If you pass a negative number, zero, or a number larger than the total number of points on the linestring, ST_PointN returns NULL.

The vertex order is based on the Well-Known Text (WKT) representation of the spatial object.

Behavior type

Immutable

Syntax

```
ST_PointN( g, n )
```

Arguments

g

Spatial object to search, type GEOMETRY or GEOGRAPHY

n

Point in the spatial object to be returned. The index is one-based, type INTEGER

Returns

GEOMETRY or GEOGRAPHY

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	No	No	No

Examples

The following examples show how to use ST_PointN.

Returns the fifth point:

```
=> SELECT ST_AsText(ST_PointN(ST_GeomFromText('
        POLYGON(( 2 6, 2 9, 6 9, 7 7, 4 6, 2 6))'), 5));
 ST_AsText
-------------
 POINT (4 6)
(1 row)
```

Returns the second point:

```
=> SELECT ST_AsText(ST_PointN(ST_GeographyFromText('
        LINESTRING(23.41 24.93,34.2 32.98,40.7 41.19)'), 2));
    ST_AsText
---------------------
 POINT (34.2 32.98)
(1 row)
```

ST_Relate

Determines if a given GEOMETRY object is spatially related to another GEOMETRY object, based on the specified DE-9IM pattern matrix string.

The DE-9IM standard identifies how two objects are spatially related to each other.

Behavior type

Immutable

Syntax

```
ST_Relate( g1, g2, matrix )
```

Arguments

g1

Spatial object, type GEOMETRY

g2

Spatial object, type GEOMETRY

matrix

DE-9IM pattern matrix string, type CHAR(9). This string represents a 3 x 3 matrix of restrictions on the dimensions of the respective intersections of the interior, boundary, and exterior of the two geometries. Must contain exactly 9 of the following characters:

- T
- F
- 0
- 1
- 2
- *

Returns

BOOLEAN

Supported data types

Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

Examples

The following examples show how to use ST_Relate.

The DE-9IM pattern for "equals" is 'T*F**FFF2' :

```
=> SELECT ST_Relate(ST_GeomFromText('LINESTRING(0 1,2 2)'),
   ST_GeomFromText('LINESTRING(2 2,0 1)'), 'T*F**FFF2');
 ST_Relate
---------------
 t
(1 row)
```

The DE-9IM pattern for "overlaps" is 'T*T***T**' :

```
=> SELECT ST_Relate(ST_GeomFromText('POLYGON((-1 -1,0 1,2 2,-1 -1))'),
   ST_GeomFromText('POLYGON((0 1,1 -1,1 1,0 1))'), 'T*T***T**');
 ST_Relate
-----------
 t
(1 row)
```

ST_SRID

Identifies the spatial reference system identifier (SRID) stored with a spatial object.

The SRID of a GEOMETRY object can only be determined when passing an SRID to either ST_GeomFromText or ST_GeomFromWKB. ST_SRID returns this stored value. SRID values of 0 to 2^{32-1} are valid.

Behavior type

Immutable

Syntax

```
ST_SRID( g )
```

Arguments

g

Spatial object for which you want the SRID, type GEOMETRY or GEOGRAPHY

Returns

INTEGER

Supported data types

Data Type	**GEOMETRY**	**GEOGRAPHY (Perfect Sphere)**	**GEOGRAPHY (WGS84)**
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	Yes	No	No

Examples

The following examples show how to use ST_SRID.

The default SRID of a GEOMETRY object is 0:

```
=> SELECT ST_SRID(ST_GeomFromText(
    'POLYGON((-1 -1,2 2,0 1,-1 -1))'));
 ST_SRID
---------
 0
(1 row)
```

The default SRID of a GEOGRAPHY object is 4326:

```
=> SELECT ST_SRID(ST_GeographyFromText(
    'POLYGON((22 35,24 35,26 32,22 35))'));
 ST_SRID
---------
 4326
(1 row)
```

ST_SymDifference

Calculates all the points in two GEOMETRY objects except for the points they have in common, but including the boundaries of both objects.

This result is called the symmetric difference and is represented mathematically as: Closure (*g1 – g2*) È Closure (*g2 – g1*)

Behavior type

Immutable

Syntax

```
ST_SymDifference( g1, g2 )
```

Arguments

g1

Spatial object, type GEOMETRY

g2

Spatial object, type GEOMETRY

Returns

GEOMETRY

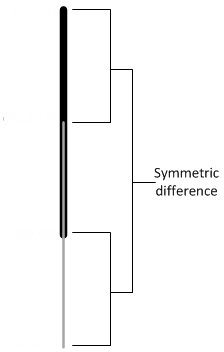
Supported data types

Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

Examples

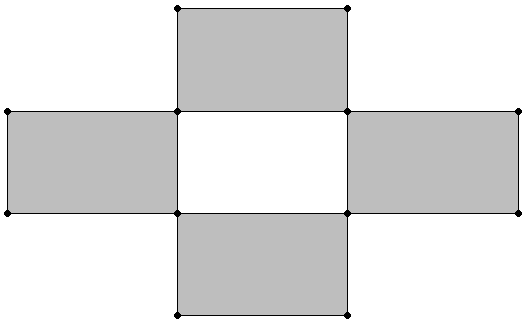
The following examples show how to use ST_SymDifference.

Returns the two linestrings:




```
=> SELECT ST_AsText(ST_SymDifference(ST_GeomFromText('LINESTRING(30 40,
  30 55)'),ST_GeomFromText('LINESTRING(30 32.5,30 47.5)')));
   ST_AsText
-----------------
MULTILINESTRING ((30 47.5, 30 55),(30 32.5,30 40))
(1 row)
```

Returns four squares:



```
=> SELECT ST_AsText(ST_SymDifference(ST_GeomFromText('POLYGON((2 1,2 4,3 4,
  3 1,2 1))'),ST_GeomFromText('POLYGON((1 2,1 3,4 3,4 2,1 2))')));
                          ST_AsText
-------------------------------------------------------------------------
 MULTIPOLYGON (((2 1, 2 2, 3 2, 3 1, 2 1)), ((1 2, 1 3, 2 3, 2 2, 1 2)),
 ((2 3, 2 4, 3 4, 3 3, 2 3)), ((3 2, 3 3, 4 3, 4 2, 3 2)))
(1 row)
```

ST_Touches

Determines if two GEOMETRY objects touch at a single point or along a boundary, but do not have interiors that intersect.

GEOGRAPHY Polygons with a vertex or border on the International Date Line (IDL) or the North or South pole are not supported.

Behavior type

Immutable

Syntax

```
ST_Touches( g1, g2
                [USING PARAMETERS spheroid={true | false}] )
```

Arguments

g1

Spatial object, value of type GEOMETRY

g2

Spatial object, value of type GEOMETRY

Parameters

spheroid = {true | false}

(Optional) BOOLEAN that specifies whether to use a perfect sphere or WGS84.
Default : False

Returns

BOOLEAN

Supported data types

Data Type	**GEOMETRY**	**GEOGRAPHY (WGS84)**
Point	Yes	Yes
Multipoint	Yes	No
Linestring	Yes	No
Multilinestring	Yes	No
Polygon	Yes	Yes
Multipolygon	Yes	No

GeometryCollection	Yes	No

Compatible GEOGRAPHY pairs:

Data Type	**GEOGRAPHY (WGS84)**
Point-Point	No
Linestring-Point	No
Polygon-Point	Yes
Multipolygon-Point	No

Examples

The following examples show how to use ST_Touches.

Two polygons touch at a single point:

```
=> SELECT ST_Touches(ST_GeomFromText('POLYGON((-1 2,0 3,0 1,-1 2))'),
   ST_GeomFromText('POLYGON((1 3,0 3,1 2,1 3))'));
 ST_Touches
------------
 t
(1 row)
```

Two polygons touch only along part of the boundary:

```
=> SELECT ST_Touches(ST_GeomFromText('POLYGON((-1 2,0 3,0 1,-1 2))'),
   ST_GeomFromText('POLYGON((1 2,0 3,0 1,1 2))'));
 ST_Touches
------------
 t
(1 row)
```

Two polygons do not touch at any point:

```
=> SELECT ST_Touches(ST_GeomFromText('POLYGON((-1 2,0 3,0 1,-1 2))'),
   ST_GeomFromText('POLYGON((0 2,-1 3,-2 0,0 2))'));
 ST_Touches
------------
 f
(1 row)
```

ST_Transform

Returns a new GEOMETRY with its coordinates converted to the spatial reference system identifier (SRID) used by the srid argument.

This function supports the following transformations:

- EPSG 4326 (WGS84) to EPSG 3857 (Web Mercator)
- EPSG 3857 (Web Mercator) to EPSG 4326 (WGS84)

For EPSG 4326 (WGS84), unless the coordinates fall within the following ranges, conversion results in failure:

- Longitude limits: -572 to +572
- Latitude limits: -89.9999999 to +89.9999999

Behavior type

Immutable

Syntax

```
ST_Transform( g1, srid )
```

Arguments

g1

Spatial object of type GEOMETRY.

srid

Spatial reference system identifier (SRID) to which you want to convert your spatial object, of type INTEGER.

Returns

GEOMETRY

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	No	No
Multipoint	Yes	No	No
Linestring	Yes	No	No
Multilinestring	Yes	No	No
Polygon	Yes	No	No
Multipolygon	Yes	No	No
GeometryCollection	Yes	No	No

Examples

The following example shows how you can transform data from Web Mercator (3857) to WGS84 (4326):

```
=> SELECT ST_AsText(ST_Transform(STV_GeometryPoint(7910240.56433, 5215074.23966, 3857), 4326));
       ST_AsText
-------------------------
 POINT (71.0589 42.3601)
(1 row)
```

The following example shows how you can transform linestring data in a table from WGS84 (4326) to Web Mercator (3857):

```
=> CREATE TABLE transform_line_example (g GEOMETRY);
CREATE TABLE
=> COPY transform_line_example (gx FILLER LONG VARCHAR, g AS ST_GeomFromText(gx, 4326)) FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> LINESTRING(0 0, 1 1, 2 2, 3 4)
>> \.
=> SELECT ST_AsText(ST_Transform(g, 3857)) FROM transform_line_example;
                                                ST_AsText
-----------------------------------------------------------------------------------------------------------------------
 LINESTRING (0 -7.08115455161e-10, 111319.490793 111325.142866, 222638.981587 222684.208506, 333958.47238 445640.109656)
(1 row)
```

The following example shows how you can transform point data in a table from WGS84 (4326) to Web Mercator (3857):

```
=> CREATE TABLE transform_example (x FLOAT, y FLOAT, srid INT);
CREATE TABLE
=> COPY transform_example FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 42.3601|71.0589|4326
>> 122.4194|37.7749|4326
>> 94.5786|39.0997|4326
>> \.
=> SELECT ST_AsText(ST_Transform(STV_GeometryPoint(x, y, srid), 3857)) FROM transform_example;
            ST_AsText
--------------------------------------
 POINT (4715504.76195 11422441.5961)
 POINT (13627665.2712 4547675.35434)
 POINT (10528441.5919 4735962.8206)
(3 rows)
```

ST_Union

Calculates the union of all points in two spatial objects.

This result is represented mathematically by: *g1* È *g2*

Behavior type

Immutable

Syntax

```
ST_Union( g1, g2 )
```

Arguments

g1

Spatial object, type GEOMETRY

g2

Spatial object, type GEOMETRY

Returns

GEOMETRY

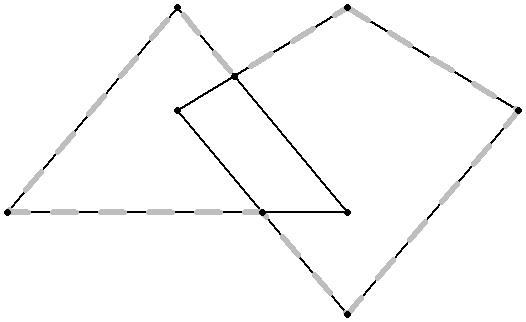
Supported data types

Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

Examples

The following example shows how to use ST_Union.

Returns a polygon that represents all the points contained in these two polygons:



```
=> SELECT ST_AsText(ST_Union(ST_GeomFromText('POLYGON((0 2,1 1,0 -1,-1 1,0 2))'),
   ST_GeomFromText('POLYGON((-1 2, 0 0, -2 0, -1 2))')));
                              ST_AsText
-------------------------------------------------------------------------------
 POLYGON ((0 2, 1 1, 0 -1, -0.5 0, -2 0, -1 2, -0.666666666667 1.33333333333, 0 2))
(1 row)
```

ST_Within

If spatial object g1 is completely inside of spatial object g2 , then ST_Within returns true. Both parameters must be the same spatial data type. Either specify two GEOMETRY objects or two GEOGRAPHY objects.

If an object such as a point or linestring only exists along a polygon's boundary, then ST_Within returns false. The interior of a linestring is all the points along the linestring except the start and end points.

ST_Within(g``g is functionally equivalent to ST_Contains( g``g .

GEOGRAPHY Polygons with a vertex or border on the International Date Line (IDL) or the North or South pole are not supported.

Behavior type

Immutable

Syntax

```
ST_Within( g1, g2
                [USING PARAMETERS spheroid={true | false}] )
```

Arguments

g1

Spatial object, type GEOMETRY or GEOGRAPHY

g2

Spatial object, type GEOMETRY or GEOGRAPHY

Parameters

spheroid = {true | false}

(Optional) BOOLEAN that specifies whether to use a perfect sphere or WGS84.
Default : False

Returns

BOOLEAN

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	No	No
Linestring	Yes	Yes	No
Multilinestring	Yes	No	No

Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	No
GeometryCollection	Yes	No	No

Compatible GEOGRAPHY pairs:

Data Type	**GEOGRAPHY (Perfect Sphere)**	**GEOGRAPHY (WGS84)**
Point-Point	Yes	No
Point-Linestring	Yes	No
Point-Polygon	Yes	Yes
Point-Multipolygon	Yes	No

Examples

The following examples show how to use ST_Within.

The first polygon is completely contained within the second polygon:

```
=> SELECT ST_Within(ST_GeomFromText('POLYGON((0 2,1 1,0 -1,0 2))'),
   ST_GeomFromText('POLYGON((-1 3,2 1,0 -3,-1 3))'));
 ST_Within
-----------
 t
(1 row)
```

The point is on a vertex of the polygon, but not in its interior:

```
=> SELECT ST_Within (ST_GeographyFromText('POINT(30 25)'),
   ST_GeographyFromText('POLYGON((25 25,25 35,32.2 35,30 25,25 25))'));
 ST_Within
-----------
 f
(1 row)
```

Two polygons are spatially equivalent:

```
=> SELECT ST_Within (ST_GeomFromText('POLYGON((-1 2, 0 3, 0 1, -1 2))'),
   ST_GeomFromText('POLYGON((0 3, -1 2, 0 1, 0 3))'));
 ST_Within
-----------
 t
(1 row)
```

See also

- ST_Contains
- ST_Overlaps

ST_X

Determines the *x* - coordinate for a GEOMETRY point or the longitude value for a GEOGRAPHY point.

Behavior type

Immutable

Syntax

```
ST_X( g )
```

Arguments

g

Point of type GEOMETRY or GEOGRAPHY

Returns

FLOAT

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	No	No	No
Linestring	No	No	No
Multilinestring	No	No	No
Polygon	No	No	No
Multipolygon	No	No	No
GeometryCollection	No	No	No

Examples

The following examples show how to use ST_X.

Returns the *x* -coordinate:

```
=> SELECT ST_X(ST_GeomFromText('POINT(3.4 1.25)'));
 ST_X
-----
 3.4
(1 row)
```

Returns the longitude value:

```
=> SELECT ST_X(ST_GeographyFromText('POINT(25.34 45.67)'));
 ST_X
-------
 25.34
(1 row)
```

ST_XMax

Returns the maximum *x* -coordinate of the minimum bounding rectangle of the GEOMETRY or GEOGRAPHY object.

For GEOGRAPHY types, Vertica Place computes maximum coordinates by calculating the maximum longitude of the great circle arc from (MAX(longitude), ST_YMin(GEOGRAPHY)) to (MAX(longitude), ST_YMax(GEOGRAPHY)). In this case, MAX(longitude) is the maximum longitude value of the geography object.

If either latitude or longitude is out of range, ST_XMax returns the maximum plain value of the geography object.

Behavior type

Immutable

Syntax

```
ST_XMax( g )
```

Arguments

g

Spatial object for which you want to find the maximum *x* -coordinate, type GEOMETRY or GEOGRAPHY.

Returns

FLOAT

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes
Polygon	Yes	Yes
Multipolygon	Yes	Yes
GeometryCollection	Yes	No

Examples

The following examples show how to use ST_XMax.

Returns the maximum *x*-coordinate within a rectangle:

```
=> SELECT ST_XMax(ST_GeomFromText('POLYGON((0 1,0 2,1 2,1 1,0 1))'));
 ST_XMax
-----------
    1
(1 row)
```

Returns the maximum longitude value within a rectangle:

```
=> SELECT ST_XMax(ST_GeographyFromText(
   'POLYGON((-71.50 42.35, -71.00 42.35, -71.00 42.38, -71.50 42.38, -71.50 42.35))'));
 ST_XMax
---------
    -71
(1 row)
```

ST_XMin

Returns the minimum *x*-coordinate of the minimum bounding rectangle of the GEOMETRY or GEOGRAPHY object.

For GEOGRAPHY types, Vertica Place computes minimum coordinates by calculating the minimum longitude of the great circle arc from (MIN(longitude), ST_YMin(GEOGRAPHY)) to (MIN(longitude), ST_YMax(GEOGRAPHY)). In this case, MIN(latitude) represents the minimum longitude value of the geography object

If either latitude or longitude is out of range, ST_XMin returns the minimum plain value of the geography object.

Behavior type

Immutable

Syntax

```
ST_XMin( g )
```

Arguments

g

Spatial object for which you want to find the minimum *x*-coordinate, type GEOMETRY or GEOGRAPHY.

Returns

FLOAT

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes
Polygon	Yes	Yes
Multipolygon	Yes	Yes
GeometryCollection	Yes	No

Examples

The following examples show how to use ST_XMin.

Returns the minimum *x* -coordinate within a rectangle:

```
=> SELECT ST_XMin(ST_GeomFromText('POLYGON((0 1,0 2,1 2,1 1,0 1))'));
 ST_XMin
----------
       0
(1 row)
```

Returns the minimum longitude value within a rectangle:

```
=> SELECT ST_XMin(ST_GeographyFromText(
   'POLYGON((-71.50 42.35, -71.00 42.35, -71.00 42.38, -71.50 42.38, -71.50 42.35))'));
 ST_XMin
----------
   -71.5
(1 row)
```

ST_Y

Determines the *y* -coordinate for a GEOMETRY point or the latitude value for a GEOGRAPHY point.

Behavior type

Immutable

Syntax

```
ST_Y( g )
```

Arguments

g

Point of type GEOMETRY or GEOGRAPHY

Returns

FLOAT

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	No	No	No
Linestring	No	No	No

Multilinestring	No	No	No
Polygon	No	No	No
Multipolygon	No	No	No
GeometryCollection	No	No	No

Examples

The following examples show how to use ST_Y.

Returns the *y* -coordinate:

```
=> SELECT ST_Y(ST_GeomFromText('POINT(3 5.25)'));
 ST_Y
------
 5.25
(1 row)
```

Returns the latitude value:

```
=> SELECT ST_Y(ST_GeographyFromText('POINT(35.44 51.04)'));
 ST_Y
-------
 51.04
(1 row)
```

ST_YMax

Returns the maximum *y* -coordinate of the minimum bounding rectangle of the GEOMETRY or GEOGRAPHY object.

For GEOGRAPHY types, Vertica Place computes maximum coordinates by calculating the maximum latitude of the great circle arc from (ST_XMin(GEOGRAPHY), MAX(latitude)) to (ST_XMax(GEOGRAPHY), MAX(latitude)). In this case, MAX(latitude) is the maximum latitude value of the geography object.

If either latitude or longitude is out of range, ST_YMax returns the maximum plain value of the geography object.

Behavior type

Immutable

Syntax

```
ST_YMax( g )
```

Arguments

g

Spatial object for which you want to find the maximum *y* -coordinate, type GEOMETRY or GEOGRAPHY.

Returns

FLOAT

Supported data types

Data Type	**GEOMETRY**	**GEOGRAPHY (Perfect Sphere)**
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes
Polygon	Yes	Yes

Multipolygon	Yes	Yes
GeometryCollection	Yes	No

Examples

The following examples show how to use ST_YMax.

Returns the maximum *y* -coordinate within a rectangle:

```
=> SELECT ST_YMax(ST_GeomFromText('POLYGON((0 1,0 4,1 4,1 1,0 1))'));
 ST_YMax
-----------
    4
(1 row)
```

Returns the maximum latitude value within a rectangle:

```
=> SELECT ST_YMax(ST_GeographyFromText(
   'POLYGON((-71.50 42.35, -71.00 42.35, -71.00 42.38, -71.50 42.38, -71.50 42.35))'));
  ST_YMax
------------------
 42.3802715689979
(1 row)
```

ST_YMin

Returns the minimum *y* -coordinate of the minimum bounding rectangle of the GEOMETRY or GEOGRAPHY object.

For GEOGRAPHY types, Vertica Place computes minimum coordinates by calculating the minimum latitude of the great circle arc from (ST_XMin(GEOGRAPHY), MIN(latitude)) to (ST_XMax(GEOGRAPHY), MIN(latitude)). In this case, MIN(latitude) represents the minimum latitude value of the geography object.

If either latitude or longitude is out of range, ST_YMin returns the minimum plain value of the geography object.

Behavior type

Immutable

Syntax

```
ST_YMin( g )
```

Arguments

g

Spatial object for which you want to find the minimum *y* -coordinate, type GEOMETRY or GEOGRAPHY.

Returns

FLOAT

Supported data types

Data Type	**GEOMETRY**	**GEOGRAPHY (Perfect Sphere)**
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes
Polygon	Yes	Yes
Multipolygon	Yes	Yes

GeometryCollection	Yes	No

Examples

The following examples show how to use ST_YMin.

Returns the minimum *y* -coordinate within a rectangle:

```
=> SELECT ST_YMin(ST_GeomFromText('POLYGON((0 1,0 4,1 4,1 1,0 1))'));
 ST_YMin
-----------
    1
(1 row)
```

Returns the minimum latitude value within a rectangle:

```
=> SELECT ST_YMin(ST_GeographyFromText(
   'POLYGON((-71.50 42.35, -71.00 42.35, -71.00 42.38, -71.50 42.38, -71.50 42.35))'));
 ST_YMin
------------------
 42.35
(1 row)
```

STV_AsGeoJSON

Returns the geometry or geography argument as a Geometry Javascript Object Notation (GeoJSON) object.

Behavior type

Immutable

Syntax

```
STV_AsGeoJSON( g, [USING PARAMETERS maxdecimals=[dec_value]])
```

Arguments

g

Spatial object of type GEOMETRY or GEOGRAPHY

maxdecimals = *dec_value*

(Optional) Integer value. Determines the maximum number of digits to output after the decimal of floating point coordinates.

Valid values **:** Between 0 and 15.
Default ** value****:** 6

Returns

LONG VARCHAR

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes	Yes
Multipoint	Yes	Yes	Yes
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	No	No	No

Examples

The following examples show how you can use STV_AsGeoJSON.

Convert a geometry polygon to GeoJSON:

```
=> SELECT STV_AsGeoJSON(ST_GeomFromText('POLYGON((3 2, 4 3, 5 1, 3 2), (3.5 2, 4 2.5, 4.5 1.5, 3.5 2))'));
                                    STV_AsGeoJSON
-------------------------------------------------------------------------------------------------
 {"type":"Polygon","coordinates":[[[3,2],[4,3],[5,1],[3,2]],[[3.5,2],[4,2.5],[4.5,1.5],[3.5,2]]]}
(1 row)
```

Convert a geography point to GeoJSON:

```
=> SELECT STV_AsGeoJSON(ST_GeographyFromText('POINT(42.36011 71.05899)') USING PARAMETERS maxdecimals=4);
                STV_AsGeoJSON
-----------------------------------------------
 {"type":"Point","coordinates":[42.3601,71.059]}
(1 row)
```

STV_Create_Index

Creates a spatial index on a set of polygons to speed up spatial intersection with a set of points.

A spatial index is created from an input polygon set, which can be the result of a query. Spatial indexes are created in a global name space. Vertica uses a distributed plan whenever the input table or projection is segmented across nodes of the cluster.

The OVER() clause must be empty.

Important

You cannot access spatial indexes on newly added nodes without rebalancing your cluster. For more information, see REBALANCE_CLUSTER .

Behavior type

Immutable

> **Note**
> Indexes are not connected to any specific table. Subsequent DML commands on the underlying table or tables of the input data source do not modify the index.

Syntax

```
STV_Create_Index( gid, g
            USING PARAMETERS index='index_name'
                        [, overwrite={ true | false } ]
                        [, max_mem_mb=maxmem_value]
              [, skip_nonindexable_polygons={true | false } ] )
           OVER()
             [ AS (polygons, srid, min_x, min_y, max_x, max_y, info) ]
```

Arguments

gid

Name of an integer column that uniquely identifies the polygon. The gid cannot be NULL.

g

Name of a geometry or geography (WGS84) column or expression that contains polygons and multipolygons. Only polygon and multipolygon can be indexed. Other shape types are excluded from the index.

Parameters

index = *'index_name'*

Name of the index, type VARCHAR. Index names cannot exceed 110 characters. The slash, backslash, and tab characters are not allowed in index names.

overwrite = [true | false]

Boolean, specifies whether to overwrite the index, if an index exists. This parameter cannot be NULL.
Default: False

max_mem_mb = ***maxmem_value***

A positive integer that assigns a limit to the amount of memory in megabytes that STV_Create_Index can allocate during index construction. On a multi-node database this is the memory limit per node. The default value is 256. Do not assign a value higher than the amount of memory in the GENERAL resource pool. For more information about this pool, see Monitoring resource pools .

Setting a value for max_mem_mb that is at or near the maximum memory available on the node can negatively affect your system's performance. For example, it could cause other queries to time out waiting for memory resources during index construction.

skip_nonindexable_polygons **= [true | false]**

(Optional) BOOLEAN
In rare cases, intricate polygons (for instance, with too high resolution or anomalous spikes) cannot be indexed. These polygons are considered non-indexable. When set to False, non-indexable polygons cause the index creation to fail. When set to True, index creation can succeed by excluding non-indexable polygons from the index.

To review the polygons that were not able to be indexed, use STV_Describe_Index with the parameter list_polygon.

Default: False

Returns

polygons

Number of polygons indexed.

SRID

Spatial reference system identifier.

min_x, min_y, max_x, max_y

Coordinates of the minimum bounding rectangle (MBR) of the indexed geometries. (min_x , min_y) are the south-west coordinates, and (max_x , max_y) are the north-east coordinates.

info

Lists the number of excluded spatial objects as well as their type that were excluded from the index.

Supported data types

Data Type	**GEOMETRY**	**GEOGRAPHY (WGS84)**
Point	No	No
Multipoint	No	No
Linestring	No	No
Multilinestring	No	No
Polygon	Yes	Yes
Multipolygon	Yes	No
GeometryCollection	No	No

Privileges

Any user with access to the STV_*_Index functions can describe, rename, or drop indexes created by any other user.

Recommendations

- Segment large polygon tables across multiple nodes. Table segmentation causes index creation to run in parallel, leveraging the Massively Parallel Processing (MPP) architecture in Vertica. This significantly reduces execution time on large tables.
  Vertica recommends that you segment the table from which you are building the index when the total number of polygons is large.
- STV_Create_Index can consume large amounts of processing time and memory.
  Vertica recommends that when indexing new data for the first time, you monitor memory usage to be sure it stays within safe limits. Memory usage depends on number of polygons, number of vertices, and the amount of overlap among polygons.

- STV_Create_Index tries to allocate memory before it starts creating the index. If it cannot allocate enough memory, the function fails. If not enough memory is available, try the following:
  - Create the index at a time of less load on the system.
  - Avoid concurrent index creation.
  - Try segmenting the input table across the nodes of the cluster.
- Ensure that all of the polygons you plan to index are valid polygons. STV_Create_Index and STV_Refresh_Index do not check polygon validity when building an index.

  For more information, see Ensuring polygon validity before creating or refreshing an index .

Limitations

- Any indexes created prior to 12.0.x need to re-created.
- Index creation fails if there are WGS84 polygons with vertices on the International Date Line (IDL) or the North and South Poles.
- The backslash or tab characters are not allowed in index names.
- Indexes cannot have names greater than 110 characters.
- The following geometries are excluded from the index:
  - Non-polygons
  - Geometries with NULL identifiers
  - NULL (multi) polygon
  - EMPTY (multi) polygon
  - Invalid (multi) polygon
- The following geographies are excluded from the index:
  - Polygons with holes
  - Polygons crossing the International Date Line
  - Polygons covering the north or south pole
  - Antipodal polygons

Usage tips

- To cancel an STV_Create_Index run, use **Ctrl** + **C** .
- If there are no valid polygons in the geom column, STV_Create_Index reports an error in vertica.log and stops index creation.
- If index creation uses a large amount of memory, consider segmenting your data to utilize parallel index creation.

Examples

The following examples show how to use STV_Create_Index.

Create an index with a single literal argument:

```
=> SELECT STV_Create_Index(1, ST_GeomFromText('POLYGON((0 0,0 15.2,3.9 15.2,3.9 0,0 0))')
    USING PARAMETERS index='my_polygon') OVER();
polygons  | SRID | min_x | min_y | max_x | max_y | info
----------+------+-------+-------+-------+-------+------
        1 |    0 |     0 |     0 |   3.9 |  15.2 |
(1 row)
```

Create an index from a table:

```
=> CREATE TABLE pols (gid INT, geom GEOMETRY(1000));
CREATE TABLE
=> COPY pols(gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1|POLYGON((-31 74,8 70,8 50,-36 53,-31 74))
>> 2|POLYGON((-38 50,4 13,11 45,0 65,-38 50))
>> 3|POLYGON((10 20,15 60,20 45,46 15,10 20))
>> 4|POLYGON((5 20,9 30,20 45,36 35,5 20))
>> 5|POLYGON((12 23,9 30,20 45,36 35,37 67,45 80,50 20,12 23))
>> \.
=> SELECT STV_Create_Index(gid, geom USING PARAMETERS index='my_polygons_1', overwrite=true,
    max_mem_mb=256) OVER() FROM pols;
 polygons | SRID | min_x | min_y | max_x | max_y | info
----------+------+-------+-------+-------+-------+------
        5 |    0 |   -38 |    13 |    50 |    80 |
(1 row)
```

Create an index in parallel from a partitioned table:

```
=> CREATE TABLE pols (p INT, gid INT, geom GEOMETRY(1000)) SEGMENTED BY HASH(p) ALL NODES;
CREATE TABLE
=> COPY pols (p, gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1|10|POLYGON((-31 74,8 70,8 50,-36 53,-31 74))
>> 1|11|POLYGON((-38 50,4 13,11 45,0 65,-38 50))
>> 3|12|POLYGON((-12 42,-12 42,27 48,14 26,-12 42))
>> \.
=> SELECT STV_Create_Index(gid, geom USING PARAMETERS index='my_polygons', overwrite=true,
    max_mem_mb=256) OVER() FROM pols;
 polygons | SRID | min_x | min_y | max_x | max_y | info
----------+------+-------+-------+-------+-------+------
        3 |    0 |   -38 |    13 |    27 |    74 |
(1 row)
```

See also

- Spatial joins with ST_Intersects and STV_Intersect
- STV_Intersect scalar function
- STV_Intersect transform function
- STV_Describe_Index
- STV_Drop_Index
- STV_Rename_Index
- Ensuring polygon validity before creating or refreshing an index

STV_Describe_Index

Retrieves information about an index that contains a set of polygons. If you do not pass any parameters, STV_Describe_Index returns all of the defined indexes.

The OVER() clause must be empty.

Behavior type

Immutable

Syntax

```
STV_Describe_Index ( [ USING PARAMETERS [index='index_name']
                                [, list_polygons={true | false } ]] ) OVER ()
```

Arguments

index = *'index_name'*

Name of the index, type VARCHAR. Index names cannot exceed 110 characters. The slash, backslash, and tab characters are not allowed in index names.

list_polygon

(Optional) BOOLEAN that specifies whether to list the polygons in the index. The index argument must be used with this argument.

Returns

polygons

Number of polygons indexed.

SRID

Spatial reference system identifier.

min_x* , *min_y* , *max_x* , *max_y

Coordinates of the minimum bounding rectangle (MBR) of the indexed geometries. (min_x , min_y) are the south-west coordinates, and (max_x , max_y) are the north-east coordinates.

name

The name of the spatial index(es).

gid

Name of an integer column that uniquely identifies the polygon. The gid cannot be NULL.

state

The spatial object's state in the index. Possible values are:

- INDEXED - The spatial object was successfully indexed.
- SELF_INTERSECT - (WGS84 Only) The spatial object was not indexed because one of its edges intersects with another of its edges.
- EDGE_CROSS_IDL - (WGS84 Only) The spatial object was not indexed because one of its edges crosses the International Date Line.
- EDGE_HALF_CIRCLE - (WGS84 Only) The spatial object was not indexed because it contains two adjacent vertices that are antipodal.
- NON_INDEXABLE - The spatial object was not able to be indexed.

geography

The Well-Known Binary (WKB) representation of the spatial object.

geometry

The Well-Known Binary (WKB) representation of the spatial object.

Privileges

Any user with access to the STV_*_Index functions can describe, rename, or drop indexes created by any other user.

Limitations

Some functionality will require the index to be rebuilt if the index was created with 12.0.x or earlier.

Examples

The following examples show how to use STV_Describe_Index.

Retrieve information about the index:

```
=> SELECT STV_Describe_Index (USING PARAMETERS index='my_polygons') OVER ();
   type   | polygons | SRID | min_x | min_y | max_x | max_y
----------+----------+------+-------+-------+-------+-------
 GEOMETRY |        4 |    0 |    -1 |    -1 |    12 |    12

(1 row)
```

Return the names of all the defined indexes:

```
=> SELECT STV_Describe_Index() OVER ();
       name
------------------
MA_counties_index
my_polygons
NY_counties_index
US_States_Index
(4 rows)
```

Return the polygons included in an index:

```
=> SELECT STV_Describe_Index(USING PARAMETERS index='my_polygons', list_polygons=TRUE) OVER ();
 gid |     state     |            geometry
-----+---------------+---------------------------------
  12 | INDEXED       | \260\000\000\000\000\000\000\ ...
  14 | INDEXED       | \200\000\000\000\000\000\000\ ...
  10 | NON_INDEXABLE | \274\000\000\000\000\000\000\ ...
  11 | INDEXED       | \260\000\000\000\000\000\000\ ...
(4 rows)
```

See also

- Spatial joins with ST_Intersects and STV_Intersect
- STV_Intersect scalar function
- STV_Intersect transform function
- STV_Drop_Index
- STV_Rename_Index

STV_Drop_Index

Deletes a spatial index. If STV_Drop_Index cannot find the specified spatial index, it returns an error.

The OVER clause must be empty.

Behavior type

Immutable

Syntax

```
STV_Drop_Index( USING PARAMETERS index = 'index_name' ) OVER ()
```

Arguments

index = *'index_name'*

Name of the index, type VARCHAR. Index names cannot exceed 110 characters. The slash, backslash, and tab characters are not allowed in index names.

Examples

The following example shows how to use STV_Drop_Index.

Drop an index:

```
=> SELECT STV_Drop_Index(USING PARAMETERS index ='my_polygons') OVER ();
 drop_index
------------
 Index dropped
(1 row)
```

See also

- Spatial joins with ST_Intersects and STV_Intersect
- STV_Create_Index
- STV_Describe_Index
- STV_Rename_Index
- STV_Intersect scalar function
- STV_Intersect transform function

STV_DWithin

Determines if the shortest distance from the boundary of one spatial object to the boundary of another object is within a specified distance.

Parameters g1 and g2 must be both GEOMETRY objects or both GEOGRAPHY objects.

Behavior type

Immutable

Syntax

```
STV_DWithin( g1, g2, d )
```

Arguments

g1

Spatial object of type GEOMETRY or GEOGRAPHY

g2

Spatial object of type GEOMETRY or GEOGRAPHY

d

Value of type FLOAT indicating a distance. For GEOMETRY objects, the distance is measured in Cartesian coordinate units. For GEOGRAPHY objects, the distance is measured in meters.

Returns

BOOLEAN

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes
Polygon	Yes	Yes
Multipolygon	Yes	Yes
GeometryCollection	Yes	No

Compatible GEOGRAPHY pairs:

Data Type
GEOGRAPHY (Perfect Sphere)

Point-Point
Yes

Point-Linestring
Yes

Point-Polygon
Yes

Point-Multilinestring
Yes

Point-Multipolygon
Yes

Examples

The following examples show how to use STV_DWithin.

Two geometries are one Cartesian coordinate unit from each other at their closest points:

```
=> SELECT STV_DWithin(ST_GeomFromText('POLYGON((-1 -1,2 2,0 1,-1 -1))'),
   ST_GeomFromText('POLYGON((4 3,2 3,4 5,4 3))'),1);
 STV_DWithin
-------------
 t
(1 row)
```

If you reduce the distance to 0.99 units:

```
=> SELECT STV_DWithin(ST_GeomFromText('POLYGON((-1 -1,2 2,0 1,-1 -1))'),
   ST_GeomFromText('POLYGON((4 3,2 3,4 5,4 3))'),0.99);
 STV_DWithin
-------------
 f
(1 row)
```

The first polygon touches the second polygon:

```
=> SELECT STV_DWithin(ST_GeomFromText('POLYGON((-1 -1,2 2,0 1,-1 -1))'),
   ST_GeomFromText('POLYGON((1 1,2 3,4 5,1 1))'),0.00001);
 STV_DWithin
-------------
 t
(1 row)
```

The first polygon is not within 1000 meters from the second polygon:

```
=> SELECT STV_DWithin(ST_GeomFromText('POLYGON((45.2 40,50.65 51.29,
   55.67 47.6,50 47.6,45.2 40))'),ST_GeomFromText('POLYGON((25 25,25 30,
   30 30,30 25,25 25))'), 1000);
 STV_DWithin
--------------
 t
(1 row)
```

STV_Export2Shapefile

Exports GEOGRAPHY or GEOMETRY data from a database table or a subquery to a shapefile. Output is written to the directory specified using STV_SetExportShapefileDirectory .

Behavior type

Immutable

Syntax

```
STV_Export2Shapefile( columns USING PARAMETERS shapefile = 'shapefile-name'
   [, overwrite = boolean ]
   [, shape = 'spatial-class'] )
OVER()
```

Arguments

columns

The columns to export to the shapefile.

A value of asterisk (*) is the equivalent to listing all columns of the FROM clause.

Parameters

shapefile

Prefix of the component names of the shapefile. The following requirements apply:

- Must end with the file extension .shp .
- Limited to 128 octets in length—for example, city-data.shp .

If you want to save the shapefile to a sub-directory you can do so by concatenating the sub-directory to *shapefile-name* —for example, visualizations/city-data.shp .

You can also export a shapefile to a mounted S3 directory where you have read and write permissions. Use the syntax *bucketname* / *path* / *filename* .

overwrite

Boolean, specifies whether to overwrite the index, if an index exists. This parameter cannot be NULL.
Default: False

shape

One of the following spatial classes:

- Point
- Polygon
- Linestring
- Multipoint
- Multipolygon
- Multilinestring

Polygons and multipolygons always have a clockwise orientation.
Default: Polygon

Returns

Three files in the shapefile export directory with the extensions .shp , .shx , and .dbf .

Limitations

- If a multipolygon, multilinestring, or multipoint contains only one element, then it is written as a polygon, line, or point, respectively.
- Column names longer than 10 characters are truncated.
- Empty POINTS cannot be exported.
- All rows with NULL geometry or geography data are skipped.
- Unsupported or invalid dates are replaced with NULLs.
- Numeric values may lose precision when they are exported. This loss occurs because the target field in the .dbf file is a 64-bit FLOAT column, which can only represent about 15 significant digits.
- Shapefiles cannot exceed 4GB in size. If your shapefile is too large, try splitting the data and exporting to multiple shapefiles.

Examples

The following example shows how you can use STV_Export2Shapefile to export all columns from the table geo_data to a shapefile named city-data.shp:

```
=> SELECT STV_Export2Shapefile(*
        USING PARAMETERS shapefile = 'visualizations/city-data.shp',
                         overwrite = true, shape = 'Point')
        OVER()
        FROM geo_data
        WHERE REVENUE > 25000;
 Rows Exported |                        File Path
---------------+-----------------------------------------------------------------
       6442892 | v_geo-db_node0001: /home/geo/temp/visualizations/city-data.shp
(1 row)
```

STV_Extent

Returns a bounding box containing all of the input data.

Use STV_Extent inside of a nested query for best results. The OVER clause must be empty.

Important
STV_Extent does not return a valid polygon when the input is a single point.

Behavior type

Immutable
Syntax

```
STV_Extent( g )
```

Arguments

g

Spatial object, type GEOMETRY.

Returns

GEOMETRY

Supported data types

Data Type	**GEOMETRY**

Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	Yes

Examples

The following examples show how you can use STV_Extent.

Return the bounding box of a linestring, and verify that it is a valid polygon:

```
=> SELECT ST_AsText(geom) AS bounding_box, ST_IsValid(geom)
   FROM (SELECT STV_Extent(ST_GeomFromText('LineString(0 0, 1 1)')) OVER() AS geom) AS g;
          bounding_box            | ST_IsValid
--------------------------------------+------------
 POLYGON ((0 0, 1 0, 1 1, 0 1, 0 0)) | t
(1 row)
```

Return the bounding box of spatial objects in a table:

```
=> CREATE TABLE misc_geo_shapes (id IDENTITY, geom GEOMETRY);
CREATE TABLE
=> COPY misc_geo_shapes (gx FILLER LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> POINT(-71.03 42.37)
>> LINESTRING(-71.058849 42.367501, -71.062240 42.371276, -71.067938 42.371246)
>> POLYGON((-71.066030 42.380617, -71.055827 42.376734, -71.060811 42.376011, -71.066030 42.380617))
>> \.
=> SELECT ST_AsText(geom_col) AS bounding_box
   FROM (SELECT STV_Extent(geom) OVER() AS geom_col FROM misc_geo_shapes) AS g;
                                          bounding_box
-----------------------------------------------------------------------------------------------------------
 POLYGON ((-71.067938 42.367501, -71.03 42.367501, -71.03 42.380617, -71.067938 42.380617, -71.067938 42.367501))
(1 row)
```

STV_ForceLHR

Alters the order of the vertices of a spatial object to follow the left-hand-rule.

Behavior type

Immutable

Syntax

```
STV_ForceLHR( g, [USING PARAMETERS skip_nonreorientable_polygons={true | false} ])
```

Arguments

g

Spatial object, type GEOGRAPHY.

skip_nonreorientable_polygons = { true | false }

(Optional) Boolean

When set to False, non-orientable polygons generate an error. For example, if you use STV_ForceLHR or STV_Reverse with skip_nonorientable_polygons set to False, a geography polygon containing a hole generates an error. When set to True, the result returned is the polygon, as passed to the API, without alteration.

This argument can help you when you are creating an index from a table containing polygons that cannot be re-oriented.

Vertica Place considers these polygons non-orientable:

- Polygons with a hole
- Multipolygons
- Multipolygons with a hole

Default value : False

Returns

GEOGRAPHY

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	No	No	No
Multipoint	No	No	No
Linestring	No	No	No
Multilinestring	No	No	No
Polygon	No	Yes	Yes
Multipolygon	No	Yes	Yes
GeometryCollection	No	No	No

Examples

The following example shows how you can use STV_ForceLHR.

Re-orient a geography polygon to left-hand orientation:

```
=> SELECT ST_AsText(STV_ForceLHR(ST_GeographyFromText('Polygon((1 1, 3 1, 2 2, 1 1))')));
          ST_AsText
---------------------------------
 POLYGON ((1 1, 3 1, 2 2, 1 1))
(1 row)
```

Reverse the orientation of a geography polygon by forcing left-hand orientation:

```
=> SELECT ST_AsText(STV_ForceLHR(ST_GeographyFromText('Polygon((1 1, 2 2, 3 1, 1 1))')));
          ST_AsText
---------------------------------
 POLYGON ((1 1, 3 1, 2 2, 1 1))
(1 row)
```

See also

STV_Reverse

STV_Geography

Casts a GEOMETRY object into a GEOGRAPHY object. The SRID value does not affect the results of Vertica Place queries.

When STV_Geography converts a GEOMETRY object to a GEOGRAPHY object, it sets its SRID to 4326.

Behavior type

Immutable

Syntax

```
STV_Geography( geom )
```

Arguments

geom

Spatial object that you want to cast into a GEOGRAPHY object, type GEOMETRY

Returns

GEOGRAPHY

Supported data types

Data Type	**GEOMETRY**
Point	Yes
Multipoint	Yes
Linestring	Yes
Multilinestring	Yes
Polygon	Yes
Multipolygon	Yes
GeometryCollection	No

Examples

The following example shows how to use STV_Geography.

To calculate the centroid of the GEOGRAPHY object, convert it to a GEOMETRY object, then convert it back to a GEOGRAPHY object:

```
=> CREATE TABLE geogs(g GEOGRAPHY);
CREATE TABLE
=> COPY geogs(gx filler LONG VARCHAR, geog AS ST_GeographyFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> MULTIPOINT(-108.619726 45.000284,-107.866813 45.00107,-106.363711 44.994223,-70.847746 41.205814)
>> \.
=> SELECT ST_AsText(STV_Geography(ST_Centroid(STV_Geometry(g)))) FROM geogs;
          ST_AsText
----------------------------------
 POINT (-98.424499 44.05034775)
(1 row)
```

STV_GeographyPoint

Returns a GEOGRAPHY point based on the input values.

This is the optimal way to convert raw coordinates to GEOGRAPHY points.

Behavior type

Immutable

Syntax

```
STV_GeographyPoint( x, y )
```

Arguments

x

x-coordinate or longitude, FLOAT.

y

y-coordinate or latitude, FLOAT.

Returns

GEOGRAPHY

Examples

The following examples show how to use STV_GeographyPoint.

Return a GEOGRAPHY point:

```
=> SELECT ST_AsText(STV_GeographyPoint(-114.101588, 47.909677));
          ST_AsText
-------------------------------
 POINT (-114.101588 47.909677)
(1 row)
```

Return GEOGRAPHY points using two columns:

```
=> CREATE TABLE geog_data (id IDENTITY, x FLOAT, y FLOAT);
CREATE TABLE
=> COPY geog_data FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> -114.101588|47.909677
>> -111.532377|46.430753
>> \.
=> SELECT id, ST_AsText(STV_GeographyPoint(x, y)) FROM geog_data;
 id |           ST_AsText
----+-------------------------------
  1 | POINT (-114.101588 47.909677)
  2 | POINT (-111.532377 46.430753)
(2 rows)
```

Create GEOGRAPHY points by manipulating data source columns during load:

```
=> CREATE TABLE geog_data_load (id IDENTITY, geog GEOGRAPHY);
CREATE TABLE
=> COPY geog_data_load (lon FILLER FLOAT,
                        lat FILLER FLOAT,
                        geog AS STV_GeographyPoint(lon, lat))
   FROM 'test_coords.csv' DELIMITER ',';
 Rows Loaded
-------------
           2
(1 row)
=> SELECT id, ST_AsText(geog) FROM geog_data_load;
 id |             ST_AsText
----+---------------------------------------
  1 | POINT (-75.101654451 43.363830536)
  2 | POINT (-75.106444487 43.367093798)
(2 rows)
```

See also

STV_GeometryPoint

STV_Geometry

Casts a GEOGRAPHY object into a GEOMETRY object.

The SRID value does not affect the results of Vertica Place queries.

Behavior type

Immutable

Syntax

```
STV_Geometry( geog )
```

Arguments

geog

Spatial object that you want to cast into a GEOMETRY object, type GEOGRAPHY

Returns

GEOMETRY

Supported data types

Data Type	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes
Polygon	Yes	Yes
Multipolygon	Yes	Yes
GeometryCollection	No	No

Examples

The following example shows how to use STV_Geometry.

Convert the GEOGRAPHY values to GEOMETRY values, then convert the result back to a GEOGRAPHY type:

```
=> CREATE TABLE geogs(g GEOGRAPHY);
CREATE TABLE
=> COPY geogs(gx filler LONG VARCHAR, geog AS ST_GeographyFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> MULTIPOINT(-108.619726 45.000284,-107.866813 45.00107,-106.363711 44.994223,-70.847746 41.205814)
>> \.
=> SELECT ST_AsText(STV_Geography(ST_Centroid(STV_Geometry(g)))) FROM geogs;
          ST_AsText
---------------------------------
 POINT (-98.424499 44.05034775)
```

STV_GeometryPoint

Returns a GEOMETRY point, based on the input values.

This approach is the most-optimal way to convert raw coordinates to GEOMETRY points.

Behavior type

Immutable

Syntax

```
STV_GeometryPoint( x, y [, srid] )
```

Arguments

x

x-coordinate or longitude, FLOAT.

y

y-coordinate or latitude, FLOAT.

srid

(Optional) Spatial Reference Identifier (SRID) assigned to the point, INTEGER.

Returns

GEOMETRY

Examples

The following examples show how to use STV_GeometryPoint.

Return a GEOMETRY point with an SRID:

```
=> SELECT ST_AsText(STV_GeometryPoint(71.148562, 42.989374, 4326));
        ST_AsText
------------------------------
 POINT (-71.148562 42.989374)
(1 row)
```

Return GEOMETRY points using two columns:

```
=> CREATE TABLE geom_data (id IDENTITY, x FLOAT, y FLOAT, SRID int);
CREATE TABLE
=> COPY geom_data FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 42.36383053600048|-71.10165445099966|4326
>> 42.3670937980005|-71.10644448699964|4326
>> \.
=> SELECT id, ST_AsText(STV_GeometryPoint(x, y, SRID)) FROM geom_data;
 id |             ST_AsText
----+------------------------------------
  1 | POINT (-71.101654451 42.363830536)
  2 | POINT (-71.106444487 42.367093798)
(2 rows)
```

Create GEOMETRY points by manipulating data source columns during load:

```
=> CREATE TABLE geom_data_load (id IDENTITY, geom GEOMETRY);
CREATE TABLE
=> COPY geom_data_load (lon FILLER FLOAT,
                        lat FILLER FLOAT,
                        geom AS STV_GeometryPoint(lon, lat))
   FROM 'test_coords.csv' DELIMITER ',';
 Rows Loaded
-------------
           2
(1 row)
=> SELECT id, ST_AsText(geom) FROM geom_data_load;
 id |             ST_AsText
----+------------------------------------
  1 | POINT (-75.101654451 43.363830536)
  2 | POINT (-75.106444487 43.367093798)
(2 rows)
```

See also

STV_GeographyPoint

STV_GetExportShapefileDirectory

Returns the path of the export directory.

Behavior type

Immutable

Syntax

```
STV_GetExportShapefileDirectory( )
```

Returns

The path of the shapefile export directory.

Examples

The following example shows how you can use STV_GetExportShapefileDirectory to query the path of the shapefile export directory:

```
=> SELECT STV_GetExportShapefileDirectory();
      STV_GetExportShapefileDirectory
-----------------------------------------------------
 Shapefile export directory: [/home/user/temp]
(1 row)
```

STV_Intersect scalar function

Spatially intersects a point or points with a set of polygons. The STV_Intersect scalar function returns the identifier associated with an intersecting polygon.

Behavior type

Immutable

Syntax

```
STV_Intersect(  { g | x , y }
              USING PARAMETERS index= 'index_name')
```

Arguments

g

A geometry or geography (WGS84) column that contains points. The *g* column can contain only point geometries or geographies. If the column contains a different geometry or geography type, STV_Intersect terminates with an error.

x

x-coordinate or longitude, FLOAT.

y

y-coordinate or latitude, FLOAT.

Parameters

index = ' *index_name* '

Name of the spatial index, of type VARCHAR.

Returns

The identifier of a matching polygon. If the point does not intersect any of the index's polygons, then the STV_Intersect scalar function returns NULL.

Examples

The following examples show how you can use STV_Intersect scalar.

Using two floats, return the gid of a matching polygon or NULL:

```
=> CREATE TABLE pols (gid INT, geom GEOMETRY(1000));
CREATE TABLE
=> COPY pols(gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1|POLYGON((31 74,8 70,8 50,36 53,31 74))
>> \.
=> SELECT STV_Create_Index(gid, geom USING PARAMETERS index='my_polygons_1', overwrite=true,
                           max_mem_mb=256) OVER() FROM pols;
   type   | polygons | SRID | min_x | min_y | max_x | max_y | info
----------+----------+------+-------+-------+-------+-------+------
 GEOMETRY |        1 |    0 |     8 |    50 |    36 |    74 |
(1 row)

=> SELECT STV_Intersect(12.5683, 55.6761 USING PARAMETERS index = 'my_polygons_1');
 STV_Intersect
---------------
             1
(1 row)
```

Using a GEOMETRY column, return the gid of a matching polygon or NULL:

```
=> CREATE TABLE polygons (gid INT, geom GEOMETRY(700));
CREATE TABLE
=> COPY polygons (gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1|POLYGON((-31 74,8 70,8 50,-36 53,-31 74))
>> 2|POLYGON((-38 50,4 13,11 45,0 65,-38 50))
>> 3|POLYGON((-18 42,-10 65,27 48,14 26,-18 42))
>> \.
=> SELECT STV_Create_Index(gid, geom USING PARAMETERS index='my_polygons', overwrite=true,
     max_mem_mb=256) OVER() FROM polygons;
   type   | polygons | SRID | min_x | min_y | max_x | max_y | info
----------+----------+------+-------+-------+-------+-------+------
 GEOMETRY |        3 |    0 |   -38 |    13 |    27 |    74 |
(1 row)

=> CREATE TABLE points (gid INT, geom GEOMETRY(700));
CREATE TABLE
=> COPY points (gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 100|POINT(-1 52)
>> 101|POINT(-20 0)
>> 102|POINT(-8 25)
>> 103|POINT(0 0)
>> 104|POINT(1 5)
>> 105|POINT(20 45)
>> 106|POINT(-20 5)
>> 107|POINT(-20 1)
>> \.
=> SELECT gid AS pt_gid, STV_Intersect(geom USING PARAMETERS index='my_polygons') AS pol_gid
      FROM points ORDER BY pt_gid;
 pt_gid | pol_gid
--------+---------
    100 |       1
    101 |
    102 |       2
    103 |
    104 |
    105 |       3
    106 |
    107 |
(8 rows)
```

See also

- Best practices for spatial joins
- STV_Intersect: scalar function vs. transform function
- STV_Intersect transform function
- STV_Create_Index

STV_Intersect transform function

Spatially intersects points and polygons. The STV_Intersect transform function returns a tuple with matching point/polygon pairs. For every point, Vertica returns either one or many matching polygons.

You can improve performance when you parallelize the computation of the STV_Intersect transform function over multiple nodes. To parallelize the computation, use an OVER(PARTITION BEST) clause.

Behavior type

Immutable

Syntax

```
STV_Intersect ( { gid | i }, { g | x , y }
    USING PARAMETERS index='index_name')
    OVER() AS (pt_gid, pol_gid)
```

Arguments

gid* | *i

An integer column or integer that uniquely identifies the spatial object(s) of *g* or *x* and *y* .

g

A geometry or geography (WGS84) column that contains points. The *g* column can contain only point geometries or geographies. If the column contains a different geometry or geography type, STV_Intersect terminates with an error.

x

x-coordinate or longitude, FLOAT.

y

y-coordinate or latitude, FLOAT.

Parameters

index = 'index_name'

Name of the spatial index, of type VARCHAR.

Returns

pt_gid

Unique identifier of the point geometry or geography, of type INTEGER.

pol_gid

Unique identifier of the polygon geometry or geography, of type INTEGER.

Examples

The following examples show how you can use STV_Intersect transform.

Using two floats, return the matching point-polygon pairs.

```
=> CREATE TABLE pols (gid INT, geom GEOMETRY(1000));
CREATE TABLE
=> COPY pols(gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1|POLYGON((31 74,8 70,8 50,36 53,31 74))
>> \.
=> SELECT STV_Create_Index(gid, geom USING PARAMETERS index='my_polygons_1', overwrite=true,
                           max_mem_mb=256) OVER() FROM pols;
   type    | polygons | SRID | min_x | min_y | max_x | max_y | info
-----------+----------+------+-------+-------+-------+-------+------
 GEOMETRY |        1 |    0 |     8 |    50 |    36 |    74 |
(1 row)

=> SELECT STV_Intersect(56, 12.5683, 55.6761 USING PARAMETERS index = 'my_polygons_1') OVER();
 pt_gid | pol_gid
--------+---------
     56 |       1
(1 row)
```

Using a GEOMETRY column, return the matching point-polygon pairs.

```
=> CREATE TABLE polygons (gid int, geom GEOMETRY(700));
CREATE TABLE
=> COPY polygons (gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 10|POLYGON((5 5, 5 10, 10 10, 10 5, 5 5))
>> 11|POLYGON((0 0, 0 2, 2 2, 2 0, 0 0))
>> 12|POLYGON((1 1, 1 3, 3 3, 3 1, 1 1))
>> 14|POLYGON((-1 -1, -1 12, 12 12, 12 -1, -1 -1))
>> \.
=> SELECT STV_Create_Index(gid, geom USING PARAMETERS index='my_polygons', overwrite=true, max_mem_mb=256)
     OVER() FROM polygons;
   type   | polygons | SRID | min_x | min_y | max_x | max_y | info
----------+----------+------+-------+-------+-------+-------+------
 GEOMETRY |        4 |    0 |    -1 |    -1 |    12 |    12 |
(1 row)

=> CREATE TABLE points (gid INT, geom GEOMETRY(700));
CREATE TABLE
=> COPY points (gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1|POINT(9 9)
>> 2|POINT(0 1)
>> 3|POINT(2.5 2.5)
>> 4|POINT(0 0)
>> 5|POINT(1 5)
>> 6|POINT(1.5 1.5)
>> \.
=> SELECT STV_Intersect(gid, geom USING PARAMETERS index='my_polygons') OVER (PARTITION BEST)
    AS (point_id, polygon_gid)
      FROM points;
 point_id | polygon_gid
----------+-------------
        5 |          14
        1 |          14
        1 |          10
        4 |          14
        4 |          11
        6 |          12
        6 |          14
        6 |          11
        2 |          14
        2 |          11
        3 |          12
        3 |          14
(12 rows)
```

You can improve query performance by using the STV_Intersect transform function in a WHERE clause. Performance improves because this syntax eliminates all points that do not intersect polygons in the index.

Return the count of points that intersect with the polygon, where gid = 14:

```
=> SELECT COUNT(pt_id) FROM
   (SELECT STV_Intersect(gid, geom USING PARAMETERS index='my_polygons')
    OVER (PARTITION BEST) AS (pt_id, pol_id) FROM points)
      AS T WHERE pol_id = 14;
 COUNT
-------
     6
(1 row)
```

See also

- Best practices for spatial joins
- STV_Intersect: scalar function vs. transform function
- STV_Create_Index
- STV_Intersect scalar function

STV_IsValidReason

Determines if a spatial object is well formed or valid. If the object is not valid, STV_IsValidReason returns a string that explains where the invalidity occurs.

A polygon or multipolygon is valid if all of the following are true:

- The polygon is closed; its start point is the same as its end point.
- Its boundary is a set of linestrings.
- The boundary does not touch or cross itself.
- Any polygons in the interior that do not have more than one point touching the boundary of the exterior polygon.

If you pass an invalid object to a Vertica Place function, the function fails or returns incorrect results. To determine if a polygon is valid, first run ST_IsValid. ST_IsValid returns TRUE if the polygon is valid, FALSE otherwise.

> **Note**
> If you pass a valid polygon to STV_IsValidReason, it returns NULL.

Behavior type

Immutable

Syntax

```
STV_IsValidReason( g )
```

Arguments

g

Geospatial object to test for validity, value of type GEOMETRY or GEOGRAPHY (WGS84).

Returns

LONG VARCHAR

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	Yes	No	No
Multipoint	Yes	No	No
Linestring	Yes	No	No
Multilinestring	Yes	No	No
Polygon	Yes	No	Yes
Multipolygon	Yes	No	No
GeometryCollection	Yes	No	No

Examples

The following example shows how to use STV_IsValidReason.

Returns a string describing where the polygon is invalid:

```
=> SELECT STV_IsValidReason(ST_GeomFromText('POLYGON((1 3,3 2,1 1,
  3 0,1 0,1 3))'));
            STV_IsValidReason
-------------------------------------------------
 Ring Self-intersection at or near POINT (1 1)
(1 row)
```

See also

ST_IsValid

STV_LineStringPoint

Retrieves the vertices of a linestring or multilinestring. The values returned are points of either GEOMETRY or GEOGRAPHY type depending on the input object's type. GEOMETRY points inherit the SRID of the input object.

STV_LineStringPoint is an analytic function. For more information, see Analytic functions .

Behavior type

Immutable

Syntax

```
STV_LineStringPoint( g )
   OVER( [PARTITION NODES] ) AS
```

Arguments

g

Linestring or multilinestring, value of type GEOMETRY or GEOGRAPHY

Returns

GEOMETRY or GEOGRAPHY

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	No	No	No
Multipoint	No	No	No
Linestring	Yes	Yes	Yes
Multilinestring	Yes	Yes	Yes
Polygon	No	No	No
Multipolygon	No	No	No
GeometryCollection	No	No	No

Examples

The following examples show how to use STV_LineStringPoint.

Returns the vertices of the geometry linestring and their SRID:

```
=> SELECT ST_AsText(Point), ST_SRID(Point)
    FROM (SELECT STV_LineStringPoint(
        ST_GeomFromText('MULTILINESTRING((1 2, 2 3, 3 1, 4 2),
            (10 20, 20 30, 30 10, 40 20))', 4269)) OVER () AS Point) AS foo;
   ST_AsText   | ST_SRID
---------------+---------
 POINT (1 2)   |    4269
 POINT (2 3)   |    4269
 POINT (3 1)   |    4269
 POINT (4 2)   |    4269
 POINT (10 20) |    4269
 POINT (20 30) |    4269
 POINT (30 10) |    4269
 POINT (40 20) |    4269
(8 rows)
```

Returns the vertices of the geography linestring:

```
=> SELECT ST_AsText(g)
    FROM (SELECT STV_LineStringPoint(
     ST_GeographyFromText('MULTILINESTRING ((42.1 71.0, 41.4 70.0, 41.3 72.9),
        (42.99 71.46, 44.47 73.21)', 4269)) OVER () AS g) AS line_geog_points;
     ST_AsText
---------------------
 POINT (42.1 71.0)
 POINT (41.4 70.0)
 POINT (41.3 72.9)
 POINT (42.99 71.46)
 POINT (44.47 73.21)
(5 rows)
```

See also

STV_PolygonPoint

STV_MemSize

Returns the length of the spatial object in bytes as an INTEGER.

Use this function to determine the optimal column width for your spatial data.

Behavior type

Immutable

Syntax

```
STV_MemSize( g )
```

Arguments

g

Spatial object, value of type GEOMETRY or GEOGRAPHY

Returns

INTEGER

Examples

The following example shows how you can optimize your table by sizing the GEOMETRY or GEOGRAPHY column to the maximum value returned by STV_MemSize:

```
=> CREATE TABLE mem_size_table (id int, geom geometry(800));
CREATE TABLE
=> COPY mem_size_table (id, gx filler LONG VARCHAR, geom as ST_GeomFromText(gx)) FROM STDIN DELIMITER '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>>1|POINT(3 5)
>>2|MULTILINESTRING((1 5, 2 4, 5 3, 6 6),(3 5, 3 7))
>>3|MULTIPOLYGON(((2 6, 2 9, 6 9, 7 7, 4 6, 2 6)),((0 0, 0 5, 1 0, 0 0)),((0 2, 2 5, 4 5, 0 2)))
>>\.
=> SELECT max(STV_MemSize(geom)) FROM mem_size_table;
 max
-----
 336
(1 row)

=> CREATE TABLE production_table(id int, geom geometry(336));
CREATE TABLE
=> INSERT INTO production_table SELECT * FROM mem_size_table;
 OUTPUT
--------
      3
(1 row)
=> DROP mem_size_table;
DROP TABLE
```

STV_NN

Calculates the distance of spatial objects from a reference object and returns (object, distance) pairs in ascending order by distance from the reference object.

Parameters g1 and g2 must be both GEOMETRY objects or both GEOGRAPHY objects.

STV_NN is an analytic function. For more information, see Analytic functions .

Behavior type

Immutable

Syntax

```
STV_NN( g, ref_obj, k ) OVER()
```

Arguments

g

Spatial object, value of type GEOMETRY or GEOGRAPHY

ref_obj

Reference object, type GEOMETRY or GEOGRAPHY

k

Number of rows to return, type INTEGER

Returns

(Object, distance) pairs, in ascending order by distance. If a parameter is EMPTY or NULL, then 0 rows are returned.

Supported data types

Data Type	**GEOMETRY**	**GEOGRAPHY (Perfect Sphere)**
Point	Yes	Yes
Multipoint	Yes	Yes
Linestring	Yes	Yes
Multilinestring	Yes	Yes

Polygon	Yes	Yes
Multipolygon	Yes	Yes
GeometryCollection	Yes	No

Examples

The following example shows how to use STV_NN.

Create a table and insert nine GEOGRAPHY points:

```
=> CREATE TABLE points (g geography);
CREATE TABLE
=> COPY points (gx filler LONG VARCHAR, g AS ST_GeographyFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> POINT (21.5 18.4)
>> POINT (21.5 19.2)
>> POINT (21.5 20.7)
>> POINT (22.5 16.4)
>> POINT (22.5 17.15)
>> POINT (22.5 18.33)
>> POINT (23.5 13.68)
>> POINT (23.5 15.9)
>> POINT (23.5 18.4)
>> \.
```

Calculate the distances (in meters) of objects in table points from the GEOGRAPHY point (23.5, 20).

Returns the five objects that are closest to that point:

```
=> SELECT ST_AsText(nn), dist FROM (SELECT STV_NN(g,
   ST_GeographyFromText('POINT(23.5 20)'),5) OVER() AS (nn,dist) FROM points) AS example;
    ST_AsText     |       dist
--------------------+------------------
 POINT (23.5 18.4)  |  177912.12757541
 POINT (22.5 18.33) | 213339.210738322
 POINT (21.5 20.7)  |  222561.43679943
 POINT (21.5 19.2)  | 227604.371833335
 POINT (21.5 18.4)  | 275239.416790128
(5 rows)
```

STV_PolygonPoint

Retrieves the vertices of a polygon as individual points. The values returned are points of either GEOMETRY or GEOGRAPHY type depending on the input object's type. GEOMETRY points inherit the SRID of the input object.

STV_PolygonPoint is an analytic function. For more information, see Analytic functions .

Behavior type

Immutable

Syntax

```
STV_PolygonPoint( g )
   OVER( [PARTITION NODES] ) AS
```

Arguments

g

Polygon, value of type GEOMETRY or GEOGRAPHY

Returns

GEOMETRY or GEOGRAPHY

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (Perfect Sphere)	GEOGRAPHY (WGS84)
Point	No	No	No
Multipoint	No	No	No
Linestring	No	No	No
Multilinestring	No	No	No
Polygon	Yes	Yes	Yes
Multipolygon	Yes	Yes	Yes
GeometryCollection	No	No	No

Examples

The following examples show how to use STV_PolygonPoint.

Returns the vertices of the geometry polygon:

```
=> SELECT ST_AsText(g) FROM (SELECT STV_PolygonPoint(ST_GeomFromText('POLYGON((1 2, 2 3, 3 1, 1 2))'))
    OVER (PARTITION NODES) AS g) AS poly_points;
 ST_AsText
-------------
 POINT (1 2)
 POINT (2 3)
 POINT (3 1)
 POINT (1 2)
(4 rows)
```

Returns the vertices of the geography polygon:

```
=> SELECT ST_AsText(g) FROM (SELECT STV_PolygonPoint(ST_GeographyFromText('
            POLYGON((25.5 28.76, 28.83 29.13, 27.2 30.99, 25.5 28.76))'))
  OVER (PARTITION NODES) AS g) AS poly_points;
     ST_AsText
----------------------
 POINT (25.5 28.76)
 POINT (28.83 29.13)
 POINT (27.2 30.99)
 POINT (25.5 28.76)
(4 rows)
```

See also

STV_LineStringPoint

STV_Refresh_Index

Appends newly added or updated polygons and removes deleted polygons from an existing spatial index.

The OVER() clause must be empty.

Behavior type

Mutable

Syntax

```
STV_Refresh_Index( gid, g
        USING PARAMETERS index='index_name'
         [, skip_nonindexable_polygons={ true | false } ] )

       OVER()
         [ AS (type, polygons, srid, min_x, min_y, max_x, max_y, info,
       indexed, appended, updated, deleted) ]
```

Arguments

gid

Name of an integer column that uniquely identifies the polygon. The gid cannot be NULL.

g

Name of a geometry or geography (WGS84) column or expression that contains polygons and multipolygons. Only polygon and multipolygon can be indexed. Other shape types are excluded from the index.

Parameters

index = *'index_name'*

Name of the index, type VARCHAR. Index names cannot exceed 110 characters. The slash, backslash, and tab characters are not allowed in index names.

***skip_nonindexable_polygons* = { true | false }**

(Optional) BOOLEAN
In rare cases, intricate polygons (for instance, with too high resolution or anomalous spikes) cannot be indexed. These polygons are considered non-indexable. When set to False, non-indexable polygons cause the index creation to fail. When set to True, index creation can succeed by excluding non-indexable polygons from the index.

To review the polygons that were not able to be indexed, use STV_Describe_Index with the parameter list_polygon.

Default: False

Returns

type

Spatial object type of the index.

polygons

Number of polygons indexed.

SRID

Spatial reference system identifier.

min_x, min_y, max_x, max_y

Coordinates of the minimum bounding rectangle (MBR) of the indexed geometries. (min_x , min_y) are the south-west coordinates, and (max_x , max_y) are the north-east coordinates.

info

Lists the number of excluded spatial objects as well as their type that were excluded from the index.

indexed

Number of polygons indexed during the operation.

appended

Number of appended polygons.

updated

Number of updated polygons.

deleted

Number of deleted polygons.

Supported data types

Data Type	GEOMETRY	GEOGRAPHY (WGS84)
Point	No	No
Multipoint	No	No

Linestring	No	No
Multilinestring	No	No
Polygon	Yes	Yes
Multipolygon	Yes	No
GeometryCollection	No	No

Privileges

Any user with access to the STV_*_Index functions can describe, rename, or drop indexes created by any other user.

Limitations

- In rare cases, intricate polygons (such as those with too-high a resolution or anomalous spikes) cannot be indexed. See the parameter skip_nonindexable_polygons .
- If you replace a valid polygon in the source table with an invalid polygon, STV_Refresh_Index ignores the invalid polygon. As a result, the polygon originally indexed persists in the index.
- The following geometries cannot be indexed:
  - Non-polygons
  - NULL gid
  - NULL (multi) polygon
  - EMPTY (multi) polygon
  - Invalid (multi) polygon
- The following geographies are excluded from the index:
  - Polygons with holes
  - Polygons crossing the International Date Line
  - Polygons covering the north or south pole
  - Antipodal polygons

Usage tips

- To cancel an STV_Refresh_Index run, use **Ctrl** + **C** .
- If you use source data not previously associated with the index, then the index will be overwritten.
- If STV_Refresh_Index has insufficient memory to process the query, then rebuild the index using STV_Create_Index.
- If there are no valid polygons in the geom column, STV_Refresh_Index reports an error in vertica.log and stops the index refresh.
- Ensure that all of the polygons you plan to index are valid polygons. STV_Create_Index and STV_Refresh_Index do not check polygon validity when building an index.

  For more information, see Ensuring polygon validity before creating or refreshing an index .

Examples

The following examples show how to use STV_Refresh_Index.

Refresh an index with a single literal argument:

```
=> SELECT STV_Create_Index(1, ST_GeomFromText('POLYGON((0 0,0 15.2,3.9 15.2,3.9 0,0 0))')
    USING PARAMETERS index='my_polygon') OVER();
  type   | polygons | SRID | min_x | min_y | max_x | max_y | info
---------+----------+------+-------+-------+-------+-------+------
 GEOMETRY |        1 |    0 |     0 |     0 |   3.9 |  15.2 |
(1 row)

=> SELECT STV_Refresh_Index(2, ST_GeomFromText('POLYGON((0 0,0 13.2,3.9 18.2,3.9 0,0 0))')
    USING PARAMETERS index='my_polygon') OVER();
  type   | polygons | SRID | min_x | min_y | max_x | max_y | info | indexed | appended | updated | deleted
---------+----------+------+-------+-------+-------+-------+------+---------+----------+---------+---------
 GEOMETRY |        1 |    0 |     0 |     0 |   3.9 |  18.2 |      |       1 |        1 |       0 |       1
(1 row)
```

Refresh an index from a table:

```
=> CREATE TABLE pols (gid INT, geom GEOMETRY);
CREATE TABLE
=> COPY pols(gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1|POLYGON((-31 74,8 70,8 50,-36 53,-31 74))
>> 2|POLYGON((5 20,9 30,20 45,36 35,5 20))
>> 3|POLYGON((12 23,9 30,20 45,36 35,37 67,45 80,50 20,12 23))
>> \.
=> SELECT STV_Create_Index(gid, geom USING PARAMETERS index='my_polygons_1', overwrite=true)
    OVER() FROM pols;
   type   | polygons | SRID | min_x | min_y | max_x | max_y | info
----------+----------+------+-------+-------+-------+-------+------
 GEOMETRY |        3 |    0 |   -36 |    20 |    50 |    80 |
(1 row)

=> COPY pols(gid, gx filler LONG VARCHAR, geom AS ST_GeomFromText(gx)) FROM stdin delimiter '|';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 6|POLYGON((-32 74,8 70,8 50,-36 53,-32 74))
>> \.
=> SELECT STV_Refresh_Index(gid, geom USING PARAMETERS index='my_polygons_1') OVER() FROM pols;
   type   | polygons | SRID | min_x | min_y | max_x | max_y | info | indexed | appended | updated | deleted
----------+----------+------+-------+-------+-------+-------+------+---------+----------+---------+---------
 GEOMETRY |        4 |    0 |   -36 |    20 |    50 |    80 |      |       1 |        1 |       0 |       0
(1 row)
```

See also

- STV_Create_Index
- STV_Describe_Index
- STV_Drop_Index
- STV_Rename_Index
- Ensuring polygon validity before creating or refreshing an index

STV_Rename_Index

Renames a spatial index. If the index format is out of date, you cannot rename the index.

A spatial index is created from an input polygon set, which can be the result of a query. Spatial indexes are created in a global name space. Vertica uses a distributed plan whenever the input table or projection is segmented across nodes of the cluster.

The OVER() clause must be empty.

Behavior type

Immutable

Syntax

```
STV_Rename_Index( USING PARAMETERS
                  source = 'old_index_name',
                  dest = 'new_index_name',
                  overwrite = [ 'true' | 'false' ]
                )
                 OVER ()
```

Arguments

source = ' *old_index_name* '

Current name of the spatial index, type VARCHAR.

dest = ' *new_index_name* '

New name of the spatial index, type VARCHAR.

overwrite = ['true' | 'false']

Boolean, specifies whether to overwrite the index, if an index exists. This parameter cannot be NULL.

Default: False

Privileges

Any user with access to the STV_*_Index functions can describe, rename, or drop indexes created by any other user.

Limitations

- Index names cannot exceed 110 characters.
- The backslash or tab characters are not allowed in index names.

Examples

The following example shows how to use STV_Rename_Index.

Rename an index:

```
=> SELECT STV_Rename_Index (
     USING PARAMETERS
     source = 'my_polygons',
     dest = 'US_states',
     overwrite = 'false'
     )
     OVER ();
 rename_index
---------------
 Index renamed
(1 Row)
```

STV_Reverse

Reverses the order of the vertices of a spatial object.

Behavior type

Immutable

Syntax

```
STV_Reverse( g, [USING PARAMETERS skip_nonreorientable_polygons={true | false} ])
```

Arguments

g

Spatial object, type GEOGRAPHY.

skip_nonreorientable_polygons = { true | false }

(Optional) Boolean
When set to False, non-orientable polygons generate an error. For example, if you use STV_ForceLHR or STV_Reverse with skip_nonorientable_polygons set to False, a geography polygon containing a hole generates an error. When set to True, the result returned is the polygon, as passed to the API, without alteration.

This argument can help you when you are creating an index from a table containing polygons that cannot be re-oriented.

Vertica Place considers these polygons non-orientable:

- Polygons with a hole
- Multipolygons
- Multipolygons with a hole

Default value : False

Returns

GEOGRAPHY

Supported data types

Data Type	**GEOMETRY**	**GEOGRAPHY (Perfect Sphere)**	**GEOGRAPHY (WGS84)**
Point	No	No	No

Multipoint	No	No	No
Linestring	No	No	No
Multilinestring	No	No	No
Polygon	No	Yes	Yes
Multipolygon	No	Yes	Yes
GeometryCollection	No	No	No

Examples

The following examples show how you can use STV_Reverse.

Reverse vertices of a geography polygon:

```
=> SELECT ST_AsText(STV_Reverse(ST_GeographyFromText('Polygon((1 1, 3 1, 2 2, 1 1))')));
          ST_AsText
--------------------------------
 POLYGON ((1 1, 2 2, 3 1, 1 1))
(1 row)
```

Force the polygon to reverse orientation:

```
=> SELECT ST_AsText(STV_Reverse(ST_GeographyFromText('Polygon((1 1, 2 2, 3 1, 1 1))')));
ST_AsText
--------------------------------
 POLYGON ((1 1, 3 1, 2 2, 1 1))
(1 row)
```

See also

STV_ForceLHR

STV_SetExportShapefileDirectory

Specifies the directory to export GEOMETRY or GEOGRAPHY data to a shapefile. The validity of the path is not checked, and the path cannot be empty.

Behavior type

Immutable

Syntax

```
STV_SetExportShapefileDirectory( USING PARAMETERS path='shapefile_path' )
```

Arguments

path = ' *shapefile_path* '

The path where you want the shapefile exported. For example, '/home/user/temp'. You can also export to a mounted S3 directory where you have read and write permissions using the convention '<bucketname>/path'.

Returns

The path of the shapefile export directory.

Privileges

Only a superuser can use this function.

Examples

The following example shows how you can use STV_SetExportShapefileDirectory to set the shapefile export directory to /home/user/temp:

```
=> SELECT STV_SetExportShapefileDirectory(USING PARAMETERS path = '/home/user/temp');
            STV_SetExportShapefileDirectory
-----------------------------------------------------------------
 SUCCESS. Set shapefile export directory: [/home/user/temp]
(1 row)
```

STV_ShpCreateTable

Returns a CREATE TABLE statement with the columns and types of the attributes found in the specified shapefile.

The column types are sized according to the shapefile metadata. The size of the column is based on the largest geometry found in the shapefile. The first column in the table is gid , which is an IDENTITY primary key column. The cache value is set to 64 by default. The last column is a GEOMETRY data type for storing the actual geometry data.

Behavior type

Immutable

Syntax

```
STV_ShpCreateTable (USING PARAMETERS file='filename') OVER()
```

Arguments

file = ' *filename* '

Fully qualified path of the .dbf , .shp , or .shx file (file extension optional).

You can also create a table using a shapefile stored on a mounted S3 directory where you have read and write permissions. Use the following syntax:

```
bucketname/path/filename
```

Returns

CREATE TABLE statement that matches the specified shapefile

Usage tips

- STV_ShpCreateTable returns a CREATE TABLE statement; but it does not create the table. Modify the CREATE TABLE statement as needed, and then create the table before loading the shapefile into the table.
- To create a table with characters other than alphanumeric and underscore (_) characters, you must specify the table name enclosed in double quotes, such as "counties%NY" .
- The name of the table is the same as the name of the shapefile, without the directory name or extension.
- The shapefile must be accessible from the initiator node.
- If the .shp and .shx files are corrupt, STV_ShpCreateTable returns an error. If the .shp and .shx files are valid, but the .dbf file is corrupt, STV_ShpCreateTable ignores the .dbf file and does not create columns for that data.
- All the mandatory files (.dbf , .shp , .shx) must be in the same directory. If not, STV_ShpCreateTable returns an error.
- If the .dbf component of a shapefile contains a Numeric attribute, this field's values may lose precision when the Vertica shapefile loader loads it into a table. The target field is a 64-bit FLOAT column, which can only represent about 15 significant digits. In a .dbf file, numeric fields can be up to 30 digits.
  Vertica records all instances of shapefile values that are too long in the vertica.log file.

Examples

The following example shows how to use STV_ShpCreateTable.

Returns a CREATE TABLE statement:

```
=> SELECT STV_ShpCreateTable
    (USING PARAMETERS file='/shapefiles/tl_2010_us_state10.shp')
    OVER() as create_table_states;
    create_table_states
 ---------------------------------
CREATE TABLE tl_2010_us_state10(
  gid IDENTITY(64) PRIMARY KEY,
  REGION10 VARCHAR(2),
  DIVISION10 VARCHAR(2),
  STATEFP10 VARCHAR(2),
  STATENS10 VARCHAR(8),
  GEOID10 VARCHAR(2),
  STUSPS10 VARCHAR(2),
  NAME10 VARCHAR(100),
  LSAD10 VARCHAR(2),
  MTFCC10 VARCHAR(5),
  FUNCSTAT10 VARCHAR(1),
  ALAND10 INT8,
  AWATER10 INT8,
  INTPTLAT10 VARCHAR(11),
  INTPTLON10 VARCHAR(12),
  geom GEOMETRY(940845)
);
(18 rows)
```

See also

- STV_ShpSource and STV_ShpParser

STV_ShpSource and STV_ShpParser

These two functions work with COPY to parse and load geometries and attributes from a shapefile into a Vertica table, and convert them to the appropriate GEOMETRY data type. You must use these two functions together.

The following restrictions apply:

- An empty multipoint or an invalid multipolygon can not be loaded from a shapefile.
- If the .dbf component of a shapefile contains a numeric attribute, this field's values might lose precision when the Vertica Place shapefile loader loads it into a table. The target field is a 64-bit FLOAT column, which can only represent about 15 significant digits; in a .dbf file, Numeric fields can be up to 30 digits.

Rejected records are saved to CopyErrorLogs subdirectory, under the Vertica catalog directory.

Behavior type

Immutable

Syntax

```
COPY table( columnslist )
   WITH SOURCE STV_ShpSource
      ( file = 'path'[[, SRID=spatial-reference-identifier] [, flatten_2d={true | false }] ] )
   PARSER STV_ShpParser()
```

Arguments

table

Name of the table in which to load the geometry data.

columnslist

Comma-delimited list of column names in the table that match fields in the external file. Run the CREATE TABLE command that STV_ShpCreateTable creates. When you do so, these columns correspond to the second through the second-to-last columns.

file = ' *pathname* '

Specifies the fully qualified path of a .dbf , .shp , or .shx file.

You can also load from a shapefile that is stored on a mounted S3 directory where you have read and write permissions. In this case, use the following the syntax:

```
bucketname/path/filename
```

SRID= ***spatial-reference-identifier***

Specifies an integer spatial reference identifier (SRID) associated with the shape file.

flatten_2d

Specifies a BOOLEAN argument that excludes 3D or 4D coordinates during COPY commands:

- true : Excludes geometries with 3D or 4D coordinates before a COPY command.
- false : Causes the load to fail if a geometry with 3D or 4D coordinate is found.

Default: false

Privileges

- Source shapefile: Read
- Shapefile directory: Execute

COPY errors

The COPY command fails under one of the following conditions:

- The shapefile cannot be located or opened.
- The number of columns or the data types of the columns that STV_ShpParser creates do not match the columns in the destination table. Use STV_ShpCreateTable to generate the appropriate CREATE TABLE command.
- One of the mandatory files is missing or cannot be opened. When opening a shapefile, you must have three files: .dbf , .shp , and .shx .

STV_ShpSource file corruption handling

- If the .shp and .shx files are corrupt, STV_ShpSource returns an error.
- If the .shp and .shx files are valid, but the .dbf file is corrupt, STV_ShpSource ignores the .dbf file and does not create columns for that data.

Examples

```
=> COPY tl_2010_us_state10 WITH SOURCE
STV_ShpSource(file='/shapefiles/tl_2010_us_state10.shp', SRID=4269) PARSER STV_ShpParser();

 Rows loaded
-------------
          52
```

Hadoop functions

This section contains functions to manage interactions with Hadoop.

In this section

- CLEAR_HDFS_CACHES
- EXTERNAL_CONFIG_CHECK
- GET_METADATA
- HADOOP_IMPERSONATION_CONFIG_CHECK
- HASH_EXTERNAL_TOKEN
- HCATALOGCONNECTOR_CONFIG_CHECK
- HDFS_CLUSTER_CONFIG_CHECK
- KERBEROS_HDFS_CONFIG_CHECK
- SYNC_WITH_HCATALOG_SCHEMA
- SYNC_WITH_HCATALOG_SCHEMA_TABLE
- VERIFY_HADOOP_CONF_DIR

CLEAR_HDFS_CACHES

Clears the configuration information copied from HDFS and any cached connections.

This function affects reads using the hdfs scheme in the following ways:

- This function flushes information loaded from configuration files copied from Hadoop (such as core-site.xml). These files are found on the path set by the HadoopConfDir configuration parameter.
- This function flushes information about which NameNode is active in a High Availability (HA) Hadoop cluster. Therefore, the first request to Hadoop after calling this function is slower than expected.

Vertica maintains a cache of open connections to NameNodes to reduce latency. This function flushes that cache.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAR_HDFS_CACHES ( )
```

Privileges

Superuser

Examples

The following example clears the Hadoop configuration information:

```
=> SELECT CLEAR_HDFS_CACHES();
 CLEAR_HDFS_CACHES
--------------
 Cleared
(1 row)
```

See also

Hadoop parameters

EXTERNAL_CONFIG_CHECK

Tests the Hadoop configuration of a Vertica cluster. This function tests HDFS configuration files, HCatalog Connector configuration, and Kerberos configuration.

This function calls the following functions:

- KERBEROS_CONFIG_CHECK
- HADOOP_IMPERSONATION_CONFIG_CHECK
- HDFS_CLUSTER_CONFIG_CHECK
- HCATALOGCONNECTOR_CONFIG_CHECK

If you call this function with an argument, it passes the argument to functions it calls that also take an argument.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
EXTERNAL_CONFIG_CHECK( ['what_to_test' ] )
```

Arguments

what_to_test

A string specifying the authorities, nameservices, and/or HCatalog schemas to test. The format is a comma-separated list of "key=value" pairs, where keys are "authority", "nameservice", and "schema". The value is passed to all of the sub-functions; see those reference pages for details on how values are interpreted.

Privileges

This function does not require privileges.

Examples

The following example tests the configuration of only the nameservice named "ns1". Output has been omitted due to length.

```
=> SELECT EXTERNAL_CONFIG_CHECK('nameservice=ns1');
```

GET_METADATA

Returns the metadata of a Parquet file. Metadata includes the number and sizes of row groups, column names, and information about chunks and compression. Metadata is returned as JSON.

This function inspects one file. Parquet data usually spans many files in a single directory; choose one. The function does not accept a directory name as an argument.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_METADATA( 'filename' )
```

Arguments

filename

The name of a Parquet file. Any path that is valid for COPY is valid for this function. This function does not operate on files in other formats.

Privileges

Superuser, or non-superuser with READ privileges on the USER-accessible storage location (see GRANT (storage location)).

Examples

You must call this function with a single file, not a directory or glob:

```
=> SELECT GET_METADATA('/data/emp-row.parquet');
               GET_METADATA
-----------------------------------------------------------------------------------------------------
 schema:
required group field_id=-1 spark_schema {
  optional int32 field_id=-1 employeeID;
  optional group field_id=-1 personal {
    optional binary field_id=-1 name (String);
    optional group field_id=-1 address {
      optional binary field_id=-1 street (String);
      optional binary field_id=-1 city (String);
      optional int32 field_id=-1 zipcode;
    }
    optional int32 field_id=-1 taxID;
  }
  optional binary field_id=-1 department (String);
}

 data page version:
  data page v1

 metadata:
{
  "FileName": "/data/emp-row.parquet",
  "FileFormat": "Parquet",
  "Version": "1.0",
  "CreatedBy": "parquet-mr version 1.10.1 (build a89df8f9932b6ef6633d06069e50c9b7970bebd1)",
  "TotalRows": "4",
  "NumberOfRowGroups": "1",
  "NumberOfRealColumns": "3",
  "NumberOfColumns": "7",
  "Columns": [
    { "Id": "0", "Name": "employeeID", "PhysicalType": "INT32", "ConvertedType": "NONE", "LogicalType": {"Type": "None"} },
    { "Id": "1", "Name": "personal.name", "PhysicalType": "BYTE_ARRAY", "ConvertedType": "UTF8", "LogicalType": {"Type": "String"} },
    { "Id": "2", "Name": "personal.address.street", "PhysicalType": "BYTE_ARRAY", "ConvertedType": "UTF8", "LogicalType": {"Type": "String"} },
    { "Id": "3", "Name": "personal.address.city", "PhysicalType": "BYTE_ARRAY", "ConvertedType": "UTF8", "LogicalType": {"Type": "String"} },
    { "Id": "4", "Name": "personal.address.zipcode", "PhysicalType": "INT32", "ConvertedType": "NONE", "LogicalType": {"Type": "None"} },
    { "Id": "5", "Name": "personal.taxID", "PhysicalType": "INT32", "ConvertedType": "NONE", "LogicalType": {"Type": "None"} },
    { "Id": "6", "Name": "department", "PhysicalType": "BYTE_ARRAY", "ConvertedType": "UTF8", "LogicalType": {"Type": "String"} }
  ]
```

```
 },
 "RowGroups": [
   {
    "Id": "0",  "TotalBytes": "642",  "TotalCompressedBytes": "0",  "Rows": "4",
    "ColumnChunks": [
      {"Id": "0", "Values": "4", "StatsSet": "True", "Stats": {"NumNulls": "0", "DistinctValues": "0", "Max": "51513", "Min": "17103" },
       "Compression": "SNAPPY", "Encodings": "PLAIN RLE BIT_PACKED ", "UncompressedSize": "67", "CompressedSize": "69" },
      {"Id": "1", "Values": "4", "StatsSet": "True", "Stats": {"NumNulls": "0", "DistinctValues": "0", "Max": "Sheldon Cooper", "Min": "Howard Wolowitz" },
       "Compression": "SNAPPY", "Encodings": "PLAIN RLE BIT_PACKED ", "UncompressedSize": "142", "CompressedSize": "145" },
      {"Id": "2", "Values": "4", "StatsSet": "True", "Stats": {"NumNulls": "0", "DistinctValues": "0", "Max": "52 Broad St", "Min": "100 Main St Apt 4A" },
       "Compression": "SNAPPY", "Encodings": "PLAIN RLE BIT_PACKED ", "UncompressedSize": "139", "CompressedSize": "123" },
      {"Id": "3", "Values": "4", "StatsSet": "True", "Stats": {"NumNulls": "0", "DistinctValues": "0", "Max": "Pasadena", "Min": "Pasadena" },
       "Compression": "SNAPPY", "Encodings": "RLE PLAIN_DICTIONARY BIT_PACKED ", "UncompressedSize": "95", "CompressedSize": "99" },
      {"Id": "4", "Values": "4", "StatsSet": "True", "Stats": {"NumNulls": "0", "DistinctValues": "0", "Max": "91021", "Min": "91001" },
       "Compression": "SNAPPY", "Encodings": "PLAIN RLE BIT_PACKED ", "UncompressedSize": "68", "CompressedSize": "70" },
      {"Id": "5", "Values": "4", "StatsSet": "True", "Stats": {"NumNulls": "4", "DistinctValues": "0", "Max": "0", "Min": "0" },
       "Compression": "SNAPPY", "Encodings": "PLAIN RLE BIT_PACKED ", "UncompressedSize": "28", "CompressedSize": "30" },
      {"Id": "6", "Values": "4", "StatsSet": "True", "Stats": {"NumNulls": "0", "DistinctValues": "0", "Max": "Physics", "Min": "Astronomy" },
       "Compression": "SNAPPY", "Encodings": "RLE PLAIN_DICTIONARY BIT_PACKED ", "UncompressedSize": "103", "CompressedSize": "107" }
    ]
   }
 ]
}

(1 row)
```


HADOOP_IMPERSONATION_CONFIG_CHECK

Reports the delegation tokens Vertica will use when accessing Kerberized data in HDFS. The HadoopImpersonationConfig configuration parameter specifies one or more authorities, nameservices, and HCatalog schemas and their associated tokens. For each tested value, the function reports what doAs user or delegation token Vertica will use for access. Use this function to confirm that you have defined your delegation tokens as you intended.

You can call this function with an argument to specify the authority, nameservice, or HCatalog schema to test, or without arguments to test all configured values.

This function does not check that you can use these delegation tokens to access HDFS.

See Proxy users and delegation tokens for more about impersonation.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
HADOOP_IMPERSONATION_CONFIG_CHECK( ['what_to_test' ] )
```

Arguments

what_to_test

A string specifying the authorities, nameservices, and/or HCatalog schemas to test. For example, a value of 'nameservice=ns1' means the function tests only access to the nameservice "ns1" and ignores any other authorities and schemas. A value of 'nameservice=ns1, schema=hcat1' means the function tests one nameservice and one HCatalog schema.

If you do not specify this argument, the function tests all authorities, nameservices, and schemas defined in HadoopImpersonationConfig .

Privileges

This function does not require privileges.

Examples

Consider the following definition of HadoopImpersonationConfig:

```
[{
      "nameservice": "ns1",
      "token": "RANDOM-TOKEN-STRING"
   },
   {
      "nameservice": "*",
      "doAs": "Paul"
   },
   {
      "schema": "hcat1",
      "doAs": "Fred"
   }
]
```

The following query tests only the "ns1" name service:

```
=> SELECT HADOOP_IMPERSONATION_CONFIG_CHECK('nameservice=ns1');

-- hadoop_impersonation_config_check --
Connections to nameservice [ns1] will use a delegation token with hash [b3dd9e71cd695d91]
```

This function returns a hash of the token for security reasons. You can call HASH_EXTERNAL_TOKEN with the expected value and compare that hash to the one in this function's output.

A query with no argument tests all values:

```
=> SELECT HADOOP_IMPERSONATION_CONFIG_CHECK();

-- hadoop_impersonation_config_check --
Connections to nameservice [ns1] will use a delegation token with hash [b3dd9e71cd695d91]
JDBC connections for HCatalog schema [hcat1] will doAs [Fred]
[!] hadoop_impersonation_config_check : [PASS]
```

HASH_EXTERNAL_TOKEN

Returns a hash of a string token, for use with HADOOP_IMPERSONATION_CONFIG_CHECK . Call HASH_EXTERNAL_TOKEN with the delegation token you expect Vertica to use and compare it to the hash in the output of HADOOP_IMPERSONATION_CONFIG_CHECK .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
HASH_EXTERNAL_TOKEN( 'token' )
```

Arguments

token

A string specifying the token to hash. The token is configured in the HadoopImpersonationConfig parameter.

Privileges

This function does not require privileges.

Examples

The following query tests the expected value shown in the example on the HADOOP_IMPERSONATION_CONFIG_CHECK reference page.

```
=> SELECT HASH_EXTERNAL_TOKEN('RANDOM-TOKEN-STRING');
hash_external_token
--------------------
b3dd9e71cd695d91
(1 row)
```

HCATALOGCONNECTOR_CONFIG_CHECK

Tests the configuration of a Vertica cluster that uses the HCatalog Connector to access Hive data. The function first verifies that the HCatalog

Connector is properly installed and reports on the values of several related configuration parameters. It then tests the connection using HiveServer2. This function does not support the WebHCat server.

If you specify an HCatalog schema, and if you have defined a delegation token for that schema, this function uses the delegation token. Otherwise, the function uses the default endpoint without a delegation token.

See Proxy users and delegation tokens for more about delegation tokens.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
HCATALOGCONNECTOR_CONFIG_CHECK( ['what_to_test' ] )
```

Arguments

what_to_test

A string specifying the HCatalog schemas to test. For example, a value of 'schema=hcat1' means the function tests only the "hcat1" schema and ignores any others that are found.

Privileges

This function does not require privileges.

Examples

The following query tests with the default endpoint and no delegation token.

```
=> SELECT HCATALOGCONNECTOR_CONFIG_CHECK();

-- hcatalogconnector_config_check --

   HCatalogConnectorUseHiveServer2 : [1]
   EnableHCatImpersonation : [1]
   HCatalogConnectorUseORCReader : [1]
   HCatalogConnectorUseParquetReader : [1]
   HCatalogConnectorUseTxtReader : [0]
 [INFO] Vertica is not configured to use its internal parsers for delimited files.
 [INFO] This is off by default, but will be changed in a future release.
   HCatalogConnectorUseLibHDFSPP : [1]

 [OK] HCatalog connector library is properly installed.
 [INFO] Creating JDBC connection as session user.
 [OK] Successful JDBC connection to HiveServer2 as user [USER].

 [!] hcatalogconnector_config_check : [PASS]
```

To test with the configured delegation token, pass the schema as an argument:

```
=> SELECT HCATALOGCONNECTOR_CONFIG_CHECK('schema=hcat1');
```

HDFS_CLUSTER_CONFIG_CHECK

Tests the configuration of a Vertica cluster that uses HDFS. The function scans the Hadoop configuration files found in HadoopConfDir and performs configuration checks on each cluster it finds. If you have more than one cluster configured, you can specify which one to test instead of testing all of them.

For each Hadoop cluster, it reports properties including:

- Nameservice name and associated NameNodes
- High-availability status
- RPC encryption status
- Kerberos authentication status
- HTTP(S) status

It then tests connections using http(s) , hdfs , and webhdfs URL schemes. It tests the latter two using both the Vertica and session user.

See Configuring HDFS access for information about configuration files and HadoopConfDir.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
HDFS_CLUSTER_CONFIG_CHECK( ['what_to_test' ] )
```

Arguments

what_to_test

A string specifying the authorities or nameservices to test. For example, a value of 'nameservice=ns1' means the function tests only "ns1" cluster. If you specify both an authority and a nameservice, the authority must be a NameNode in the specified nameservice for the check to pass.

If you do not specify this argument, the function tests all cluster configurations found in HadoopConfDir.

Privileges

This function does not require privileges.

Examples

The following example tests all clusters.

```
=> SELECT HDFS_CLUSTER_CONFIG_CHECK();

-- hdfs_cluster_config_check --

   Hadoop Conf Path : [/conf/hadoop_conf]
 [OK] HadoopConfDir verified on all nodes
   Connection Timeout (seconds) : [60]
   Token Refresh Frequency (seconds) : [0]
   HadoopFSBlockSizeBytes (MiB) : [64]

 [OK] Found [1] hadoop cluster configurations

------------- Cluster 1 -------------
   Is DefaultFS : [true]
   Nameservice : [vmns]
   Namenodes : [node1.example.com:8020, node2.example.com:8020]
   High Availability : [true]
   RPC Encryption : [false]
   Kerberos Authentication : [true]
   HTTPS Only : [false]
 [INFO] Checking connections to [hdfs:///]
   vertica : [OK]
   dbuser : [OK]

 [INFO] Checking connections to [http://node1.example.com:50070]
 [INFO] Node is in standby
 [INFO] Checking connections to [http://node2.example.com:50070]
 [OK] Can make authenticated external curl connection
 [INFO] Checking webhdfs
   vertica : [OK]
   USER : [OK]

 [!] hdfs_cluster_config_check : [PASS]
```

KERBEROS_HDFS_CONFIG_CHECK

> **Deprecated**
> This function is deprecated and will be removed in a future release. Instead, use EXTERNAL_CONFIG_CHECK.

Tests the Kerberos configuration of a Vertica cluster that uses HDFS. The function succeeds if it can use both the Vertica keytab file and the session user to access HDFS, and reports errors otherwise. This function is a more specific version of KERBEROS_CONFIG_CHECK .

If the current session is not Kerberized, this function will not be able to use secured HDFS connections and will fail.

You can call this function with arguments to specify an HDFS configuration to test, or without arguments. If you call it with no arguments, this function reads the HDFS configuration files and fails if it does not find them. See Configuring HDFS access . If it finds configuration files, it tests all configured nameservices.

The function performs the following tests, in order:

- Are Kerberos services available?
- Does a keytab file exist and are the Kerberos and HDFS configuration parameters set in the database?
- Can Vertica read and invoke kinit with the keys to authenticate to HDFS and obtain the database Kerberos ticket?
- Can Vertica perform hdfs and webhdfs operations using both the database Kerberos ticket and user-forwardable tickets for the current session?
- Can Vertica connect to HiveServer2? (This function does not support WebHCat.)

If any test fails, the function returns a descriptive error message.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
KERBEROS_HDFS_CONFIG_CHECK( ['hdfsHost:hdfsPort',
 'webhdfsHost:webhdfsPort', 'webhcatHost' ] )
```

Arguments

hdfsHost* , *hdfsPort

The hostname or IP address and port of the HDFS NameNode. Vertica uses this server to access data that is specified with hdfs URLs. If the value is ' ', the function skips this part of the check.

webhdfsHost* , *webhdfsPort

The hostname or IP address and port of the WebHDFS server. Vertica uses this server to access data that is specified with webhdfs URLs. If the value is ' ', the function skips this part of the check.

webhcatHost

Pass any value in this position. WebHCat is deprecated and this value is ignored but must be present.

Privileges

This function does not require privileges.

SYNC_WITH_HCATALOG_SCHEMA

Copies the structure of a Hive database schema available through the HCatalog Connector to a Vertica schema. If the HCatalog schema and the target Vertica schema have matching table names, SYNC_WITH_HCATALOG_SCHEMA overwrites the Vertica tables.

This function can synchronize the HCatalog schema directly. In this case, call it with the same schema name for the *vertica_schema* and *hcatalog_schema* parameters. The function can also synchronize a different schema to the HCatalog schema.

If you change the settings of HCatalog Connector configuration parameters , you must call this function again.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SYNC_WITH_HCATALOG_SCHEMA( vertica_schema, hcatalog_schema, [drop_non_existent] )
```

Parameters

vertica_schema

The target Vertica schema to store the copied HCatalog schema's metadata. This can be the same schema as *hcatalog_schema* , or it can be a separate one created with CREATE SCHEMA .

> **Caution**
> Do not use the Vertica schema to store other data.

hcatalog_schema

The HCatalog schema to copy, created with CREATE HCATALOG SCHEMA

drop_non_existent

If true , drop any tables in *vertica_schema* that do not correspond to a table in *hcatalog_schema*

Privileges

Non-superuser: CREATE privileges on *vertica_schema* .

Users also require access to Hive data, one of the following:

- USAGE permissions on *hcat_schema* , if Hive does not use an authorization service to manage access.
- Permission through an authorization service (Sentry or Ranger), and access to the underlying files in HDFS. (Sentry can provide that access through ACL synchronization.)
- dbadmin user privileges, with or without an authorization service.

Data type matching

Hive STRING and BINARY data types are matched, in Vertica, to the VARCHAR(65000) and VARBINARY(65000) types. Adjust the data types with ALTER TABLE as needed after creating the schema. The maximum size of a VARCHAR or VARBINARY in Vertica is 65000, but you can use LONG VARCHAR and LONG VARBINARY to specify larger values.

Hive and Vertica define string length in different ways. In Hive the length is the number of characters; in Vertica it is the number of bytes. Thus, a character encoding that uses more than one byte, such as Unicode, can cause mismatches between the two. To avoid data truncation, set values in Vertica based on bytes, not characters.

If data size exceeds the column size, Vertica logs an event at read time in the QUERY_EVENTS system table.

Examples

The following example uses SYNC_WITH_HCATALOG_SCHEMA to synchronize an HCatalog schema named hcat:

```
=> CREATE HCATALOG SCHEMA hcat WITH hostname='hcathost' HCATALOG_SCHEMA='default'
   HCATALOG_USER='hcatuser';
CREATE SCHEMA
=> SELECT sync_with_hcatalog_schema('hcat', 'hcat');
sync_with_hcatalog_schema
-----------------------------------------
Schema hcat synchronized with hcat
tables in hcat = 56
tables altered in hcat = 0
tables created in hcat = 56
stale tables in hcat = 0
table changes erred in hcat = 0
(1 row)

=> -- Use vsql's \d command to describe a table in the synced schema

=> \d hcat.messages
List of Fields by Tables
 Schema  |  Table  | Column |      Type      | Size  | Default | Not Null | Primary Key | Foreign Key
---------+---------+--------+----------------+-------+---------+----------+-------------+-------------
hcat     | messages | id     | int            |     8 |         | f        | f           |
hcat     | messages | userid  | varchar(65000) | 65000 |         | f        | f           |
hcat     | messages | "time"  | varchar(65000) | 65000 |         | f        | f           |
hcat     | messages | message | varchar(65000) | 65000 |         | f        | f           |
(4 rows)
```

The following example uses SYNC_WITH_HCATALOG_SCHEMA followed by ALTER TABLE to adjust a column value:

```
=> CREATE HCATALOG SCHEMA hcat WITH hostname='hcathost' HCATALOG_SCHEMA='default'
-> HCATALOG_USER='hcatuser';
CREATE SCHEMA
=> SELECT sync_with_hcatalog_schema('hcat', 'hcat');
...
=> ALTER TABLE hcat.t ALTER COLUMN a1 SET DATA TYPE long varchar(1000000);
=> ALTER TABLE hcat.t ALTER COLUMN a2 SET DATA TYPE long varbinary(1000000);
```

The following example uses SYNC_WITH_HCATALOG_SCHEMA with a local (non-HCatalog) schema:

```
=> CREATE HCATALOG SCHEMA hcat WITH hostname='hcathost' HCATALOG_SCHEMA='default'
-> HCATALOG_USER='hcatuser';
CREATE SCHEMA
=> CREATE SCHEMA hcat_local;
CREATE SCHEMA
=> SELECT sync_with_hcatalog_schema('hcat_local', 'hcat');
```

SYNC_WITH_HCATALOG_SCHEMA_TABLE

Copies the structure of a single table in a Hive database schema available through the HCatalog Connector to a Vertica table.

This function can synchronize the HCatalog schema directly. In this case, call it with the same schema name for the *vertica_schema* and *hcatalog_schema* parameters. The function can also synchronize a different schema to the HCatalog schema.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SYNC_WITH_HCATALOG_SCHEMA_TABLE( vertica_schema, hcatalog_schema, table_name )
```

Parameters

vertica_schema

The existing Vertica schema to store the copied HCatalog schema's metadata. This can be the same schema as *hcatalog_schema* , or it can be a separate one created with CREATE SCHEMA .

hcatalog_schema

The HCatalog schema to copy, created with CREATE HCATALOG SCHEMA .

table_name

The table in *hcatalog_schema* to copy. If *table_name* already exists in *vertica_schema* , the function overwrites it.

Privileges

Non-superuser: CREATE privileges on *vertica_schema* .

Users also require access to Hive data, one of the following:

- USAGE permissions on *hcat_schema* , if Hive does not use an authorization service to manage access.
- Permission through an authorization service (Sentry or Ranger), and access to the underlying files in HDFS. (Sentry can provide that access through ACL synchronization.)
- dbadmin user privileges, with or without an authorization service.

Data type matching

Hive STRING and BINARY data types are matched, in Vertica, to the VARCHAR(65000) and VARBINARY(65000) types. Adjust the data types with ALTER TABLE as needed after creating the schema. The maximum size of a VARCHAR or VARBINARY in Vertica is 65000, but you can use LONG VARCHAR and LONG VARBINARY to specify larger values.

Hive and Vertica define string length in different ways. In Hive the length is the number of characters; in Vertica it is the number of bytes. Thus, a character encoding that uses more than one byte, such as Unicode, can cause mismatches between the two. To avoid data truncation, set values in Vertica based on bytes, not characters.

If data size exceeds the column size, Vertica logs an event at read time in the QUERY_EVENTS system table.

Examples

The following example uses SYNC_WITH_HCATALOG_SCHEMA_TABLE to synchronize the "nation" table:

```
=> CREATE SCHEMA 'hcat_local';
CREATE SCHEMA

=> CREATE HCATALOG SCHEMA hcat WITH hostname='hcathost' HCATALOG_SCHEMA='hcat'
  HCATALOG_USER='hcatuser';
CREATE SCHEMA

=> SELECT sync_with_hcatalog_schema_table('hcat_local', 'hcat', 'nation');
sync_with_hcatalog_schema_table
--------------------------------------------------------------------------------
  Schema hcat_local synchronized with hcat for table nation
  table nation is created in schema hcat_local
  (1 row)
```

The following example shows the behavior if the "nation" table already exists in the local schema:

```
=> SELECT sync_with_hcatalog_schema_table('hcat_local','hcat','nation');
sync_with_hcatalog_schema_table
--------------------------------------------------------------------------------
  Schema hcat_local synchronized with hcat for table nation
  table nation is altered in schema hcat_local
  (1 row)
```

VERIFY_HADOOP_CONF_DIR

Verifies that the Hadoop configuration that is used to access HDFS is valid on all Vertica nodes. The configuration is valid if:

- all required configuration files are found on the path defined by the HadoopConfDir configuration parameter
- all properties needed by Vertica are set in those files

This function does not attempt to validate the settings of those properties; it only verifies that they have values.

It is possible for Hadoop configuration to be valid on some nodes and invalid on others. The function reports a validation failure if the value is invalid on any node; the rest of the output reports the details.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
VERIFY_HADOOP_CONF_DIR( )
```

Parameters

This function has no parameters.

Privileges

This function does not require privileges.

Examples

The following example shows the results when the Hadoop configuration is valid.

```
=> SELECT VERIFY_HADOOP_CONF_DIR();
   verify_hadoop_conf_dir
--------------------------------------------------------------------------
Validation Success
v_vmart_node0001: HadoopConfDir [PG_TESTOUT/config] is valid
v_vmart_node0002: HadoopConfDir [PG_TESTOUT/config] is valid
v_vmart_node0003: HadoopConfDir [PG_TESTOUT/config] is valid
v_vmart_node0004: HadoopConfDir [PG_TESTOUT/config] is valid
  (1 row)
```

In the following example, the Hadoop configuration is valid on one node, but on other nodes a needed value is missing.

```
=> SELECT VERIFY_HADOOP_CONF_DIR();
   verify_hadoop_conf_dir
------------------------------------------------------------------------
Validation Failure
v_vmart_node0001: HadoopConfDir [PG_TESTOUT/test_configs/config] is valid
v_vmart_node0002: No fs.defaultFS parameter found in config files in [PG_TESTOUT/config]
v_vmart_node0003: No fs.defaultFS parameter found in config files in [PG_TESTOUT/config]
v_vmart_node0004: No fs.defaultFS parameter found in config files in [PG_TESTOUT/config]
   (1 row)
```

Machine learning functions

Machine learning functions let you work with your data set in different stages of the data analysis process:

- Preparing models
- Training models
- Evaluating models
- Applying models
- Managing models

Some Vertica machine learning functions are implemented as Vertica UDx functions, while others are implemented as meta-functions:

- A UDx function accepts an input relation name from a FROM clause. The SELECT statement that calls the functions is composable—it can be used as a sub-query in another SELECT statement.
- A meta-function accepts the input relation name as a single-quoted string passed to it as an argument or a named parameter. The data that the SELECT statement returns cannot be used in a sub-query. Machine learning meta-functions do not support temporary tables.

All machine learning functions automatically cast NUMERIC arguments to FLOAT.

Important
Before using a machine learning function, be aware that any open transaction on the current session might be committed.

In this section

- Data preparation
- Machine learning algorithms
- Model evaluation
- Model management
- Transformation functions

Data preparation

Vertica supports machine learning functions that prepare data as needed before subjecting it to analysis.

In this section

- BALANCE
- CORR_MATRIX
- DETECT_OUTLIERS
- IFOREST
- IMPUTE
- NORMALIZE
- NORMALIZE_FIT
- ONE_HOT_ENCODER_FIT
- PCA
- SUMMARIZE_CATCOL
- SUMMARIZE_NUMCOL
- SVD

BALANCE

Returns a view with an equal distribution of the input data based on the response_column.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
BALANCE ( 'output-view', 'input-relation', 'response-column', 'balance-method'
    [ USING PARAMETERS sampling_ratio=ratio ] )
```

Arguments

output-view

The name of the view where Vertica saves the balanced data from the input relation.

> **Note**
> **Note** : The view that results from this function employs a random function. Its content can differ each time it is used in a query. To make the operations on the view predictable, store it in a regular table.

input-relation

The table or view that contains the data the function uses to create a more balanced data set. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

Name of the input column that represents the dependent variable, of type VARCHAR or INTEGER.

balance-method

Specifies a method to select data from the minority and majority classes, one of the following.

- hybrid_sampling : Performs over-sampling and under-sampling on different classes so each class is equally represented.
- over_sampling : Over-samples on all classes, with the exception of the most majority class, towards the most majority class's cardinality.
- under_sampling : Under-samples on all classes, with the exception of the most minority class, towards the most minority class's cardinality.
- weighted_sampling : An alias of under_sampling .

Parameters

ratio

The desired ratio between the majority class and the minority class. This value has no effect when used with balance method hybrid_sampling .

Default: 1.0

Privileges

Non-superusers:

- SELECT privileges on the input relation
- CREATE privileges on the output view schema

Examples

```
=> CREATE TABLE backyard_bugs (id identity, bug_type int, finder varchar(20));
CREATE TABLE

=> COPY backyard_bugs FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1|Ants
>> 1|Beetles
>> 3|Ladybugs
>> 3|Ants
>> 3|Beetles
>> 3|Caterpillars
>> 2|Ladybugs
>> 3|Ants
>> 3|Beetles
>> 1|Ladybugs
>> 3|Ladybugs
>> \.

=> SELECT bug_type, COUNT(bug_type) FROM backyard_bugs GROUP BY bug_type;
 bug_type | COUNT
----------+-------
        2 |     1
        1 |     3
        3 |     7
(3 rows)

=> SELECT BALANCE('backyard_bugs_balanced', 'backyard_bugs', 'bug_type', 'under_sampling');
         BALANCE
--------------------------
 Finished in 1 iteration

(1 row)

=> SELECT bug_type, COUNT(bug_type) FROM backyard_bugs_balanced GROUP BY bug_type;
----------+-------
        2 |     1
        1 |     2
        3 |     1
(3 rows)
```

See also

- Balancing imbalanced data

CORR_MATRIX

Takes an input relation with numeric columns, and calculates the *Pearson Correlation Coefficient* between each pair of its input columns. The function is implemented as a Multi-Phase Transform function.

Syntax

```
CORR_MATRIX ( input-columns ) OVER()
```

Arguments

input-columns

A comma-separated list of the columns in the input table. The input columns can be of any numeric type or BOOL, but they will be converted internally to FLOAT. The number of input columns must be more than 1 and not more than 1600.

Returns

CORR_MATRIX returns the correlation matrix in triplet format. That is, each pair-wise correlation is identified by three returned columns: name of the first variable, name of the second variable, and the correlation value of the pair. The function also returns two extra columns: number_of_ignored_input_rows and number_of_processed_input_rows . The value of the fourth/fifth column indicates the number of rows from the

input which are ignored/used to calculate the corresponding correlation value. Any input pair with NULL, Inf, or NaN is ignored.

The correlation matrix is symmetric with a value of 1 on all diagonal elements; therefore, it can return only the value of elements above the diagonals —that is, the upper triangle. Nevertheless, the function returns the entire matrix to simplify any later operations. Then, the number of output rows is:

```
(#input-columns)^2
```

The first two output columns are of type VARCHAR(128), the third one is of type FLOAT, and the last two are of type INT.

Notes

- The contents of the OVER clause must be empty.
- The function returns no rows when the input table is empty.
- When any of X_i and Y_i is NULL, Inf, or NaN, the pair will not be included in the calculation of CORR(X, Y). That is, any input pair with NULL, Inf, or NaN is ignored.
- For the pair of (X,X), regardless of the contents of X: CORR(X,X) = 1, number_of_ignored_input_rows = 0, and number_of_processed_input_rows = #input_rows.
- When (N *SUMX2* == *SUMX* SUMX) or (N *SUMY2* == *SUMY* SUMY) then value of CORR(X, Y) will be NULL. In theory it can happen in case of a column with constant values; nevertheless, it may not be always observed because of rounding error.
- In the special case where all pair values of (X_i,Y_i) contain NULL, inf, or NaN, and X != Y: CORR(X,Y)=NULL.

Examples

The following example uses the iris dataset.*

```
SELECT CORR_MATRIX("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width") OVER() FROM iris;
variable_name_1 | variable_name_2 | corr_value        | number_of_ignored_input_rows | number_of_processed_input_rows
----------------+-----------------+-------------------+------------------------------+-------------------------------
Sepal.Length    | Sepal.Width     |-0.117569784133002 | 0                            | 150
Sepal.Width     | Sepal.Length    |-0.117569784133002 | 0                            | 150
Sepal.Length    | Petal.Length    |0.871753775886583  | 0                            | 150
Petal.Length    | Sepal.Length    |0.871753775886583  | 0                            | 150
Sepal.Length    | Petal.Width     |0.817941126271577  | 0                            | 150
Petal.Width     | Sepal.Length    |0.817941126271577  | 0                            | 150
Sepal.Width     | Petal.Length    |-0.42844010433054  | 0                            | 150
Petal.Length    | Sepal.Width     |-0.42844010433054  | 0                            | 150
Sepal.Width     | Petal.Width     |-0.366125932536439 | 0                            | 150
Petal.Width     | Sepal.Width     |-0.366125932536439 | 0                            | 150
Petal.Length    | Petal.Width     |0.962865431402796  | 0                            | 150
Petal.Width     | Petal.Length    |0.962865431402796  | 0                            | 150
Sepal.Length    | Sepal.Length    |1                  | 0                            | 150
Sepal.Width     | Sepal.Width     |1                  | 0                            | 150
Petal.Length    | Petal.Length    |1                  | 0                            | 150
Petal.Width     | Petal.Width     |1                  | 0                            | 150
(16 rows)
```

DETECT_OUTLIERS

Returns the outliers in a data set based on the outlier threshold. The output is a table that contains the outliers. DETECT_OUTLIERS uses the detection method robust_szcore to normalize each input column. The function then identifies as outliers all rows that contain a normalized value greater than the default or specified threshold.

> **Note**
> You can calculate normalized column values with Vertica functions NORMALIZE and NORMALIZE_FIT .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DETECT_OUTLIERS ( 'output-table', 'input-relation','input-columns', 'detection-method'
    [ USING PARAMETERS
        [outlier_threshold = threshold]
        [, exclude_columns = 'excluded-columns']
        [, partition_columns = 'partition-columns'] ] )
```

Arguments

output-table

The name of the table where Vertica saves rows that are outliers along the chosen input_columns . All columns are present in this table.

input-relation

The table or view that contains outlier data. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. Input columns must be of type numeric .

detection-method

The outlier detection method to use, set to robust_zscore .

Parameters

outlier_threshold

The minimum normalized value in a row that is used to identify that row as an outlier.

Default: 3.0

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

partition_columns

Comma-separated list of column names from the input table or view that defines the partitions. DETECT_OUTLIERS detects outliers among each partition separately.

Default: empty list

Privileges

Non-superusers:

- SELECT privileges on the input relation
- CREATE privileges on the output table

Examples

The following example shows how to use DETECT_OUTLIERS :

```
=> CREATE TABLE baseball_roster (id identity, last_name varchar(30), hr int, avg float);
CREATE TABLE

=> COPY baseball_roster FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> Polo|7|.233
>> Gloss|45|.170
>> Gus|12|.345
>> Gee|1|.125
>> Laus|3|.095
>> Hilltop|16|.222
>> Wicker|78|.333
>> Scooter|0|.121
>> Hank|999999|.8888
>> Popup|35|.378
>> \.


=> SELECT * FROM baseball_roster;
 id | last_name |   hr   |  avg
----+-----------+--------+--------
  3 | Gus       |     12 |  0.345
  4 | Gee       |      1 |  0.125
  6 | Hilltop   |     16 |  0.222
 10 | Popup     |     35 |  0.378
  1 | Polo      |      7 |  0.233
  7 | Wicker    |     78 |  0.333
  9 | Hank      | 999999 | 0.8888
  2 | Gloss     |     45 |   0.17
  5 | Laus      |      3 |  0.095
  8 | Scooter   |      0 |  0.121
(10 rows)

=> SELECT DETECT_OUTLIERS('baseball_outliers', 'baseball_roster', 'id, hr, avg', 'robust_zscore' USING PARAMETERS
outlier_threshold=3.0);

    DETECT_OUTLIERS
---------------------------
 Detected 2 outliers

(1 row)

=> SELECT * FROM baseball_outliers;
 id | last_name | hr         | avg
----+-----------+------------+-------------
  7 | Wicker    |         78 |       0.333
  9 | Hank      |     999999 |      0.8888
(2 rows)
```

IFOREST

Trains and returns an isolation forest (iForest) model. After you train the model, you can use the APPLY_IFOREST function to predict outliers in an input relation.

For more information about how the iForest algorithm works, see Isolation Forest .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

IFOREST(*'model-name'*, *'input-relation'*, *'input-columns'* [USING PARAMETERS *param=value*[,...]])

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the input data for IFOREST.

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. Columns must be of types CHAR, VARCHAR, BOOL, INT, or FLOAT.

Columns of types CHAR, VARCHAR, and BOOL are treated as categorical features; all others are treated as numeric features.

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

Default: Empty string ('')

ntree

Integer in the range [1, 1000], specifies the number of trees in the forest.

Default: 100

sampling_size

Float in the range (0.0, 1.0], specifies the portion of the input data set that is randomly picked, without replacement, for training each tree.

Default: 0.632

col_sample_by_tree

Float in the range (0.0, 1.0], specifies the fraction of columns that are randomly picked for training each tree.

Default: 1.0

max_depth

Integer in the range [1, 100], specifies the maximum depth for growing each tree.

Default: 10

nbins

Integer in the range [2, 1000], specifies the number of bins used to discretize continuous features.

Default: 32

Model Attributes

details

Details about the function's predictor columns, including:

- predictor : Names of the predictors in the same order specified when training the model.
- type : Types of the predictors in the same order as their names in predictor .

tree_count

Number of trees in the model.

rejected_row_count

Number of rows in *input-relation* that were skipped because they contained an invalid value.

accepted_row_count

Total number of rows in *input-relation* minus rejected_row_count .

call_string

Value of all input arguments that were specified at the time the function was called.

Privileges

Non-superusers:

- CREATE privileges on the schema where the model is created

- SELECT privileges on the input relation

Examples

In the following example, the input data to the function contains columns of type INT, VARCHAR, and FLOAT:

```
=> SELECT IFOREST('baseball_anomalies','baseball','team, hr, hits, avg, salary' USING PARAMETERS ntree=75, sampling_size=0.7,
max_depth=15);
IFOREST
----------
Finished
(1 row)
```

You can verify that all the input columns were read in correctly by calling GET_MODEL_SUMMARY and checking the details section:

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='baseball_anomalies');
GET_MODEL_SUMMARY
---------------------------------------------------------------------------------------------------------------------------------

===========
call_string
===========
SELECT iforest('public.baseball_anomalies', 'baseball', 'team, hr, hits, avg, salary' USING PARAMETERS exclude_columns='', ntree=75,
sampling_size=0.7, col_sample_by_tree=1, max_depth=15, nbins=32);

=======
details
=======
predictor|      type
---------+----------------
  team   |char or varchar
   hr    |      int
  hits   |      int
   avg   |float or numeric
 salary  |float or numeric


===============
Additional Info
===============
       Name       |Value
------------------+-----
    tree_count    | 75
rejected_row_count|  0
accepted_row_count|1000

(1 row)
```

See also

- Detect outliers
- APPLY_IFOREST
- READ_TREE

IMPUTE

Imputes missing values in a data set with either the mean or the mode, based on observed values for a variable in each column. This function supports numeric and categorical data types.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
IMPUTE( 'output-view', 'input-relation', 'input-columns', 'method'
    [ USING PARAMETERS [exclude_columns = 'excluded-columns'] [, partition_columns = 'partition-columns'] ] )
```

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Arguments

output-view

Name of the view that shows the input table with imputed values in place of missing values. In this view, rows without missing values are kept intact while the rows with missing values are modified according to the specified method.

input-relation

The table or view that contains the data for missing-value imputation. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

input-columns

Comma-separated list of input columns where missing values will be replaced, or asterisk (*) to specify all columns. All columns must be of type numeric or BOOLEAN.

method

The method to compute the missing value replacements, one of the following:

- mean : The missing values in each column will be replaced by the mean of that column. This method can be used for numeric data only.
- mode : The missing values in each column will be replaced by the most frequent value in that column. This method can be used for categorical data only.

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

partition_columns

Comma-separated list of column names from the input relation that defines the partitions.

Privileges

Non-superusers:

- SELECT privileges on the input relation
- CREATE privileges on the output view schema

Examples

Execute IMPUTE on the small_input_impute table, specifying the mean method:

```
=> SELECT impute('output_view','small_input_impute', 'pid, x1,x2,x3,x4','mean'
USING PARAMETERS exclude_columns='pid');
impute
--------------------------
Finished in 1 iteration
(1 row)
```

Execute IMPUTE , specifying the mode method:

```
=> SELECT impute('output_view3','small_input_impute', 'pid, x5,x6','mode' USING PARAMETERS exclude_columns='pid');
impute
--------------------------
Finished in 1 iteration
(1 row)
```

See also

Imputing missing values

NORMALIZE

Runs a normalization algorithm on an input relation. The output is a view with the normalized data.

> **Note**
> **Note** : This function differs from NORMALIZE_FIT, which creates and stores a model rather than creating a view definition. This can lead to

different performance characteristics between the two functions.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
NORMALIZE ( 'output-view', 'input-relation', 'input-columns', 'normalization-method'
        [ USING PARAMETERS exclude_columns = 'excluded-columns' ] )
```

Arguments

output-view

The name of the view showing the input relation with normalized data replacing the specified input columns. .

input-relation

The table or view that contains the data to normalize. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

input-columns

Comma-separated list of numeric input columns that contain the values to normalize, or asterisk (*) to select all columns.

normalization-method

The normalization method to use, one of the following:

- minmax
- zscore
- robust_zscore

If infinity values appear in the table, the method ignores those values.

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

Privileges

Non-superusers:

- SELECT privileges on the input relation
- CREATE privileges on the output view schema

Examples

These examples show how you can use the NORMALIZE function on the wt and hp columns in the mtcars table.

Execute the NORMALIZE function, and specify the minmax method:

```
=> SELECT NORMALIZE('mtcars_norm', 'mtcars',
                    'wt, hp', 'minmax');
       NORMALIZE
-------------------------
 Finished in 1 iteration

(1 row)
```

Execute the NORMALIZE function, and specify the zscore method:

```
=> SELECT NORMALIZE('mtcars_normz','mtcars',
                    'wt, hp', 'zscore');
       NORMALIZE
-------------------------
 Finished in 1 iteration

(1 row)
```

Execute the NORMALIZE function, and specify the robust_zscore method:

```
=> SELECT NORMALIZE('mtcars_normz', 'mtcars',
                    'wt, hp', 'robust_zscore');
       NORMALIZE
-------------------------
 Finished in 1 iteration

(1 row)
```

See also

Normalizing data
NORMALIZE_FIT

> **Note**
> This function differs from NORMALIZE , which directly outputs a view with normalized results, rather than storing normalization parameters into a model for later operation.

NORMALIZE_FIT computes normalization parameters for each of the specified columns in an input relation. The resulting model stores the normalization parameters. For example, for MinMax normalization, the minimum and maximum value of each column are stored in the model. The generated model serves as input to functions APPLY_NORMALIZE and REVERSE_NORMALIZE .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile
Syntax

```
NORMALIZE_FIT ( 'model-name', 'input-relation', 'input-columns', 'normalization-method'
    [ USING PARAMETERS  [exclude_columns = 'excluded-columns'] [, output_view = 'output-view'] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the data to normalize. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. Input columns must be of data type numeric .

normalization-method

The normalization method to use, one of the following:

- minmax
- zscore
- robust_zscore

If you specify robust_zscore , NORMALIZE_FIT uses the function APPROXIMATE_MEDIAN [aggregate] .
All normalization methods ignore infinity, negative infinity, or NULL values in the input relation.

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

output_view

Name of the view that contains all columns from the input relation, with the specified input columns normalized.

Model attributes

data

Normalization method set to minmax :

- colNames : Model column names
- mins : Minimum value of each column
- maxes : Maximum value of each column

Privileges

Non-superusers:

- CREATE privileges on the schema where the model is created
- SELECT privileges on the input relation
- CREATE privileges on the output view schema

Examples

The following example creates a model with NORMALIZE_FIT using the wt and hp columns in table mtcars , and then uses this model in successive calls to APPLY_NORMALIZE and REVERSE_NORMALIZE .

```
=> SELECT NORMALIZE_FIT('mtcars_normfit', 'mtcars', 'wt,hp', 'minmax');
NORMALIZE_FIT
---------------
Success
(1 row)
```

The following call to APPLY_NORMALIZE specifies the hp and cyl columns in table mtcars , where hp is in the normalization model and cyl is not in the normalization model:

```
=> CREATE TABLE mtcars_normalized AS SELECT APPLY_NORMALIZE (hp, cyl USING PARAMETERS model_name = 'mtcars_normfit') FROM mtcars;
CREATE TABLE
=> SELECT * FROM mtcars_normalized;
         hp         | cyl
--------------------+-----
  0.434628975265018 | 8
  0.681978798586572 | 8
  0.434628975265018 | 6
                  1 | 8
  0.540636042402827 | 8
                  0 | 4
  0.681978798586572 | 8
 0.0459363957597173 | 4
  0.434628975265018 | 8
  0.204946996466431 | 6
  0.250883392226148 | 6
  0.049469964664311 | 4
  0.204946996466431 | 6
  0.201413427561837 | 4
  0.204946996466431 | 6
  0.250883392226148 | 6
  0.049469964664311 | 4
  0.215547703180212 | 4
 0.0353356890459364 | 4
  0.187279151943463 | 6
  0.452296819787986 | 8
  0.628975265017668 | 8
  0.346289752650177 | 8
  0.137809187279152 | 4
  0.749116607773852 | 8
  0.144876325088339 | 4
  0.151943462897526 | 4
  0.452296819787986 | 8
  0.452296819787986 | 8
  0.575971731448763 | 8
  0.159010600706714 | 4
  0.346289752650177 | 8
(32 rows)
```

```
=> SELECT REVERSE_NORMALIZE (hp, cyl USING PARAMETERS model_name='mtcars_normfit') FROM mtcars_normalized;
 hp | cyl
-----+-----
 175 | 8
 245 | 8
 175 | 6
 335 | 8
 205 | 8
  52 | 4
 245 | 8
  65 | 4
 175 | 8
 110 | 6
 123 | 6
  66 | 4
 110 | 6
 109 | 4
 110 | 6
 123 | 6
  66 | 4
 113 | 4
  62 | 4
 105 | 6
 180 | 8
 230 | 8
 150 | 8
  91 | 4
 264 | 8
  93 | 4
  95 | 4
 180 | 8
 180 | 8
 215 | 8
  97 | 4
 150 | 8
(32 rows)
```

The following call to REVERSE_NORMALIZE also specifies the hp and cyl columns in table mtcars , where hp is in normalization model mtcars_normfit , and cyl is not in the normalization model.

```
=> SELECT REVERSE_NORMALIZE (hp, cyl USING PARAMETERS model_name='mtcars_normfit') FROM mtcars_normalized;
       hp         | cyl
------------------+-----
205.000005722046 |   8
150.000000357628 |   8
150.000000357628 |   8
93.0000016987324 |   4
 174.99999666214 |   8
94.9999992102385 |   4
214.999997496605 |   8
97.0000009387732 |   4
245.000006556511 |   8
 174.99999666214 |   6
             335 |   8
245.000006556511 |   8
62.0000002086163 |   4
 174.99999666214 |   8
230.000002026558 |   8
              52 |   4
263.999997675419 |   8
109.999999523163 |   6
123.000002324581 |   6
64.9999996386468 |   4
66.0000005029142 |   4
112.999997898936 |   4
109.999999523163 |   6
180.000000983477 |   8
180.000000983477 |   8
108.999998658895 |   4
109.999999523163 |   6
104.999999418855 |   6
123.000002324581 |   6
180.000000983477 |   8
66.0000005029142 |   4
90.9999999701977 |   4
(32 rows)
```

See also

Normalizing data

ONE_HOT_ENCODER_FIT

Generates a sorted list of each of the category levels for each feature to be encoded, and stores the model.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ONE_HOT_ENCODER_FIT ( 'model-name', 'input-relation','input-columns'
      [ USING PARAMETERS
          [exclude_columns = 'excluded-columns']
          [, output_view = 'output-view']
          [, extra_levels = 'category-levels'] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers. It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the data for one hot encoding. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. Input columns must be INTEGER, BOOLEAN, VARCHAR, or dates.

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

output_view

The name of the view that stores the input relation and the one hot encodings. Columns are returned in the order they appear in the input relation, with the one-hot encoded columns appended after the original columns.

extra_levels

Additional levels in each category that are not in the input relation. This parameter should be passed as a string that conforms with the JSON standard, with category names as keys, and lists of extra levels in each category as values.

Model attributes

call_string

The value of all input arguments that were specified at the time the function was called.

varchar_categories integer_categories boolean_categories date_categories

Settings for all:

- category_name : Column name
- category_level : Levels of the category, sorted for each category
- category_level_index : Index of this categorical level in the sorted list of levels for the category.

Privileges

Non-superusers:

- CREATE privileges on the schema where the model is created
- SELECT privileges on the input relation
- CREATE privileges on the output view schema

Examples

```
=> SELECT ONE_HOT_ENCODER_FIT ('one_hot_encoder_model','mtcars','*'
USING PARAMETERS exclude_columns='mpg,disp,drat,wt,qsec,vs,am');
ONE_HOT_ENCODER_FIT
---------------------
Success
(1 row)
```

See also

- APPLY_ONE_HOT_ENCODER
- Encoding categorical columns

PCA

Computes principal components from the input table/view. The results are saved in a PCA model. Internally, PCA finds the components by using SVD on the co-variance matrix built from the input date. The singular values of this decomposition are also saved as part of the PCA model. The signs of all elements of a principal component could be flipped all together on different runs.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
PCA ( 'model-name', 'input-relation', 'input-columns'
    [ USING PARAMETERS
        [exclude_columns = 'excluded-columns']
        [, num_components = num-components]
        [, scale = is-scaled]
        [, method = 'method'] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the input data for PCA.

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. All input columns must be a numeric data type.

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

num_components

The number of components to keep in the model. If this value is not provided, all components are kept. The maximum number of components is the number of non-zero singular values returned by the internal call to SVD. This number is less than or equal to SVD (number of columns, number of rows).

scale

A Boolean value that specifies whether to standardize the columns during the preparation step:

- True : Use a correlation matrix instead of a covariance matrix.
- False (default)

method

The method used to calculate PCA, can be set to LAPACK .

Model attributes

columns

The information about columns from the input relation used for creating the PCA model:

- index
- name

singular_values

The information about singular values found. They are sorted in descending order:

- index
- value
- explained_variance : percentage of the variance in data that can be attributed to this singular value
- accumulated_explained_variance : percentage of the variance in data that can be retained if we drop all singular values after this current one

principal_components

The principal components corresponding to the singular values mentioned above:

- index: indies of the elements in each component
- PC1
- PC2
- ...

counters

The information collected during training the model, stored as name-value pairs:

- counter_name
  - accepted_row_count: number of valid rows in the data
  - rejected_row_count: number of invalid rows (having NULL, INF or NaN) in the data
  - iteration_count: number of iterations, always 1 for the current implementation of PCA
- counter_value

call_string

The function call that created the model.

Privileges

Non-superusers:

- CREATE privileges on the schema where the model is created
- SELECT privileges on the input relation

Examples

```
=> SELECT PCA ('pcamodel', 'world','country,HDI,em1970,em1971,em1972,em1973,em1974,em1975,em1976,em1977,
em1978,em1979,em1980,em1981,em1982,em1983,em1984 ,em1985,em1986,em1987,em1988,em1989,em1990,em1991,em1992,
em1993,em1994,em1995,em1996,em1997,em1998,em1999,em2000,em2001,em2002,em2003,em2004,em2005,em2006,em2007,
em2008,em2009,em2010,gdp1970,gdp1971,gdp1972,gdp1973,gdp1974,gdp1975,gdp1976,gdp1977,gdp1978,gdp1979,gdp1980,
gdp1981,gdp1982,gdp1983,gdp1984,gdp1985,gdp1986,gdp1987,gdp1988,gdp1989,gdp1990,gdp1991,gdp1992,gdp1993,
gdp1994,gdp1995,gdp1996,gdp1997,gdp1998,gdp1999,gdp2000,gdp2001,gdp2002,gdp2003,gdp2004,gdp2005,gdp2006,
gdp2007,gdp2008,gdp2009,gdp2010' USING PARAMETERS exclude_columns='HDI,country');
PCA
--------------------------------------------------------------
Finished in 1 iterations.
Accepted Rows: 96  Rejected Rows: 0
(1 row)
=> CREATE TABLE worldPCA AS SELECT
APPLY_PCA (HDI,country,em1970,em1971,em1972,em1973,em1974,em1975,em1976,em1977,em1978,em1979,
em1980,em1981,em1982,em1983,em1984 ,em1985,em1986,em1987,em1988,em1989,em1990,em1991,em1992,em1993,em1994,
em1995,em1996,em1997,em1998,em1999,em2000,em2001,em2002,em2003,em2004,em2005,em2006,em2007,em2008,em2009,
em2010,gdp1970,gdp1971,gdp1972,gdp1973,gdp1974,gdp1975,gdp1976,gdp1977,gdp1978,gdp1979,gdp1980,gdp1981,gdp1982,
gdp1983,gdp1984,gdp1985,gdp1986,gdp1987,gdp1988,gdp1989,gdp1990,gdp1991,gdp1992,gdp1993,gdp1994,gdp1995,
gdp1996,gdp1997,gdp1998,gdp1999,gdp2000,gdp2001,gdp2002,gdp2003,gdp2004,gdp2005,gdp2006,gdp2007,gdp2008,
gdp2009,gdp2010 USING PARAMETERS model_name='pcamodel', exclude_columns='HDI, country', key_columns='HDI,
country',cutoff=.3)OVER () FROM world;
CREATE TABLE

=> SELECT * FROM worldPCA;
HDI   |       country       |       col1
------+---------------------+--------------------
0.886 | Belgium             |  79002.2946705704
0.699 | Belize              | -25631.6670012556
0.427 | Benin               | -40373.4104598122
0.805 | Chile               | -16805.7940082156
0.687 | China               | -37279.2893141103
0.744 | Costa Rica          | -19505.5631231635
0.4   | Cote d'Ivoire       | -38058.2060339272
0.776 | Cuba                | -23724.5779612041
0.895 | Denmark             |  117325.594028813
0.644 | Egypt               | -34609.9941604549
...
(96 rows)

=> SELECT APPLY_INVERSE_PCA (HDI, country, col1
   USING PARAMETERS model_name = 'pcamodel', exclude_columns='HDI,country',
   key_columns = 'HDI, country') OVER () FROM worldPCA;
HDI |       country       |      em1970       |      em1971       |      em1972      |      em1973      |
     em1974      |      em1975       |      em1976|      em1977      |      em1978       |      em1979
  |      em1980       |      em1981       |      em1982       |      em1983       |      em1984       |em1985
|      em1986       |      em1987       |      em1988       |      em1989      |      em1990      |      em1991
|      em1992       |      em1993|      em1994      |      em1995       |      em1996       |      em1997
   |      em1998       |      em1999       |      em2000       |      em2001       |em2002       |
em2003      |      em2004       |      em2005      |      em2006       |      em2007       |      em2008
|      em2009      |      em2010       |      gdp1970      |      gdp1971      |      gdp1972      |      gdp1973
|      gdp1974      |      gdp1975      |      gdp1976      |      gdp1977      |gdp1978      |      gdp1979
 |      gdp1980      |      gdp1981      |      gdp1982      |      gdp1983      |      gdp1984      |      gdp1985
     |      gdp1986|      gdp1987      |      gdp1988      |      gdp1989      |      gdp1990      |      gdp1991
    |      gdp1992      |      gdp1993      |      gdp1994      |      gdp1995      |      gdp1996      |
gdp1997      |      gdp1998      |      gdp1999      |      gdp2000      |      gdp2001      |      gdp2002
|      gdp2003      |gdp2004      |      gdp2005      |      gdp2006      |      gdp2007      |      gdp2008
```

```
 |    gdp2009     |    gdp2010
------+----------------+----------------+----------------+----------------+----------------
+----------------+----------------+----------------+----------------+----------------+--------
---------+----------------+----------------+----------------+----------------+----------------
--+----------------+----------------+----------------+----------------+----------------+-------
----------+----------------+----------------+----------------+----------------+----------------
-+----------------+----------------+----------------+----------------+----------------+-------
-----------+----------------+----------------+----------------+----------------+--------------
---+----------------+----------------+----------------+----------------+----------------+------
----------+----------------+----------------+----------------+----------------+----------------+
----------------+----------------+----------------+----------------+----------------+------------
-----+----------------+----------------+----------------+----------------+----------------+-------
----------+----------------+----------------+----------------+----------------+----------------+-
---------------+----------------+----------------+----------------+----------------+------------
----+----------------+----------------+----------------+----------------+----------------+-------
---------+----------------+----------------+----------------+----------------+----------------
0.886 | Belgium            |  18585.6613572407 | -16145.6374560074 |  26938.956253415 | 8094.30475779595 |
 12073.5461203817 | -11069.0567600181 | 19133.8584911727|   5500.312894949 | -4227.94863799987 |  6265.77925410752
|  -10884.749295608 | 30929.4669575201 | -7831.49439429977 |  3235.81760508742 | -22765.9285442662 | 27200
.6767714485 | -10554.9550160917 |    1169.4144482273 | -16783.7961289161 | 27932.2660829329 | 17227.9083196848
| 13956.0524012749 | -40175.6286481088 | -10889.4785920499 | 22703.6576872859 | -14635.5832197402 |
2857.12270512168 | 20473.5044214494 | -52199.4895696423 | -11038.7346460738 |  18466.7298633088 | -17410.4225137703 |
-3475.63826305462 | 29305.6753822341 |    1242.5724942049 | 17491.0096310849 | -12609.9984515902 | -17909.3603476248
|  6276.58431412381 | 21851.9475485178 | -2614.33738160397 | 3777.74134131349 | 4522.08854282736 | 4251.90446379366
| 4512.15101396876 | 4265.49424538129 | 5190.06845330997 | 4543.80444817989 | 5639.81122679089 | 4420.44705213467
|  5658.8820279283 | 5172.69025294376 | 5019.63640408663 | 5938.84979495903 | 4976.57073629812 | 4710.49525137591
| 6523.65700286465 | 5067.82520773578 | 6789.13070219317 | 5525.94643553563 | 6894.68336419297 | 5961.58442474331
| 5661.21093840818 | 7721.56088518218 |  5959.7301109143 | 6453.43604137202 | 6739.39384033096 | 7517.97645468455
| 6907.49136910647 | 7049.03921764209 | 7726.49091035527 | 8552.65909911844 | 7963.94487647115 | 7187.45827585515
| 7994.02955410523 | 9532.89844418041 | 7962.25713582666 | 7846.68238907624 | 10230.9878908643 | 8642.76044946519
| 8886.79860331866 |  8718.3731386891
...
(96 rows)
```

See also

- APPLY_INVERSE_PCA
- APPLY_PCA

SUMMARIZE_CATCOL

Returns a statistical summary of categorical data input, in three columns:

- CATEGORY: Categorical levels, of the same SQL data type as the summarized column
- COUNT: The number of category levels, of type INTEGER
- PERCENT: Represents category percentage, of type FLOAT

Syntax

```
SUMMARIZE_CATCOL (target-column
    [ USING PARAMETERS TOPK = topk-value [, WITH_TOTALCOUNT = show-total] ] )
OVER()
```

Arguments

target-column

The name of the input column to summarize, one of the following data types:

- BOOLEAN
- FLOAT
- INTEGER
- DATE
- CHAR/VARCHAR

Parameters

TOPK

Integer, specifies how many of the most frequent rows to include in the output.

WITH_TOTALCOUNT

A Boolean value that specifies whether the table contains a heading row that displays the total number of rows displayed in the target column, and a percent equal to 100.

Default: true

Examples

This example shows the categorical summary for the current_salary column in the salary_data table. The output of the query shows the column category, count, and percent. The first column gives the categorical levels, with the same SQL data type as the input column, the second column gives a count of that value, and the third column gives a percentage.

```
=> SELECT SUMMARIZE_CATCOL (current_salary USING PARAMETERS TOPK = 5) OVER() FROM salary_data;
CATEGORY | COUNT | PERCENT
---------+-------+---------
       | 1000 |     100
 39004 |     2 |     0.2
 35321 |     1 |     0.1
 36313 |     1 |     0.1
 36538 |     1 |     0.1
 36562 |     1 |     0.1
(6 rows)
```

SUMMARIZE_NUMCOL

Returns a statistical summary of columns in a Vertica table:

- Count
- Mean
- Standard deviation
- Min/max values
- Approximate percentile
- Median

All summary values are FLOAT data types, except INTEGER for count.

Syntax

```
SUMMARIZE_NUMCOL (input-columns [ USING PARAMETERS exclude_columns = 'excluded-columns'] ) OVER()
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. All columns must be a numeric data type. If you select all columns, SUMMARIZE_NUMCOL normalizes all columns in the model

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

Examples

Show the statistical summary for the age and salary columns in the employee table:

```
=> SELECT SUMMARIZE_NUMCOL(* USING PARAMETERS exclude_columns='id,name,gender,title') OVER() FROM employee;
COLUMN        | COUNT |    MEAN    |      STDDEV      |  MIN    | PERC25  | MEDIAN  |  PERC75   |  MAX
--------------+-------+------------+------------------+---------+---------+---------+-----------+-------
age           |     5 |    63.4    | 19.3209730603818 |      44 |      45 |      67 |      71   |    90
salary        |     5 | 3456.76    | 1756.78754300285 | 1234.56 | 2345.67 | 3456.78 | 4567.89   | 5678.9
(2 rows)
```

SVD

Computes singular values (the diagonal of the S matrix) and right singular vectors (the V matrix) of an SVD decomposition of the input relation. The results are saved as an SVD model. The signs of all elements of a singular vector in SVD could be flipped all together on different runs.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SVD ( 'model-name', 'input-relation', 'input-columns'
   [ USING PARAMETERS
        [exclude_columns = 'excluded-columns']
        [, num_components = num-components]
        [, method = 'method'] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the input data for SVD.

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. Input columns must be a numeric data type.

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

num_components

The number of components to keep in the model. The maximum number of components is the number of non-zero singular values computed, which is less than or equal to min (number of columns, number of rows). If you omit this parameter, all components are kept.

method

The method used to calculate SVD, can be set to LAPACK .

Model attributes

columns

The information about columns from the input relation used for creating the SVD model:

- index
- name

singular_values

The information about singular values found. They are sorted in descending order:

- index
- value
- explained_variance : percentage of the variance in data that can be attributed to this singular value
- accumulated_explained_variance : percentage of the variance in data that can be retained if we drop all singular values after this current one

right_singular_vectors

The right singular vectors corresponding to the singular values mentioned above:

- index: indices of the elements in each vector
- vector1
- vector2
- ...

counters

The information collected during training the model, stored as name-value pairs:

- counter_name
  - accepted_row_count: number of valid rows in the data
  - rejected_row_count: number of invalid rows (having NULL, INF or NaN) in the data
  - iteration_count: number of iterations, always 1 for the current implementation of SVD
- counter_value

call_string

The function call that created the model.

Privileges

Non-superusers:

- CREATE privileges on the schema where the model is created
- SELECT privileges on the input relation

Examples

```
=> SELECT SVD ('svdmodel', 'small_svd', 'x1,x2,x3,x4');
SVD
-------------------------------------------------------------------
Finished in 1 iterations.
Accepted Rows: 8  Rejected Rows: 0
(1 row)

=> CREATE TABLE transform_svd AS SELECT
    APPLY_SVD (id, x1, x2, x3, x4 USING PARAMETERS model_name='svdmodel', exclude_columns='id', key_columns='id')
    OVER () FROM small_svd;
CREATE TABLE

=> SELECT * FROM transform_svd;
id |       col1        |       col2        |       col3        |       col4
---+-------------------+-------------------+-------------------+-------------------
4  |  0.44849499240202 |  -0.347260956311326 |   0.186958376368345 |  0.378561270493651
6  |  0.17652411036246 | -0.0753183783382909 |  -0.678196192333598 | 0.0567124770173372
1  | 0.494871802886819 |   0.161721379259287 |  0.0712816417153664 | -0.473145877877408
2  |  0.17652411036246 | -0.0753183783382909 |  -0.678196192333598 | 0.0567124770173372
3  | 0.150974762654569 |   0.589561842046029 | 0.00392654610109522 |  0.360011163271921
5  | 0.494871802886819 |   0.161721379259287 |  0.0712816417153664 | -0.473145877877408
8  |  0.44849499240202 |  -0.347260956311326 |   0.186958376368345 |  0.378561270493651
7  | 0.150974762654569 |   0.589561842046029 | 0.00392654610109522 |  0.360011163271921
(8 rows)

=> SELECT APPLY_INVERSE_SVD (* USING PARAMETERS model_name='svdmodel', exclude_columns='id',
key_columns='id') OVER () FROM transform_svd;
id |       x1        |       x2        |       x3        |       x4
---+-----------------+-----------------+-----------------+-----------------
4 | 91.4056627665577 | 44.7629617207482 | 83.1704961993117 | 38.9274292265543
6 | 20.6468626294368 | 9.30974906868751 | 8.71006863405534 |  6.5855928603967
7 | 31.2494347777156 | 20.6336519003026 | 27.5668287751507 | 5.84427645886865
1 |  107.93376580719 | 51.6980548011917 | 97.9665796560552 | 40.4918236881051
2 | 20.6468626294368 | 9.30974906868751 | 8.71006863405534 |  6.5855928603967
3 | 31.2494347777156 | 20.6336519003026 | 27.5668287751507 | 5.84427645886865
5 |  107.93376580719 | 51.6980548011917 | 97.9665796560552 | 40.4918236881051
8 | 91.4056627665577 | 44.7629617207482 | 83.1704961993117 | 38.9274292265543
(8 rows)
```

See also

- APPLY_INVERSE_SVD
- APPLY_SVD

Machine learning algorithms

Vertica supports a full range of machine learning functions that train a model on a set of data, and return a model that can be saved for later execution.

These functions require the following privileges for non-superusers:

- CREATE privileges on the schema where the model is created
- SELECT privileges on the input relation

Note
Machine learning algorithms contain a subset of four classification functions:

- LOGISTIC_REG
- NAIVE_BAYES
- RF_CLASSIFIER
- SVM_CLASSIFIER

In this section

- ARIMA
- AUTOREGRESSOR
- BISECTING_KMEANS
- KMEANS
- KPROTOTYPES
- LINEAR_REG
- LOGISTIC_REG
- MOVING_AVERAGE
- NAIVE_BAYES
- POISSON_REG
- RF_CLASSIFIER
- RF_REGRESSOR
- SVM_CLASSIFIER
- SVM_REGRESSOR
- XGB_CLASSIFIER
- XGB_REGRESSOR

ARIMA

Creates and trains an autoregressive integrated moving average (ARIMA) model from a time series with consistent timesteps. ARIMA models combine the abilities of AUTOREGRESSOR and MOVING_AVERAGE models by making future predictions based on both preceding time series values and errors of previous predictions. After the model is trained, you can make predictions with the PREDICT_ARIMA function.

In Vertica, ARIMA is implemented using a Kalman Filter state-space approach, similar to Gardner, G., et al. This approach updates the state-space model with each element in the training data in order to calculate a loss score over the training data. A BFGS optimizer is then used to adjust the coefficients, and the state-space estimation is rerun until convergence. Because of this repeated estimation process, ARIMA consumes large amounts of memory when called with high values of p and q.

Given that the input data must be sorted by timestamp, this algorithm is single-threaded.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Immutable

Syntax

```
ARIMA( 'model-name', 'input-relation', 'timeseries-column', 'timestamp-column'
  USING PARAMETERS param=value[,...] )
```

Arguments

model-name

Model to create, where *model-name* conforms to conventions described in Identifiers. It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

Name of the table or view containing *timeseries-column* and *timestamp-column*.

timeseries-column

Name of a NUMERIC column in *input-relation* that contains the dependent variable or outcome.

timestamp-column

Name of an INTEGER, FLOAT, or TIMESTAMP column in *input-relation* that represents the timestamp variable. The timestep between consecutive entries should be consistent throughout the *timestamp-column*.

> **Tip**
> If your *timestamp-column* has varying timesteps, consider standardizing the step size with the TIME_SLICE function.

Parameters

p

Integer in the range [0, 1000], the number of lags to include in the autoregressive component of the computation. If q is unspecified or set to zero, p must be set to a nonzero value. In some cases, using a large p value can result in a memory overload error.

Default: 0

d

Difference order of the model.

> **Caution**
> Currently, this value must be zero. If you attempt to train an ARIMA model with a nonzero d value, the function errors.

Default: 0

q

Integer in the range [0, 1000], the number of lags to include in the moving average component of the computation. If p is unspecified or set to zero, q must be set to a nonzero value. In some cases, using a large q value can result in a memory overload error.

Default: 0

missing

Method for handling missing values, one of the following strings:

- 'drop': Missing values are ignored.
- 'raise': Missing values raise an error.
- 'zero': Missing values are set to zero.
- 'linear_interpolation': Missing values are replaced by a linearly interpolated value based on the nearest valid entries before and after the missing value. In cases where the first or last values in a dataset are missing, the function errors.

Default: 'linear_interpolation'

init_method

Initialization method, one of the following strings:

- 'Zero': Coefficients are initialized to zero.
- 'Hannan-Rissanen' or 'HR': Coefficients are initialized using the Hannan-Rissanen algorithm.

Default: 'Zero'

epsilon

Float in the range (0.0, 1.0), controls the convergence criteria of the optimization algorithm.

Default: 1e-6

max_iterations

Integer in the range [1, 1000000), the maximum number of training iterations. If you set this value too low, the algorithm might not converge.

Default: 100

Model attributes

coefficients

Coefficients of the model:

- phi : parameters for the autoregressive component of the computation. The number of returned phi values is equal to the value of p .
- theta : parameters for the moving average component of the computation. The number of returned theta values is equal to the value of q .

p, q, d

ARIMA component values:

- p : number of lags included in the autoregressive component of the computation
- d : difference order of the model
- q : number of lags included in the moving average component of the computation

mean

The model mean, average of the accepted sample values from *timeseries-column*

regularization

Type of regularization used when training the model

lambda

Regularization parameter. Higher values indicates stronger regularization.

mean_squared_error

Mean squared error of the model on the training set

rejected_row_count

Number of samples rejected during training

accepted_row_count

Number of samples accepted for training from the data set

timeseries_name

Name of the *timeseries-column* used to train the model

timestamp_name

Name of the *timestamp-column* used to train the model

missing_method

Method used for handling missing values

call_string

SQL statement used to train the model

Examples

The function requires that at least one of the p and q parameters be a positive, nonzero integer. The following example trains a model where both of these parameters are set to two:

```
=> SELECT ARIMA('arima_temp', 'temp_data', 'temperature', 'time' USING PARAMETERS p=2, q=2);
            ARIMA
-----------------------------------
Finished in 24 iterations.
3650 elements accepted, 0 elements rejected.
(1 row)
```

To see a summary of the model, including all model coefficients and parameter values, call GET_MODEL_SUMMARY :

```
=> SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='arima_temp');
                GET_MODEL_SUMMARY
-------------------------------------------------------------------


============
coefficients
============
parameter| value
---------+--------
  phi_1  | 1.23639
  phi_2  |-0.24201
 theta_1 |-0.64535
 theta_2 |-0.23046


==============
regularization
==============
none

===============
timeseries_name
===============
temperature

==============
timestamp_name
==============
time

==============
missing_method
==============
linear_interpolation

===========
call_string
===========
ARIMA('public.arima_temp', 'temp_data', 'temperature', 'time' USING PARAMETERS p=2, d=0, q=2, missing='linear_interpolation', init_method='Zero',
epsilon=1e-06, max_iterations=100);

===============
Additional Info
===============
       Name       | Value
------------------+--------
        p         |   2
        q         |   2
        d         |   0
       mean       |11.17775
      lambda      | 1.00000
mean_squared_error| 5.80628
rejected_row_count|   0
accepted_row_count|  3650

(1 row)
```

For an in-depth example that trains and makes predictions with an ARIMA model, see ARIMA model example .

See also

- Time series forecasting

- GET_MODEL_ATTRIBUTE

AUTOREGRESSOR

Creates an autoregressive (AR) model from a stationary time series with consistent timesteps that can then be used for prediction via PREDICT_AUTOREGRESSOR .

Autoregressive models predict future values of a time series based on the preceding values. More specifically, the user-specified *lag* determines how many previous timesteps it takes into account during computation, and predicted values are linear combinations of the values at each lag.

Since its input data must be sorted by timestamp, this algorithm is single-threaded.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
AUTOREGRESSOR ('model-name', 'input-relation', 'data-column', 'timestamp-column'
    [ USING PARAMETERS
        [ p = lags ]
        [, method = 'training-algorithm' ]
        [, missing = 'imputation-method' ]
        [, regularization = 'regularization-method' ]
        [, lambda = regularization-value ]
        [, compute_mse = boolean ]
    ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view containing the *timestamp-column* .

This algorithm expects a stationary time series as input; using a time series with a mean that shifts over time may lead to weaker results.

data-column

An input column of type NUMERIC that contains the dependent variables or outcomes.

timestamp-column

One INTEGER, FLOAT, or TIMESTAMP column that represents the timestamp variable. Timesteps must be consistent.

Parameters

p

INTEGER in the range [1, 1999], the number of lags to consider in the computation. Larger values for p weaken the correlation.

Default: 3

method

One of the following algorithms for training the model:

- 'OLS' (Ordinary Least Squares)
- 'Yule-Walker'

> **Note**
> If you train a model with the Yule-Walker algorithm, the alpha constant in the trained model is set to 0.0.

Default: 'OLS'

missing

One of the following methods for handling missing values:

- 'drop': Missing values are ignored.
- 'error': Missing values raise an error.
- 'zero': Missing values are replaced with 0.

- 'linear_interpolation': Missing values are replaced by linearly-interpolated values based on the nearest valid entries before and after the missing value. This means that in cases where the first or last values in a dataset are missing, they will simply be dropped.

Default: ' linear_interpolation'

regularization

One of the following regularization methods used when fitting the data:

- None
- 'L2': Weight regularization term which penalizes the squared weight value

Default: None

lambda

FLOAT in the range [0, 100000], the regularization value, lambda.

Default: 1.0

compute_mse

BOOLEAN, whether to calculate and output the mean squared error (MSE).

Default: False

Examples

The following example creates and trains an autoregression model using the Yule-Walker training algorithm and a lag of 3:

```
=> SELECT AUTOREGRESSOR('AR_temperature_yw', 'temp_data', 'Temperature', 'time' USING PARAMETERS p=3, method='yule-walker');
              AUTOREGRESSOR
-------------------------------------------------------------
Finished. 3650 elements accepted, 0 elements rejected.

(1 row)
```

See Autoregressive model example for a walk-through of how to train and make predictions with an autoregression model.

See also

- PREDICT_AUTOREGRESSOR
- GET_MODEL_SUMMARY

BISECTING_KMEANS

Executes the bisecting k-means algorithm on an input relation. The result is a trained model with a hierarchy of cluster centers, with a range of *k* values, each of which can be used for prediction.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
BISECTING_KMEANS('model-name', 'input-relation', 'input-columns', 'num-clusters'
    [ USING PARAMETERS
        [exclude_columns = 'exclude-columns']
        [, bisection_iterations = bisection-iterations]
        [, split_method = 'split-method']
        [, min_divisible_cluster_size = min-cluster-size]
        [, kmeans_max_iterations = kmeans-max-iterations]
        [, kmeans_epsilon = kmeans-epsilon]
        [, kmeans_center_init_method = 'kmeans-init-method']
        [, distance_method = 'distance-method']
        [, output_view = 'output-view']
        [, key_columns = 'key-columns'] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers. It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

Table or view that contains the input data for k-means. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. Input columns must be of data type numeric .

num-clusters

Number of clusters to create, an integer ≤ 10,000. This argument represents the *k* in k-means.

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

bisection_iterations

Integer between 1 - 1MM inclusive, specifies number of iterations the bisecting k-means algorithm performs for each bisection step. This corresponds to how many times a standalone k-means algorithm runs in each bisection step.

A setting >1 allows the algorithm to run and choose the best k-means run within each bisection step. If you use kmeanspp, the value of bisection_iterations is always 1, because kmeanspp is more costly to run but also better than the alternatives, so it does not require multiple runs.

Default: 1

split_method

The method used to choose a cluster to bisect/split, one of:

- size : Choose the largest cluster to bisect.
- sum_squares : Choose the cluster with the largest within-cluster sum of squares to bisect.

Default: sum_squares

min_divisible_cluster_size

Integer ≥ 2, specifies minimum number of points of a divisible cluster.

Default: 2

kmeans_max_iterations

Integer between 1 and 1MM inclusive, specifies the maximum number of iterations the k-means algorithm performs. If you set this value to a number lower than the number of iterations needed for convergence, the algorithm might not converge.

Default: 10

kmeans_epsilon

Integer between 1 and 1MM inclusive, determines whether the k-means algorithm has converged. The algorithm is considered converged after no center has moved more than a distance of *epsilon* from the previous iteration.

Default: 1e-4

kmeans_center_init_method

The method used to find the initial cluster centers in k-means, one of:

- kmeanspp (default): kmeans++ algorithm
- pseudo : Uses "pseudo center" approach used by Spark, bisects given center without iterating over points

distance_method

The measure for distance between two data points. Only Euclidean distance is supported at this time.

Default: euclidean

output_view

Name of the view where you save the assignment of each point to its cluster. You must have CREATE privileges on the view schema.

key_columns

Comma-separated list of column names that identify the output rows. Columns must be in the *input-columns* argument list. To exclude these and other input columns from being used by the algorithm, list them in parameter exclude_columns .

Model attributes

centers

A list of centers of the K centroids.

hierarchy

The hierarchy of K clusters, including:

- ParentCluster: Parent cluster centroid of each centroid—that is, the centroid of the cluster from which a cluster is obtained by bisection.
- LeftChildCluster: Left child cluster centroid of each centroid—that is, the centroid of the first sub-cluster obtained by bisecting a cluster.
- RightChildCluster: the right child cluster centroid of each centroid—that is, the centroid of the second sub-cluster obtained by bisecting a cluster.
- BisectionLevel: Specifies which bisection step a cluster is obtained from.
- WithinSS: Within-cluster sum of squares for the current cluster
- TotalWithinSS: Total within-cluster sum of squares of leaf clusters thus far obtained.

metrics

Several metrics related to the quality of the clustering, including

- Total sum of squares
- Total within-cluster sum of squares
- Between-cluster sum of squares
- Between-cluster sum of squares / Total sum of squares
- Sum of squares for cluster *x* , center_id *y* [...]

Examples

```
SELECT BISECTING_KMEANS('myModel', 'iris1', '*', '5'
    USING PARAMETERS exclude_columns = 'Species,id', split_method ='sum_squares', output_view = 'myBKmeansView');
```

See also

- Clustering data hierarchically using bisecting k-means
- APPLY_BISECTING_KMEANS

KMEANS

Executes the k-means algorithm on an input relation. The result is a model with a list of cluster centers.

You can export the resulting k-means model in VERTICA_MODELS or PMML format to apply it on data outside Vertica. You can also train a k-means model elsewhere, then import it to Vertica in PMML format to predict on data in Vertica.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
KMEANS ( 'model-name', 'input-relation', 'input-columns', 'num-clusters'
    [ USING PARAMETERS
      [exclude_columns = 'excluded-columns']
      [, max_iterations = max-iterations]
      [, epsilon = epsilon-value]
      [, { init_method = 'init-method' } | { initial_centers_table = 'init-table' } ]
      [, output_view = 'output-view']
      [, key_columns = 'key-columns'] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the input data for k-means. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. Input columns must be of data type numeric .

num-clusters

The number of clusters to create, an integer ≤ 10,000. This argument represents the *k* in k-means.

Parameters

Important

Parameters init_method and initial_centers_table are mutually exclusive. If you set both, the function returns an error.

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

max_iterations

The maximum number of iterations the algorithm performs. If you set this value to a number lower than the number of iterations needed for convergence, the algorithm may not converge.

Default: 10

epsilon

Determines whether the algorithm has converged. The algorithm is considered converged after no center has moved more than a distance of 'epsilon' from the previous iteration.

Default: 1e-4

init_method

The method used to find the initial cluster centers, one of the following:

- random
- kmeanspp (default): kmeans++ algorithm

  This value can be memory intensive for high k. If the function returns an error that not enough memory is available, decrease the value of k or use the random method.

initial_centers_table

The table with the initial cluster centers to use. Supply this value if you know the initial centers to use and do not want Vertica to find the initial cluster centers for you.

output_view

The name of the view where you save the assignments of each point to its cluster. You must have CREATE privileges on the schema where the view is saved.

key_columns

Comma-separated list of column names from *input-columns* that will appear as the columns of output_view . These columns should be picked such that their contents identify each input data point. This parameter is only used if output_view is specified. Columns listed in *input-columns* that are only meant to be used as key_columns and not for training should be listed in exclude_columns .

Model attributes

centers

A list that contains the center of each cluster.

metrics

A string summary of several metrics related to the quality of the clustering.

Examples

The following example creates k-means model myKmeansModel and applies it to input table iris1 . The call to APPLY_KMEANS mixes column names and constants. When a constant is passed in place of a column name, the constant is substituted for the value of the column in all rows:

```
=> SELECT KMEANS('myKmeansModel', 'iris1', '*', 5
USING PARAMETERS max_iterations=20, output_view='myKmeansView', key_columns='id', exclude_columns='Species, id');
        KMEANS
-----------------------------
 Finished in 12 iterations

(1 row)
=> SELECT id, APPLY_KMEANS(Sepal_Length, 2.2, 1.3, Petal_Width
USING PARAMETERS model_name='myKmeansModel', match_by_pos='true') FROM iris2;
 id  | APPLY_KMEANS
-----+--------------
  5 |            1
 10 |            1
 14 |            1
 15 |            1
 21 |            1
 22 |            1
 24 |            1
 25 |            1
 32 |            1
 33 |            1
 34 |            1
 35 |            1
 38 |            1
 39 |            1
 42 |            1
...
 (60 rows)
```

See also

- Clustering data using k-means
- APPLY_KMEANS
- IMPORT_MODELS
- EXPORT_MODELS
- PREDICT_PMML

KPROTOTYPES

Executes the k-prototypes algorithm on an input relation. The result is a model with a list of cluster centers.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Syntax

```
SELECT KPROTOTYPES ('`*`model-name`*`', '`*`input-relation`*`', '`*`input-columns`*`', `*`num-clusters`*`
        [USING PARAMETERS [exclude_columns = '`*`exclude-columns`*`']
        [, max_iterations = '`*`max-iterations`*`']
        [, epsilon = `*`epsilon`*`]
        [, {[init_method = '`*`init-method`*`'] } | { initial_centers_table = '`*`init-table`*`' } ]
        [, gamma = '`*`gamma`*`']
        [, output_view = '`*`output-view`*`']
        [, key_columns = '`*`key-columns`*`']]);
```

Behavior type

Volatile

Arguments

model-name

Name of the model resulting from the training.

input-relation

Name of the table or view containing the training samples.

input-columns

String containing a comma-separated list of columns to use from the input-relation, or asterisk (*) to select all columns.

num-clusters

Integer ≤ 10,000 representing the number of clusters to create. This argument represents the k in k-prototypes.

Parameters

exclude-columns

String containing a comma-separated list of column names from input-columns to exclude from processing.

Default: (empty)

max_iterations

Integer ≤ 1M representing the maximum number of iterations the algorithm performs.

Default: Integer ≤ 1M

epsilon

Integer which determines whether the algorithm has converged.

Default: 1e-4

init_method

String specifying the method used to find the initial k-prototypes cluster centers.

Default: "random"

initial_centers_table

The table with the initial cluster centers to use.

gamma

Float between 0 and 10000 specifying the weighing factor for categorical columns. It can determine relative importance of numerical and categorical attributes

Default: Inferred from data.

output_view

The name of the view where you save the assignments of each point to its cluster

key_columns

Comma-separated list of column names that identify the output rows. Columns must be in the input-columns argument list

Examples

The following example creates k-prototypes model small_model and applies it to input table small_test_mixed :

```
=> SELECT KPROTOTYPES('small_model_initcenters', 'small_test_mixed', 'x0, country', 3 USING PARAMETERS
initial_centers_table='small_test_mixed_centers', key_columns='pid');
     KPROTOTYPES
---------------------------
Finished in 2 iterations

(1 row)

=> SELECT country, x0, APPLY_KPROTOTYPES(country, x0
USING PARAMETERS model_name='small_model')
FROM small_test_mixed;
  country   | x0  | apply_kprototypes
------------+-----+-------------------
 'China'    | 20 |                 0
 'US'       | 85 |                 2
 'Russia'   | 80 |                 1
 'Brazil'   | 78 |                 1
 'US'       | 23 |                 0
 'US'       | 50 |                 0
 'Canada'   | 24 |                 0
 'Canada'   | 18 |                 0
 'Russia'   | 90 |                 2
 'Russia'   | 98 |                 2
 'Brazil'   | 89 |                 2
...
(45 rows)
```

See also

- APPLY_KPROTOTYPES
- KMEANS
- GET_MODEL_SUMMARY

LINEAR_REG

Executes linear regression on an input relation, and returns a linear regression model.

You can export the resulting linear regression model in VERTICA_MODELS or PMML format to apply it on data outside Vertica. You can also train a linear regression model elsewhere, then import it to Vertica in PMML format to model on data inside Vertica.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
LINEAR_REG ( 'model-name', 'input-relation', 'response-column', 'predictor-columns'
    [ USING PARAMETERS
        [exclude_columns = 'excluded-columns']
        [, optimizer = 'optimizer-method']
        [, regularization = 'regularization-method']
        [, epsilon = epsilon-value]
        [, max_iterations = iterations]
        [, lambda = lamda-value]
        [, alpha = alpha-value]
        [, fit_intercept = boolean-value] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

Table or view that contains the training data for building the model. If the input relation is defined in Hive, use

SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

Name of the input column that represents the dependent variable or outcome. All values in this column must be numeric , otherwise the model is invalid.

predictor-columns

Comma-separated list of columns in the input relation that represent independent variables for the model, or asterisk (*) to select all columns. If you select all columns, the argument list for parameter exclude_columns must include *response-column* , and any columns that are invalid as predictor columns.
All predictor columns must be of type numeric or BOOLEAN; otherwise the model is invalid.

> **Note**
> All BOOLEAN predictor values are converted to FLOAT values before training: 0 for false, 1 for true. No type checking occurs during prediction, so you can use a BOOLEAN predictor column in training, and during prediction provide a FLOAT column of the same name. In this case, all FLOAT values must be either 0 or 1.

Parameters

exclude_columns

Comma-separated list of columns from *predictor-columns* to exclude from processing.

optimizer

Optimizer method used to train the model, one of the following:

- Newton
- BFGS
- CGD

  Important
  If you select CGD , *regularization-method* must be set to L1 or ENet , otherwise the function returns an error.

Default: CGD if *regularization-method* is set to L1 or ENet , otherwise Newton .

regularization

Method of regularization, one of the following:

- None (default)
- L1
- L2
- ENet

epsilon

FLOAT in the range (0.0, 1.0), the error value at which to stop training. Training stops if either the difference between the actual and predicted values is less than or equal to epsilon or if the number of iterations exceeds max_iterations .
Default: 1e-6

max_iterations

INTEGER in the range (0, 1000000), the maximum number of training iterations. Training stops if either the number of iterations exceeds max_iterations or if the difference between the actual and predicted values is less than or equal to epsilon .
Default: 100

lambda

Integer ≥ 0, specifies the value of the regularization parameter.

Default: 1

alpha

Integer ≥ 0, specifies the value of the ENET regularization parameter, which defines how much L1 versus L2 regularization to provide. A value of 1 is equivalent to L1 and a value of 0 is equivalent to L2.

Value range: [0,1]
Default: 0.5

fit_intercept

Boolean, specifies whether the model includes an intercept. By setting to false, no intercept will be used in training the model. Note that setting fit_intercept to false does not work well with the BFGS optimizer.

Default: True

Model attributes

data

The data for the function, including:

- coeffNames : Name of the coefficients. This starts with intercept and then follows with the names of the predictors in the same order specified in the call.
- coeff : Vector of estimated coefficients, with the same order as coeffNames
- stdErr : Vector of the standard error of the coefficients, with the same order as coeffNames
- zValue (for logistic regression): Vector of z-values of the coefficients, in the same order as coeffNames
- tValue (for linear regression): Vector of t-values of the coefficients, in the same order as coeffNames
- pValue : Vector of p-values of the coefficients, in the same order as coeffNames

regularization

Type of regularization to use when training the model.

lambda

Regularization parameter. Higher values enforce stronger regularization. This value must be nonnegative.

alpha

Elastic net mixture parameter.

iterations

Number of iterations that actually occur for the convergence before exceeding max_iterations .

skippedRows

Number of rows of the input relation that were skipped because they contained an invalid value.

processedRows

Total number of input relation rows minus skippedRows .

callStr

Value of all input arguments specified when the function was called.

Examples

```
=> SELECT LINEAR_REG('myLinearRegModel', 'faithful', 'eruptions', 'waiting'
                     USING PARAMETERS optimizer='BFGS', fit_intercept=true);
       LINEAR_REG
-----------------------------
 Finished in 10 iterations

(1 row)
```

See also

- Building a linear regression model
- PREDICT_LINEAR_REG
- PREDICT_PMML
- IMPORT_MODELS
- EXPORT_MODELS

LOGISTIC_REG

Executes logistic regression on an input relation. The result is a logistic regression model.

You can export the resulting logistic regression model in VERTICA_MODELS or PMML format to apply it on data outside Vertica. You can also train a logistic regression model elsewhere, then import it to Vertica in PMML format to predict on data in Vertica.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
LOGISTIC_REG ( 'model-name', 'input-relation', 'response-column', 'predictor-columns'
    [ USING PARAMETERS [exclude_columns = 'excluded-columns']
        [, optimizer = 'optimizer-method']
        [, regularization = 'regularization-method']
        [, epsilon = epsilon-value]
        [, max_iterations = iterations]
        [, lambda = lamda-value]
        [, alpha = alpha-value]
        [, fit_intercept = boolean-value] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the training data for building the model. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

The input column that represents the dependent variable or outcome. The column value must be 0 or 1, and of type numeric or BOOLEAN. The function automatically skips all other values.

predictor-columns

Comma-separated list of columns in the input relation that represent independent variables for the model, or asterisk (*) to select all columns. If you select all columns, the argument list for parameter exclude_columns must include *response-column* , and any columns that are invalid as predictor columns.
All predictor columns must be of type numeric or BOOLEAN; otherwise the model is invalid.

> **Note**
> All BOOLEAN predictor values are converted to FLOAT values before training: 0 for false, 1 for true. No type checking occurs during prediction, so you can use a BOOLEAN predictor column in training, and during prediction provide a FLOAT column of the same name. In this case, all FLOAT values must be either 0 or 1.

Parameters

exclude_columns

Comma-separated list of columns from *predictor-columns* to exclude from processing.

optimizer

The optimizer method used to train the model, one of the following:

- Newton
- BFGS
- CGD
  Important
  If you select CGD , *regularization-method* must be set to L1 or ENet , otherwise the function returns an error.

Default: CGD if *regularization-method* is set to L1 or ENet , otherwise Newton .

regularization

The method of regularization, one of the following:

- None (default)
- L1
- L2
- ENet

epsilon

FLOAT in the range (0.0, 1.0), the error value at which to stop training. Training stops if either the difference between the actual and predicted values is less than or equal to epsilon or if the number of iterations exceeds max_iterations .

Default: 1e-6

max_iterations

INTEGER in the range (0, 1000000), the maximum number of training iterations. Training stops if either the number of iterations exceeds max_iterations or if the difference between the actual and predicted values is less than or equal to epsilon .

Default: 100

lambda

Integer ≥ 0, specifies the value of the regularization parameter.

Default: 1

alpha

Integer ≥ 0, specifies the value of the ENET regularization parameter, which defines how much L1 versus L2 regularization to provide. A value of 1 is equivalent to L1 and a value of 0 is equivalent to L2.

Value range: [0,1]
Default: 0.5

fit_intercept

Boolean, specifies whether the model includes an intercept. By setting to false, no intercept will be used in training the model. Note that setting fit_intercept to false does not work well with the BFGS optimizer.

Default: True

Model attributes

data

The data for the function, including:

- coeffNames : Name of the coefficients. This starts with intercept and then follows with the names of the predictors in the same order specified in the call.
- coeff : Vector of estimated coefficients, with the same order as coeffNames
- stdErr : Vector of the standard error of the coefficients, with the same order as coeffNames
- zValue (for logistic regression): Vector of z-values of the coefficients, in the same order as coeffNames
- tValue (for linear regression): Vector of t-values of the coefficients, in the same order as coeffNames
- pValue : Vector of p-values of the coefficients, in the same order as coeffNames

regularization

Type of regularization to use when training the model.

lambda

Regularization parameter. Higher values enforce stronger regularization. This value must be nonnegative.

alpha

Elastic net mixture parameter.

iterations

Number of iterations that actually occur for the convergence before exceeding max_iterations .

skippedRows

Number of rows of the input relation that were skipped because they contained an invalid value.

processedRows

Total number of input relation rows minus skippedRows .

callStr

Value of all input arguments specified when the function was called.

Privileges

Superuser, or SELECT privileges on the input relation

Examples

```
=> SELECT LOGISTIC_REG('myLogisticRegModel', 'mtcars', 'am',
                       'mpg, cyl, disp, hp, drat, wt, qsec, vs, gear, carb'
                        USING PARAMETERS exclude_columns='hp', optimizer='BFGS', fit_intercept=true);
      LOGISTIC_REG
---------------------------
 Finished in 20 iterations

(1 row)
```

See also

- Building a logistic regression model
- PREDICT_LOGISTIC_REG
- PREDICT_PMML
- IMPORT_MODELS
- EXPORT_MODELS

MOVING_AVERAGE

Creates a moving-average (MA) model from a stationary time series with consistent timesteps that can then be used for prediction via PREDICT_MOVING_AVERAGE .

Moving average models use the errors of previous predictions to make future predictions. More specifically, the user-specified *lag* determines how many previous predictions and errors it takes into account during computation.

Since its input data must be sorted by timestamp, this algorithm is single-threaded.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
MOVING_AVERAGE ('model-name', 'input-relation', 'data-column', 'timestamp-column'
    [ USING PARAMETERS
        [ q = lags ]
        [, missing = "imputation-method" ]
        [, regularization = "regularization-method" ]
        [, lambda = regularization-value ]
        [, compute_mse = boolean ]
    ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view containing the *timestamp-column* .

This algorithm expects a stationary time series as input; using a time series with a mean that shifts over time may lead to weaker results.

data-column

An input column of type NUMERIC that contains the dependent variables or outcomes.

timestamp-column

One INTEGER, FLOAT, or TIMESTAMP column that represent the timestamp variable. Timesteps must be consistent.

Parameters

q

INTEGER in the range [1, 67), the number of lags to consider in the computation.

Default: 1

missing

One of the following methods for handling missing values:

- **drop** : Missing values are ignored.

- **error** : Missing values raise an error.
- **zero** : Missing values are replaced with 0.
- **linear_interpolation** : Missing values are replaced by linearly interpolated values based on the nearest valid entries before and after the missing value. This means that in cases where the first or last values in a dataset are missing, they will simply be dropped.

Default: linear_interpolation

regularization

One of the following regularization methods used when fitting the data:

- **None**
- **L2** : weight regularization term which penalizes the squared weight value

Default: None

lambda

FLOAT in the range [0, 100000], the regularization value, lambda.

Default: 1.0

compute_mse

BOOLEAN, whether to calculate and output the mean squared error (MSE).

This parameter only accepts "true" or "false" rather than the standard literal equivalents for BOOLEANs like 1 or 0.
Default: False

Examples

See Moving-average model example .

See also

- PREDICT_MOVING_AVERAGE
- GET_MODEL_SUMMARY

NAIVE_BAYES

Executes the Naive Bayes algorithm on an input relation and returns a Naive Bayes model.

Columns are treated according to data type:

- FLOAT: Values are assumed to follow some Gaussian distribution.
- INTEGER: Values are assumed to belong to one multinomial distribution.
- CHAR/VARCHAR: Values are assumed to follow some categorical distribution. The string values stored in these columns must not be greater than 128 characters.
- BOOLEAN: Values are treated as categorical with two values.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
NAIVE_BAYES ( 'model-name', 'input-relation', 'response-column', 'predictor-columns'
    [ USING PARAMETERS [exclude_columns = 'excluded-columns'] [, alpha = alpha-value] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the training data for building the model. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

Name of the input column that represents the dependent variable, or outcome. This column must contain discrete labels that represent different class labels.

The response column must be of type numeric , CHAR/VARCHAR, or BOOLEAN; otherwise the model is invalid.

> **Note**
> Vertica automatically casts numeric response column values to VARCHAR.

predictor-columns

Comma-separated list of columns in the input relation that represent independent variables for the model, or asterisk (*) to select all columns. If you select all columns, the argument list for parameter exclude_columns must include *response-column* , and any columns that are invalid as predictor columns.
All predictor columns must be of type numeric , CHAR/VARCHAR, or BOOLEAN; otherwise the model is invalid. BOOLEAN column values are converted to FLOAT values before training: 0 for false, 1 for true.

Parameters

exclude_columns

Comma-separated list of columns from *predictor-columns* to exclude from processing.

alpha

Float, specifies use of Laplace smoothing if the event model is categorical, multinomial, or Bernoulli.

Default: 1.0

Model attributes

colsInfo

The information from the response and predictor columns used in training:

- index: The index (starting at 0) of the column as provided in training. Index 0 is used for the response column.
- name: The column name.
- type: The label used for response with a value of Gaussian, Multinominal, Categorical, or Bernoulli.

alpha

The smooth parameter value.

prior

The percentage of each class among all training samples:

- label: The class label.
- value: The percentage of each class.

nRowsTotal

The number of samples accepted for training from the data set.

nRowsRejected

The number of samples rejected for training.

callStr

The SQL statement used to replicate the training.

Gaussian

The Gaussian model conditioned on the class indicated by the class_name:

- index: The index of the predictor column.
- mu: The mean value of the model.
- sigmaSq: The squared standard deviation of the model.

Multinominal

The Multinomial model conditioned on the class indicated by the class_name:

- index: The index of the predictor column.
- prob: The probability conditioned on the class indicated by the class_name.

Bernoulli

The Bernoulli model conditioned on the class indicated by the class_name:

- index: The index of the predictor column.
- probTrue: The probability of having the value TRUE in this predictor column.

Categorical

The Gaussian model conditioned on the class indicated by the class_name:

- category: The value in the predictor name.
- <class_name>: The probability of having that value conditioned on the class indicated by the class_name.

Privileges

Superuser, or SELECT privileges on the input relation.

Examples

```
=> SELECT NAIVE_BAYES('naive_house84_model', 'house84_train', 'party', '*'
                      USING PARAMETERS exclude_columns='party, id');
                              NAIVE_BAYES
-------------------------------------------------------
 Finished. Accepted Rows: 324  Rejected Rows: 0
(1 row)
```

See also

- Classifying data using naive bayes
- PREDICT_NAIVE_BAYES
- PREDICT_NAIVE_BAYES_CLASSES

POISSON_REG

Executes Poisson regression on an input relation, and returns a Poisson regression model.

You can export the resulting Poisson regression model in VERTICA_MODELS or PMML format to apply it on data outside Vertica. You can also train a Poisson regression model elsewhere, then import it to Vertica in PMML format to apply it on data inside Vertica.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
POISSON_REG ( 'model-name', 'input-table', 'response-column', 'predictor-columns'
    [ USING PARAMETERS
        [exclude_columns = 'excluded-columns']
        [, optimizer = 'optimizer-method']
        [, regularization = 'regularization-method']
        [, epsilon = epsilon-value]
        [, max_iterations = iterations]
        [, lambda = lamda-value]
        [, fit_intercept = boolean-value] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-table

Table or view that contains the training data for building the model. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

Name of input column that represents the dependent variable or outcome. All values in this column must be numeric , otherwise the model is invalid.

predictor-columns

Comma-separated list of columns in the input relation that represent independent variables for the model, or asterisk (*) to select all columns. If you select all columns, the argument list for parameter exclude_columns must include *response-column* , and any columns that are invalid as predictor columns.
All predictor columns must be of type numeric or BOOLEAN; otherwise the model is invalid.

> **Note**
> All BOOLEAN predictor values are converted to FLOAT values before training: 0 for false, 1 for true. No type checking occurs during prediction, so you can use a BOOLEAN predictor column in training, and during prediction provide a FLOAT column of the same name. In this case, all FLOAT values must be either 0 or 1.

Parameters

exclude_columns

Comma-separated list of columns from *predictor-columns* to exclude from processing.

optimizer

Optimizer method used to train the model. The currently supported method is Newton .

regularization

Method of regularization, one of the following:

- None (default)
- L2

epsilon

FLOAT in the range (0.0, 1.0), the error value at which to stop training. Training stops if either the relative change in Poisson deviance is less than or equal to epsilon or if the number of iterations exceeds max_iterations .

Default: 1e-6

max_iterations

INTEGER in the range (0, 1000000), the maximum number of training iterations. Training stops if either the number of iterations exceeds max_iterations or the relative change in Poisson deviance is less than or equal to epsilon.

lambda

FLOAT ≥ 0, specifies the regularization strength.

Default: 1.0

fit_intercept

Boolean, specifies whether the model includes an intercept. By setting to false, no intercept will be used in training the model."

Default: True

Model attributes

data

Data for the function, including:

- coeffNames : Name of the coefficients. This starts with intercept and then follows with the names of the predictors in the same order specified in the call.
- coeff : Vector of estimated coefficients, with the same order as coeffNames
- stdErr : Vector of the standard error of the coefficients, with the same order as coeffNames
- zValue : (for logistic and Poisson regression): Vector of z-values of the coefficients, in the same order as coeffNames
- tValue (for linear regression): Vector of t-values of the coefficients, in the same order as coeffNames
- pValue : Vector of p-values of the coefficients, in the same order as coeffNames

regularization

Type of regularization to use when training the model.

lambda

Regularization parameter. Higher values enforce stronger regularization. This value must be nonnegative.

iterations

Number of iterations that actually occur for the convergence before exceeding max_iterations .

skippedRows

Number of rows of the input relation that were skipped because they contained an invalid value.

processedRows

Total number of input relation rows minus skippedRows .

callStr

Value of all input arguments specified when the function was called.

Examples

```
=> SELECT POISSON_REG('myModel', 'numericFaithful', 'eruptions', 'waiting' USING PARAMETERS epsilon=1e-8);
poisson_reg
---------------------------
Finished in 7 iterations

(1 row)
```

See also

- Building a linear regression model
- PREDICT_LINEAR_REG
- PREDICT_PMML
- IMPORT_MODELS
- EXPORT_MODELS

RF_CLASSIFIER

Trains a random forest model for classification on an input relation.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RF_CLASSIFIER ( 'model-name', input-relation, 'response-column', 'predictor-columns'
    [ USING PARAMETERS
        [exclude_columns = 'excluded-columns']
        [, ntree = num-trees]
        [, mtry = num-features]
        [, sampling_size = sampling-size]
        [, max_depth = depth]
        [, max_breadth = breadth]
        [, min_leaf_size = leaf-size]
        [, min_info_gain = threshold]
        [, nbins = num-bins] ] )
```

Arguments

model-name

Identifies the model stored as a result of the training, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the training samples. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

An input column of type numeric , CHAR/VARCHAR, or BOOLEAN that represents the dependent variable.

> **Note**
> Vertica automatically casts numeric response column values to VARCHAR.

predictor-columns

Comma-separated list of columns in the input relation that represent independent variables for the model, or asterisk (*) to select all columns. If you select all columns, the argument list for parameter exclude_columns must include *response-column* , and any columns that are invalid as predictor columns.
All predictor columns must be of type numeric , CHAR/VARCHAR, or BOOLEAN; otherwise the model is invalid.

Vertica XGBoost and Random Forest algorithms offer native support for categorical columns (BOOL/VARCHAR). Simply pass the categorical columns as predictors to the models and the algorithm will automatically treat the columns as categorical and will not attempt to split them into bins in the same manner as numerical columns; Vertica treats these columns as true categorical values and does not simply cast them to continuous values under-the-hood.

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

ntree

Integer in the range [1, 1000], the number of trees in the forest.
Default: 20

mtry

Integer in the range [1, *number-predictors*], the number of randomly chosen features from which to pick the best feature to split on a given tree node.

Default: Square root of the total number of predictors

sampling_size

Float in the range (0.0, 1.0], the portion of the input data set that is randomly picked for training each tree.
Default: 0.632

max_depth

Integer in the range [1, 100], the maximum depth for growing each tree. For example, a max_depth of 0 represents a tree with only a root node, and a max_depth of 2 represents a tree with four leaf nodes.
Default: 5

max_breadth

Integer in the range [1, 1e9], the maximum number of leaf nodes a tree can have.
Default: 32

min_leaf_size

Integer in the range [1, 1e6], the minimum number of samples each branch must have after splitting a node. A split that results in fewer remaining samples in its left or right branch is be discarded, and the node is treated as a leaf node.
Default: 1

min_info_gain

Float in the range [0.0, 1.0), the minimum threshold for including a split. A split with information gain less than this threshold is discarded.
Default: 0.0

nbins

Integer in the range [2, 1000], the number of bins to use for discretizing continuous features.
Default: 32

Model attributes

data

Data for the function, including:

- predictorNames : The name of the predictors in the same order they were specified for training the model.
- predictorTypes : The type of the predictors in the same order as their names in predictorNames.

ntree

Number of trees in the model.

skippedRows

Number of rows in input_relation that were skipped because they contained an invalid value.

processedRows

Total number of rows in input_relation minus skippedRows .

callStr

Value of all input arguments that were specified at the time the function was called.

Examples

```
=> SELECT RF_CLASSIFIER ('myRFModel', 'iris', 'Species', 'Sepal_Length, Sepal_Width,
Petal_Length, Petal_Width' USING PARAMETERS ntree=100, sampling_size=0.3);

RF_CLASSIFIER
-------------------------------------------------------
Finished training
(1 row)
```

See also

- Classifying data using random forest
- PREDICT_RF_CLASSIFIER
- PREDICT_RF_CLASSIFIER_CLASSES

RF_REGRESSOR

Trains a random forest model for regression on an input relation.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RF_REGRESSOR ( 'model-name', input-relation, 'response-column', 'predictor-columns'
    [ USING PARAMETERS
        [exclude_columns = 'excluded-columns']
        [, ntree = num-trees]
        [, mtry = num-features]
        [, sampling_size = sampling-size]
        [, max_depth = depth]
        [, max_breadth = breadth]
        [, min_leaf_size = leaf-size]
        [, min_info_gain = threshold]
        [, nbins = num-bins] ] )
```

Arguments

model-name

The model that is stored as a result of training, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the training samples. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

A numeric input column that represents the dependent variable.

predictor-columns

Comma-separated list of columns in the input relation that represent independent variables for the model, or asterisk (*) to select all columns. If you select all columns, the argument list for parameter exclude_columns must include *response-column* , and any columns that are invalid as predictor columns.
All predictor columns must be of type numeric , CHAR/VARCHAR, or BOOLEAN; otherwise the model is invalid.

Vertica XGBoost and Random Forest algorithms offer native support for categorical columns (BOOL/VARCHAR). Simply pass the categorical columns as predictors to the models and the algorithm will automatically treat the columns as categorical and will not attempt to split them into bins in the same manner as numerical columns; Vertica treats these columns as true categorical values and does not simply cast them to continuous values under-the-hood.

Parameters

exclude_columns

Comma-separated list of columns from *predictor-columns* to exclude from processing.

ntree

Integer in the range [1, 1000], the number of trees in the forest.
Default: 20

mtry

Integer in the range [1, *number-predictors*], the number of features to consider at the split of a tree node.

Default: One-third the total number of predictors

sampling_size

Float in the range (0.0, 1.0], the portion of the input data set that is randomly picked for training each tree.
Default: 0.632

max_depth

Integer in the range [1, 100], the maximum depth for growing each tree. For example, a max_depth of 0 represents a tree with only a root node, and a max_depth of 2 represents a tree with four leaf nodes.
Default: 5

max_breadth

Integer in the range [1, 1e9], the maximum number of leaf nodes a tree can have.
Default: 32

min_leaf_size

Integer in the range [1, 1e6], the minimum number of samples each branch must have after splitting a node. A split that results in fewer remaining samples in its left or right branch is be discarded, and the node is treated as a leaf node.

The default value of this parameter differs from that of analogous parameters in libraries like sklearn and will therefore yield a model with predicted values that differ from the original response values.
Default: 5

min_info_gain

Float in the range [0.0, 1.0), the minimum threshold for including a split. A split with information gain less than this threshold is discarded.
Default: 0.0

nbins

Integer in the range [2, 1000], the number of bins to use for discretizing continuous features.
Default: 32

Model attributes

data

Data for the function, including:

- predictorNames : The name of the predictors in the same order they were specified for training the model.
- predictorTypes : The type of the predictors in the same order as their names in predictorNames.

ntree

Number of trees in the model.

skippedRows

Number of rows in input_relation that were skipped because they contained an invalid value.

processedRows

Total number of rows in input_relation minus skippedRows .

callStr

Value of all input arguments that were specified at the time the function was called.

Examples

```
=> SELECT RF_REGRESSOR ('myRFRegressorModel', 'mtcars', 'carb', 'mpg, cyl, hp, drat, wt' USING PARAMETERS
ntree=100, sampling_size=0.3);
RF_REGRESSOR
--------------
Finished
(1 row)
```

See also

- Building a random forest regression model
- GET_MODEL_SUMMARY
- PREDICT_RF_REGRESSOR

SVM_CLASSIFIER

Trains the SVM model on an input relation.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SVM_CLASSIFIER ( 'model-name', input-relation, 'response-column', 'predictor-columns'
    [ USING PARAMETERS
        [exclude_columns = 'excluded-columns']
        [, C = 'cost']
        [, epsilon = 'epsilon-value']
        [, max_iterations = 'max-iterations']
        [, class_weights = 'weight']
        [, intercept_mode = 'intercept-mode']
        [, intercept_scaling = 'scale'] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the training data. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

The input column that represents the dependent variable or outcome. The column value must be 0 or 1, and of type numeric or BOOLEAN, otherwise the function returns with an error.

predictor-columns

Comma-separated list of columns in the input relation that represent independent variables for the model, or asterisk (*) to select all columns. If you select all columns, the argument list for parameter exclude_columns must include *response-column* , and any columns that are invalid as predictor columns.
All predictor columns must be of type numeric or BOOLEAN; otherwise the model is invalid.

> **Note**
> All BOOLEAN predictor values are converted to FLOAT values before training: 0 for false, 1 for true. No type checking occurs during prediction, so you can use a BOOLEAN predictor column in training, and during prediction provide a FLOAT column of the same name. In this case, all FLOAT values must be either 0 or 1.

Parameters

exclude_columns

Comma-separated list of columns from *predictor-columns* to exclude from processing.

C

Weight for misclassification cost. The algorithm minimizes the regularization cost and the misclassification cost.

Default: 1.0

epsilon

Used to control accuracy.

Default: 1e-3

max_iterations

Maximum number of iterations that the algorithm performs.

Default: 100

class_weights

Specifies how to determine weights of the two classes, one of the following:

- None (default): No weights are used
- *value0* , *value1* : Two comma-delimited strings that specify two positive FLOAT values, where *value0* assigns a weight to class 0, and *value1* assigns a weight to class 1.
- auto : Weights each class according to the number of samples.

intercept_mode

Specifies how to treat the intercept, one of the following:

- regularized (default): Fits the intercept and applies a regularization on it.
- unregularized : Fits the intercept but does not include it in regularization.

intercept_scaling

Float value that serves as the value of a dummy feature whose coefficient Vertica uses to calculate the model intercept. Because the dummy feature is not in the training data, its values are set to a constant, by default 1.

Model attributes

coeff

Coefficients in the model:

- colNames : Intercept, or predictor column name
- coefficients : Coefficient value

nAccepted

Number of samples accepted for training from the data set

nRejected

Number of samples rejected when training

nIteration

Number of iterations used in training

callStr

SQL statement used to replicate the training

Examples

The following example uses SVM_CLASSIFIER on the mtcars table:

```
=> SELECT SVM_CLASSIFIER(
     'mySvmClassModel', 'mtcars', 'am', 'mpg,cyl,disp,hp,drat,wt,qsec,vs,gear,carb'
     USING PARAMETERS exclude_columns = 'hp,drat');
SVM_CLASSIFIER
---------------------------------------------------------------
Finished in 15 iterations.
Accepted Rows: 32  Rejected Rows: 0
(1 row)
```

See also

- Classifying data using SVM (support vector machine)
- SVM (support vector machine) for classification
- PREDICT_SVM_CLASSIFIER

SVM_REGRESSOR

Trains the SVM model on an input relation.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SVM_REGRESSOR ( 'model-name', input-relation, 'response-column', 'predictor-columns'
    [ USING PARAMETERS
        [exclude_columns = 'excluded-columns']
        [, error_tolerance = error-tolerance]
        [, C = cost]
        [, epsilon = epsilon-value]
        [, max_iterations = max-iterations]
        [, intercept_mode = 'mode']
        [, intercept_scaling = 'scale'] ] )
```

Arguments

model-name

Identifies the model to create, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

input-relation

The table or view that contains the training data. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

An input column that represents the dependent variable or outcome. The column must be a numeric data type.

predictor-columns

Comma-separated list of columns in the input relation that represent independent variables for the model, or asterisk (*) to select all columns. If you select all columns, the argument list for parameter exclude_columns must include *response-column* , and any columns that are invalid as predictor columns.
All predictor columns must be of type numeric or BOOLEAN; otherwise the model is invalid.

> **Note**
> All BOOLEAN predictor values are converted to FLOAT values before training: 0 for false, 1 for true. No type checking occurs during prediction, so you can use a BOOLEAN predictor column in training, and during prediction provide a FLOAT column of the same name. In this case, all FLOAT values must be either 0 or 1.

Parameters

exclude_columns

Comma-separated list of columns from *predictor-columns* to exclude from processing.

error_tolerance

Defines the acceptable error margin. Any data points outside this region add a penalty to the cost function.

Default: 0.1

C

The weight for misclassification cost. The algorithm minimizes the regularization cost and the misclassification cost.

Default: 1.0

epsilon

Used to control accuracy.

Default: 1e-3

max_iterations

The maximum number of iterations that the algorithm performs.

Default: 100

intercept_mode

A string that specifies how to treat the intercept, one of the following

- regularized (default): Fits the intercept and applies a regularization on it.
- unregularized : Fits the intercept but does not include it in regularization.

intercept_scaling

A FLOAT value, serves as the value of a dummy feature whose coefficient Vertica uses to calculate the model intercept. Because the dummy feature is not in the training data, its values are set to a constant, by default set to 1.

Model attributes

coeff

Coefficients in the model:

- colNames : Intercept, or predictor column name
- coefficients : Coefficient value

nAccepted

Number of samples accepted for training from the data set

nRejected

Number of samples rejected when training

nIteration

Number of iterations used in training

callStr

SQL statement used to replicate the training

Examples

```
=> SELECT SVM_REGRESSOR('mySvmRegModel', 'faithful', 'eruptions', 'waiting'
                        USING PARAMETERS error_tolerance=0.1, max_iterations=100);
SVM_REGRESSOR
---------------------------------------------------------------------------
Finished in 5 iterations.
Accepted Rows: 272  Rejected Rows: 0
(1 row)
```

See also

- Building an SVM for regression model
- SVM (support vector machine) for regression
- PREDICT_SVM_REGRESSOR

XGB_CLASSIFIER

Trains an XGBoost model for classification on an input relation.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
XGB_CLASSIFIER ('model-name', 'input-relation', 'response-column', 'predictor-columns'
    [ USING PARAMETERS param=value[,...] ] )
```

Arguments

model-name

Name of the model (case-insensitive).

input-relation

The table or view that contains the training samples. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

An input column of type CHAR or VARCHAR that represents the dependent variable or outcome.

predictor-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. Columns must be of data types CHAR, VARCHAR, BOOL, INT, or FLOAT.

Columns of type CHAR, VARCHAR, and BOOL are treated as categorical features; all others are treated as numeric features.

Vertica XGBoost and Random Forest algorithms offer native support for categorical columns (BOOL/VARCHAR). Simply pass the categorical columns as predictors to the models and the algorithm will automatically treat the columns as categorical and will not attempt to split them into bins in the same manner as numerical columns; Vertica treats these columns as true categorical values and does not simply cast them to continuous values under-the-hood.

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

max_ntree

Integer in the range [1,1000] that sets the maximum number of trees to create.

Default: 10

max_depth

Integer in the range [1,20] that specifies the maximum depth of each tree.

Default: 6

objective

The objective/loss function used to iteratively improve the model. 'crossentropy' is the only option.

Default: 'crossentropy'

split_proposal_method

The splitting strategy for the feature columns. 'global' is the only option. This method calculates the split for each feature column only at the beginning of the algorithm. The feature columns are split into the number of bins specified by nbins .

Default: 'global'

learning_rate

Float in the range (0,1] that specifies the weight for each tree's prediction. Setting this parameter can reduce each tree's impact and thereby prevent earlier trees from monopolizing improvements at the expense of contributions from later trees.

Default: 0.3

min_split_loss

Float in the range [0,1000] that specifies the minimum amount of improvement each split must achieve on the model's objective function value to avoid being pruned.

If set to 0 or omitted, no minimum is set. In this case, trees are pruned according to positive or negative objective function values.
Default: 0.0 (disable)

weight_reg

Float in the range [0,1000] that specifies the regularization term applied to the weights of classification tree leaves. The higher the setting, the sparser or smoother the weights are, which can help prevent over-fitting.

Default: 1.0

nbins

Integer in the range (1,1000] that specifies the number of bins to use for finding splits in each column. More bins leads to longer runtime but more fine-grained and possibly better splits.

Default: 32

sampling_size

Float in the range (0,1] that specifies the fraction of rows to use in each training iteration.

A value of 1 indicates that all rows are used.
Default: 1.0

col_sample_by_tree

Float in the range (0,1] that specifies the fraction of columns (features), chosen at random, to use when building each tree.

A value of 1 indicates that all columns are used.
col_sample_by parameters "stack" on top of each other if several are specified. That is, given a set of 24 columns, for col_sample_by_tree=0.5 and col_sample_by_node=0.5 , col_sample_by_tree samples 12 columns, reducing the available, unsampled column pool to 12. col_sample_by_node then samples half of the remaining pool, so each node samples 6 columns.

This algorithm will always sample at least one column.

Default: 1

col_sample_by_node

Float in the range (0,1] that specifies the fraction of columns (features), chosen at random, to use when evaluating each split.

A value of 1 indicates that all columns are used.
col_sample_by parameters "stack" on top of each other if several are specified. That is, given a set of 24 columns, for col_sample_by_tree=0.5 and col_sample_by_node=0.5 , col_sample_by_tree samples 12 columns, reducing the available, unsampled column pool to 12.
col_sample_by_node then samples half of the remaining pool, so each node samples 6 columns.

This algorithm will always sample at least one column.

Default: 1

Examples

See XGBoost for classification .

XGB_REGRESSOR

Trains an XGBoost model for regression on an input relation.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
XGB_REGRESSOR ('model-name', 'input-relation', 'response-column', 'predictor-columns'
    [ USING PARAMETERS param=value[,...] ] )
```

Arguments

model-name

Name of the model (case-insensitive).

input-relation

The table or view that contains the training samples. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

An input column of type INTEGER or FLOAT that represents the dependent variable or outcome.

predictor-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. Columns must be of data types CHAR, VARCHAR, BOOL, INT, or FLOAT.

Columns of type CHAR, VARCHAR, and BOOL are treated as categorical features; all others are treated as numeric features.
Vertica XGBoost and Random Forest algorithms offer native support for categorical columns (BOOL/VARCHAR). Simply pass the categorical columns as predictors to the models and the algorithm will automatically treat the columns as categorical and will not attempt to split them into bins in the same manner as numerical columns; Vertica treats these columns as true categorical values and does not simply cast them to continuous values under-the-hood.

Parameters

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

max_ntree

Integer in the range [1,1000] that sets the maximum number of trees to create.

Default: 10

max_depth

Integer in the range [1,20] that specifies the maximum depth of each tree.

Default: 6

objective

The objective/loss function used to iteratively improve the model. 'squarederror' is the only option.

Default: 'squarederror'

split_proposal_method

The splitting strategy for the feature columns. 'global' is the only option. This method calculates the split for each feature column only at the beginning of the algorithm. The feature columns are split into the number of bins specified by nbins .

Default: 'global'

learning_rate

Float in the range (0,1] that specifies the weight for each tree's prediction. Setting this parameter can reduce each tree's impact and thereby prevent earlier trees from monopolizing improvements at the expense of contributions from later trees.

Default: 0.3

min_split_loss

Float in the range [0,1000] that specifies the minimum amount of improvement each split must achieve on the model's objective function value to avoid being pruned.

If set to 0 or omitted, no minimum is set. In this case, trees are pruned according to positive or negative objective function values.
Default: 0.0 (disable)

weight_reg

Float in the range [0,1000] that specifies the regularization term applied to the weights of classification tree leaves. The higher the setting, the sparser or smoother the weights are, which can help prevent over-fitting.

Default: 1.0

nbins

Integer in the range (1,1000] that specifies the number of bins to use for finding splits in each column. More bins leads to longer runtime but more fine-grained and possibly better splits.

Default: 32

sampling_size

Float in the range (0,1] that specifies the fraction of rows to use in each training iteration.

A value of 1 indicates that all rows are used.
Default: 1.0

col_sample_by_tree

Float in the range (0,1] that specifies the fraction of columns (features), chosen at random, to use when building each tree.

A value of 1 indicates that all columns are used.
col_sample_by parameters "stack" on top of each other if several are specified. That is, given a set of 24 columns, for col_sample_by_tree=0.5 and col_sample_by_node=0.5 , col_sample_by_tree samples 12 columns, reducing the available, unsampled column pool to 12. col_sample_by_node then samples half of the remaining pool, so each node samples 6 columns.

This algorithm will always sample at least one column.

Default: 1

col_sample_by_node

Float in the range (0,1] that specifies the fraction of columns (features), chosen at random, to use when evaluating each split.

A value of 1 indicates that all columns are used.
col_sample_by parameters "stack" on top of each other if several are specified. That is, given a set of 24 columns, for col_sample_by_tree=0.5 and col_sample_by_node=0.5 , col_sample_by_tree samples 12 columns, reducing the available, unsampled column pool to 12. col_sample_by_node then samples half of the remaining pool, so each node samples 6 columns.

This algorithm will always sample at least one column.

Default: 1

Examples

See XGBoost for regression .

Model evaluation

A set of Vertica machine learning functions evaluate the prediction data that is generated by trained models, or return information about the models themselves.

In this section

- CONFUSION_MATRIX
- CROSS_VALIDATE
- ERROR_RATE
- LIFT_TABLE
- MSE
- PRC
- READ_TREE
- RF_PREDICTOR_IMPORTANCE
- ROC
- RSQUARED
- XGB_PREDICTOR_IMPORTANCE

CONFUSION_MATRIX

Computes the confusion matrix of a table with observed and predicted values of a response variable. CONFUSION_MATRIX produces a table with the following dimensions:

- Rows: Number of classes
- Columns: Number of classes + 2

Syntax

```
CONFUSION_MATRIX ( targets, predictions [ USING PARAMETERS num_classes = num-classes ] OVER()
```

Arguments

targets

An input column that contains the true values of the response variable.

predictions

An input column that contains the predicted class labels.

Arguments *targets* and *predictions* must be set to input columns of the same data type, one of the following: INTEGER, BOOLEAN, or CHAR/VARCHAR. Depending on their data type, these columns identify classes as follows:

- INTEGER: Zero-based consecutive integers between 0 and (*num-classes* -1) inclusive, where *num-classes* is the number of classes. For example, given the following input column values— {0, 1, 2, 3, 4 }—Vertica assumes five classes.

  > **Note**
  > If input column values are not consecutive, Vertica interpolates the missing values. Thus, given the following input values— {0, 1, 3, 5, 6,} — Vertica assumes seven classes.

- BOOLEAN: Yes or No
- CHAR/VARCHAR: Class names. If the input columns are of type CHAR/VARCHAR columns, you must also set parameter num_classes to the number of classes.

  > **Note**
  > Vertica computes the number of classes as the union of values in both input columns. For example, given the following sets of values in the *targets* and *predictions* input columns, Vertica counts four classes:
  >
  > ```
  > {'milk', 'soy milk', 'cream'}
  > {'soy milk', 'almond milk'}
  > ```

Parameters

num_classes

An integer > 1, specifies the number of classes to pass to the function.
You must set this parameter if the specified input columns are of type CHAR/VARCHAR. Otherwise, the function processes this parameter according to the column data types:

- INTEGER: By default set to 2, you must set this parameter correctly if the number of classes is any other value.
- BOOLEAN: By default set to 2, cannot be set to any other value.

Examples

This example computes the confusion matrix for a logistic regression model that classifies cars in the mtcars data set as automatic or manual transmission. Observed values are in input column obs , while predicted values are in input column pred . Because this is a binary classification problem, all values are either 0 or 1.

In the table returned, all 19 cars with a value of 0 in column am are correctly predicted by PREDICT_LOGISTIC_REGRESSION as having a value of 0. Of the 13 cars with a value of 1 in column am , 12 are correctly predicted to have a value of 1, while 1 car is incorrectly classified as having a value of 0:

```
=> SELECT CONFUSION_MATRIX(obs::int, pred::int USING PARAMETERS num_classes=2) OVER()
   FROM (SELECT am AS obs, PREDICT_LOGISTIC_REG(mpg, cyl, disp,drat, wt, qsec, vs, gear, carb
         USING PARAMETERS model_name='myLogisticRegModel')AS PRED
         FROM mtcars) AS prediction_output;

actual_class | predicted_0 | predicted_1 |        comment
-------------+-------------+-------------+---------------------------------------------
0            |          19 |           0 |
1            |           0 |          13 | Of 32 rows, 32 were used and 0 were ignored
(2 rows)
```

CROSS_VALIDATE

Performs k-fold cross validation on a learning algorithm using an input relation, and grid search for hyper parameters. The output is an average performance indicator of the selected algorithm. This function supports SVM classification, naive bayes, and logistic regression.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CROSS_VALIDATE ( 'algorithm', 'input-relation', 'response-column', 'predictor-columns'
      [ USING PARAMETERS
          [exclude_columns = 'excluded-columns']
        [, cv_model_name = 'model']
        [, cv_metrics = 'metrics']
        [, cv_fold_count = num-folds]
        [, cv_hyperparams = 'hyperparams']
        [, cv_prediction_cutoff = prediction-cutoff] ] )
```

Arguments

algorithm

Name of the algorithm training function, one of the following:

- LINEAR_REG
- LOGISTIC_REG
- NAIVE_BAYES
- SVM_CLASSIFIER
- SVM_REGRESSOR

input-relation

The table or view that contains data used for training and testing. If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

response-column

Name of the input column that contains the response.

predictor-columns

Comma-separated list of columns in the input relation that represent independent variables for the model, or asterisk (*) to select all columns. If you select all columns, the argument list for parameter exclude_columns must include *response-column* , and any columns that are invalid as predictor columns.

Parameters

exclude_columns

Comma-separated list of columns from *predictor-columns* to exclude from processing.

cv_model_name

The name of a model that lets you retrieve results of the cross validation process. If you omit this parameter, results are displayed but not saved. If you set this parameter to a model name, you can retrieve the results with summary functions GET_MODEL_ATTRIBUTE and GET_MODEL_SUMMARY

cv_metrics

The metrics used to assess the algorithm, specified either as a comma-separated list of metric names or in a JSON array . In both cases, you specify one or more of the following metric names:

- accuracy (default)
- error_rate
- TP : True positive, the number of cases of class 1 predicted as class 1
- FP : False positive, the number of cases of class 0 predicted as class 1
- TN : True negative, the number of cases of class 0 predicted as class 0
- FN : False negative, the number of cases of class 1 predicted as class 0
- TPR or recall : True positive rate, the correct predictions among class 1
- FPR : False positive rate, the wrong predictions among class 0
- TNR : True negative rate, the correct predictions among class 0
- FNR : False negative rate, the wrong predictions among class 1
- PPV or precision : The positive predictive value, the correct predictions among cases predicted as class 1
- NPV : Negative predictive value, the correct predictions among cases predicted as class 0
- MSE : Mean squared error

  $$\mathrm{MSE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2.$$
- MAE : Mean absolute error

  $$\mathrm{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} |y_i - \hat{y}_i| .$$
- rsquared : coefficient of determination

  $$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2}{\sum_{i=0}^{n_{\text{samples}}-1} (y_i - \bar{y})^2}$$
- explained_variance

  $$\texttt{explained\_variance}\ (y, \hat{y}) = 1 - \frac{Var\{y - \hat{y}\}}{Var\{y\}}$$
- fscore

  ```
  (1 + beta^2) * precison * recall / (beta^2 * precision + recall)
  ```

  beta equals 1 by default
- auc_roc : AUC of ROC using the specified number of bins, by default 100
- auc_prc : AUC of PRC using the specified number of bins, by default 100
- counts : Shortcut that resolves to four other metrics: TP , FP , TN , and FN
- count : Valid only in JSON syntax, counts the number of cases labeled by one class (*case-class-label*) but predicted as another class (*predicted-class-label*):

  cv_metrics='[{"count":[*case-class-label*, *predicted-class-label*]}]'

cv_fold_count

The number of folds to split the data.

Default: 5

cv_hyperparams

A JSON string that describes the combination of parameters for use in grid search of hyper parameters. The JSON string contains pairs of the hyper parameter name. The value of each hyper parameter can be specified as an array or sequence. For example:

```
{"param1":[value1,value2,...], "param2":{"first":first_value, "step":step_size, "count":number_of_values} }
```

Hyper parameter names and string values should be quoted using the JSON standard. These parameters are passed to the training function.

cv_prediction_cutoff

The cutoff threshold that is passed to the prediction stage of logistic regression, a FLOAT between 0 and 1, exclusive

Default: 0.5

Model attributes

call_string

The value of all input arguments that were specified at the time CROSS_VALIDATE was called.

run_average

The average across all folds of all metrics specified in parameter cv_metrics , if specified; otherwise, average accuracy.

fold_info

The number of rows in each fold:

- fold_id : The index of the fold.
- row_count : The number of rows held out for testing in the fold.

counters

All counters for the function, including:

- accepted_row_count : The total number of rows in the input_relation , minus the number of rejected rows.
- rejected_row_count : The number of rows of the input_relation that were skipped because they contained an invalid value.
- feature_count : The number of features input to the machine learning model.

run_details

Information about each run, where a run means training a single model, and then testing that model on the one held-out fold:

- fold_id : The index of the fold held out for testing.
- iteration_count : The number of iterations used in model training on non-held-out folds.
- accuracy : All metrics specified in parameter cv_metrics , or accuracy if cv_metrics is not provided.
- error_rate : All metrics specified in parameter cv_metrics , or accuracy if the parameter is omitted.

Privileges

Non-superusers:

- SELECT privileges on the input relation
- CREATE and USAGE privileges on the default schema where machine learning algorithms generate models. If cv_model_name is provided, the cross validation results are saved as a model in the same schema.

Specifying metrics in JSON

Parameter cv_metrics can specify metrics as an array of JSON objects , where each object specifies a metric name . For example, the following expression sets cv_metrics to two metrics specified as JSON objects, accuracy and error_rate :

```
cv_metrics='["accuracy", "error_rate"]'
```

In the next example, cv_metrics is set to two metrics, accuracy and TPR (true positive rate). Here, the TPR metric is specified as a JSON object that takes an array of two class label arguments, 2 and 3:

```
cv_metrics='[ "accuracy", {"TPR":[2,3] } ]'
```

Metrics specified as JSON objects can accept parameters. In the following example, the fscore metric specifies parameter beta , which is set to 0.5:

```
cv_metrics='[ {"fscore":{"beta":0.5} } ]'
```

Parameter support can be especially useful for certain metrics. For example, metrics auc_roc and auc_prc build a curve, and then compute the area under that curve. For ROC , the curve is formed by plotting metrics TPR against FPR ; for PRC , PPV (precision) against TPR (recall). The accuracy of such curves can be increased by setting parameter num_bins to a value greater than the default value of 100. For example, the following expression computes AUC for an ROC curve built with 1000 bins:

```
cv_metrics='[{"auc_roc":{"num_bins":1000}}]'
```

Using metrics with Multi-class classifier functions

All supported metrics are defined for binary classifier functions LOGISTIC_REG and SVM_CLASSIFIER . For multi-class classifier functions such as NAIVE_BAYES , these metrics can be calculated for each *one-versus-the-rest* binary classifier. Use arguments to request the metrics for each classifier. For example, if training data has integer class labels, you can set cv_metrics with the precision (PPV) metric as follows:

```
cv_metrics='[{"precision":[0,4]}]'
```

This setting specifies to return two columns with precision computed for two classifiers:

- Column 1: classifies 0 versus not 0
- Collumn 2: classifies 4 versus not 4

If you omit class label arguments, the class with index 1 is used. Instead of computing metrics for individual *one-versus-the-rest* classifiers, the average is computed in one of the following styles: macro , micro , or weighted (default). For example, the following cv_metrics setting returns the average weighted by class sizes:

```
cv_metrics='[{"precision":{"avg":"weighted"}}]'
```

AUC-type metrics can be similarly defined for multi-class classifiers. For example, the following cv_metrics setting computes the area under the ROC curve for each *one-versus-the-rest* classifier, and then returns the average weighted by class sizes.

```
cv_metrics='[{"auc_roc":{"avg":"weighted", "num_bins":1000}}]'
```

Examples

```
=> SELECT CROSS_VALIDATE('svm_classifier', 'mtcars', 'am', 'mpg'
    USING PARAMETERS cv_fold_count= 6,
                     cv_hyperparams='{"C":[1,5]}',
                     cv_model_name='cv_svm',
                     cv_metrics='accuracy, error_rate');
       CROSS_VALIDATE
----------------------------
 Finished

============
run_average
============
C  |accuracy      |error_rate
---+--------------+----------
1 | 0.75556       |  0.24444
5 | 0.78333       |  0.21667
(1 row)
```

ERROR_RATE

Using an input table, returns a table that calculates the rate of incorrect classifications and displays them as FLOAT values. ERROR_RATE returns a table with the following dimensions:

- Rows: Number of classes plus one row that contains the total error rate across classes
- Columns: 2

Syntax

```
ERROR_RATE ( targets, predictions [ USING PARAMETERS num_classes = num-classes ] ) OVER()
```

Arguments

targets

An input column that contains the true values of the response variable.

predictions

An input column that contains the predicted class labels.

Arguments *targets* and *predictions* must be set to input columns of the same data type, one of the following: INTEGER, BOOLEAN, or CHAR/VARCHAR. Depending on their data type, these columns identify classes as follows:

- INTEGER: Zero-based consecutive integers between 0 and (*num-classes* -1) inclusive, where *num-classes* is the number of classes. For example, given the following input column values— {0, 1, 2, 3, 4 }—Vertica assumes five classes.

  > **Note**
  > If input column values are not consecutive, Vertica interpolates the missing values. Thus, given the following input values— {0, 1, 3, 5, 6,} — Vertica assumes seven classes.

- BOOLEAN: Yes or No
- CHAR/VARCHAR: Class names. If the input columns are of type CHAR/VARCHAR columns, you must also set parameter num_classes to the number of classes.

> **Note**
> Vertica computes the number of classes as the union of values in both input columns. For example, given the following sets of values in the *targets* and *predictions* input columns, Vertica counts four classes:
>
> ```
> {'milk', 'soy milk', 'cream'}
> {'soy milk', 'almond milk'}
> ```

Parameters

num_classes

An integer > 1, specifies the number of classes to pass to the function.
You must set this parameter if the specified input columns are of type CHAR/VARCHAR. Otherwise, the function processes this parameter according to the column data types:

- INTEGER: By default set to 2, you must set this parameter correctly if the number of classes is any other value.
- BOOLEAN: By default set to 2, cannot be set to any other value.

Privileges

Non-superusers: model owner, or USAGE privileges on the model

Examples

This example shows how to execute the ERROR_RATE function on an input table named mtcars . The response variables appear in the column obs , while the prediction variables appear in the column pred . Because this example is a classification problem, all response variable values and prediction variable values are either 0 or 1, indicating binary classification.

In the table returned by the function, the first column displays the class id column. The second column displays the corresponding error rate for the class id. The third column indicates how many rows were successfully used by the function and whether any rows were ignored.

```
=> SELECT ERROR_RATE(obs::int, pred::int USING PARAMETERS num_classes=2) OVER()
   FROM (SELECT am AS obs, PREDICT_LOGISTIC_REG (mpg, cyl, disp, drat, wt, qsec, vs, gear, carb
              USING PARAMETERS model_name='myLogisticRegModel', type='response') AS pred
           FROM mtcars) AS prediction_output;
 class |     error_rate     |                  comment
-------+--------------------+-----------------------------------------------
     0 |                  0 |
     1 | 0.0769230797886848 |
       |            0.03125 | Of 32 rows, 32 were used and 0 were ignored
(3 rows)
```

LIFT_TABLE

Returns a table that compares the predictive quality of a machine learning model. This function is also known as a *lift chart* .

Syntax

```
LIFT_TABLE ( targets, probabilities
      [ USING PARAMETERS [num_bins = num-bins] [, main_class = class-name ] ] )
OVER()
```

Arguments

targets

An input column that contains the true values of the response variable, one of the following data types: INTEGER, BOOLEAN, or CHAR/VARCHAR.
Depending on the column data type, the function processes column data as follows:

- INTEGER: Uses the input column as containing the true value of the response variable.
- BOOLEAN: Resolves Yes to 1, 0 to No.
- CHAR/VARCHAR: Resolves the value specified by parameter main_class to 1, all other values to 0.

> **Note**
> If the input column is of data type INTEGER or BOOLEAN, the function ignores parameter main_class .

probabilities

A FLOAT input column that contains the predicted probability of response being the main class, set to 1 if *targets* is of type INTEGER.

Parameters

num_bins

An integer value that determines the number of decision boundaries. Decision boundaries are set at equally spaced intervals between 0 and 1, inclusive. The function computes the table at each *num-bin* + 1 point.
Default : 100

main_class

Used only if *targets* is of type CHAR/VARCHAR, specifies the class to associate with the *probabilities* argument.

Examples

Execute LIFT_TABLE on an input table mtcars .

```
=> SELECT LIFT_TABLE(obs::int, prob::float USING PARAMETERS num_bins=2) OVER()
   FROM (SELECT am AS obs, PREDICT_LOGISTIC_REG(mpg, cyl, disp, drat, wt, qsec, vs, gear, carb
                                        USING PARAMETERS model_name='myLogisticRegModel',
                                        type='probability') AS prob
         FROM mtcars) AS prediction_output;
 decision_boundary | positive_prediction_ratio |       lift       |                 comment
-------------------+---------------------------+------------------+-----------------------------------------------
                 1 |                         0 |              NaN |
               0.5 |                   0.40625 | 2.46153846153846 |
                 0 |                         1 |                1 | Of 32 rows, 32 were used and 0 were ignored
(3 rows)
```

The first column, decision_boundary , indicates the cut-off point for whether to classify a response as 0 or 1. For instance, for each row, if prob is greater than or equal to decision_boundary , the response is classified as 1. If prob is less than decision_boundary , the response is classified as 0.

The second column, positive_prediction_ratio , shows the percentage of samples in class 1 that the function classified correctly using the corresponding decision_boundary value.

For the third column, lift , the function divides the positive_prediction_ratio by the percentage of rows correctly or incorrectly classified as class 1.

MSE

Returns a table that displays the mean squared error of the prediction and response columns in a machine learning model.

Syntax

```
MSE ( targets, predictions ) OVER()
```

Arguments

targets

The model response variable, of type FLOAT.

predictions

A FLOAT input column that contains predicted values for the response variable.

Examples

Execute the MSE function on input table faithful_testing . The response variables appear in the column obs , while the prediction variables appear in the column prediction .

```
=> SELECT MSE(obs, prediction) OVER()
  FROM (SELECT eruptions AS obs,
               PREDICT_LINEAR_REG (waiting USING PARAMETERS model_name='myLinearRegModel') AS prediction
        FROM faithful_testing) AS prediction_output;
       mse       |                  Comments
-----------------+----------------------------------------------
 0.252925741352641 | Of 110 rows, 110 were used and 0 were ignored
(1 row)
```

PRC

Returns a table that displays the points on a receiver precision recall (PR) curve.

Syntax

```
PRC ( targets, probabilities
    [ USING PARAMETERS
        [num_bins = num-bins]
        [, f1_score = return-score ]
        [, main_class = class-name ] )
OVER()
```

Arguments

targets

An input column that contains the true values of the response variable, one of the following data types: INTEGER, BOOLEAN, or CHAR/VARCHAR. Depending on the column data type, the function processes column data as follows:

- INTEGER: Uses the input column as containing the true value of the response variable.
- BOOLEAN: Resolves Yes to 1, 0 to No.
- CHAR/VARCHAR: Resolves the value specified by parameter main_class to 1, all other values to 0.

> **Note**
> If the input column is of data type INTEGER or BOOLEAN, the function ignores parameter main_class .

probabilities

A FLOAT input column that contains the predicted probability of response being the main class, set to 1 if *targets* is of type INTEGER.

Parameters

num_bins

An integer value that determines the number of decision boundaries. Decision boundaries are set at equally spaced intervals between 0 and 1, inclusive. The function computes the table at each *num-bin* + 1 point.

Default : 100

f1_score

A Boolean that specifies whether to return a column that contains the f1 score—the harmonic average of the precision and recall measures, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.

Default: false

main_class

Used only if *targets* is of type CHAR/VARCHAR, specifies the class to associate with the *probabilities* argument.

Examples

Execute the PRC function on an input table named mtcars . The response variables appear in the column obs , while the prediction variables appear in column pred .

```
=> SELECT PRC(obs::int, prob::float USING PARAMETERS num_bins=2, f1_score=true) OVER()
   FROM (SELECT am AS obs,
                PREDICT_LOGISTIC_REG (mpg, cyl, disp, drat, wt, qsec, vs, gear, carb
                    USING PARAMETERS model_name='myLogisticRegModel',
                                     type='probability') AS prob
         FROM mtcars) AS prediction_output;
decision_boundary | recall | precision |     f1_score      |     comment
------------------+--------+-----------+-------------------+-----------------------------------------
0                 |      1 |   0.40625 | 0.577777777777778 |
0.5               |      1 |         1 |                 1 | Of 32 rows, 32 were used and 0 were ignored
(2 rows)
```

The first column, decision_boundary , indicates the cut-off point for whether to classify a response as 0 or 1. For example, in each row, if the probability is equal to or greater than decision_boundary , the response is classified as 1. If the probability is less than decision_boundary , the response is classified as 0.

READ_TREE

Reads the contents of trees within the random forest or XGBoost model.

Syntax

```
READ_TREE ( USING PARAMETERS model_name = 'model-name' [, tree_id = tree-id] [, format = 'format'] )
```

Parameters

model_name

Identifies the model that is stored as a result of training, where *model-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

tree_id

The tree identifier, an integer between 0 and *n* -1, where *n* is the number of trees in the random forest or XGBoost model. If you omit this parameter, all trees are returned.

format

Output format of the returned tree, one of the following:

- tabular : Returns a table with the twelve output columns.
- graphviz : Returns DOT language source that can be passed to a graphviz tool and render a graphic visualization of the tree.

Privileges

Non-superusers: USAGE privileges on the model

Examples

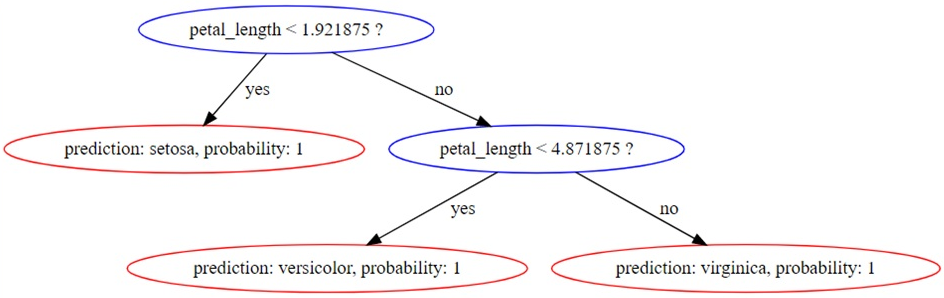
Get tabular output from READ_TREE for a random forest model:

```
=> SELECT READ_TREE ( USING PARAMETERS model_name='myRFModel', tree_id=1 ,
format= 'tabular') LIMIT 2;
-[ RECORD 1 ]-------------+-------------------
tree_id                   | 1
node_id                   | 1
node_depth                | 0
is_leaf                   | f
is_categorical_split      | f
split_predictor           | petal_length
split_value               | 1.921875
weighted_information_gain | 0.111242236024845
left_child_id             | 2
right_child_id            | 3
prediction                |
probability/variance      |

-[ RECORD 2 ]-------------+-------------------
tree_id                   | 1
node_id                   | 2
node_depth                | 1
is_leaf                   | t
is_categorical_split      |
split_predictor           |
split_value               |
weighted_information_gain |
left_child_id             |
right_child_id            |
prediction                | setosa
probability/variance      | 1
```

Get graphviz -formatted output from READ_TREE:

```
=> SELECT READ_TREE ( USING PARAMETERS model_name='myRFModel', tree_id=1 ,
format= 'graphviz')LIMIT 1;

-[ RECORD 1 ]+-----------------------------------------------------------------
-------------------------------------------------------------------------------
tree_id      | 1
tree_digraph | digraph Tree{
1 [label="petal_length < 1.921875 ?", color="blue"];
1 -> 2 [label="yes", color="black"];
1 -> 3 [label="no", color="black"];
2 [label="prediction: setosa, probability: 1", color="red"];
3 [label="petal_length < 4.871875 ?", color="blue"];
3 -> 6 [label="yes", color="black"];
3 -> 7 [label="no", color="black"];
6 [label="prediction: versicolor, probability: 1", color="red"];
7 [label="prediction: virginica, probability: 1", color="red"];
}
```

This renders as follows:



See also

- RF_CLASSIFIER
- RF_REGRESSOR
- XGB_CLASSIFIER
- XGB_REGRESSOR

RF_PREDICTOR_IMPORTANCE

Measures the importance of the predictors in a random forest model using the Mean Decrease Impurity (MDI) approach. The importance vector is normalized to sum to 1.

Syntax

```
RF_PREDICTOR_IMPORTANCE ( USING PARAMETERS model_name = 'model-name' [, tree_id = tree-id] )
```

Parameters

model_name

Identifies the model that is stored as a result of the training, where *model-name* must be of type rf_classifier or rf_regressor .

tree_id

Identifies the tree to process, an integer between 0 and *n* -1, where *n* is the number of trees in the forest. If you omit this parameter, the function uses all trees to measure importance values.

Privileges

Non-superusers: USAGE privileges on the model

Examples

This example shows how you can use the RF_PREDICTOR_IMPORTANCE function.

```
=> SELECT RF_PREDICTOR_IMPORTANCE ( USING PARAMETERS model_name = 'myRFModel');
 predictor_index | predictor_name | importance_value
-----------------+----------------+--------------------
               0 | sepal.length   | 0.106763318092655
               1 | sepal.width    | 0.0279536658041994
               2 | petal.length   | 0.499198722346586
               3 | petal.width    | 0.366084293756561
(4 rows)
```

See also

- RF_CLASSIFIER
- RF_REGRESSOR

ROC

Returns a table that displays the points on a receiver operating characteristic curve. The ROC function tells you the accuracy of a classification model as you raise the discrimination threshold for the model.

Syntax

```
ROC ( targets, probabilities
      [ USING PARAMETERS
            [num_bins = num-bins]
            [, AUC = output]
            [, main_class = class-name ] ) ] )
OVER()
```

Arguments

targets

An input column that contains the true values of the response variable, one of the following data types: INTEGER, BOOLEAN, or CHAR/VARCHAR. Depending on the column data type, the function processes column data as follows:

- INTEGER: Uses the input column as containing the true value of the response variable.
- BOOLEAN: Resolves Yes to 1, 0 to No.
- CHAR/VARCHAR: Resolves the value specified by parameter main_class to 1, all other values to 0.

> **Note**
> If the input column is of data type INTEGER or BOOLEAN, the function ignores parameter main_class .

probabilities

A FLOAT input column that contains the predicted probability of response being the main class, set to 1 if *targets* is of type INTEGER.

Parameters

num_bins

An integer value that determines the number of decision boundaries. Decision boundaries are set at equally spaced intervals between 0 and 1, inclusive. The function computes the table at each *num-bin* + 1 point.
Default : 100

Greater values result in more precise approximations of the AUC.

AUC

A Boolean value that specifies whether to output the area under the curve (AUC) value.

Default: True

main_class

Used only if *targets* is of type CHAR/VARCHAR, specifies the class to associate with the *probabilities* argument.

Examples

Execute ROC on input table mtcars . Observed class labels are in column obs , predicted class labels are in column prob :

```
=> SELECT ROC(obs::int, prob::float USING PARAMETERS num_bins=5, AUC = True) OVER()
   FROM (SELECT am AS obs,
        PREDICT_LOGISTIC_REG (mpg, cyl, disp, drat, wt, qsec, vs, gear, carb
            USING PARAMETERS
              model_name='myLogisticRegModel', type='probability') AS prob
  FROM mtcars) AS prediction_output;
 decision_boundary | false_positive_rate | true_positive_rate | AUC |comment
-------------------+---------------------+--------------------+-----+---------------------------------
0                  |                   1 |                  1 |     |
0.5                |                   0 |                  1 |     |
1                  |                   0 |                  0 |   1 | Of 32 rows,32 were used and 0 were ignoreded
(3 rows)
```

The function returns a table with the following results:

- decision_boundary indicates the cut-off point for whether to classify a response as 0 or 1. In each row, if prob is equal to or greater than decision_boundary , the response is classified as 1. If prob is less than decision_boundary , the response is classified as 0.
- false_positive_rate shows the percentage of false positives (when 0 is classified as 1) in the corresponding decision_boundary .
- true_positive_rate shows the percentage of rows that were classified as 1 and also belong to class 1.

RSQUARED

Returns a table with the R-squared value of the predictions in a regression model.

Syntax

```
RSQUARED ( targets, predictions ) OVER()
```

Important
The OVER() clause must be empty.

Arguments

targets

A FLOAT response variable for the model.

predictions

A FLOAT input column that contains the predicted values for the response variable.

Examples

This example shows how to execute the RSQUARED function on an input table named faithful_testing . The observed values of the response variable appear in the column, obs , while the predicted values of the response variable appear in the column, pred .

```
=> SELECT RSQUARED(obs, prediction) OVER()
    FROM (SELECT eruptions AS obs,
                 PREDICT_LINEAR_REG (waiting
                                     USING PARAMETERS model_name='myLinearRegModel') AS prediction
          FROM faithful_testing) AS prediction_output;
       rsq        |                  comment
------------------+-----------------------------------------------
 0.801392981147911 | Of 110 rows, 110 were used and 0 were ignored
(1 row)
```

XGB_PREDICTOR_IMPORTANCE

Measures the importance of the predictors in an XGBoost model. The function outputs three measures of importance for each predictor:

- frequency : relative number of times the model uses a predictor to split the data.
- total_gain : relative contribution of a predictor to the model based on the total information gain across a predictor's splits. A higher value means more predictive importance.
- avg_gain : relative contribution of a predictor to the model based on the average information gain across a predictor's splits.

The sum of each importance measure is normalized to one across all predictors.

Syntax

```
XGB_PREDICTOR_IMPORTANCE ( USING PARAMETERS param=value[,...] )
```

Parameters

model_name

Name of the model, which must be of type xgb_classifier or xgb_regressor .

tree_id

Integer in the range [0, *n* -1], where *n* is the number of trees in model_name , that specifies the tree to process. If you omit this parameter, the function uses all trees in the model to measure predictor importance values.

Privileges

Non-superusers: USAGE privileges on the model

Examples

The following example measures the importance of the predictors in the model 'xgb_iris', an XGBoost classifier model, across all trees:

```
=> SELECT XGB_PREDICTOR_IMPORTANCE( USING PARAMETERS model_name = 'xgb_iris' );
 predictor_index | predictor_name |     frequency     |     total_gain     |      avg_gain
-----------------+----------------+-------------------+--------------------+--------------------
               0 | sepal_length   |  0.15384615957737 |    0.0183021749937 | 0.0370849960701401
               1 | sepal_width    | 0.215384617447853 | 0.0154729501420881 | 0.0223944615251752
               2 | petal_length   | 0.369230777025223 |  0.607349886817728 |  0.512770753876444
               3 | petal_width    | 0.261538475751877 |  0.358874988046484 |  0.427749788528241
(4 rows)
```

To sort the predictors by importance values, you can use a nested query with an ORDER BY clause. The following sorts the model predictors by descending avg_gain :

```
=> SELECT * FROM (SELECT XGB_PREDICTOR_IMPORTANCE( USING PARAMETERS model_name = 'xgb_iris' )) AS importances ORDER BY
avg_gain DESC;
 predictor_index | predictor_name |     frequency     |     total_gain     |      avg_gain
-----------------+----------------+-------------------+--------------------+--------------------
               2 | petal_length   | 0.369230777025223 |  0.607349886817728 |  0.512770753876444
               3 | petal_width    | 0.261538475751877 |  0.358874988046484 |  0.427749788528241
               0 | sepal_length   |  0.15384615957737 |     0.0183021749937 | 0.0370849960701401
               1 | sepal_width    | 0.215384617447853 | 0.0154729501420881 | 0.0223944615251752
(4 rows)
```

See also

- XGB_CLASSIFIER
- XGB_REGRESSOR

Model management

Vertica provides several functions for managing models.

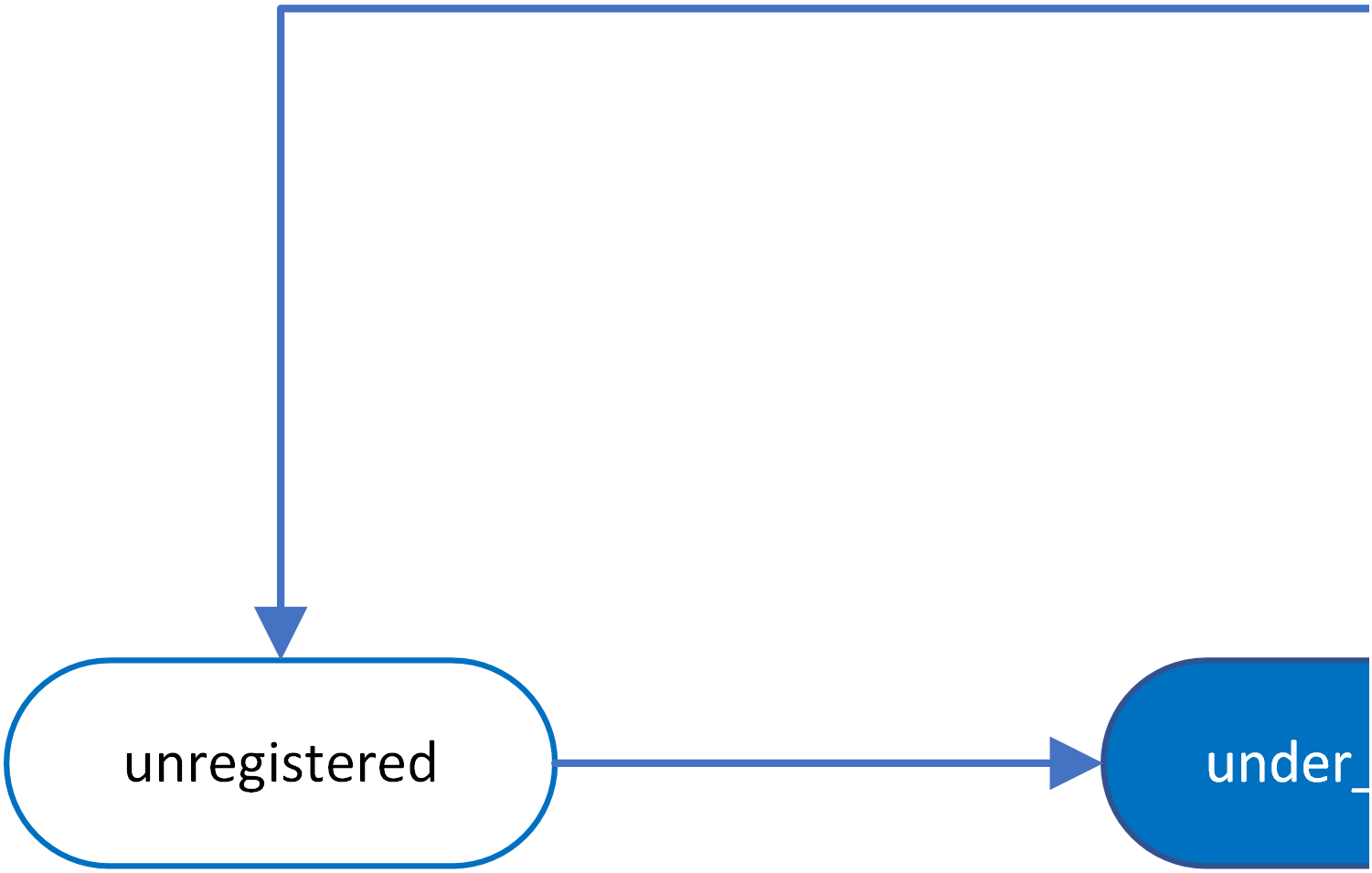
In this section

- CHANGE_MODEL_STATUS
- EXPORT_MODELS
- GET_MODEL_ATTRIBUTE
- GET_MODEL_SUMMARY
- IMPORT_MODELS
- REGISTER_MODEL
- UPGRADE_MODEL

CHANGE_MODEL_STATUS

Changes the status of a registered model. Only dbadmin and users with the MLSUPERVISOR role can call this function.

The following diagram depicts the valid status transitions:



This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Stable

Syntax

```
CHANGE_MODEL_STATUS( 'registered_name', registered_version, 'new_status' )
```

Arguments

registered_name

Identifies the abstract name to which the model is registered. This *registered_name* can represent a group of models for a higher-level

application, where each model in the group has a unique version number.

registered_version

Unique version number of the model under the specified *registered_name* .

If there is no registered model with the given *registered_name* and *registered_version* , the function errors.

new_status

New status of the registered model. Must be one of the following strings and adhere to the valid status transitions depicted in the above diagram :

- under_review : Status assigned to newly registered models.
- staging : Model is targeted for A/B testing against the model currently in production.
- production : Model is in production for its specified application. Only one model can be in production for a given *registered_name* at one time.
- archived : Status of models that were previously in production. Archived models can be returned to production at any time.
- declined : Model is no longer in consideration for production.
- unregistered : Model is removed from the versioning environment. The model does not appear in the REGISTERED_MODELS system table.

If you change the status of a model to 'production' and there is already a model in production under the given *registered_name* , the status of the model in production is set to 'archived' and the status of the new model is set to 'production'.

Privileges

One of the following:

- Superuser
- MLSUPERVISOR

Examples

In the following example, the linear_reg_spark1 model, which is uniquely identified by the *registered_name* 'linear_reg_app' and the *registered_version* of two, is set to 'production' status:

```
=> SELECT * FROM REGISTERED_MODELS;
  registered_name | registered_version |    status    |        registered_time        |      model_id     | schema_name |    model_name     |      model_type       |
category
------------------+--------------------+--------------+-------------------------------+-------------------+-------------+-------------------+-----------------------+----------------
 linear_reg_app   |                  2 | STAGING      | 2023-01-29 05:49:00.082166-04 | 45035996273714020 | public      | linear_reg_spark1 |
PMML_REGRESSION_MODEL | PMML
 linear_reg_app   |                  1 | PRODUCTION   | 2023-01-24 09:19:04.553102-05 | 45035996273850350 | public      | native_linear_reg |
LINEAR_REGRESSION     | VERTICA_MODELS
 logistic_reg_app |                  1 | DECLINED     | 2023-01-11 02:47:25.990626-02 | 45035996273853740 | public      | log_reg_bfgs      |
LOGISTIC_REGRESSION   | VERTICA_MODELS
(3 rows)

=> SELECT CHANGE_MODEL_STATUS('linear_reg_app', 2, 'production');
                    CHANGE_MODEL_STATUS
-------------------------------------------------------------------------------
The status of model [linear_reg_app] - version [2] is changed to [production]
(1 row)
```

You can query the REGISTERED_MODELS system table to confirm that the linear_reg_spark1 model is now in 'production' and the native_linear_reg model, which was currently in 'production', is moved to 'archived':

```
=> SELECT * FROM REGISTERED_MODELS;
  registered_name | registered_version |    status    |        registered_time        |      model_id     | schema_name |    model_name     |      model_type       |
category
------------------+--------------------+--------------+-------------------------------+-------------------+-------------+-------------------+-----------------------+----------------
 linear_reg_app   |                  2 | PRODUCTION   | 2023-01-29 05:49:00.082166-04 | 45035996273714020 | public      | linear_reg_spark1 |
PMML_REGRESSION_MODEL | PMML
 linear_reg_app   |                  1 | ARCHIVED     | 2023-01-24 09:19:04.553102-05 | 45035996273850350 | public      | native_linear_reg |
LINEAR_REGRESSION     | VERTICA_MODELS
 logistic_reg_app |                  1 | DECLINED     | 2023-01-11 02:47:25.990626-02 | 45035996273853740 | public      | log_reg_bfgs      |
LOGISTIC_REGRESSION   | VERTICA_MODELS
(2 rows)
```

If you change a model's status to 'unregistered', the model is removed from the model versioning environment and no longer appears in the REGISTERED_MODELS system table:

```
=> SELECT CHANGE_MODEL_STATUS('logistic_reg_app', 1, 'unregistered');
                        CHANGE_MODEL_STATUS
-------------------------------------------------------------------------------
The status of model [logistic_reg_app] - version [1] is changed to [unregistered]
(1 row)

=> SELECT * FROM REGISTERED_MODELS;
 registered_name | registered_version |    status    |       registered_time        |     model_id     | schema_name |    model_name    |      model_type      |
category
-----------------+--------------------+--------------+------------------------------+------------------+-------------+------------------+----------------------+----------------
 linear_reg_app  |                  2 | STAGING      | 2023-01-29 05:49:00.082166-04 | 45035996273714020 | public      | linear_reg_spark1 |
PMML_REGRESSION_MODEL | PMML
 linear_reg_app  |                  1 | PRODUCTION   | 2023-01-24 09:19:04.553102-05 | 45035996273850350 | public      | native_linear_reg |
LINEAR_REGRESSION     | VERTICA_MODELS
(2 rows)
```

See also

- REGISTER_MODEL
- Model versioning

EXPORT_MODELS

Exports machine learning models. Vertica supports three model formats:

- Native Vertica (VERTICA_MODELS)
- PMML
- TensorFlow

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
EXPORT_MODELS ( 'output-dir', 'export-target' [ USING PARAMETERS category = 'model-category' ] )
```

Arguments

output-dir

Absolute path of an output directory to store the exported models.

export-target

Specifies which models to export as follows:

```
[schema.]{ model-name | * }
```

where *schema* optionally specifies to export models from the specified schema. If omitted, EXPORT_MODELS uses the default schema. Supply * (asterisk) to export all models from the schema.

Parameters

category

The category of models to export, one of the following:

- VERTICA_MODELS
- PMML
- TENSORFLOW

EXPORT_MODELS exports models of the specified category according to the scope of the export operation—that is, whether it applies to a single model, or to all models within a schema. See Export Scope and Category Processing below.
Exported Files below describes the files that EXPORT_MODELS exports for each category.

If you omit this parameter, EXPORT_MODELS exports the model, or models in the specified schema, according to their model type.

Privileges

Superuser

Export scope and category processing

EXPORT_MODELS executes according to the following parameter settings:

- Scope of the export operation: single model, or all models within a given schema
- Category specified or omitted

The following table shows how these two parameters control the export process:

Export scope	**If category specified...**	**If category omitted...**
Single model	Convert the model to the specified category, provided the model and category are compatible ; otherwise, return with a mismatch error.	Export the model according to model type.
All models in schema	Export only models that are compatible with the specified category and issue mismatch warnings on all other models in the schema.	Export all models in the schema according to model type.

Exported files

EXPORT_MODELS exports the following files for each model category:

Model category	**Exported files**
VERTICA_MODELS	• Multiple binary files (exact number dependent on model type) • metadata.json : Metadata file with model information —model name, category, type, Vertica version on export. • crc.json : Used on import to validate other files of this model.
PMML	• XML file with the same name as the model and complying with PMML standard. • metadata.json : Metadata file with model information —model name, category, type, Vertica version on export. • crc.json : Used on import to validate other files of this model.
TENSORFLOW	• *model-name* .pb : Contains the TensorFlow model, saved in 'frozen graph' format. • metadata.json : Metadata file with model information —model name, category, type, Vertica version on export. • tf_model_desc.json : Summary model description. • model.json : Verbose model description. • crc.json : Used on import to validate other files of this model.

Categories and compatible models

If EXPORT_MODELS specifies a single model and also sets the category parameter, the function succeeds if the model type and category are compatible; otherwise, it returns with an error:

Model type	**Compatible categories**
PMML	PMML
TensorFlow	TENSORFLOW
VERTICA_MODELS	PMML VERTICA_MODELS

If EXPORT_MODELS specifies to export all models from a schema and sets a category, it issues a warning message on each model that is incompatible with that category. The function then continues to process remaining models in that schema.

EXPORT_MODELS logs all errors and warnings in *output-dir* /export_log.json .

Examples

Export models without changing their category:

- Export model myschema.mykmeansmodel without changing its category:

```
=> SELECT EXPORT_MODELS ('/home/dbadmin', 'myschema.mykmeansmodel');
EXPORT_MODELS
----------------
Success
(1 row)
```

- Export all models in schema myschema without changing their categories:

```
=> SELECT EXPORT_MODELS ('/home/dbadmin', 'myschema.*');
EXPORT_MODELS
----------------
Success
(1 row)
```

Export models that are compatible with the specified category:

> **Note**
> When you import a model of category VERTICA_MODELS trained in a different version of Vertica, Vertica automatically upgrades the model version to match that of the database. If this fails, you must run UPGRADE_MODEL .
>
> If both methods fail, the model cannot be used for in-database scoring and cannot be exported as a PMML model.

- The category is set to PMML. Models of type PMML and VERTICA_MODELS are compatible with the PMML category, so the export operation succeeds if my_keans is of either type:

```
=> SELECT EXPORT_MODELS ('/tmp/', 'my_kmeans' USING PARAMETERS category='PMML');
```

- The category is set to VERTICA_MODELS. Only models of type VERTICA_MODELS are compatible with the VERTICA_MODELS category, so the export operation succeeds only if my_keans is of that type:

```
=> SELECT EXPORT_MODELS ('/tmp/', 'public.my_kmeans' USING PARAMETERS category='VERTICA_MODELS');
```

- The category is set to TENSORFLOW. Only models of type TensorFlow are compatible with the TENSORFLOW category, so the model tf_mnist_keras must be of type TensorFlow:

```
=> SELECT EXPORT_MODELS ('/tmp/', 'tf_mnist_keras', USING PARAMETERS category='TENSORFLOW');
export_models
---------------
Success
(1 row)
```

After exporting the TensorFlow model tf_mnist_keras, list the exported files:

```
$ ls tf_mnist_keras/
crc.json  metadata.json  mnist_keras.pb  model.json  tf_model_desc.json
```

See also

IMPORT_MODELS

GET_MODEL_ATTRIBUTE

Extracts either a specific attribute from a model or all attributes from a model. Use this function to view a list of attributes and row counts or view detailed information about a single attribute. The output of GET_MODEL_ATTRIBUTE is a table format where users can select particular columns or rows.

Syntax

```
GET_MODEL_ATTRIBUTE ( USING PARAMETERS model_name = 'model-name' [, attr_name = 'attribute' ] )
```

Parameters

model_name

Name of the model (case-insensitive).

attr_name

Name of the model attribute to extract. If omitted, the function shows all available attributes. Attribute names are case-sensitive.

Privileges

Non-superusers: model owner, or USAGE privileges on the model

Examples

This example returns a summary of all model attributes.

```
=> SELECT GET_MODEL_ATTRIBUTE ( USING PARAMETERS model_name='myLinearRegModel');
attr_name          |               attr_fields                | #_of_rows
-------------------+------------------------------------------+-----------
details            | predictor, coefficient, std_err, t_value, p_value |         2
regularization     | type, lambda                             |         1
iteration_count    | iteration_count                          |         1
rejected_row_count | rejected_row_count                       |         1
accepted_row_count | accepted_row_count                       |         1
call_string        | call_string                              |         1
(6 rows)
```

This example extracts the details attribute from the myLinearRegModel model.

```
=> SELECT GET_MODEL_ATTRIBUTE ( USING PARAMETERS model_name='myLinearRegModel', attr_name='details');
coeffNames |       coeff        |       stdErr        |      zValue       |        pValue
-----------+--------------------+---------------------+-------------------+-----------------------
Intercept  |  -1.87401598641074 |   0.160143331525544 | -11.7021169008952 |   7.3592939615234e-26
waiting    | 0.0756279479518627 | 0.00221854185633525 |  34.0890336307608 | 8.13028381124448e-100
(2 rows)
```

GET_MODEL_SUMMARY

Returns summary information of a model.

Syntax

```
GET_MODEL_SUMMARY ( USING PARAMETERS model_name = 'model-name' )
```

Parameters

model_name

Name of the model (case-insensitive).

Privileges

Non-superusers: model owner, or USAGE privileges on the model

Examples

This example shows how you can view the summary of a linear regression model.

```
=> SELECT GET_MODEL_SUMMARY( USING PARAMETERS model_name='myLinearRegModel');

--------------------------------------------------------------------------------
=======
details
=======
predictor|coefficient|std_err |t_value |p_value
---------+-----------+--------+--------+--------
Intercept| -2.06795  | 0.21063|-9.81782| 0.00000
waiting  |  0.07876  | 0.00292|26.96925| 0.00000

==============
regularization
==============
type| lambda
----+--------
none| 1.00000

===========
call_string
===========
linear_reg('public.linear_reg_faithful', 'faithful_training', '"eruptions"', 'waiting'
USING PARAMETERS optimizer='bfgs', epsilon=1e-06, max_iterations=100,
regularization='none', lambda=1)

===============
Additional Info
===============
Name             |Value
-----------------+-----
iteration_count   | 3
rejected_row_count|  0
accepted_row_count| 162
(1 row)
```

IMPORT_MODELS

Imports models into Vertica, either Vertica models that were exported with EXPORT_MODELS , or models in Predictive Model Markup Language (PMML) or TensorFlow format. You can use this function to move models between Vertica clusters, or to import PMML and TensorFlow models trained elsewhere.

Other Vertica model management operations such as GET_MODEL_SUMMARY and GET_MODEL_ATTRIBUTE support imported models.

> **Caution**
> Changing the exported model files causes the import functionality to fail on attempted re-import.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
IMPORT_MODELS ( 'source'
      [ USING PARAMETERS [ new_schema = 'schema-name' ] [, category = 'model-category' ] ] )
```

Arguments

source

The absolute path of the location from which to import models, one of the following:

- The directory of a single model:

  ```
  path/model-directory
  ```

- The parent directory of multiple model directories:
  ```
  parent-dir-path/*
  ```

Parameters

new_schema

An existing schema where the machine learning models are imported. If omitted, models are imported to the default schema.

IMPORT_MODELS extracts the name of the imported model from its metadata.json file, if it exists. Otherwise, the function uses the name of the model directory.

category

Specifies the category of the model to import, one of the following:

- VERTICA_MODELS
- PMML
- TENSORFLOW

This parameter is required if the model directory has no metadata.json file. IMPORT_MODELS returns with an error if one of the following cases is true:

- No category is specified and the model directory has no metadata.json .
- The specified category does not match the model type.

> **Note**
> If the category is TENSORFLOW, IMPORT_MODELS only imports the following files from the model directory:
>
> - *model-name* .pb
> - *model-name* .json
> - *model-name* .pbtxt (optional)

Privileges

Superuser

Requirements and restrictions

The following requirements and restrictions apply:

- If you export a model, then import it again, the export and import model directory names must match. If naming conflicts occur, import the model to a different schema by using the new_schema parameter, and then rename the model.
- The machine learning configuration parameter MaxModelSizeKB sets the maximum size of a model that can be imported into Vertica.
- Some PMML features and attributes are not currently supported. See PMML features and attributes for details.
- If you import a PMML model with both metadata.json and crc.json files, the CRC file must contain the metadata file's CRC value. Otherwise, the import operation returns with an error.

Examples

Import models into the specified schema:

In both examples no model category is specified, so IMPORT_MODEL uses the model's metadata.json file to determine its category:

- Import a single model mykmeansmodel into the newschema schema:
  ```
  => SELECT IMPORT_MODELS ('/home/dbadmin/myschema/mykmeansmodel' USING PARAMETERS new_schema='newschema')
  IMPORT_MODELS
  ----------------
  Success
  (1 row)
  ```
- Import all models in the myschema directory into the newschema schema:
  ```
  => SELECT IMPORT_MODELS ('/home/dbadmin/myschema/*' USING PARAMETERS new_schema='newschema')
  IMPORT_MODELS
  ----------------
  Success
  (1 row)
  ```

Specify the category of models to import:

In the first two examples, IMPORT_MODELS returns with success only if the specified model and category match; otherwise, it returns an error:

- Import kmeans_pmml as a PMML model:

```
SELECT IMPORT_MODELS ('/root/user/kmeans_pmml' USING PARAMETERS category='PMML')
 import_models
---------------
 Success
(1 row)
```

- Import tf_mnist_estimator as a TensorFlow model:

```
=> SELECT IMPORT_MODELS ( '/path/tf_models/tf_mnist_estimator' USING PARAMETERS category='TENSORFLOW');
 import_models
---------------
 Success
(1 row)
```

- Import all TensorFlow models from the specified directory:

```
=> SELECT IMPORT_MODELS ( '/path/tf_models/*' USING PARAMETERS category='TENSORFLOW');
 import_models
---------------
 Success
(1 row)
```

See also

EXPORT_MODELS

REGISTER_MODEL

Registers a trained model and adds it to Model versioning environment with a status of 'under_review'. The model must be registered by the owner of the model or dbadmin.

After a model is registered, the model owner is automatically changed to Superuser and the previous owner is given USAGE privileges. Users with the MLSUPERVISOR role or dbamin can call the CHANGE_MODEL_STATUS function to alter the status of registered models.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Stable

Syntax

```
REGISTER_MODEL( 'model_name', 'registered_name' )
```

Arguments

model_name

Identifies the model to register. If the model has already been registered, the function throws an error.

registered_name

Identifies an abstract name to which the model is registered. This *registered_name* can represent a group of models for a higher-level application, where each model in the group has a unique version number.

If a model is the first to be registered to a given *registered_name* , the model is assigned a *registered_version* of one. Otherwise, newly registered models are assigned an incremented *registered_version* of n + 1, where n is the number of models already registered to the given *registered_name* . Each registered model can be uniquely identified by the combination of *registered_name* and *registered_version* .

Privileges

Non-superusers: model owner

Examples

In the following example, the model log_reg_bfgs is registered to the logistic_reg_app application:

```
=> SELECT REGISTER_MODEL('log_reg_bfgs', 'logistic_reg_app');
                    REGISTER_MODEL
-----------------------------------------------------------------------
Model [log_reg_bfgs] is registered as [logistic_reg_app], version [1]
(1 row)
```

You can query the REGISTERED_MODELS system table to view details about the newly registered model:

```
=> SELECT * FROM REGISTERED_MODELS;
 registered_name | registered_version |    status    |        registered_time        |     model_id     | schema_name |    model_name    |      model_type      |
category
-----------------+--------------------+--------------+-------------------------------+------------------+-------------+------------------+----------------------+----------------
 logistic_reg_app |                  1 | UNDER_REVIEW | 2023-01-22 09:49:25.990626-02 | 45035996273853740 | public      | log_reg_bfgs     |
LOGISTIC_REGRESSION   | VERTICA_MODELS
(1 row)
```

See also

- CHANGE_MODEL_STATUS
- Model versioning

UPGRADE_MODEL

Upgrades a model from a previous Vertica version. Vertica automatically runs this function during a database upgrade and if you run the IMPORT_MODELS function. Manually call this function to upgrade models after a backup or restore.

If UPGRADE_MODEL fails to upgrade the model and the model is of category VERTICA_MODELS, it cannot be used for in-database scoring and cannot be exported as a PMML model.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
UPGRADE_MODEL ( [ USING PARAMETERS [model_name = 'model-name'] ] )
```

Parameters

model_name

Name of the model to upgrade. If you omit this parameter, Vertica upgrades all models on which you have privileges.

Privileges

Non-superuser: Upgrades only models that the user owns.

Examples

Upgrade model myLogisticRegModel :

```
=> SELECT UPGRADE_MODEL( USING PARAMETERS model_name = 'myLogisticRegModel');
      UPGRADE_MODEL
----------------------------
 1 model(s) upgrade

(1 row)
```

Upgrade all models that the user owns:

```
=> SELECT UPGRADE_MODEL();
      UPGRADE_MODEL
----------------------------
 20 model(s) upgrade

(1 row)
```

Transformation functions

The machine learning API includes a set of UDx functions that transform the columns of each input row to one or more corresponding output columns. These transformations follow rules that are defined in models that were created earlier. For example, APPLY_SVD uses an SVD model to transform input data.

Unless otherwise indicated, these functions require the following privileges for non-superusers:

- USAGE privileges on the model

- SELECT privileges on the input relation

In general, given an invalid input row, the return value for these functions is NULL.

In this section

- APPLY_BISECTING_KMEANS
- APPLY_IFOREST
- APPLY_INVERSE_PCA
- APPLY_INVERSE_SVD
- APPLY_KMEANS
- APPLY_KPROTOTYPES
- APPLY_NORMALIZE
- APPLY_ONE_HOT_ENCODER
- APPLY_PCA
- APPLY_SVD
- PREDICT_ARIMA
- PREDICT_AUTOREGRESSOR
- PREDICT_LINEAR_REG
- PREDICT_LOGISTIC_REG
- PREDICT_MOVING_AVERAGE
- PREDICT_NAIVE_BAYES
- PREDICT_NAIVE_BAYES_CLASSES
- PREDICT_PMML
- PREDICT_POISSON_REG
- PREDICT_RF_CLASSIFIER
- PREDICT_RF_CLASSIFIER_CLASSES
- PREDICT_RF_REGRESSOR
- PREDICT_SVM_CLASSIFIER
- PREDICT_SVM_REGRESSOR
- PREDICT_TENSORFLOW
- PREDICT_XGB_CLASSIFIER
- PREDICT_XGB_CLASSIFIER_CLASSES
- PREDICT_XGB_REGRESSOR
- REVERSE_NORMALIZE

APPLY_BISECTING_KMEANS

Applies a trained bisecting k-means model to an input relation, and assigns each new data point to the closest matching cluster in the trained model.

> **Note**
> If the input relation is defined in Hive, use SYNC_WITH_HCATALOG_SCHEMA to sync the hcatalog schema, and then run the machine learning function.

Syntax

```
SELECT APPLY_BISECTING_KMEANS( 'input-columns'
    USING PARAMETERS model_name = 'model-name'
      [, num_clusters = 'num-clusters']
      [, match_by_pos = match-by-position] ] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. Input columns must be of data type numeric .

Parameters

model_name

Name of the model (case-insensitive).

num_clusters

Integer between 1 and *k* inclusive, where *k* is the number of centers in the model, specifies the number of clusters to use for prediction.

Default: Value that the model specifies for *k*

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Privileges

Non-superusers: model owner, or USAGE privileges on the model

APPLY_IFOREST

Applies an isolation forest (iForest) model to an input relation. For each input row, the function returns an output row with two fields:

- anomaly_score : A float value that represents the average path length across all trees in the model normalized by the training sample size.
- is_anomaly : A Boolean value that indicates whether the input row is an anomaly. This value is true when anomaly_score is equal to or larger than a given threshold; otherwise, it's false.

Syntax

```
APPLY_IFOREST( input-columns USING PARAMETERS param=value[,...] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. Column types must match the types of the predictors in model_name .

Parameters

model_name

Name of the model (case-insensitive).

threshold

Optional. Float in the range (0.0, 1.0), specifies the threshold that determines if a data point is an anomaly. If the anomaly_score for a data point is equal to or larger than the value of threshold , the data point is marked as an outlier.

Alternatively, you can specify a contamination value that sets a threshold where the percentage of training data points labeled as outliers is approximately equal to the value of contamination . You cannot set both contamination and threshold in the same function call.
Default: 0.7

match_by_pos

Optional. Boolean value that specifies how input columns are matched to model columns:

- false : Match by name.
- true : Match by the position of columns in the input columns list.

Default: false

contamination

Optional. Float in the range (0.0, 1.0), the approximate ratio of data points in the training data that are labeled as outliers. The function calculates a threshold based on this contamination value. If you do not set this parameter, the function marks outliers using the specified or default threshold value.

You cannot set both contamination and threshold in the same function call.

Privileges

Non-superusers:

- USAGE privileges on the model
- SELECT privileges on the input relation

Examples

The following example demonstrates how different threshold values can affect outlier detection on an input relation:

```
=> SELECT * FROM (SELECT first_name, last_name, APPLY_IFOREST(team, hr, hits, avg, salary USING PARAMETERS
model_name='baseball_anomalies',
  threshold=0.75) AS predictions FROM baseball) AS outliers WHERE predictions.is_anomaly IS true;
 first_name | last_name |                    predictions
------------+-----------+-----------------------------------------------------
 Jacqueline | Richards  | {"anomaly_score":0.777757463074347,"is_anomaly":true}
(1 row)

=> SELECT * FROM (SELECT first_name, last_name, APPLY_IFOREST(team, hr, hits, avg, salary USING PARAMETERS
model_name='baseball_anomalies',
  threshold=0.55) AS predictions FROM baseball) AS outliers WHERE predictions.is_anomaly IS true;
 first_name | last_name |                    predictions
------------+-----------+-----------------------------------------------------
 Jacqueline | Richards  | {"anomaly_score":0.777757463074347,"is_anomaly":true}
 Debra      | Hall      | {"anomaly_score":0.5714649698133808,"is_anomaly":true}
 Gerald     | Fuller    | {"anomaly_score":0.5980549926114661,"is_anomaly":true}
(3 rows)
```

You can also use different contamination values to alter the outlier threshold:

```
=> SELECT * FROM (SELECT first_name, last_name, APPLY_IFOREST(team, hr, hits, avg, salary USING PARAMETERS
model_name='baseball_anomalies',
  contamination = 0.1) AS predictions FROM baseball) AS outliers WHERE predictions.is_anomaly IS true;
 first_name | last_name |                    predictions
------------+-----------+-----------------------------------------------------
 Marie      | Fields    | {"anomaly_score":0.5307715717521868,"is_anomaly":true}
 Jacqueline | Richards  | {"anomaly_score":0.777757463074347,"is_anomaly":true}
 Debra      | Hall      | {"anomaly_score":0.5714649698133808,"is_anomaly":true}
 Gerald     | Fuller    | {"anomaly_score":0.5980549926114661,"is_anomaly":true}
(4 rows)

=> SELECT * FROM (SELECT first_name, last_name, APPLY_IFOREST(team, hr, hits, avg, salary USING PARAMETERS
model_name='baseball_anomalies',
  contamination = 0.01) AS predictions FROM baseball) AS outliers WHERE predictions.is_anomaly IS true;
 first_name | last_name |                    predictions
------------+-----------+-----------------------------------------------------
 Jacqueline | Richards  | {"anomaly_score":0.777757463074347,"is_anomaly":true}
 Debra      | Hall      | {"anomaly_score":0.5714649698133808,"is_anomaly":true}
 Gerald     | Fuller    | {"anomaly_score":0.5980549926114661,"is_anomaly":true}
(3 rows)
```

See also

- Detect outliers
- IFOREST
- READ_TREE

APPLY_INVERSE_PCA

Inverts the APPLY_PCA -generated transform back to the original coordinate system.

Syntax

```
APPLY_INVERSE_PCA ( input-columns
    USING PARAMETERS model_name = 'model-name'
      [, exclude_columns = 'excluded-columns']
      [, key_columns = 'key-columns'] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. The following requirements apply:

- All columns must be a numeric data type.
- Enclose the column name in double quotes if it contains special characters.

Parameters

model_name

Name of the model (case-insensitive).

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

key_columns

Comma-separated list of column names from *input-columns* that identify its data rows. These columns are included in the output table.

Examples

The following example shows how to use the APPLY_INVERSE_PCA function. It shows the output for the first record.

```
=> SELECT PCA ('pcamodel', 'world','country,HDI,em1970,em1971,em1972,em1973,em1974,em1975,em1976,em1977,
em1978,em1979,em1980,em1981,em1982,em1983,em1984 ,em1985,em1986,em1987,em1988,em1989,em1990,em1991,em1992,
em1993,em1994,em1995,em1996,em1997,em1998,em1999,em2000,em2001,em2002,em2003,em2004,em2005,em2006,em2007,
em2008,em2009,em2010,gdp1970,gdp1971,gdp1972,gdp1973,gdp1974,gdp1975,gdp1976,gdp1977,gdp1978,gdp1979,gdp1980,
gdp1981,gdp1982,gdp1983,gdp1984,gdp1985,gdp1986,gdp1987,gdp1988,gdp1989,gdp1990,gdp1991,gdp1992,gdp1993,
gdp1994,gdp1995,gdp1996,gdp1997,gdp1998,gdp1999,gdp2000,gdp2001,gdp2002,gdp2003,gdp2004,gdp2005,gdp2006,
gdp2007,gdp2008,gdp2009,gdp2010' USING PARAMETERS exclude_columns='HDI,country');
PCA
------------------------------------------------------------------
Finished in 1 iterations.
Accepted Rows: 96  Rejected Rows: 0
(1 row)
=> CREATE TABLE worldPCA AS SELECT
APPLY_PCA (HDI,country,em1970,em1971,em1972,em1973,em1974,em1975,em1976,em1977,em1978,em1979,
em1980,em1981,em1982,em1983,em1984 ,em1985,em1986,em1987,em1988,em1989,em1990,em1991,em1992,em1993,em1994,
em1995,em1996,em1997,em1998,em1999,em2000,em2001,em2002,em2003,em2004,em2005,em2006,em2007,em2008,em2009,
em2010,gdp1970,gdp1971,gdp1972,gdp1973,gdp1974,gdp1975,gdp1976,gdp1977,gdp1978,gdp1979,gdp1980,gdp1981,gdp1982,
gdp1983,gdp1984,gdp1985,gdp1986,gdp1987,gdp1988,gdp1989,gdp1990,gdp1991,gdp1992,gdp1993,gdp1994,gdp1995,
gdp1996,gdp1997,gdp1998,gdp1999,gdp2000,gdp2001,gdp2002,gdp2003,gdp2004,gdp2005,gdp2006,gdp2007,gdp2008,
gdp2009,gdp2010 USING PARAMETERS model_name='pcamodel', exclude_columns='HDI, country', key_columns='HDI,
country',cutoff=.3)OVER () FROM world;
CREATE TABLE

=> SELECT * FROM worldPCA;
HDI   |       country       |       col1
------+---------------------+-------------------
0.886 | Belgium             |  79002.2946705704
0.699 | Belize              | -25631.6670012556
0.427 | Benin               | -40373.4104598122
0.805 | Chile               | -16805.7940082156
0.687 | China               | -37279.2893141103
0.744 | Costa Rica          | -19505.5631231635
0.4   | Cote d'Ivoire       | -38058.2060339272
0.776 | Cuba                | -23724.5779612041
0.895 | Denmark             |  117325.594028813
0.644 | Egypt               | -34609.9941604549
...
(96 rows)

=> SELECT APPLY_INVERSE_PCA (HDI, country, col1
    USING PARAMETERS model_name = 'pcamodel', exclude_columns='HDI,country',
    key_columns = 'HDI, country') OVER () FROM worldPCA;
HDI |       country       |      em1970       |      em1971       |      em1972      |      em1973      |
     em1974      |      em1975       |      em1976|      em1977      |      em1978       |      em1979
  |      em1980       |      em1981       |      em1982       |      em1983       |      em1984       |em1985
|      em1986       |      em1987       |      em1988       |      em1989       |      em1990       |      em1991
|      em1992       |      em1993|      em1994      |      em1995       |      em1996       |      em1997
```

```
   |     em1998     |     em1999     |     em2000     |     em2001     |em2002      |
em2003     |     em2004     |     em2005     |     em2006     |     em2007     |     em2008
|     em2009     |     em2010     |     gdp1970     |     gdp1971     |     gdp1972     |     gdp1973
|     gdp1974     |     gdp1975     |     gdp1976     |     gdp1977     |gdp1978     |     gdp1979
 |     gdp1980     |     gdp1981     |     gdp1982     |     gdp1983     |     gdp1984     |     gdp1985
      |     gdp1986|     gdp1987     |     gdp1988     |     gdp1989     |     gdp1990     |     gdp1991
    |     gdp1992     |     gdp1993     |     gdp1994     |     gdp1995     |     gdp1996     |
gdp1997     |     gdp1998     |     gdp1999     |     gdp2000     |     gdp2001     |     gdp2002
|     gdp2003     |gdp2004     |     gdp2005     |     gdp2006     |     gdp2007     |     gdp2008
 |     gdp2009     |     gdp2010
------+-----------------+----------------+----------------+----------------+----------------
+----------------+----------------+----------------+----------------+----------------+--------
---------+----------------+----------------+----------------+----------------+---------------
--+----------------+----------------+----------------+----------------+----------------+-------
----------+----------------+----------------+----------------+----------------+----------------
-+----------------+----------------+----------------+----------------+----------------+-------
-----------+-----------------+----------------+----------------+----------------+---------------
---+----------------+----------------+----------------+----------------+----------------+------
----------+----------------+----------------+----------------+----------------+----------------+
----------------+----------------+----------------+----------------+----------------+------------
-----+----------------+----------------+----------------+----------------+----------------+-------
----------+----------------+----------------+----------------+----------------+----------------+-
---------------+----------------+----------------+----------------+----------------+------------
----+----------------+----------------+----------------+----------------+----------------+-------
---------+----------------+----------------+----------------+----------------+----------------
0.886 | Belgium             |  18585.6613572407 | -16145.6374560074 |  26938.956253415 | 8094.30475779595 |
 12073.5461203817 | -11069.0567600181 | 19133.8584911727|   5500.312894949 | -4227.94863799987 |  6265.77925410752
|  -10884.749295608 | 30929.4669575201 | -7831.49439429977 |  3235.81760508742 | -22765.9285442662 | 27200
.6767714485 | -10554.9550160917 |    1169.4144482273 | -16783.7961289161 | 27932.2660829329 | 17227.9083196848
| 13956.0524012749 | -40175.6286481088 | -10889.4785920499 | 22703.6576872859 | -14635.5832197402 |
2857.12270512168 | 20473.5044214494 | -52199.4895696423 | -11038.7346460738 |  18466.7298633088 | -17410.4225137703 |
-3475.63826305462 | 29305.6753822341 |    1242.5724942049 | 17491.0096310849 | -12609.9984515902 | -17909.3603476248
|  6276.58431412381 | 21851.9475485178 | -2614.33738160397 | 3777.74134131349 | 4522.08854282736 | 4251.90446379366
| 4512.15101396876 | 4265.49424538129 | 5190.06845330997 | 4543.80444817989 | 5639.81122679089 | 4420.44705213467
|  5658.8820279283 | 5172.69025294376 | 5019.63640408663 | 5938.84979495903 | 4976.57073629812 | 4710.49525137591
| 6523.65700286465 | 5067.82520773578 | 6789.13070219317 | 5525.94643553563 | 6894.68336419297 | 5961.58442474331
| 5661.21093840818 | 7721.56088518218 |  5959.7301109143 | 6453.43604137202 | 6739.39384033096 | 7517.97645468455
| 6907.49136910647 | 7049.03921764209 | 7726.49091035527 | 8552.65909911844 | 7963.94487647115 | 7187.45827585515
| 7994.02955410523 | 9532.89844418041 | 7962.25713582666 | 7846.68238907624 | 10230.9878908643 | 8642.76044946519
| 8886.79860331866 |  8718.3731386891
...
(96 rows)
```

See also

- APPLY_PCA
- PCA

APPLY_INVERSE_SVD

Transforms the data back to the original domain. This essentially computes the approximated version of the original data by multiplying three matrices: matrix U (input to this function), matrices S and V (stored in the model).

Syntax

```
APPLY_INVERSE_SVD ( 'input-columns'
    USING PARAMETERS model_name = 'model-name'
      [, match_by_pos = match-by-position]
      [, exclude_columns = 'excluded-columns']
      [, key_columns = 'key-columns'] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. The following requirements apply:

- All columns must be a numeric data type.
- Enclose the column name in double quotes if it contains special characters.

Parameters

model_name

Name of the model (case-insensitive).

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

key_columns

Comma-separated list of column names from *input-columns* that identify its data rows. These columns are included in the output table.

Examples

```
=> SELECT SVD ('svdmodel', 'small_svd', 'x1,x2,x3,x4');
SVD
-----------------------------------------------------------------
Finished in 1 iterations.
Accepted Rows: 8  Rejected Rows: 0
(1 row)

=> CREATE TABLE transform_svd AS SELECT
    APPLY_SVD (id, x1, x2, x3, x4 USING PARAMETERS model_name='svdmodel', exclude_columns='id', key_columns='id')
    OVER () FROM small_svd;
CREATE TABLE

=> SELECT * FROM transform_svd;
id |       col1        |       col2        |       col3        |       col4
----+-------------------+-------------------+-------------------+-------------------
4  |  0.44849499240202 |  -0.347260956311326 |   0.186958376368345 |  0.378561270493651
6  |  0.17652411036246 | -0.0753183783382909 |  -0.678196192333598 | 0.0567124770173372
1  | 0.494871802886819 |   0.161721379259287 |  0.0712816417153664 | -0.473145877877408
2  |  0.17652411036246 | -0.0753183783382909 |  -0.678196192333598 | 0.0567124770173372
3  | 0.150974762654569 |   0.589561842046029 | 0.00392654610109522 |  0.360011163271921
5  | 0.494871802886819 |   0.161721379259287 |  0.0712816417153664 | -0.473145877877408
8  |  0.44849499240202 |  -0.347260956311326 |   0.186958376368345 |  0.378561270493651
7  | 0.150974762654569 |   0.589561842046029 | 0.00392654610109522 |  0.360011163271921
(8 rows)

=> SELECT APPLY_INVERSE_SVD (* USING PARAMETERS model_name='svdmodel', exclude_columns='id',
key_columns='id') OVER () FROM transform_svd;
id |        x1        |        x2        |        x3        |        x4
----+------------------+------------------+------------------+------------------
4 | 91.4056627665577 | 44.7629617207482 | 83.1704961993117 | 38.9274292265543
6 | 20.6468626294368 | 9.30974906868751 | 8.71006863405534 |  6.5855928603967
7 | 31.2494347777156 | 20.6336519003026 | 27.5668287751507 | 5.84427645886865
1 |  107.93376580719 | 51.6980548011917 | 97.9665796560552 | 40.4918236881051
2 | 20.6468626294368 | 9.30974906868751 | 8.71006863405534 |  6.5855928603967
3 | 31.2494347777156 | 20.6336519003026 | 27.5668287751507 | 5.84427645886865
5 |  107.93376580719 | 51.6980548011917 | 97.9665796560552 | 40.4918236881051
8 | 91.4056627665577 | 44.7629617207482 | 83.1704961993117 | 38.9274292265543
(8 rows)
```

See also

- APPLY_SVD
- SVD

APPLY_KMEANS

Assigns each row of an input relation to a cluster center from an existing k-means model.

Syntax

```
APPLY_KMEANS ( input-columns
    USING PARAMETERS model_name = 'model-name' [, match_by_pos = match-by-position] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Privileges

Non-superusers: model owner, or USAGE privileges on the model

Examples

The following example creates k-means model myKmeansModel and applies it to input table iris1 . The call to APPLY_KMEANS mixes column names and constants. When a constant is passed in place of a column name, the constant is substituted for the value of the column in all rows:

```
=> SELECT KMEANS('myKmeansModel', 'iris1', '*', 5
USING PARAMETERS max_iterations=20, output_view='myKmeansView', key_columns='id', exclude_columns='Species, id');
        KMEANS
------------------------------
 Finished in 12 iterations

(1 row)
=> SELECT id, APPLY_KMEANS(Sepal_Length, 2.2, 1.3, Petal_Width
USING PARAMETERS model_name='myKmeansModel', match_by_pos='true') FROM iris2;
 id  | APPLY_KMEANS
-----+--------------
  5 |            1
 10 |            1
 14 |            1
 15 |            1
 21 |            1
 22 |            1
 24 |            1
 25 |            1
 32 |            1
 33 |            1
 34 |            1
 35 |            1
 38 |            1
 39 |            1
 42 |            1
...
(60 rows)
```

See also

- Clustering data using k-means
- KMEANS

APPLY_KPROTOTYPES

Assigns each row of an input relation to a cluster center from an existing k-prototypes model.

Syntax

```
APPLY_KPROTOTYPES ( input-columns
    USING PARAMETERS model_name = 'model-name' [, match_by_pos = match-by-position] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Privileges

Non-superusers: model owner, or USAGE privileges on the model

Examples

The following example creates k-prototypes model small_model and applies it to input table small_test_mixed :

```
=> SELECT KPROTOTYPES('small_model_initcenters', 'small_test_mixed', 'x0, country', 3 USING PARAMETERS
initial_centers_table='small_test_mixed_centers', key_columns='pid');
    KPROTOTYPES
---------------------------
Finished in 2 iterations

(1 row)

=> SELECT country, x0, APPLY_KPROTOTYPES(country, x0
USING PARAMETERS model_name='small_model')
FROM small_test_mixed;
  country   | x0  | apply_kprototypes
------------+-----+-------------------
 'China'    |  20 |                 0
 'US'       |  85 |                 2
 'Russia'   |  80 |                 1
 'Brazil'   |  78 |                 1
 'US'       |  23 |                 0
 'US'       |  50 |                 0
 'Canada'   |  24 |                 0
 'Canada'   |  18 |                 0
 'Russia'   |  90 |                 2
 'Russia'   |  98 |                 2
 'Brazil'   |  89 |                 2
...
(45 rows)
```

See also

- KPROTOTYPES
- Clustering data using k-means
- KMEANS

APPLY_NORMALIZE

A UDTF function that applies the normalization parameters saved in a model to a set of specified input columns. If any column specified in the function is not in the model, its data passes through unchanged to APPLY_NORMALIZE .

> **Note**
> **Note** : If a column contains only one distinct value, APPLY_NORMALIZE returns NaN for values in that column.

Syntax

```
APPLY_NORMALIZE ( input-columns USING PARAMETERS model_name = 'model-name');
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns. If you supply an asterisk, APPLY_NORMALIZE normalizes all columns in the model.

Parameters

model_name

Name of the model (case-insensitive).

Examples

The following example creates a model with NORMALIZE_FIT using the wt and hp columns in table mtcars , and then uses this model in successive calls to APPLY_NORMALIZE and REVERSE_NORMALIZE .

```
=> SELECT NORMALIZE_FIT('mtcars_normfit', 'mtcars', 'wt,hp', 'minmax');
NORMALIZE_FIT
---------------
Success
(1 row)
```

The following call to APPLY_NORMALIZE specifies the hp and cyl columns in table mtcars , where hp is in the normalization model and cyl is not in the normalization model:

```
=> CREATE TABLE mtcars_normalized AS SELECT APPLY_NORMALIZE (hp, cyl USING PARAMETERS model_name = 'mtcars_normfit') FROM mtcars;
CREATE TABLE
=> SELECT * FROM mtcars_normalized;
         hp         | cyl
--------------------+-----
  0.434628975265018 | 8
  0.681978798586572 | 8
  0.434628975265018 | 6
                  1 | 8
  0.540636042402827 | 8
                  0 | 4
  0.681978798586572 | 8
 0.0459363957597173 | 4
  0.434628975265018 | 8
  0.204946996466431 | 6
  0.250883392226148 | 6
  0.049469964664311 | 4
  0.204946996466431 | 6
  0.201413427561837 | 4
  0.204946996466431 | 6
  0.250883392226148 | 6
  0.049469964664311 | 4
  0.215547703180212 | 4
 0.0353356890459364 | 4
  0.187279151943463 | 6
  0.452296819787986 | 8
```

```
 0.628975265017668 | 8
 0.346289752650177 | 8
 0.137809187279152 | 4
 0.749116607773852 | 8
 0.144876325088339 | 4
 0.151943462897526 | 4
 0.452296819787986 | 8
 0.452296819787986 | 8
 0.575971731448763 | 8
 0.159010600706714 | 4
 0.346289752650177 | 8
(32 rows)

=> SELECT REVERSE_NORMALIZE (hp, cyl USING PARAMETERS model_name='mtcars_normfit') FROM mtcars_normalized;
  hp | cyl
-----+-----
 175 | 8
 245 | 8
 175 | 6
 335 | 8
 205 | 8
  52 | 4
 245 | 8
  65 | 4
 175 | 8
 110 | 6
 123 | 6
  66 | 4
 110 | 6
 109 | 4
 110 | 6
 123 | 6
  66 | 4
 113 | 4
  62 | 4
 105 | 6
 180 | 8
 230 | 8
 150 | 8
  91 | 4
 264 | 8
  93 | 4
  95 | 4
 180 | 8
 180 | 8
 215 | 8
  97 | 4
 150 | 8
(32 rows)
```

The following call to REVERSE_NORMALIZE also specifies the hp and cyl columns in table mtcars , where hp is in normalization model mtcars_normfit , and cyl is not in the normalization model.

```
=> SELECT REVERSE_NORMALIZE (hp, cyl USING PARAMETERS model_name='mtcars_normfit') FROM mtcars_normalized;
       hp         | cyl
------------------+-----
205.000005722046 |   8
150.000000357628 |   8
150.000000357628 |   8
93.0000016987324 |   4
 174.99999666214 |   8
94.9999992102385 |   4
214.999997496605 |   8
97.0000009387732 |   4
245.000006556511 |   8
 174.99999666214 |   6
             335 |   8
245.000006556511 |   8
62.0000002086163 |   4
 174.99999666214 |   8
230.000002026558 |   8
              52 |   4
263.999997675419 |   8
109.999999523163 |   6
123.000002324581 |   6
64.9999996386468 |   4
66.0000005029142 |   4
112.999997898936 |   4
109.999999523163 |   6
180.000000983477 |   8
180.000000983477 |   8
108.999998658895 |   4
109.999999523163 |   6
104.999999418855 |   6
123.000002324581 |   6
180.000000983477 |   8
66.0000005029142 |   4
90.9999999701977 |   4
(32 rows)
```

See also

- NORMALIZE
- NORMALIZE_FIT
- Normalizing data
- REVERSE_NORMALIZE

APPLY_ONE_HOT_ENCODER

A user-defined transform function (UDTF) that loads the one hot encoder model and writes out a table that contains the encoded columns.

Syntax

```
APPLY_ONE_HOT_ENCODER( input-columns
      USING PARAMETERS model_name = 'model-name'
        [, drop_first = 'is-first']
        [, ignore_null = 'ignore']
        [, separator = 'separator-character']
        [, column_naming = 'name-output']
        [, null_column_name = 'null-column-name'] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).
, stores the categories and their corresponding levels.

drop_first

Boolean value, one of the following:

- true (default): Treat the first level of the categorical variable as the reference level.
- false : Every level of the categorical variable has a corresponding column in the output view

ignore_null

Boolean value, one of the following:

- true (default): Null values set all corresponding one-hot binary columns to null.
- false : Null values in *input-columns* are treated as a categorical level

separator

The character that separates the input variable name and the indicator variable level in the output table.To avoid using any separator, set this parameter to null value.

Default: Underscore (_)

column_naming

Appends categorical levels to column names according to the specified method:

- indices (default): Uses integer indices to represent categorical levels.
- values / values_relaxed : Both methods use categorical level names. If duplicate column names occur, the function attempts to disambiguate them by appending _ *n* , where *n* is a zero-based integer index (_0 , _1 ,...).
  If the function cannot produce unique column names , it handles this according to the chosen method:
  - values returns an error.
  - values_relaxed reverts to using indices.

Important
The following column naming rules apply if column_naming is set to values or values_relaxed :

- Input column names with more than 128 characters are truncated.
- Column names can contain special characters.
- If parameter ignore_null is set to true, APPLY_ONE_HOT_ENCODER constructs the column name from the value set in parameter null_column_name . If this parameter is omitted, the string null is used.

null_column_name

The string used in naming the indicator column for null values, used only if ignore_null is set to false and column_naming is set to values or values_relaxed .

Default: null

> **Note**
> **Note** : If an input row contains a level not stored in the model, the output row columns corresponding to that categorical level are returned as null values.

Examples

```
=> SELECT APPLY_ONE_HOT_ENCODER(cyl USING PARAMETERS model_name='one_hot_encoder_model',
drop_first='true', ignore_null='false') FROM mtcars;
cyl | cyl_1 | cyl_2
----+-------+-------
8   |     0 |     1
4   |     0 |     0
4   |     0 |     0
8   |     0 |     1
8   |     0 |     1
8   |     0 |     1
4   |     0 |     0
8   |     0 |     1
8   |     0 |     1
4   |     0 |     0
8   |     0 |     1
6   |     1 |     0
4   |     0 |     0
4   |     0 |     0
6   |     1 |     0
6   |     1 |     0
8   |     0 |     1
8   |     0 |     1
4   |     0 |     0
4   |     0 |     0
6   |     1 |     0
8   |     0 |     1
8   |     0 |     1
6   |     1 |     0
4   |     0 |     0
8   |     0 |     1
8   |     0 |     1
8   |     0 |     1
6   |     1 |     0
6   |     1 |     0
4   |     0 |     0
4   |     0 |     0
(32 rows)
```

See also

- Encoding categorical columns
- ONE_HOT_ENCODER_FIT

APPLY_PCA

Transforms the data using a PCA model. This returns new coordinates of each data point.

Syntax

```
APPLY_PCA ( input-columns
    USING PARAMETERS model_name = 'model-name'
      [, num_components = num-components]
      [, cutoff = cutoff-value]
      [, match_by_pos = match-by-position]
      [, exclude_columns = 'excluded-columns']
      [, key_columns = 'key-columns'] )
```

Arguments

input-columns

Comma-separated list of columns that contain the data matrix, or asterisk (*) to select all columns. The following requirements apply:

- All columns must be a numeric data type.
- Enclose the column name in double quotes if it contains special characters.

Parameters

model_name

Name of the model (case-insensitive).

num_components

The number of components to keep in the model. This is the number of output columns that will be generated. If you omit this parameter and the cutoff parameter, all model components are kept.

cutoff

Set to 1, specifies the minimum accumulated explained variance. Components are taken until the accumulated explained variance reaches this value.

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

key_columns

Comma-separated list of column names from *input-columns* that identify its data rows. These columns are included in the output table.

Examples

```
=> SELECT PCA ('pcamodel', 'world','country,HDI,em1970,em1971,em1972,em1973,em1974,em1975,em1976,em1977,
em1978,em1979,em1980,em1981,em1982,em1983,em1984 ,em1985,em1986,em1987,em1988,em1989,em1990,em1991,em1992,
em1993,em1994,em1995,em1996,em1997,em1998,em1999,em2000,em2001,em2002,em2003,em2004,em2005,em2006,em2007,
em2008,em2009,em2010,gdp1970,gdp1971,gdp1972,gdp1973,gdp1974,gdp1975,gdp1976,gdp1977,gdp1978,gdp1979,gdp1980,
gdp1981,gdp1982,gdp1983,gdp1984,gdp1985,gdp1986,gdp1987,gdp1988,gdp1989,gdp1990,gdp1991,gdp1992,gdp1993,
gdp1994,gdp1995,gdp1996,gdp1997,gdp1998,gdp1999,gdp2000,gdp2001,gdp2002,gdp2003,gdp2004,gdp2005,gdp2006,
gdp2007,gdp2008,gdp2009,gdp2010' USING PARAMETERS exclude_columns='HDI,country');
PCA
--------------------------------------------------------------------
Finished in 1 iterations.
Accepted Rows: 96  Rejected Rows: 0
(1 row)
=> CREATE TABLE worldPCA AS SELECT
APPLY_PCA (HDI,country,em1970,em1971,em1972,em1973,em1974,em1975,em1976,em1977,em1978,em1979,
em1980,em1981,em1982,em1983,em1984 ,em1985,em1986,em1987,em1988,em1989,em1990,em1991,em1992,em1993,em1994,
em1995,em1996,em1997,em1998,em1999,em2000,em2001,em2002,em2003,em2004,em2005,em2006,em2007,em2008,em2009,
em2010,gdp1970,gdp1971,gdp1972,gdp1973,gdp1974,gdp1975,gdp1976,gdp1977,gdp1978,gdp1979,gdp1980,gdp1981,gdp1982,
gdp1983,gdp1984,gdp1985,gdp1986,gdp1987,gdp1988,gdp1989,gdp1990,gdp1991,gdp1992,gdp1993,gdp1994,gdp1995,
gdp1996,gdp1997,gdp1998,gdp1999,gdp2000,gdp2001,gdp2002,gdp2003,gdp2004,gdp2005,gdp2006,gdp2007,gdp2008,
gdp2009,gdp2010 USING PARAMETERS model_name='pcamodel', exclude_columns='HDI, country', key_columns='HDI,
country',cutoff=.3)OVER () FROM world;
CREATE TABLE

=> SELECT * FROM worldPCA;
HDI   |       country       |       col1
------+---------------------+-------------------
0.886 | Belgium             |  79002.2946705704
0.699 | Belize              | -25631.6670012556
0.427 | Benin               | -40373.4104598122
0.805 | Chile               | -16805.7940082156
0.687 | China               | -37279.2893141103
0.744 | Costa Rica          | -19505.5631231635
0.4   | Cote d'Ivoire       | -38058.2060339272
0.776 | Cuba                | -23724.5779612041
0.895 | Denmark             |  117325.594028813
0.644 | Egypt               | -34609.9941604549
...
(96 rows)
```

```
(96 rows)

=> SELECT APPLY_INVERSE_PCA (HDI, country, col1
    USING PARAMETERS model_name = 'pcamodel', exclude_columns='HDI,country',
    key_columns = 'HDI, country') OVER () FROM worldPCA;
HDI |      country      |     em1970      |     em1971      |     em1972      |     em1973      |
     em1974     |     em1975      |     em1976|     em1977      |     em1978      |     em1979
  |     em1980      |     em1981      |     em1982      |     em1983      |     em1984      |em1985
|     em1986      |     em1987      |     em1988      |     em1989      |     em1990      |     em1991
|     em1992      |     em1993|     em1994      |     em1995      |     em1996      |     em1997
   |     em1998      |     em1999      |     em2000      |     em2001      |em2002      |
em2003     |     em2004      |     em2005      |     em2006      |     em2007      |     em2008
|     em2009      |     em2010      |     gdp1970     |     gdp1971     |     gdp1972     |     gdp1973
|     gdp1974     |     gdp1975     |     gdp1976     |     gdp1977     |gdp1978     |     gdp1979
 |     gdp1980     |     gdp1981     |     gdp1982     |     gdp1983     |     gdp1984     |     gdp1985
     |     gdp1986|     gdp1987     |     gdp1988     |     gdp1989     |     gdp1990     |     gdp1991
    |     gdp1992     |     gdp1993     |     gdp1994     |     gdp1995     |     gdp1996     |
gdp1997     |     gdp1998     |     gdp1999     |     gdp2000     |     gdp2001     |     gdp2002
|     gdp2003     |gdp2004     |     gdp2005     |     gdp2006     |     gdp2007     |     gdp2008
  |     gdp2009     |     gdp2010
-------+-----------------+-----------------+-----------------+-----------------+-----------------
+-----------------+-----------------+-----------------+-----------------+-----------------+---------
----------+-----------------+-----------------+-----------------+-----------------+----------------
--+-----------------+-----------------+-----------------+-----------------+-----------------+-------
----------+-----------------+-----------------+-----------------+-----------------+-----------------
-+-----------------+-----------------+-----------------+-----------------+-----------------+-------
-----------+-----------------+-----------------+-----------------+-----------------+----------------
---+-----------------+-----------------+-----------------+-----------------+-----------------+------
-----------+-----------------+-----------------+-----------------+-----------------+-----------------+
----------------+-----------------+-----------------+-----------------+-----------------+------------
-----+-----------------+-----------------+-----------------+-----------------+-----------------+-------
----------+-----------------+-----------------+-----------------+-----------------+-----------------+-
----------------+-----------------+-----------------+-----------------+-----------------+-------------
----+-----------------+-----------------+-----------------+-----------------+-----------------+--------
---------+-----------------+-----------------+-----------------+-----------------+-----------------
0.886 | Belgium             |  18585.6613572407 | -16145.6374560074 |  26938.956253415 | 8094.30475779595 |
 12073.5461203817 | -11069.0567600181 | 19133.8584911727|   5500.312894949 | -4227.94863799987 |  6265.77925410752
|  -10884.749295608 | 30929.4669575201 | -7831.49439429977 |  3235.81760508742 | -22765.9285442662 | 27200
.6767714485 | -10554.9550160917 |   1169.4144482273 | -16783.7961289161 | 27932.2660829329 | 17227.9083196848
| 13956.0524012749 | -40175.6286481088 | -10889.4785920499 | 22703.6576872859 | -14635.5832197402 |
2857.12270512168 | 20473.5044214494 | -52199.4895696423 | -11038.7346460738 |  18466.7298633088 | -17410.4225137703 |
-3475.63826305462 | 29305.6753822341 |   1242.5724942049 | 17491.0096310849 | -12609.9984515902 | -17909.3603476248
|  6276.58431412381 | 21851.9475485178 | -2614.33738160397 | 3777.74134131349 | 4522.08854282736 | 4251.90446379366
| 4512.15101396876 | 4265.49424538129 | 5190.06845330997 | 4543.80444817989 | 5639.81122679089 | 4420.44705213467
|  5658.8820279283 | 5172.69025294376 | 5019.63640408663 | 5938.84979495903 | 4976.57073629812 | 4710.49525137591
| 6523.65700286465 | 5067.82520773578 | 6789.13070219317 | 5525.94643553563 | 6894.68336419297 | 5961.58442474331
| 5661.21093840818 | 7721.56088518218 |  5959.7301109143 | 6453.43604137202 | 6739.39384033096 | 7517.97645468455
| 6907.49136910647 | 7049.03921764209 | 7726.49091035527 | 8552.65909911844 | 7963.94487647115 | 7187.45827585515
| 7994.02955410523 | 9532.89844418041 | 7962.25713582666 | 7846.68238907624 | 10230.9878908643 | 8642.76044946519
| 8886.79860331866 |  8718.3731386891
...
(96 rows)
```

See also

- APPLY_INVERSE_PCA
- PCA

APPLY_SVD

Transforms the data using an SVD model. This computes the matrix U of the SVD decomposition.

Syntax

```
APPLY_SVD ( input-columns
    USING PARAMETERS model_name = 'model-name'
      [, num_components = num-components]
      [, cutoff = cutoff-value]
      [, match_by_pos = match-by-position]
      [, exclude_columns = 'excluded-columns']
      [, key_columns = 'key-columns'] )
```

Arguments

input-columns

Comma-separated list of columns that contain the data matrix, or asterisk (*) to select all columns. The following requirements apply:

- All columns must be a numeric data type.
- Enclose the column name in double quotes if it contains special characters.

Parameters

model_name

Name of the model (case-insensitive).

num_components

The number of components to keep in the model. This is the number of output columns that will be generated. If neither this parameter nor the cutoff parameter is provided, all components from the model are kept.

cutoff

Set to 1, specifies the minimum accumulated explained variance. Components are taken until the accumulated explained variance reaches this value. If you omit this parameter and the num_components parameter, all model components are kept.

match_by_pos

Boolean value that specifies how input columns are matched to model columns:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

exclude_columns

Comma-separated list of column names from *input-columns* to exclude from processing.

key_columns

Comma-separated list of column names from *input-columns* that identify its data rows. These columns are included in the output table.

Examples

```
=> SELECT SVD ('svdmodel', 'small_svd', 'x1,x2,x3,x4');
SVD
-------------------------------------------------------------------
Finished in 1 iterations.
Accepted Rows: 8  Rejected Rows: 0
(1 row)

=> CREATE TABLE transform_svd AS SELECT
    APPLY_SVD (id, x1, x2, x3, x4 USING PARAMETERS model_name='svdmodel', exclude_columns='id', key_columns='id')
    OVER () FROM small_svd;
CREATE TABLE

=> SELECT * FROM transform_svd;
id |       col1        |       col2        |       col3        |       col4
---+-------------------+-------------------+-------------------+-------------------
4  |  0.44849499240202 |  -0.347260956311326 |   0.186958376368345 |  0.378561270493651
6  |  0.17652411036246 | -0.0753183783382909 |  -0.678196192333598 | 0.0567124770173372
1  | 0.494871802886819 |   0.161721379259287 |  0.0712816417153664 | -0.473145877877408
2  |  0.17652411036246 | -0.0753183783382909 |  -0.678196192333598 | 0.0567124770173372
3  | 0.150974762654569 |   0.589561842046029 | 0.00392654610109522 |  0.360011163271921
5  | 0.494871802886819 |   0.161721379259287 |  0.0712816417153664 | -0.473145877877408
8  |  0.44849499240202 |  -0.347260956311326 |   0.186958376368345 |  0.378561270493651
7  | 0.150974762654569 |   0.589561842046029 | 0.00392654610109522 |  0.360011163271921
(8 rows)

=> SELECT APPLY_INVERSE_SVD (* USING PARAMETERS model_name='svdmodel', exclude_columns='id',
key_columns='id') OVER () FROM transform_svd;
id |       x1        |       x2        |       x3        |       x4
---+-----------------+-----------------+-----------------+-----------------
4 | 91.4056627665577 | 44.7629617207482 | 83.1704961993117 | 38.9274292265543
6 | 20.6468626294368 | 9.30974906868751 | 8.71006863405534 |  6.5855928603967
7 | 31.2494347777156 | 20.6336519003026 | 27.5668287751507 | 5.84427645886865
1 |  107.93376580719 | 51.6980548011917 | 97.9665796560552 | 40.4918236881051
2 | 20.6468626294368 | 9.30974906868751 | 8.71006863405534 |  6.5855928603967
3 | 31.2494347777156 | 20.6336519003026 | 27.5668287751507 | 5.84427645886865
5 |  107.93376580719 | 51.6980548011917 | 97.9665796560552 | 40.4918236881051
8 | 91.4056627665577 | 44.7629617207482 | 83.1704961993117 | 38.9274292265543
(8 rows)
```

See also

- APPLY_INVERSE_SVD
- SVD

PREDICT_ARIMA

Applies an autoregressive integrated moving average (ARIMA) model to an input relation or makes predictions using the in-sample data. ARIMA models make predictions based on preceding time series values and errors of previous predictions. The function, by default, returns the predicted values plus the mean of the model.

Behavior type

Immutable

Syntax

Apply to an input relation:

```
PREDICT_ARIMA ( 'timeseries-column'
    USING PARAMETERS param=value[,...] )
    OVER (ORDER BY 'timestamp-column')
    FROM input-relation
```

Make predictions using the in-sample data:

```
PREDICT_ARIMA ( USING PARAMETERS model_name = 'ARIMA-model'
    [, start = prediction-start ]
    [, npredictions = num-predictions ]
    [, output_standard_errors = boolean ] )
    OVER ()
```

Arguments

timeseries-column

Name of a NUMERIC column in *input-relation* used to make predictions.

timestamp-column

Name of an INTEGER, FLOAT, or TIMESTAMP column in *input-relation* that represents the timestamp variable. The timestep between consecutive entries should be consistent throughout the *timestamp-column* .

input-relation

Input relation containing *timeseries-column* and *timestamp-column* .

Parameters

model_name

Name of a trained ARIMA model.

start

The behavior of the start parameter and its range of accepted values depends on whether you provide a *timeseries-column* :

- No provided *timeseries-column* : start must be an integer ≥0, where zero indicates to start prediction at the end of the in-sample data. If start is a positive value, the function predicts the values between the end of the in-sample data and the start index, and then uses the predicted values as time series inputs for the subsequent npredictions .
- *timeseries-column* provided: start must be an integer ≥1 and identifies the index (row) of the *timeseries-column* at which to begin prediction. If the start index is greater than the number of rows, N , in the input data, the function predicts the values between N and start and uses the predicted values as time series inputs for the subsequent npredictions .

Default :

- No provided *timeseries-column* : prediction begins from the end of the in-sample data.
- *timeseries-column* provided: prediction begins from the end of the provided input data.

npredictions

Integer ≥1, the number of predicted timesteps.

Default : 10

missing

Methods for handling missing values, one of the following strings:

- 'drop': Missing values are ignored.
- 'error': Missing values raise an error.
- 'zero': Missing values are replaced with 0.
- 'linear_interpolation': Missing values are replaced by linearly-interpolated values based on the nearest valid entries before and after the missing value. If all values before or after a missing value in the prediction range are missing or invalid, interpolation is impossible and the function errors.

Default: Method used when training the model

add_mean

Boolean, whether to add the model mean to the predicted value.

Default: True

output_standard_errors

Boolean, whether to return estimates of the standard error of each prediction.

Default: False

Examples

The following example makes predictions using the in-sample data that the arima_temp model was trained on:

```
=> SELECT PREDICT_ARIMA(USING PARAMETERS model_name='arima_temp', npredictions=10) OVER();
   prediction
------------------
12.9797364979873
13.3768377212635
13.460660717892
13.468204126011
13.4572461558472
13.4418721036084
13.425515187182
13.4090117135945
13.3925648829068
13.3762235523779
(10 rows)
```

You can also apply the model to an input relation:

```
=> SELECT PREDICT_ARIMA(temperature USING PARAMETERS model_name='arima_temp', start=100, npredictions=10) OVER(ORDER BY time) FROM
temp_data;
   prediction
------------------
15.0373229398431
13.4709102391534
10.5720766977885
13.1971253722069
13.5615497506689
13.1613971089657
13.4008120147841
12.612020423044
12.9026197179173
13.2392824099367
(10 rows)
```

For an in-depth example that trains and makes predictions with an ARIMA model, see ARIMA model example .

See also

- Time series forecasting
- GET_MODEL_SUMMARY
- GET_MODEL_ATTRIBUTE

PREDICT_AUTOREGRESSOR

Applies an autoregressor (AR) model to an input relation.

Autoregressive models use previous values to make predictions. More specifically, the user-specified "lag" determines how many previous timesteps it takes into account during computation, and predicted values are linear combinations of those lags.

Syntax

```
PREDICT_AUTOREGRESSOR ( timeseries-column
    USING PARAMETERS
      model-name = 'model-name'
      [, start = starting-index]
      [, npredictions = npredictions]
      [, missing = "imputation-method" ] )
    OVER (ORDER BY timestamp-column)
    FROM input-relation
```

Note

The following argument, as written, is required and cannot be omitted nor substituted with another type of clause.

```
OVER (ORDER BY timestamp-column)
```

Arguments

timeseries-column

The timeseries column used to make the prediction (only the last p values, specified during model creation, are used).

timestamp-column

The timestamp column, with consistent timesteps, used to make the prediction.

input-relation

The input relation containing the *timeseries-column* and *timestamp-column* .

Note that *input-relation* cannot have missing values in any of the p (set during training) rows preceding start . To handle missing values, see IMPUTE or Linear interpolation .

Parameters

model_name

Name of the model (case-insensitive).

start

INTEGER >p or ≤0, the index (row) of the *input-relation* at which to start the prediction. If omitted, the prediction starts at the end of the *input-relation* .

If the start index is greater than the number of rows N in *timeseries-column* , then the values between N and start are predicted and used for the prediction.
If negative, the start index is identified by counting backwards from the end of the *input-relation* .

For an *input-relation* of N rows, negative values have a lower limit of either -1000 or -(N-p), whichever is greater.

Default: the end of *input-relation*

npredictions

INTEGER ≥1, the number of predicted timesteps.

Default: 10

missing

One of the following methods for handling missing values:

- **drop** : Missing values are ignored.
- **error** : Missing values raise an error.
- **zero** : Missing values are replaced with 0.
- **linear_interpolation** : Missing values are replaced by linearly-interpolated values based on the nearest valid entries before and after the missing value. If all values before or after a missing value in the prediction range are missing or invalid, interpolation is impossible and the function errors.

Default: linear_interpolation

Examples

See Autoregressive model example .

See also

- AUTOREGRESSOR
- GET_MODEL_SUMMARY

PREDICT_LINEAR_REG

Applies a linear regression model on an input relation and returns the predicted value as a FLOAT.

Syntax

```
PREDICT_LINEAR_REG ( input-columns
    USING PARAMETERS model_name = 'model-name' [, match_by_pos = match-by-position] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Examples

```
=> SELECT PREDICT_LINEAR_REG(waiting USING PARAMETERS model_name='myLinearRegModel')FROM
faithful ORDER BY id;


 PREDICT_LINEAR_REG
--------------------
  4.15403481386324
  2.18505296804024
  3.76023844469864
   2.8151271587036
  4.62659045686076
  2.26381224187316
  4.86286827835952
  4.62659045686076
  1.94877514654148
  4.62659045686076
  2.18505296804024
...
 (272 rows)
```

The following example shows how to use the PREDICT_LINEAR_REG function on an input table, using the match_by_pos parameter. Note that you can replace the column argument with a constant that does not match an input column:

```
=> SELECT PREDICT_LINEAR_REG(55 USING PARAMETERS model_name='linear_reg_faithful',
                  match_by_pos='true')FROM faithful ORDER BY id;
 PREDICT_LINEAR_REG
--------------------
2.28552115094171
2.28552115094171
2.28552115094171
2.28552115094171
2.28552115094171
2.28552115094171
2.28552115094171
...
 (272 rows)
```

PREDICT_LOGISTIC_REG

Applies a logistic regression model on an input relation.

PREDICT_LOGISTIC_REG returns as a FLOAT the predicted class or the probability of the predicted class, depending on how the type parameter is set. You can cast the return value to INTEGER or another numeric type when the return is in the probability of the predicted class.

Syntax

```
PREDICT_LOGISTIC_REG ( input-columns
    USING PARAMETERS model_name = 'model-name'
      [, type = 'prediction-type']
      [, cutoff = probability-cutoff]
      [, match_by_pos = match-by-position] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

type

Type of prediction for logistic regression, one of the following:

- response (default): Predicted values are 0 or 1.
- probability : Output is the probability of the predicted category to be 1.

cutoff

Used in conjunction with the type parameter, a FLOAT between 0 and 1, exclusive. When type is set to response , the returned value of prediction is 1 if its corresponding probability is greater than or equal to the value of cutoff ; otherwise, it is 0.

Default: 0.5

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Examples

```
=> SELECT car_model,
          PREDICT_LOGISTIC_REG(mpg, cyl, disp, drat, wt, qsec, vs, gear, carb
                               USING PARAMETERS model_name='myLogisticRegModel')
         FROM mtcars;
     car_model      | PREDICT_LOGISTIC_REG
--------------------+----------------------
 Camaro Z28         |                    0
 Fiat 128           |                    1
 Fiat X1-9          |                    1
 Ford Pantera L     |                    1
 Merc 450SE         |                    0
 Merc 450SL         |                    0
 Toyota Corona      |                    0
 AMC Javelin        |                    0
 Cadillac Fleetwood |                    0
 Datsun 710         |                    1
 Dodge Challenger   |                    0
 Hornet 4 Drive     |                    0
 Lotus Europa       |                    1
 Merc 230           |                    0
 Merc 280           |                    0
 Merc 280C          |                    0
 Merc 450SLC        |                    0
 Pontiac Firebird   |                    0
 Porsche 914-2      |                    1
 Toyota Corolla     |                    1
 Valiant            |                    0
 Chrysler Imperial  |                    0
 Duster 360         |                    0
 Ferrari Dino       |                    1
 Honda Civic        |                    1
 Hornet Sportabout  |                    0
 Lincoln Continental|                    0
 Maserati Bora      |                    1
 Mazda RX4          |                    1
 Mazda RX4 Wag      |                    1
 Merc 240D          |                    0
 Volvo 142E         |                    1
(32 rows)
```

The following example shows how to use PREDICT_LOGISTIC_REG on an input table, using the match_by_pos parameter. Note that you can replace any of the column inputs with a constant that does not match an input column. In this example, column mpg was replaced with the constant 20:

```
=> SELECT car_model,
          PREDICT_LOGISTIC_REG(20, cyl, disp, drat, wt, qsec, vs, gear, carb
                               USING PARAMETERS model_name='myLogisticRegModel', match_by_pos='true')
         FROM mtcars;
     car_model     | PREDICT_LOGISTIC_REG
-------------------+----------------------
AMC Javelin        |                    0
Cadillac Fleetwood |                    0
Camaro Z28         |                    0
Chrysler Imperial  |                    0
Datsun 710         |                    1
Dodge Challenger   |                    0
Duster 360         |                    0
Ferrari Dino       |                    1
Fiat 128           |                    1
Fiat X1-9          |                    1
Ford Pantera L     |                    1
Honda Civic        |                    1
Hornet 4 Drive     |                    0
Hornet Sportabout  |                    0
Lincoln Continental|                    0
Lotus Europa       |                    1
Maserati Bora      |                    1
Mazda RX4          |                    1
Mazda RX4 Wag      |                    1
Merc 230           |                    0
Merc 240D          |                    0
Merc 280           |                    0
Merc 280C          |                    0
Merc 450SE         |                    0
Merc 450SL         |                    0
Merc 450SLC        |                    0
Pontiac Firebird   |                    0
Porsche 914-2      |                    1
Toyota Corolla     |                    1
Toyota Corona      |                    0
Valiant            |                    0
Volvo 142E         |                    1
(32 rows)
```

PREDICT_MOVING_AVERAGE

Applies a moving-average (MA) model, created by MOVING_AVERAGE , to an input relation.

Moving average models use the errors of previous predictions to make future predictions. More specifically, the user-specified "lag" determines how many previous predictions and errors it takes into account during computation.

Syntax

```
PREDICT_MOVING_AVERAGE ( timeseries-column
    USING PARAMETERS
      model_name = 'model-name'
      [, start = starting-index]
      [, npredictions = npredictions]
      [, missing = "imputation-method" ] )
    OVER (ORDER BY timestamp-column)
    FROM input-relation
```

Note

The following argument, as written, is required and cannot be omitted nor substituted with another type of clause.

```
OVER (ORDER BY timestamp-column)
```

Arguments

timeseries-column

The timeseries column used to make the prediction (only the last q values, specified during model creation, are used).

timestamp-column

The timestamp column, with consistent timesteps, used to make the prediction.

input-relation

The input relation containing the *timeseries-column* and *timestamp-column* .

Note that *input-relation* cannot have missing values in any of the q (set during training) rows preceding start . To handle missing values, see IMPUTE or Linear interpolation .

Parameters

model_name

Name of the model (case-insensitive).

start

INTEGER >q or ≤0, the index (row) of the *input-relation* at which to start the prediction. If omitted, the prediction starts at the end of the *input-relation* .

If the start index is greater than the number of rows N in *timeseries-column* , then the values between N and start are predicted and used for the prediction.
If negative, the start index is identified by counting backwards from the end of the *input-relation* .

For an *input-relation* of N rows, negative values have a lower limit of either -1000 or -(N-q), whichever is greater.

Default: the end of *input-relation*

npredictions

INTEGER ≥1, the number of predicted timesteps.

Default: 10

missing

One of the following methods for handling missing values:

- **drop** : Missing values are ignored.
- **error** : Missing values raise an error.
- **zero** : Missing values are replaced with 0.
- **linear_interpolation** : Missing values are replaced by linearly-interpolated values based on the nearest valid entries before and after the missing value. If all values before or after a missing value in the prediction range are missing or invalid, interpolation is impossible and the function errors.

Default: linear_interpolation

Examples

See Moving-average model example .

See also

- MOVING_AVERAGE
- GET_MODEL_SUMMARY

PREDICT_NAIVE_BAYES

Applies a Naive Bayes model on an input relation.

Depending on how the type parameter is set, PREDICT_NAIVE_BAYES returns a VARCHAR that specifies either the predicted class or probability of the predicted class. If the function returns probability, you can cast the return value to an INTEGER or another numeric data type.

Syntax

```
PREDICT_NAIVE_BAYES ( input-columns
    USING PARAMETERS model_name = 'model-name'
      [, type = ' return-type ']
      [, class = 'user-input-class']
      [, match_by_pos = match-by-position] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

type

One of the following:

- response (default): Returns the class with the highest probability.
- probability : Valid only if class parameter is set, returns the probability of belonging to the specified class argument.

class

Required if type parameter is set to probability . If you omit this parameter, PREDICT_NAIVE_BAYES returns the class that it predicts as having the highest probability.

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Examples

```
=> SELECT party, PREDICT_NAIVE_BAYES (vote1, vote2, vote3
                                USING PARAMETERS model_name='naive_house84_model',
                                        type='response')
                                AS Predicted_Party
                                FROM house84_test;
  party    | Predicted_Party
-----------+-----------------
democrat   | democrat
democrat   | democrat
democrat   | democrat
republican | republican
democrat   | democrat
democrat   | democrat
democrat   | democrat
democrat   | democrat
democrat   | democrat
republican | republican
democrat   | democrat
democrat   | democrat
democrat   | democrat
democrat   | republican
republican | republican
democrat   | democrat
republican | republican
...
(99 rows)
```

See also

- Classifying data using naive bayes
- NAIVE_BAYES
- PREDICT_NAIVE_BAYES_CLASSES

PREDICT_NAIVE_BAYES_CLASSES

Applies a Naive Bayes model on an input relation and returns the probabilities of classes:

- VARCHAR predicted column contains the class label with the highest probability.
- Multiple FLOAT columns, where the first probability column contains the probability for the class specified in the predicted column. Other columns contain the probability of belonging to each class specified in the classes parameter.

Syntax

```
PREDICT_NAIVE_BAYES_CLASSES ( predictor-columns
      USING PARAMETERS model_name = 'model-name'
        [, key_columns = 'key-columns']
        [, exclude_columns = 'excluded-columns]
        [, classes = 'classes']
        [, match_by_pos = match-by-position] )
OVER( [window-partition-clause] )
```

Arguments

predictor-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

key_columns

Comma-separated list of predictor column names that identify the output rows. To exclude these and other predictor columns from being used for prediction, include them in the argument list for parameter exclude_columns .

exclude_columns

Comma-separated list of columns from *predictor-columns* to exclude from processing.

classes

Comma-separated list of class labels in the model. The probability of belonging to this given class as predicted by the classifier. The values are case sensitive.

match_by_pos

Boolean value that specifies how predictor columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the predictor columns list.

Examples

```
=> SELECT PREDICT_NAIVE_BAYES_CLASSES (id, vote1, vote2 USING PARAMETERS
model_name='naive_house84_model',key_columns='id',exclude_columns='id',
classes='democrat, republican', match_by_pos='false')
      OVER() FROM house84_test;
 id  | Predicted |    Probability    |     democrat      |    republican
-----+-----------+-------------------+-------------------+-------------------
  21 | democrat  | 0.775473383353576 | 0.775473383353576 | 0.224526616646424
  28 | democrat  | 0.775473383353576 | 0.775473383353576 | 0.224526616646424
  83 | republican | 0.592510497724379 | 0.407489502275621 | 0.592510497724379
 102 | democrat  | 0.779889432167111 | 0.779889432167111 | 0.220110567832889
 107 | republican | 0.598662714551597 | 0.401337285448403 | 0.598662714551597
 125 | republican | 0.598662714551597 | 0.401337285448403 | 0.598662714551597
 132 | republican | 0.592510497724379 | 0.407489502275621 | 0.592510497724379
 136 | republican | 0.592510497724379 | 0.407489502275621 | 0.592510497724379
 155 | republican | 0.598662714551597 | 0.401337285448403 | 0.598662714551597
 174 | republican | 0.592510497724379 | 0.407489502275621 | 0.592510497724379
...
(1 row)
```

See also

- Classifying data using naive bayes
- PREDICT_NAIVE_BAYES
- NAIVE_BAYES

PREDICT_PMML

Applies an imported PMML model on an input relation. The function returns the result that would be expected for the model type encoded in the PMML model.

PREDICT_PMML returns NULL in the following cases:

- The predictor is an invalid or NULL value.
- The categorical predictor is of an unknown class.

> **Note**
> PREDICT_PMML returns values of complex type ROW for models that use the Output tag . Currently, Vertica does not support directly inserting this data into a table.
>
> You can work around this limitation by changing the output to JSON with TO_JSON before inserting it into a table:
>
> ```
> => CREATE TABLE predicted_output AS SELECT TO_JSON(PREDICT_PMML(X1,X2,X3
> USING PARAMETERS model_name='pmml_imported_model'))
> AS predicted_value
> FROM input_table;
> ```

Syntax

```
PREDICT_PMML ( input-columns
    USING PARAMETERS model_name = 'model-name' [, match_by_pos = match-by-position] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive). For a list of supported PMML model types and tags, see PMML features and attributes .

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Examples

In this example, the function call uses all the columns from the table as predictors and predicts the value using the ' my_kmeans ' model in PMML format:

```
SELECT PREDICT_PMML(* USING PARAMETERS model_name='my_kmeans') AS predicted_label FROM table;
```

In this example, the function call takes only columns col1, col2 as predictors, and predicts the value for each row using the ' my_kmeans ' model from schema ' my_schema ':

```
SELECT PREDICT_PMML(col1, col2 USING PARAMETERS model_name='my_schema.my_kmeans') AS predicted_label FROM table;
```

In this example, the function call returns an error as neither *schema* nor *model-name* can accept * as a value:

```
SELECT PREDICT_PMML(* USING PARAMETERS model_name='*.*') AS predicted_label FROM table;
SELECT PREDICT_PMML(* USING PARAMETERS model_name='*') AS predicted_label FROM table;
SELECT PREDICT_PMML(* USING PARAMETERS model_name='models.*') AS predicted_label FROM table;
```

See also

- IMPORT_MODELS

- EXPORT_MODELS

PREDICT_POISSON_REG

Applies a Poisson regression model on an input relation and returns the predicted value as a FLOAT.

Syntax

```
PREDICT_POISSON_REG ( input-columns
    USING PARAMETERS model_name = 'model-name' [, match_by_pos = match-by-position] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Examples

```
=> SELECT PREDICT_POISSON_REG(waiting USING PARAMETERS model_name='MYModel')::numeric(20,10) FROM lin.faithful ORDER BY id;
predict_poisson_reg
---------------------
4.0230080811
2.2284857176
3.5747254723
2.6921731651
4.6357580051
2.2817680621
4.9762900161
4.6357580051
2.0759884314
(9 rows)
```

PREDICT_RF_CLASSIFIER

Applies a random forest model on an input relation. PREDICT_RF_CLASSIFIER returns a VARCHAR data type that specifies one of the following, as determined by how the type parameter is set:

- The predicted class (based on popular votes)
- Probability of a class for each input instance.

> **Note**
> The predicted class is selected only based on the popular vote of the decision trees in the forest. Therefore, in special cases the calculated probability of the predicted class may not be the highest.

Syntax

```
PREDICT_RF_CLASSIFIER ( input-columns
    USING PARAMETERS model_name = 'model-name'
      [, type = 'prediction-type']
      [, class = 'user-input-class']
      [, match_by_pos = match-by-position] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

type

Type of prediction to return, one of the following:

- response (default): The class with the highest probability among all possible classes.
- probability : Valid only if the class parameter is set, returns the probability of the specified class.

class

Class to use when the type parameter is set to probability . If you omit this parameter, the function uses the predicted class—the one with the popular vote. Thus, the predict function returns the probability that the input instance belongs to its predicted class.

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Examples

```
=> SELECT PREDICT_RF_CLASSIFIER (Sepal_Length, Sepal_Width, Petal_Length, Petal_Width
    USING PARAMETERS model_name='myRFModel') FROM iris;
PREDICT_RF_CLASSIFIER
-----------------------
setosa
setosa
setosa
...
versicolor
versicolor
versicolor
...
virginica
virginica
virginica
...
(150 rows)
```

This example shows how you can use the PREDICT_RF_CLASSIFIER function, using the match_by_pos parameter:

```
=> SELECT PREDICT_RF_CLASSIFIER (Sepal_Length, Sepal_Width, Petal_Length, Petal_Width
                        USING PARAMETERS model_name='myRFModel', match_by_pos='true') FROM iris;
PREDICT_RF_CLASSIFIER
-----------------------
setosa
setosa
setosa
...
versicolor
versicolor
versicolor
...
virginica
virginica
virginica
...
(150 rows)
```

See also

- Classifying data using random forest

- RF_CLASSIFIER
- PREDICT_RF_CLASSIFIER_CLASSES
- GET_MODEL_SUMMARY

PREDICT_RF_CLASSIFIER_CLASSES

Applies a random forest model on an input relation and returns the probabilities of classes:

- VARCHAR predicted column contains the class label with the highest vote (popular vote).
- Multiple FLOAT columns, where the first probability column contains the probability for the class reported in the predicted column. Other columns contain the probability of each class specified in the classes parameter.
- Key columns with the same value and data type as matching input columns specified in parameter key_columns .

> **Note**
> Selection of the predicted class is based on the popular vote of decision trees in the forest. Thus, in special cases the calculated probability of the predicted class might not be the highest.

Syntax

```
PREDICT_RF_CLASSIFIER_CLASSES ( predictor-columns
    USING PARAMETERS model_name = 'model-name'
      [, key_columns = 'key-columns']
      [, exclude_columns = 'excluded-columns']
      [, classes = 'classes']
      [, match_by_pos = match-by-position] )
OVER( [window-partition-clause] )
```

Arguments

predictor-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

key_columns

Comma-separated list of predictor column names that identify the output rows. To exclude these and other predictor columns from being used for prediction, include them in the argument list for parameter exclude_columns .

exclude_columns

Comma-separated list of columns from *predictor-columns* to exclude from processing.

classes

Comma-separated list of class labels in the model. The probability of belonging to this given class is predicted by the classifier. Values are case sensitive.

match_by_pos

Boolean value that specifies how predictor columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the predictor columns list.

Examples

```
=> SELECT PREDICT_RF_CLASSIFIER_CLASSES(Sepal_Length, Sepal_Width, Petal_Length, Petal_Width
                        USING PARAMETERS model_name='myRFModel') OVER () FROM iris;
predicted |    probability
-----------+-------------------
setosa    |                1
setosa    |             0.99
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |             0.97
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |             0.99
...
(150 rows)
```

This example shows how to use function PREDICT_RF_CLASSIFIER_CLASSES , using the match_by_pos parameter:

```
=> SELECT PREDICT_RF_CLASSIFIER_CLASSES(Sepal_Length, Sepal_Width, Petal_Length, Petal_Width
                    USING PARAMETERS model_name='myRFModel', match_by_pos='true') OVER () FROM iris;
predicted |    probability
-----------+-------------------
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
setosa    |                1
...
(150 rows)s
```

See also

- Classifying data using random forest
- RF_CLASSIFIER
- PREDICT_RF_CLASSIFIER
- GET_MODEL_SUMMARY

PREDICT_RF_REGRESSOR

Applies a random forest model on an input relation, and returns with a FLOAT data type that specifies the predicted value of the random forest model —the average of the prediction of the trees in the forest.

Syntax

```
PREDICT_RF_REGRESSOR ( input-columns
    USING PARAMETERS model_name = 'model-name' [, match_by_pos = match-by-position] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Examples

```
=> SELECT PREDICT_RF_REGRESSOR (mpg,cyl,hp,drat,wt
USING PARAMETERS model_name='myRFRegressorModel')FROM mtcars;
PREDICT_RF_REGRESSOR
----------------------
2.94774203574204
2.6954087024087
2.6954087024087
2.89906346431346
2.97688489288489
2.97688489288489
2.7086587024087
2.92078965478965
2.97688489288489
2.7086587024087
2.95621822621823
2.82255155955156
2.7086587024087
2.7086587024087
2.85650394050394
2.85650394050394
2.97688489288489
2.95621822621823
2.6954087024087
2.6954087024087
2.84493251193251
2.97688489288489
2.97688489288489
2.8856467976468
2.6954087024087
2.92078965478965
2.97688489288489
2.97688489288489
2.7934087024087
2.7934087024087
2.7086587024087
2.72469441669442
(32 rows)
```

See also

- Building a random forest regression model
- GET_MODEL_SUMMARY
- RF_REGRESSOR

PREDICT_SVM_CLASSIFIER

Uses an SVM model to predict class labels for samples in an input relation, and returns the predicted value as a FLOAT data type.

Syntax

```
PREDICT_SVM_CLASSIFIER (input-columns
   USING PARAMETERS model_name = 'model-name'
      [, match_by_pos = match-by-position]
      [, type = 'return-type']
      [, cutoff = 'cutoff-value'] ] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

type

A string that specifies the output to return for each input row, one of the following:

- response : Outputs the predicted class of 0 or 1.
- probability : Outputs a value in the range (0,1), the prediction score transformed using the logistic function.

cutoff

Valid only if the type parameter is set to probability , a FLOAT value that is compared to the transformed prediction score to determine the predicted class.

Default: 0

Examples

```
=> SELECT PREDICT_SVM_CLASSIFIER (mpg,cyl,disp,wt,qsec,vs,gear,carb
USING PARAMETERS model_name='mySvmClassModel') FROM mtcars;
PREDICT_SVM_CLASSIFIER
------------------------
0
0
1
0
0
1
1
1
1
0
0
1
0
0
1
0
0
0
0
0
0
1
1
0
0
1
1
1
1
0
0
0
(32 rows)
```

This example shows how to use PREDICT_SVM_CLASSIFIER on the mtcars table, using the match_by_pos parameter. In this example, column mpg was replaced with the constant 40:

```
=> SELECT PREDICT_SVM_CLASSIFIER (40,cyl,disp,wt,qsec,vs,gear,carb
USING PARAMETERS model_name='mySvmClassModel', match_by_pos ='true') FROM mtcars;
PREDICT_SVM_CLASSIFIER
------------------------
0
0
0
0
1
0
0
1
1
1
1
1
0
0
0
1
1
1
1
0
0
0
0
0
0
0
0
1
1
0
0
1
(32 rows)
```

See also

- Classifying data using SVM (support vector machine)
- SVM (support vector machine) for classification
- SVM_CLASSIFIER
- GET_MODEL_SUMMARY

PREDICT_SVM_REGRESSOR

Uses an SVM model to perform regression on samples in an input relation, and returns the predicted value as a FLOAT data type.

Syntax

```
PREDICT_SVM_REGRESSOR(input-columns
    USING PARAMETERS model_name = 'model-name' [, match_by_pos = match-by-position] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Examples

```
=> SELECT PREDICT_SVM_REGRESSOR(waiting USING PARAMETERS model_name='mySvmRegModel')
        FROM faithful ORDER BY id;
 PREDICT_SVM_REGRESSOR
--------------------
  4.06488248694445
  2.30392277646291
  3.71269054484815
  2.867429883817
  4.48751281746003
  2.37436116488217
  4.69882798271781
  4.48751281746003
  2.09260761120512
...
 (272 rows)
```

This example shows how you can use the PREDICT_SVM_REGRESSOR function on the faithful table, using the match_by_pos parameter. In this example, the waiting column was replaced with the constant 40:

```
=> SELECT PREDICT_SVM_REGRESSOR(40 USING PARAMETERS model_name='mySvmRegModel', match_by_pos='true')
        FROM faithful ORDER BY id;
 PREDICT_SVM_REGRESSOR
--------------------
  1.31778533859324
  1.31778533859324
  1.31778533859324
  1.31778533859324
  1.31778533859324
  1.31778533859324
  1.31778533859324
  1.31778533859324
  1.31778533859324
...
 (272 rows)
```

See also

- Building an SVM for regression model
- SVM (support vector machine) for regression
- SVM_REGRESSOR
- GET_MODEL_SUMMARY

PREDICT_TENSORFLOW

Applies a TensorFlow model on an input relation, and returns with the result expected for the encoded model type.

Syntax

```
PREDICT_TENSORFLOW ( input-columns
      USING PARAMETERS model_name = 'model-name' [, num_passthru_cols = 'n-first-columns-to-ignore'] )
OVER( [window-partition-clause] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

num_passthru_cols

Integer that specifies the number of input columns to skip.

Examples

Use PREDICT_TENSORFLOW with the num_passthru_cols parameter to skip the first two input columns:

```
SELECT PREDICT_TENSORFLOW ( pid,label,x1,x2
      USING PARAMETERS model_name='spiral_demo', num_passthru_cols=2 )
OVER(PARTITION BEST) as predicted_class FROM points;

--example output, the skipped columns are displayed as the first columns of the output
 pid  | label |        col0         |        col1
------+-------+---------------------+---------------------
   0 |     0 |   0.990638732910156 |  0.00936129689216614
   1 |     0 |   0.999036073684692 | 0.000963933940511197
   2 |     1 |  0.0103802494704723 | 0.989619791507721
```

See also

- TensorFlow example
- Classifying data using naive bayes
- NAIVE_BAYES
- PREDICT_NAIVE_BAYES_CLASSES

PREDICT_XGB_CLASSIFIER

Applies an XGBoost classifier model on an input relation. PREDICT_XGB_CLASSIFIER returns a VARCHAR data type that specifies one of the following, as determined by how the type parameter is set:

- The predicted class (based on probability scores)
- Probability of a class for each input instance.

Syntax

```
PREDICT_XGB_CLASSIFIER ( input-columns
      USING PARAMETERS model_name = 'model-name'
        [, type = 'prediction-type' ]
        [, class = 'user-input-class' ]
        [, match_by_pos = 'match-by-position' ]
        [, probability_normalization = 'prob-normalization' ] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

type

Type of prediction to return, one of the following:

- response (default): The class with the highest probability among all possible classes.
- probability : Valid only if the class parameter is set, returns for each input instance the probability of the specified class or predicted class.

class

Class to use when the type parameter is set to probability . If you omit this parameter, the function uses the predicted class—the one with the highest probability score. Thus, the predict function returns the probability that the input instance belongs to the specified or predicted class.

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

probability_normalization

The classifier's normalization method, either softmax (multi-class classifier) or logit (binary classifier). If unspecified, the default logit function is used for normalization.

Examples

Use PREDICT_XGB_CLASSIFIER to apply the classifier to the test data:

```
=> SELECT PREDICT_XGB_CLASSIFIER (Sepal_Length, Sepal_Width, Petal_Length, Petal_Width
   USING PARAMETERS model_name='xgb_iris', probability_normalization='logit') FROM iris1;
PREDICT_XGB_CLASSIFIER
------------------------
setosa
setosa
setosa
.
.
.
versicolor
versicolor
versicolor
.
.
.
virginica
virginica
virginica
.
.
.

(90 rows)
```

See XGBoost for classification for more examples.

PREDICT_XGB_CLASSIFIER_CLASSES

Applies an XGBoost classifier model on an input relation and returns the probabilities of classes:

- VARCHAR predicted column contains the class label with the highest probability.
- Multiple FLOAT columns, where the first probability column contains the probability for the class reported in the predicted column. Other columns contain the probability of each class specified in the classes parameter.
- Key columns with the same value and data type as matching input columns specified in parameter key_columns .

All trees contribute to a predicted probability for each response class, and the highest probability class is chosen.

Syntax

```
PREDICT_XGB_CLASSIFIER_CLASSES ( predictor-columns)
      USING PARAMETERS model_name = 'model-name'
        [, key_columns = 'key-columns']
        [, exclude_columns = 'excluded-columns']
        [, classes = 'classes']
        [, match_by_pos = match-by-position]
        [, probability_normalization = 'prob-normalization' ] )
OVER( [window-partition-clause] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

key_columns

Comma-separated list of predictor column names that identify the output rows. To exclude these and other predictor columns from being used for prediction, include them in the argument list for parameter exclude_columns .

exclude_columns

Comma-separated list of columns from *predictor-columns* to exclude from processing.

classes

Comma-separated list of class labels in the model. The probability of belonging to each given class is predicted by the classifier. Values are case sensitive.

match_by_pos

Boolean value that specifies how predictor columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the predictor columns list.

probability_normalization

The classifier's normalization method, either softmax (multi-class classifier) or logit (binary classifier). If unspecified, the default logit function is used for normalization.

Examples

After creating an XGBoost classifier model with XGB_CLASSIFIER , you can use PREDICT_XGB_CLASSIFIER_CLASSES to view the probability of each classification. In this example, the XGBoost classifier model "xgb_iris" is used to predict the probability that a given flower belongs to a species of iris:

```
=> SELECT PREDICT_XGB_CLASSIFIER_CLASSES(Sepal_Length, Sepal_Width, Petal_Length, Petal_Width
   USING PARAMETERS model_name='xgb_iris') OVER (PARTITION BEST) FROM iris1;
 predicted |    probability
-----------+-------------------
 setosa    |  0.9999650465368
 setosa    |  0.9999650465368
 setosa    |  0.9999650465368
 setosa    |  0.9999650465368
 setosa    | 0.999911552783011
 setosa    |  0.9999650465368
 setosa    |  0.9999650465368
 setosa    |  0.9999650465368
 setosa    |  0.9999650465368
 setosa    |  0.9999650465368
 setosa    |  0.9999650465368
 setosa    |  0.9999650465368
 versicolor |  0.99991871763563
 .
 .
 .
(90 rows)
```

You can also specify additional classes. In this example, PREDICT_XGB_CLASSIFIER_CLASSES makes the same prediction as the previous example, but also returns the probability that a flower belongs to the specified classes "virginica" and "versicolor":

```
=> SELECT PREDICT_XGB_CLASSIFIER_CLASSES(Sepal_Length, Sepal_Width, Petal_Length, Petal_Width
   USING PARAMETERS model_name='xgb_iris', classes='virginica,versicolor', probability_normalization='logit') OVER (PARTITION BEST) FROM iris1;
 predicted |    probability    |      virginica       |      versicolor
-----------+-------------------+----------------------+----------------------
 setosa    |  0.9999650465368 | 1.16160301545536e-05 | 2.33374330460065e-05
 setosa    |  0.9999650465368 | 1.16160301545536e-05 | 2.33374330460065e-05
 setosa    |  0.9999650465368 | 1.16160301545536e-05 | 2.33374330460065e-05
 .
 .
 .
 versicolor |  0.99991871763563 | 6.45697562080953e-05 |     0.99991871763563
 versicolor | 0.999967282051702 | 1.60052775404199e-05 |    0.999967282051702
 versicolor | 0.999648819964864 |  0.00028366342010669 |    0.999648819964864
 .
 .
 .
 virginica  | 0.999977039257386 |    0.999977039257386 | 1.13305901169304e-05
 virginica  | 0.999977085131063 |    0.999977085131063 | 1.12847163501674e-05
 virginica  | 0.999977039257386 |    0.999977039257386 | 1.13305901169304e-05
(90 rows)
```

PREDICT_XGB_REGRESSOR

Applies an XGBoost regressor model on an input relation. PREDICT_XGB_REGRESSOR returns a FLOAT data type that specifies the predicted value by the XGBoost model: a weighted sum of contributions by each tree in the model.

Syntax

```
PREDICT_XGB_REGRESSOR ( input-columns
    USING PARAMETERS model_name = 'model-name' [, match_by_pos = match-by-position] )
```

Arguments

input-columns

Comma-separated list of columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

match_by_pos

Boolean value that specifies how input columns are matched to model features:

- false (default): Match by name.
- true : Match by the position of columns in the input columns list.

Examples

See XGBoost for regression .

REVERSE_NORMALIZE

Reverses the normalization transformation on normalized data, thereby de-normalizing the normalized data. If you specify a column that is not in the specified model, REVERSE_NORMALIZE returns that column unchanged.

Syntax

```
REVERSE_NORMALIZE ( input-columns USING PARAMETERS model_name = 'model-name' );
```

Arguments

input-columns

The columns to use from the input relation, or asterisk (*) to select all columns.

Parameters

model_name

Name of the model (case-insensitive).

Examples

Use REVERSE_NORMALIZE on the hp and cyl columns in table mtcars , where hp is in normalization model mtcars_normfit , and cyl is not in the normalization model.

```
=> SELECT REVERSE_NORMALIZE (hp, cyl USING PARAMETERS model_name='mtcars_normfit') FROM mtcars;
hp    | cyl
------+-----
42502 |   8
58067 |   8
26371 |   4
42502 |   8
31182 |   6
32031 |   4
26937 |   4
34861 |   6
34861 |   6
50992 |   8
50992 |   8
49577 |   8
25805 |   4
18447 |   4
29767 |   6
65142 |   8
69387 |   8
14768 |   4
49577 |   8
60897 |   8
94857 |   8
31182 |   6
31182 |   6
30899 |   4
69387 |   8
49577 |   6
18730 |   4
18730 |   4
74764 |   8
17598 |   4
50992 |   8
27503 |   4
(32 rows)
```

See also

- APPLY_NORMALIZE
- NORMALIZE
- NORMALIZE_FIT
- Normalizing data

Management functions

Vertica has functions to manage various aspects of database operation, such as sessions, privileges, projections, and the catalog.

In this section

- Catalog functions
- Cloud functions
- Cluster functions
- Data collector functions
- Database functions
- Eon Mode functions
- Epoch functions
- LDAP link functions

- License functions
- Notifier functions
- Partition functions
- Privileges and access functions
- Projection functions
- Session functions
- Storage functions
- Table functions

Catalog functions

This section contains catalog management functions specific to Vertica.

In this section

- DROP_LICENSE
- DUMP_CATALOG
- EXPORT_CATALOG
- EXPORT_OBJECTS
- EXPORT_TABLES
- INSTALL_LICENSE
- MARK_DESIGN_KSAFE
- RELOAD_ADMINTOOLS_CONF

DROP_LICENSE

Drops a license key from the global catalog. Dropping expired keys is optional. Vertica automatically ignores expired license keys if a valid, alternative license key is installed.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DROP_LICENSE( 'license-name' )
```

Parameters

license-name

The name of the license to drop. Use the name (or long license key) in the NAME column of system table LICENSES .

Privileges

Superuser

Examples

```
=> SELECT DROP_LICENSE('9b2d81e2-aab1-4cfb-bc07-fa9a696e8f5e');
```

See also

Managing licenses

DUMP_CATALOG

Returns an internal representation of the Vertica catalog. This function is used for diagnostic purposes.

DUMP_CATALOG returns only the objects that are visible to the user.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DUMP_CATALOG()
```

Privileges

None

Examples

The following query obtains an internal representation of the Vertica catalog:

```
=> SELECT DUMP_CATALOG();
```

The output is written to the specified file:

```
\o /tmp/catalog.txt
SELECT DUMP_CATALOG();
\o
```

EXPORT_CATALOG

> **Note**
> This function and EXPORT_OBJECTS return equivalent output.

Generates a SQL script for recreating a physical schema design on another cluster.

The SQL script conforms to the following requirements:

- Only includes objects to which the user has access.
- Orders CREATE statements according to object dependencies so they can be recreated in the correct sequence. For example, if a table is in a non-PUBLIC schema, the required CREATE SCHEMA statement precedes the CREATE TABLE statement. Similarly, a table's CREATE ACCESS POLICY statement follows the table's CREATE TABLE statement.
- If possible, creates projections with their KSAFE clause, if any, otherwise with their OFFSET clause.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
EXPORT_CATALOG ( [ '[destination]' [, 'scope'] ] )
```

Arguments

> **Note**
> If you omit all arguments, this function exports to standard output all objects to which you have access.

destination

Specifies where to send output, one of the following:

- Empty string, writes the script to standard output.
- Path and name of an SQL output file, valid only for superusers. If you specify a file that does not exist, the function creates one. If you specify only a file name, Vertica creates it in the catalog directory. If the file already exists, the function silently overwrites its contents.

scope

Determines what to export. Within the specified scope, EXPORT_CATALOG exports all the objects to which you have access:

- DESIGN: Exports all catalog objects, including schemas, tables, constraints, views, access policies, projections, SQL macros, and stored procedures.
- DESIGN_ALL: Deprecated.
- TABLES: Exports all tables and their access policies. See also EXPORT_TABLES.
- DIRECTED_QUERIES: Exports all directed queries that are stored in the database. For details, see Managing directed queries.

Default : DESIGN

Privileges

None

Examples

See Exporting the catalog.

See also

- EXPORT_OBJECTS
- EXPORT_TABLES
- Exporting the catalog

EXPORT_OBJECTS

> **Note**
> This function and EXPORT_CATALOG return equivalent output.

Generates a SQL script you can use to recreate non-virtual catalog objects on another cluster.

The SQL script conforms to the following requirements:

- Only includes objects to which the user has access.
- Orders CREATE statements according to object dependencies so they can be recreated in the correct sequence. For example, if a table is in a non-PUBLIC schema, the required CREATE SCHEMA statement precedes the CREATE TABLE statement. Similarly, a table's CREATE ACCESS POLICY statement follows the table's CREATE TABLE statement.
- If possible, creates projections with their KSAFE clause, if any, otherwise with their OFFSET clause.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
EXPORT_OBJECTS( ['[destination]' [, '[scope]'] [, 'mark-ksafe']] )
```

Parameters

> **Note**
> If you omit all parameters, this function exports to standard output all objects to which you have access.

destination

Specifies where to send output, one of the following:

- An empty string ('') writes the script to standard output.
- The path and name of a SQL output file. This option is valid only for superusers. If you specify a file that does not exist, the function creates one. If you specify only a file name, Vertica creates it in the catalog directory. If the file already exists, the function silently overwrites its contents.

scope

Specifies one or more objects to export as a comma-delimited list:

```
{ [database.]schema[.object] | [[database.]schema.]object }[,...]
```

- If set to an empty string, Vertica exports all objects to which the user has access.
- If you specify a schema only, Vertica exports all objects in that schema.
- If you specify a database, it must be the current database.

For stored procedures with the same name but different formal parameters, you can export all implementations by exporting its parent schema:

```
mydb.myschema
```

Specifying the types or both the names and types of a particular implementation's formal parameters exports that implementation:

```
mydb.myschema.my_procedure()  -- no formal parameters
mydb.myschema.my_procedure(int, int) -- formal parameter types (parameter names are optional)
```

mark-ksafe

Boolean argument, specifies whether the generated script calls the Vertica function MARK_DESIGN_KSAFE. If set to true (default), MARK_DESIGN_KSAFE uses the correct K-safe argument for the current database.

Privileges

None

Examples

See Exporting objects .

See also

- EXPORT_CATALOG
- EXPORT_TABLES

EXPORT_TABLES

Generates a SQL script that can be used to recreate a logical schema—schemas, tables, constraints, and views—on another cluster. EXPORT_TABLES only exports objects to which the user has access.

The SQL script conforms to the following requirements:

- Only includes objects to which the user has access.
- Orders CREATE statements according to object dependencies so they can be recreated in the correct sequence. For example, if a table references a named sequence, a CREATE SEQUENCE statement precedes the CREATE TABLE statement. Similarly, a table's CREATE ACCESS POLICY statement follows the table's CREATE TABLE statement.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
EXPORT_TABLES( ['[destination]' [, '[scope]']] )
```

> **Note**
> If you omit all parameters, EXPORT_CATALOG exports to standard output all tables to which you have access.

Parameters

destination

Specifies where to send output, one of the following:

- An empty string ('') writes the script to standard output.
- The path and name of a SQL output file. This option is valid only for superusers . If you specify a file that does not exist, the function creates one. If you specify only a file name, Vertica creates it in the catalog directory. If the file already exists, the function silently overwrites its contents.

scope

Specifies one or more tables to export, as follows:

```
[database.]schema[.table][,...]
```

- If set to an empty string, Vertica exports all non-virtual table objects to which you have access, including table schemas, sequences, and constraints.
- If you specify a schema, Vertica exports all non-virtual table objects in that schema.
- If you specify a database, it must be the current database.

Privileges

None

Examples

See Exporting tables .

See also

- Exporting tables
- EXPORT_CATALOG
- EXPORT_OBJECTS

INSTALL_LICENSE

Installs the license key in the global catalog.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
INSTALL_LICENSE( 'filename' )
```

Parameters

filename

The absolute path name of a valid license file.

Privileges

Superuser

Examples

```
=> SELECT INSTALL_LICENSE('/tmp/vlicense.dat');
```

See also

Managing licenses

MARK_DESIGN_KSAFE

Enables or disables high availability in your environment, in case of a failure. Before enabling recovery, MARK_DESIGN_KSAFE queries the catalog to determine whether a cluster's physical schema design meets the following requirements:

- Small, unsegmented tables are replicated on all nodes.
- Large table superprojections are segmented with each segment on a different node.
- Each large table projection has at least one buddy projection for K-safety=1 (or two buddy projections for K-safety=2).
  Buddy projections are also segmented across database nodes, but the distribution is modified so segments that contain the same data are distributed to different nodes. See High availability with projections .

MARK_DESIGN_KSAFE does not change the physical schema.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
MARK_DESIGN_KSAFE ( k )
```

Parameters

k

Specifies the level of K-safety, one of the following:

- 2: Enables high availability if the schema design meets requirements for K-safety=2
- 1: Enables high availability if the schema design meets requirements for K-safety=1
- 0: Disables high availability

Privileges

Superuser

Return messages

If you specify a *k* value of 1 or 2, Vertica returns one of the following messages.

Success:

```
Marked design n-safe
```

Failure:

```
The schema does not meet requirements for K=n.
Fact table projection projection-name
has insufficient "buddy" projections.
```

where *n* is a K-safety setting.

Notes

- The database's internal recovery state persists across database restarts but it is not checked at startup time.
- When one node fails on a system marked K-safe=1, the remaining nodes are available for DML operations.

Examples

```
=> SELECT MARK_DESIGN_KSAFE(1);
 mark_design_ksafe
----------------------
 Marked design 1-safe
(1 row)
```

If the physical schema design is not K-safe, messages indicate which projections do not have a buddy:

```
=> SELECT MARK_DESIGN_KSAFE(1);
The given K value is not correct;
the schema is 0-safe
Projection pp1 has 0 buddies,
which is smaller that the given K of 1
Projection pp2 has 0 buddies,
which is smaller that the given K of 1
.
.
.
(1 row)
```

See also

- Identical segmentation
- Failure recovery

RELOAD_ADMINTOOLS_CONF

Updates the admintools.conf on each UP node in the cluster. Updates include:

- IP addresses and catalog paths
- Node names for all nodes in the current database

This function provides a manual method to instruct the server to update admintools.conf on all UP nodes. For example, if you restart a node, call this function to confirm its admintools.conf file is accurate.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RELOAD_ADMINTOOLS_CONF()
```

Privileges

Superuser

Examples

Update admintools.conf on each UP node in the cluster:

```
=> SELECT RELOAD_ADMINTOOLS_CONF();
 RELOAD_ADMINTOOLS_CONF
---------------------------
 admintools.conf reloaded
(1 row)
```

Cloud functions

This section contains functions for managing cloud integrations. See also Hadoop functions for HDFS.

In this section

- AZURE_TOKEN_CACHE_CLEAR

AZURE_TOKEN_CACHE_CLEAR

Clears the cached access token for Azure. Call this function after changing the configuration of Azure managed identities.

An Azure object store can support and manage multiple identities. If multiple identities are in use, Vertica looks for an Azure tag with a key of VerticaManagedIdentityClientId, the value of which must be the client_id attribute of the managed identity to be used. If the Azure configuration changes, use this function to clear the cache.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
AZURE_TOKEN_CACHE_CLEAR ( )
```

Privileges

Superuser

Cluster functions

This section contains functions that manage spread deployment on large, distributed database clusters and functions that control how the cluster organizes data for rebalancing.

In this section

- CANCEL_REBALANCE_CLUSTER
- DISABLE_LOCAL_SEGMENTS
- ENABLE_ELASTIC_CLUSTER
- ENABLE_LOCAL_SEGMENTS
- REALIGN_CONTROL_NODES
- REBALANCE_CLUSTER
- RELOAD_SPREAD
- SET_CONTROL_SET_SIZE
- SET_SCALING_FACTOR
- START_REBALANCE_CLUSTER

CANCEL_REBALANCE_CLUSTER

Stops any rebalance task that is currently in progress or is waiting to execute.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CANCEL_REBALANCE_CLUSTER()
```

Privileges

Superuser

Examples

```
=> SELECT CANCEL_REBALANCE_CLUSTER();
 CANCEL_REBALANCE_CLUSTER
--------------------------
 CANCELED
(1 row)
```

See also

- START_REBALANCE_CLUSTER
- REBALANCE_CLUSTER

DISABLE_LOCAL_SEGMENTS

Disables local data segmentation, which breaks projections segments on nodes into containers that can be easily moved to other nodes. See Local data segmentation for details.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DISABLE_LOCAL_SEGMENTS()
```

Privileges

Superuser

Examples

```
=> SELECT DISABLE_LOCAL_SEGMENTS();
 DISABLE_LOCAL_SEGMENTS
------------------------
 DISABLED
(1 row)
```

ENABLE_ELASTIC_CLUSTER

Enables elastic cluster scaling, which makes enlarging or reducing the size of your database cluster more efficient by segmenting a node's data into chunks that can be easily moved to other hosts.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ENABLE_ELASTIC_CLUSTER()
```

Privileges

Superuser

Examples

```
=> SELECT ENABLE_ELASTIC_CLUSTER();
 ENABLE_ELASTIC_CLUSTER
------------------------
 ENABLED
(1 row)
```

ENABLE_LOCAL_SEGMENTS

Enables local storage segmentation, which breaks projections segments on nodes into containers that can be easily moved to other nodes. See Local data segmentation for more information.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ENABLE_LOCAL_SEGMENTS()
```

Privileges

Superuser

Examples

```
=> SELECT ENABLE_LOCAL_SEGMENTS();
 ENABLE_LOCAL_SEGMENTS
-----------------------
 ENABLED
(1 row)
```

REALIGN_CONTROL_NODES

Causes Vertica to re-evaluate which nodes in the cluster or subcluster are control nodes and which nodes are assigned to them as dependents when large cluster is enabled. Call this function after altering fault groups in an Enterprise Mode database, or changing the number of control nodes in either database mode. After calling this function, query the V_CATALOG.CLUSTER_LAYOUT system table to see the proposed new layout for nodes in the cluster. You must also take additional steps before the new control node assignments take effect. See Changing the number of control nodes and realigning for details.

> **Note**
> In Vertica versions prior to 10.0.1, control node assignments weren't restricted to be within the same Eon Mode subcluster. If you attempt to realign control nodes in a subcluster whose control nodes have dependents in other subclusters, this function returns an error. In this case, you must realign the control nodes in those other subclusters first. Realigning the other subclusters fixes the cross-subcluster dependencies, allowing you to realign the control nodes in the original subcluster you attempted to realign.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

In Enterprise Mode:

```
REALIGN_CONTROL_NODES()
```

In Eon Mode:

```
REALIGN_CONTROL_NODES('subcluster_name')
```

Parameters

subcluster_name

The name of the subcluster where you want to realign control nodes. Only the nodes in this subcluster are affected. Other subclusters are unaffected. Only allowed when the database is running in Eon Mode.

Privileges

Superuser

Examples

In an Enterprise Mode database, choose control nodes from all nodes and assign the remaining nodes to a control node:

```
=> SELECT REALIGN_CONTROL_NODES();
```

In an Eon Mode database, re-evaluate the control node assignments in the subcluster named analytics:

```
=> SELECT REALIGN_CONTROL_NODES('analytics');
```

See also

- Changing the number of control nodes and realigning
- Fault groups

REBALANCE_CLUSTER

Rebalances the database cluster synchronously as a session foreground task. REBALANCE_CLUSTER returns only after the rebalance operation is complete. If the current session ends, the operation immediately aborts. To rebalance the cluster as a background task, call START_REBALANCE_CLUSTER.

On large cluster arrangements, you typically call REBALANCE_CLUSTER in a flow (see Changing the number of control nodes and realigning). After you change the number and distribution of control nodes (spread hosts), run REBALANCE_CLUSTER to achieve fault tolerance.

For detailed information about rebalancing tasks, see Rebalancing data across nodes.

> **Tip**
> By default, before performing a rebalance, Vertica queries system tables to compute the size of all projections involved in the rebalance task. This query can add significant overhead to the rebalance operation. To disable this query, set projection configuration parameter RebalanceQueryStorageContainers to 0.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
REBALANCE_CLUSTER()
```

Privileges

Superuser

Examples

```
=> SELECT REBALANCE_CLUSTER();
REBALANCE_CLUSTER
-------------------
 REBALANCED
(1 row)
```

RELOAD_SPREAD

Updates cluster changes to the catalog's Spread configuration file. These changes include:

- New or realigned control nodes
- New Spread hosts or fault group
- New or dropped cluster nodes

This function is often used in a multi-step process for large and elastic cluster arrangements. Calling it might require you to restart the database. You must then rebalance the cluster to realize fault tolerance. For details, see Defining and Realigning Control Nodes .

Caution

In an Eon Mode database, using this function could result in the database becoming read-only. Nodes may become disconnected after you call this function. If the database no longer has primary shard coverage without these nodes, it goes into read-only mode to maintain data integrity. Once the nodes rejoin the cluster, the database will resume normal operation. See Maintaining Shard Coverage .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RELOAD_SPREAD( true )
```

Parameters

true

Updates cluster changes related to control message responsibilities to the Spread configuration file.

Privileges

Superuser

Examples

Update the cluster with changes to control messaging:

```
=> SELECT reload_spread(true);
 reload_spread
---------------
 reloaded
(1 row)
```

See also

REBALANCE_CLUSTER

SET_CONTROL_SET_SIZE

Sets the number of control nodes that participate in the spread service when large cluster is enabled. If the database is running in Enterprise Mode, this function sets the number of control nodes for the entire database cluster. If the database is running in Eon Mode, this function sets the number of control nodes in the subcluster you specify. See Large cluster for more information.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

In Enterprise Mode :

```
SET_CONTROL_SET_SIZE( control_nodes )
```

In Eon Mode:

```
SET_CONTROL_SET_SIZE('subcluster_name', control_nodes )
```

Parameters

subcluster_name

The name of the subcluster where you want to set the number of control nodes. Only allowed when the database is running in Eon Mode.

control_nodes

The number of control nodes to assign to the cluster (when in Enterprise Mode) or subcluster (when in Eon Mode). Value can be one of the following:

- Positive integer value: Vertica assigns the number of control nodes you specify to the cluster or subcluster. This value can be larger than the current node count. This value cannot be larger than 120 (the maximum number of control nodes for a database). In Eon Mode, the total of this value plus the number of control nodes set for all other subclusters cannot be more than 120.
- -1 : Makes every node in the cluster or subcluster into control nodes. This value effectively disables large cluster for the cluster or subcluster.

Privileges

Superuser

Examples

In an Enterprise Mode database, set the number of control nodes for the entire cluster to 5:

```
=> SELECT set_control_set_size(5);
 SET_CONTROL_SET_SIZE
----------------------
 Control size set
(1 row)
```

See also

- Changing the number of control nodes and realigning

SET_SCALING_FACTOR

Sets the scaling factor that determines the number of storage containers used when rebalancing the database and when using local data segmentation is enabled. See Cluster Scaling for details.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_SCALING_FACTOR( factor )
```

Parameters

factor

An integer value between 1 and 32. Vertica uses this value to calculate the number of storage containers each projection is broken into when rebalancing or when local data segmentation is enabled.

Privileges

Superuser

Best practices

The scaling factor determines the number of storage containers that Vertica uses to store each projection across the database during rebalancing when local segmentation is enabled. When setting the scaling factor, follow these guidelines:

- The number of storage containers should be greater than or equal to the number of partitions multiplied by the number of local segments: *num-storage-containers* >= (*num-partitions* * *num-local-segments*)
- Set the scaling factor high enough so rebalance can transfer local segments to satisfy the skew threshold, but small enough so the number of storage containers does not result in too many ROS containers, and cause ROS pushback . The maximum number of ROS containers (by default 1024) is set by configuration parameter ContainersPerProjectionLimit .

Examples

```
=> SELECT SET_SCALING_FACTOR(12);
 SET_SCALING_FACTOR
--------------------
 SET
(1 row)
```

START_REBALANCE_CLUSTER

Asynchronously rebalances the database cluster as a background task. This function returns immediately after the rebalancing operation is complete. Rebalancing persists until the operation is complete, even if you close the current session or the database shuts down. In the case of shutdown, rebalancing resumes after the cluster restarts. To stop the rebalance operation, call CANCEL_REBALANCE_CLUSTER .

For detailed information about rebalancing tasks, see Rebalancing data across nodes .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
START_REBALANCE_CLUSTER()
```

Privileges

Superuser

Examples

```
=> SELECT START_REBALANCE_CLUSTER();
 START_REBALANCE_CLUSTER
-------------------------
 REBALANCING
(1 row)
```

See also

REBALANCE_CLUSTER

Data collector functions

The Vertica Data Collector is a utility that extends system table functionality by providing a framework for recording events. It gathers and retains monitoring information about your database cluster and makes that information available in system tables, requiring few configuration parameter tweaks, and having negligible impact on performance.

Collected data is stored on disk in the DataCollector directory under the Vertica /catalog path. You can use the information the Data Collector retains to query the past state of system tables and extract aggregate information, as well as do the following:

- See what actions users have taken
- Locate performance bottlenecks
- Identify potential improvements to Vertica configuration

Data Collector works in conjunction with an advisor tool called Workload Analyzer , which intelligently monitors the performance of SQL queries and workloads and recommends tuning actions based on observations of the actual workload history.

By default, Data Collector is on and retains information for all sessions . If performance issues arise, a superuser can disable Data Collector by setting set configuration parameter EnableDataCollector to 0.

In this section

- CLEAR_DATA_COLLECTOR
- DATA_COLLECTOR_HELP
- FLUSH_DATA_COLLECTOR
- GET_DATA_COLLECTOR_POLICY

- SET_DATA_COLLECTOR_POLICY
- SET_DATA_COLLECTOR_TIME_POLICY

CLEAR_DATA_COLLECTOR

Clears all memory and disk records from Data Collector tables and logs, and resets collection statistics in system table DATA_COLLECTOR .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAR_DATA_COLLECTOR( [ 'component' ] )
```

Parameters

component

Clears memory and disk records for the specified component. If you provide no argument, the function clears memory and disk records for all components.

Query system table DATA_COLLECTOR for component names. For example:

```
=> SELECT DISTINCT component, description FROM data_collector WHERE component ilike '%Depot%' ORDER BY component;
  component    |          description
---------------+-------------------------------
 DepotEvictions | Files evicted from the Depot
 DepotFetches   | Files fetched to the Depot
 DepotUploads   | Files Uploaded from the Depot
(3 rows)
```

Privileges

Superuser

Examples

The following command clears memory and disk records for the ResourceAcquisitions component:

```
=> SELECT clear_data_collector('ResourceAcquisitions');
 clear_data_collector
----------------------
 CLEAR
(1 row)
```

The following command clears data collection for all components:

```
=> SELECT clear_data_collector();
 clear_data_collector
----------------------
 CLEAR
(1 row)
```

See also

Data collector utility

DATA_COLLECTOR_HELP

Returns online usage instructions about the Data Collector, the DATA_COLLECTOR system table, and the Data Collector control functions.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DATA_COLLECTOR_HELP()
```

Privileges

None

Returns

The DATA_COLLECTOR_HELP() function returns the following information:

```
=> SELECT DATA_COLLECTOR_HELP();

--------------------------------------------------------------------------------
Usage Data Collector
The data collector retains history of important system activities.
  This data can be used as a reference of what actions have been taken
    by users, but it can also be used to locate performance bottlenecks,
    or identify potential improvements to the Vertica configuration.
  This data is queryable via Vertica system tables.
Acccess a list of data collector components, and some statistics, by running:
  SELECT * FROM v_monitor.data_collector;

The amount of data retained by size and time can be controlled with several
functions.
  To just set the size amount:
    set_data_collector_policy(<component>,
                              <memory retention (KB)>,
                              <disk retention (KB)>);

  To set both the size and time amounts (the smaller one will dominate):
    set_data_collector_policy(<component>,
                              <memory retention (KB)>,
                              <disk retention (KB)>,
                              <interval>);

  To set just the time amount:
    set_data_collector_time_policy(<component>,
                                   <interval>);

  To set the time amount for all tables:
    set_data_collector_time_policy(<interval>);

The current retention policy for a component can be queried with:
  get_data_collector_policy(<component>);

Data on disk is kept in the "DataCollector" directory under the Vertica
\catalog path. This directory also contains instructions on how to load
the monitoring data into another Vertica database.

To move the data collector logs and instructions to other storage locations,
create labeled storage locations using add_location and then use:

  set_data_collector_storage_location(<storage_label>);

Additional commands can be used to configure the data collection logs.
The log can be cleared with:
clear_data_collector([<optional component>]);
The log can be synchronized with the disk storage using:
flush_data_collector([<optional component>]);
```

See also

- DATA_COLLECTOR
- TUNING_RECOMMENDATIONS
- Analyzing workloads
- Data collector utility

FLUSH_DATA_COLLECTOR

Waits until memory logs are moved to disk and then flushes the Data Collector, synchronizing the log with disk storage.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
FLUSH_DATA_COLLECTOR( [ 'component' ] )
```

Parameters

component

Flushes data for the specified component. If you omit this argument, the function flushes data for all components.

Query system table DATA_COLLECTOR for component names. For example:

```
=> SELECT DISTINCT component, description FROM data_collector WHERE component ilike '%Depot%' ORDER BY component;
  component    |          description
---------------+-------------------------------
 DepotEvictions | Files evicted from the Depot
 DepotFetches   | Files fetched to the Depot
 DepotUploads   | Files Uploaded from the Depot
(3 rows)
```

Privileges

Superuser

Examples

The following command flushes the Data Collector for the ResourceAcquisitions component:

```
=> SELECT flush_data_collector('ResourceAcquisitions');
 flush_data_collector
----------------------
 FLUSH
(1 row)
```

The following command flushes data collection for all components:

```
=> SELECT flush_data_collector();
 flush_data_collector
----------------------
 FLUSH
(1 row)
```

See also

Data collector utility

GET_DATA_COLLECTOR_POLICY

Retrieves a brief statement about the retention policy for the specified component.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_DATA_COLLECTOR_POLICY( 'component' )
```

Parameters

component

Returns the retention policy of the specified component.

Query system table DATA_COLLECTOR for component names. For example:

```
=> SELECT DISTINCT component, description FROM data_collector WHERE component ilike '%Depot%' ORDER BY component;
  component    |          description
---------------+-------------------------------
 DepotEvictions | Files evicted from the Depot
 DepotFetches   | Files fetched to the Depot
 DepotUploads   | Files Uploaded from the Depot
(3 rows)
```

Privileges

None

Examples

The following query returns the history of all resource acquisitions by specifying the ResourceAcquisitions component:

```
=> SELECT get_data_collector_policy('ResourceAcquisitions');
        get_data_collector_policy
-----------------------------------------------
 1000KB kept in memory, 10000KB kept on disk.
(1 row)
```

See also

- DATA_COLLECTOR
- Data collector utility

SET_DATA_COLLECTOR_POLICY

Updates the following retention policy properties for the specified component:

- MEMORY_BUFFER_SIZE_KB
- DISK_SIZE_KB
- INTERVAL_TIME

Before you change a retention policy, you can view its current settings by querying system table DATA_COLLECTOR or by calling meta-function GET_DATA_COLLECTOR_POLICY .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_DATA_COLLECTOR_POLICY('component', 'memory-buffer-size', 'disk-size' ['interval-time']  )
```

Parameters

component

Specifies the retention policy to update.

Query system table DATA_COLLECTOR for component names. For example:

```
=> SELECT DISTINCT component, description FROM data_collector WHERE component ilike '%Depot%' ORDER BY component;
  component    |          description
---------------+-------------------------------
 DepotEvictions | Files evicted from the Depot
 DepotFetches   | Files fetched to the Depot
 DepotUploads   | Files Uploaded from the Depot
(3 rows)
```

memory-buffer-size

Specifies in kilobytes the maximum amount of data that is buffered in memory before moving it to disk. The policy retention policy property MEMORY_BUFFER_SIZE_KB is set from this value.

> **Caution**
> If you set this parameter to 0, the function returns with a warning that the Data Collector cannot retain any data for this component in memory or on disk.

Consider setting this parameter to a high value in the following cases:

- Unusually high levels of data collection. If *memory-buffer-size* is set too low, the Data Collector might be unable to flush buffered data to disk fast enough to keep up with the activity level, which can lead to loss of in-memory data.
- Very large data collector records—for example, records with very long query strings. The Data Collector uses double-buffering, so it cannot retain in memory records that are more than 50 percent larger than *memory-buffer-size* .

disk-size

Specifies in kilobytes the maximum disk space allocated for this component's Data Collector table. The policy retention policy property DISK_SIZE_KB is set from this value. If set to 0, the Data Collector retains only as much component data as it can buffer in memory, as specified by *memory-buffer-size* .

interval-time

INTERVAL data type that specifies how long data of a given component is retained in that component's Data Collector table. The retention policy property INTERVAL_TIME is set from this value. If you set this parameter to a positive value, it also changes the policy property INTERVAL_SET to t (true).
For example, if you specify component TupleMoverEvents and set interval-time to an interval of two days ('2 days'::interval), the Data Collector table dc_tuple_mover_events retains records of Tuple Mover activity over the last 48 hours. Older Tuple Mover data are automatically dropped from this table.

> **Note**
> Setting a component's policy's INTERVAL_TIME property has no effect on how much data storage the Data Collector retains on disk for that component. Maximum disk storage capacity is determined by the DISK_SIZE_KB property. Setting the INTERVAL_TIME property only affects how long data is retained by the component's Data Collector table. For details, see Configuring data retention policies .

To disable the INTERVAL_TIME policy property, set this parameter to a negative integer. Doing so reverts two retention policy properties to their default settings:

- INTERVAL_SET: f
- INTERVAL_TIME: 0

With these two properties thus set, the component's Data Collector table retains data on all component events until it reaches its maximum limit, as set by retention policy property DISK_SIZE_KB.

Privileges

Superuser

Examples

See Configuring data retention policies .

SET_DATA_COLLECTOR_TIME_POLICY

Updates the retention policy property INTERVAL_TIME for the specified component. Calling this function has no effect on other properties of the same component. You can use this function to update the INTERVAL_TIME property of all component retention policies.

To set other retention policy properties, call SET_DATA_COLLECTOR_POLICY .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_DATA_COLLECTOR_TIME_POLICY( ['component',] 'interval-time' )
```

Parameters

component

Specifies the retention policy to update. If you omit this argument, Vertica updates the retention policy of all Data Collector components.

Query system table DATA_COLLECTOR for component names. For example:

```
=> SELECT DISTINCT component, description FROM data_collector WHERE component ilike '%Depot%' ORDER BY component;
  component    |          description
---------------+-------------------------------
 DepotEvictions | Files evicted from the Depot
 DepotFetches   | Files fetched to the Depot
 DepotUploads   | Files Uploaded from the Depot
(3 rows)
```

interval-time

INTERVAL data type that specifies how long data of a given component is retained in that component's Data Collector table. The retention policy property INTERVAL_TIME is set from this value. If you set this parameter to a positive value, it also changes the policy property INTERVAL_SET to t (true).
For example, if you specify component TupleMoverEvents and set interval-time to an interval of two days ('2 days'::interval), the Data Collector table dc_tuple_mover_events retains records of Tuple Mover activity over the last 48 hours. Older Tuple Mover data are automatically dropped from this table.

> **Note**
> Setting a component's policy's INTERVAL_TIME property has no effect on how much data storage the Data Collector retains on disk for that component. Maximum disk storage capacity is determined by the DISK_SIZE_KB property. Setting the INTERVAL_TIME property only affects how long data is retained by the component's Data Collector table. For details, see Configuring data retention policies .

To disable the INTERVAL_TIME policy property, set this parameter to a negative integer. Doing so reverts two retention policy properties to their default settings:

- INTERVAL_SET: f
- INTERVAL_TIME: 0

With these two properties thus set, the component's Data Collector table retains data on all component events until it reaches its maximum limit, as set by retention policy property DISK_SIZE_KB.

Privileges

Superuser

Examples

See Configuring data retention policies .

Database functions

This section contains the database management functions specific to Vertica.

In this section

- CLEAR_RESOURCE_REJECTIONS
- COMPACT_STORAGE
- DUMP_LOCKTABLE
- DUMP_PARTITION_KEYS
- GET_CONFIG_PARAMETER
- KERBEROS_CONFIG_CHECK
- MEMORY_TRIM
- PURGE
- RUN_INDEX_TOOL
- SECURITY_CONFIG_CHECK
- SET_CONFIG_PARAMETER
- SET_SPREAD_OPTION
- SHUTDOWN

CLEAR_RESOURCE_REJECTIONS

Clears the content of the RESOURCE_REJECTIONS and DISK_RESOURCE_REJECTIONS system tables. Normally, these tables are only cleared during a node restart. This function lets you clear the tables whenever you need. For example, you might want to clear the system tables after you resolved a disk space issue that was causing disk resource rejections.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Immutable

Syntax

```
CLEAR_RESOURCE_REJECTIONS();
```

Privileges

Superuser

Examples

The following command clears the content of the RESOURCE_REJECTIONS and DISK_RESOURCE_REJECTIONS system tables:

```
=> SELECT clear_resource_rejections();
clear_resource_rejections
---------------------------
 OK
(1 row)
```

See also

- DISK_RESOURCE_REJECTIONS
- RESOURCE_REJECTIONS

COMPACT_STORAGE

Bundles existing data (.fdb) and index (.pidx) files into the .gt file format. The .gt format is enabled by default for data files created version 7.2 or later. If you upgrade a database from an earlier version, use COMPACT_STORAGE to bundle storage files into the .gt format. Your database can continue to operate with a mix of file storage formats.

If the settings you specify for COMPACT_STORAGE vary from the limit specified in configuration parameter MaxBundleableROSSizeKB , Vertica does not change the size of the automatically created bundles.

> **Note**
> Run this function during periods of low demand.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SELECT COMPACT_STORAGE ('[[[database.]schema.]object-name]', min-ros-filesize-kb, 'small-or-all-files', 'simulate');
```

Parameters

[database .] schema

Database and schema . The default schema is public . If you specify a database, it must be the current database.

object-name

Specifies the table or projection to bundle. If set to an empty string, COMPACT_STORAGE evaluates the data of all projections in the database for bundling.

min-ros-filesize-kb

Integer ≥ 1, specifies in kilobytes the minimum size of an independent ROS file. COMPACT_STORAGE bundles storage container ROS files below this size into a single file.

small-or-all-files

One of the following:

- small : Bundles only files smaller than the limit specified in *min-ros-filesize-kb*
- all : Bundles files smaller than the limit specified in *min-ros-filesize-kb* and bundles the .fdb and .pidx files for larger storage containers.

simulate

Specifies whether to simulate the storage settings and produce a report describing the impact of those settings.

- true : Produces a report on the impact of the specified bundle settings without actually bundling storage files.
- false : Performs the bundling as specified.

Privileges

Superuser

Storage and performance impact

Bundling reduces the number of files in your file system by at least fifty percent and improves the performance of file-intensive operations. Improved operations include backups, restores, and mergeout.

Vertica creates small files for the following reasons:

- Tables contain hundreds of columns.
- Partition ranges are small (partition by minute).
- Local segmentation is enabled and your factor is set to a high value.

Examples

The following example describes the impact of bundling the table EMPLOYEES :

```
=> SELECT COMPACT_STORAGE('employees', 1024,'small','true');
Task: compact_storage

On node v_vmart_node0001:
Projection Name :public.employees_b0 | selected_storage_containers :0 |
selected_files_to_compact :0 | files_after_compact : 0 | modified_storage_KB :0

On node v_vmart_node0002:
Projection Name :public.employees_b0 | selected_storage_containers :1 |
selected_files_to_compact :6 | files_after_compact : 1 | modified_storage_KB :0

On node v_vmart_node0003:
Projection Name :public.employees_b0 | selected_storage_containers :2 |
selected_files_to_compact :12 | files_after_compact : 2 | modified_storage_KB :0

On node v_vmart_node0001:
Projection Name :public.employees_b1 | selected_storage_containers :2 |
selected_files_to_compact :12 | files_after_compact : 2 | modified_storage_KB :0

On node v_vmart_node0002:
Projection Name :public.employees_b1 | selected_storage_containers :0 |
selected_files_to_compact :0 | files_after_compact : 0 | modified_storage_KB :0

On node v_vmart_node0003:
Projection Name :public.employees_b1 | selected_storage_containers :1 |
selected_files_to_compact :6 | files_after_compact : 1 | modified_storage_KB :0

Success

(1 row)
```

DUMP_LOCKTABLE

Returns information about deadlocked clients and the resources they are waiting for.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DUMP_LOCKTABLE()
```

Privileges

None

Notes

Use DUMP_LOCKTABLE if Vertica becomes unresponsive:

1. Open an additional vsql connection.
2. Execute the query:

```
=> SELECT DUMP_LOCKTABLE();
```

   The output is written to vsql. See Monitoring the Log Files .

You can also see who is connected using the following command:

```
=> SELECT * FROM SESSIONS;
```

Close all sessions using the following command:

```
=> SELECT CLOSE_ALL_SESSIONS();
```

Close a single session using the following command:

```
=> SELECT CLOSE_SESSION('session_id');
```

You get the session_id value from the V_MONITOR.SESSIONS system table.

See also

- CLOSE_ALL_SESSIONS
- CLOSE_SESSION
- LOCKS
- SESSIONS

DUMP_PARTITION_KEYS

Dumps the partition keys of all projections in the system.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DUMP_PARTITION_KEYS( )
```

> **Note**
> The ROS objects of partitioned tables without partition keys are ignored by the tuple mover and are not merged during automatic tuple mover operations.

Privileges

User must have select privileges on the table or usage privileges on the schema.

Examples

```
=> SELECT DUMP_PARTITION_KEYS( );
Partition keys on node v_vmart_node0001
  Projection 'states_b0'
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: NH
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: MA
  Projection 'states_b1'
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: VT
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: ME
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: CT
```

See also

- DUMP_PROJECTION_PARTITION_KEYS
- DUMP_TABLE_PARTITION_KEYS
- PARTITION_PROJECTION
- PARTITION_TABLE
- PARTITIONS
- Partitioning tables in the Administrator's Guide

GET_CONFIG_PARAMETER

Gets the value of a configuration parameter at the specified level. If no value is set at that level, the function returns an empty row.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_CONFIG_PARAMETER( 'parameter-name' [, 'level' | NULL] )
```

Parameters

parameter-name

Name of the configuration parameter value to get.

level

Level at which to get *parameter-name* 's setting, one of the following string values:

- user : Current user
- session : Current session
- *node-name* : Name of database node

If *level* is omitted or set to NULL, GET_CONFIG_PARAMETER returns the database setting.

Privileges

None

Examples

Get the AnalyzeRowCountInterval parameter at the database level:

```
=> SELECT GET_CONFIG_PARAMETER ('AnalyzeRowCountInterval');
 GET_CONFIG_PARAMETER
----------------------
 3600
```

Get the MaxSessionUDParameterSize parameter at the session level:

```
=> SELECT GET_CONFIG_PARAMETER ('MaxSessionUDParameterSize','session');
 GET_CONFIG_PARAMETER
----------------------
 2000
(1 row)
```

Get the UseDepotForReads parameter at the user level:

```
=> SELECT GET_CONFIG_PARAMETER ('UseDepotForReads', 'user');
 GET_CONFIG_PARAMETER
----------------------
 1
(1 row)
```

See also

- SET_CONFIG_PARAMETER
- CONFIGURATION_PARAMETERS

KERBEROS_CONFIG_CHECK

Tests the Kerberos configuration of a Vertica cluster. The function succeeds if it can kinit with both the keytab file and the current user's credential, and reports errors otherwise.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
KERBEROS_CONFIG_CHECK( )
```

Parameters

This function has no parameters.

Privileges

This function does not require privileges.

Examples

The following example shows the results when the Kerberos configuration is valid.

```
=> SELECT KERBEROS_CONFIG_CHECK();
   kerberos_config_check
-------------------------------------------------------------------------------
 ok: krb5 exists at [/etc/krb5.conf]
 ok: Vertica Keytab file is set to [/etc/vertica.keytab]
 ok: Vertica Keytab file exists at [/etc/vertica.keytab]
[INFO] KerberosCredentialCache [/tmp/vertica_D4/vertica450676899262134963.cc]
 Kerberos configuration parameters set in the database
       KerberosServiceName : [vertica]
       KerberosHostname : [data.hadoop.com]
       KerberosRealm : [EXAMPLE.COM]
       KerberosKeytabFile : [/etc/vertica.keytab]
 Vertica Principal: [vertica/data.hadoop.com@EXAMPLE.COM]
 [OK] Vertica can kinit using keytab file
 [OK] User [bob] has valid client authentication for kerberos principal [bob@EXAMPLE.COM]]

(1 row)
```

MEMORY_TRIM

Calls glibc function malloc_trim() to reclaim free memory from malloc and return it to the operating system. Details on the trim operation are written to system table MEMORY_EVENTS .

Unless you turn off memory polling, Vertica automatically detects when glibc accumulates an excessive amount of free memory in its allocation arena. When this occurs, Vertica consolidates much of this memory and returns it to the operating system. Call this function if you disable memory polling and wish to reduce glibc-allocated memory manually.

For more information, see Memory trimming .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
MEMORY_TRIM()
```

Privileges

Superuser

Examples

```
=> SELECT memory_trim();
                        memory_trim
-------------------------------------------------------------------
 Pre-RSS: [378822656] Post-RSS: [372129792] Benefit: [0.0176675]
(1 row)
```

PURGE

Permanently removes delete vectors from ROS storage containers so disk space can be reused. PURGE removes all historical data up to and including the Ancient History Mark epoch.

PURGE does not delete temporary tables.

> **Caution**
> PURGE can temporarily use significant disk space.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SELECT PURGE()
```

Privileges

- Table owner
- USAGE privilege on schema

Examples

After you delete data from a Vertica table, that data is marked for deletion. To see the data that is marked for deletion, query system table DELETE_VECTORS .

Run PURGE to remove the delete vectors from ROS containers.

```
=> SELECT * FROM test1;
 number
--------
     3
    12
    33
    87
    43
    99
(6 rows)
=> DELETE FROM test1 WHERE number > 50;
 OUTPUT
--------
     2
(1 row)
=> SELECT * FROM test1;
 number
--------
    43
     3
    12
    33
(4 rows)
=> SELECT node_name, projection_name, deleted_row_count FROM DELETE_VECTORS;
    node_name     | projection_name | deleted_row_count
------------------+-----------------+-------------------
 v_vmart_node0002 | test1_b1        |                 1
 v_vmart_node0001 | test1_b1        |                 1
 v_vmart_node0001 | test1_b0        |                 1
 v_vmart_node0003 | test1_b0        |                 1
(4 rows)
=> SELECT PURGE();
...
(Table: public.test1) (Projection: public.test1_b0)
(Table: public.test1) (Projection: public.test1_b1)
...
(4 rows)
```

After the ancient history mark (AHM) advances:

```
=> SELECT * FROM DELETE_VECTORS;
 (No rows)
```

See also

- Purging deleted data
- PURGE_PARTITION
- PURGE_PROJECTION
- PURGE_TABLE

RUN_INDEX_TOOL

Runs the Index tool on a Vertica database to perform one of these tasks:

- Run a per-block cyclic redundancy check (CRC) on data storage to verify data integrity.
- Check that the sort order in ROS containers is correct.

The function writes summary information about its operation to standard output; detailed information on results is logged in vertica.log on the current node. For more about evaluating tool output, see:

- Evaluating CRC errors
- Evaluating sort order errors

You can also run the Index tool on a database that is down, from the Linux command line. For details, see CRC and sort order check .

> **Caution**
> Use this function only under guidance from Vertica Support.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RUN_INDEX_TOOL ( 'taskType', global, '[projFilter]' [, numThreads ] );
```

Parameters

taskType

Specifies the operation to run, one of the following:

- checkcrc : Run a cyclic redundancy check (CRC) on each block of existing data storage to check the data integrity of ROS data blocks.
- checksort : Evaluate each ROS row to determine whether it is sorted correctly. If ROS data is not sorted correctly in the projection's order, query results that rely on sorted data will be incorrect.

global

Boolean, specifies whether to run the specified task on all nodes (true), or the current one (false).

projFilter

Specifies the scope of the operation:

- Empty string (''): Run the check on all projections.
- A string that specifies one or more projections as follows:
  - *projection-name* : Run the check on this projection
  - *projection-prefix* * : Run the check on all projections that begin with the string *projection-prefix* .

numThreads

An unsigned (positive) or signed (negative) integer that specifies the number of threads used to run this operation:

- *n* : Number of threads, ≥ 1
- - *n* : Negative integer, denotes a fraction of all CPU cores as follows:

  ```
  num-cores / n
  ```

  Thus, -1 specifies all cores, -2 , half the cores, -3 , a third of all cores, and so on.
  Default: 1

Privileges

Superuser

Optimizing performance

You can optimize meta-function performance by setting two parameters:

- *projFilter* : Narrows the scope of the operation to one or more projections.
- *numThreads* : Specifies the number of threads used to execute the function.

SECURITY_CONFIG_CHECK

Returns the status of various security-related parameters. Use this function to verify completeness of your TLS configuration.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SECURITY_CONFIG_CHECK( 'db-component' )
```

Parameters

db-component

The component to check. Currently, NETWORK is the only supported component.

NETWORK : Returns the status and parameters for spread encryption, internode TLS, and client-server TLS.

Examples

In this example, SECURITY_CONFIG_CHECK shows that spread encryption and data channel TLS are disabled because EncryptSpreadComm is disabled and the data_channel TLS Configuration is not configured.

Similarly, client-server TLS is disabled because the TLS Configuration "server" has a server certificate, but its TLSMODE is disabled. Setting TLSMODE to 'Enable' enables server mode client-server TLS. See TLS protocol for details.

```
=> SELECT SECURITY_CONFIG_CHECK('NETWORK');
                                    SECURITY_CONFIG_CHECK
-------------------------------------------------------------------------------------------------------------------
Spread security details:
* EncryptSpreadComm = []
Spread encryption is disabled
It is NOT safe to set/change other security config parameters while spread is not encrypted!
Please set EncryptSpreadComm to enable spread encryption first

Data Channel security details:
 TLS Configuration 'data_channel' TLSMODE is DISABLE
TLS on the data channel is disabled
Please set EncryptSpreadComm and configure TLS Configuration 'data_channel' to enable TLS on the data channel

Client-Server network security details:
* TLS Configuration 'server' TLSMODE is DISABLE
* TLS Configuration 'server' has a certificate set
Client-Server TLS is disabled
To enable Client-Server TLS set a certificate on TLS Configuration 'server' and/or set the tlsmode to 'ENABLE' or higher

(1 row)
```

See also

- Internode TLS
- TLS protocol

SET_CONFIG_PARAMETER

Sets or clears a configuration parameter at the specified level.

Important

You can only use this function to set configuration parameters with string or integer values. To set configuration parameters that accept other data types, use the appropriate ALTER statement .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_CONFIG_PARAMETER( 'param-name', { param-value | NULL}, ['level'| NULL])
```

Arguments

param-name

Name of the configuration parameter to set.

param-value

Value to set for *param-name* , either a string or integer. If a string, enclose in single quotes; if an integer, single quotes are optional.

To clear *param-name* at the specified level, set to NULL.

level

Level at which to set *param-name* , one of the following string values:

- user : Current user.
- session : Current session, overrides the database setting.
- *node-name* : Name of database node, overrides session and database settings.

If *level* is omitted or set to NULL, *param-name* is set at the database level.

> **Note**
> Some parameters require restart for the value to take effect.

Privileges

Superuser

Examples

Set the AnalyzeRowCountInterval parameter to 3600 at the database level:

```
=> SELECT SET_CONFIG_PARAMETER('AnalyzeRowCountInterval',3600);
   SET_CONFIG_PARAMETER
----------------------------
 Parameter set successfully
(1 row)
```

> **Note**
> You can achieve the same result with ALTER DATABASE:
>
> ```
> ALTER DATABASE DEFAULT SET PARAMETER AnalyzeRowCountInterval = 3600;
> ```

Set the MaxSessionUDParameterSize parameter to 2000 at the session level.

```
=> SELECT SET_CONFIG_PARAMETER('MaxSessionUDParameterSize',2000,'SESSION');
   SET_CONFIG_PARAMETER
----------------------------
 Parameter set successfully
(1 row)
```

See also

- GET_CONFIG_PARAMETER
- CONFIGURATION_PARAMETERS

SET_SPREAD_OPTION

Changes spread daemon settings. This function is mainly used to set the timeout before spread assumes a node has gone down.

> **Note**
> Changing Spread settings with SET_SPREAD_OPTION has minor impact on your cluster as it pauses while the new settings are propagated across the cluster. Because of this delay, changes to the Spread timeout are not immediately visible in system table SPREAD_STATE .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_SPREAD_OPTION( option-name, option-value )
```

Parameters

option-name

String containing the spread daemon setting to change.

Currently, this function supports only one option: TokenTimeout . This setting controls how long spread waits for a node to respond to a message before assuming it is lost. See Adjusting Spread Daemon timeouts for virtual environments for more information.

option-value

The new setting for *option-name* .

Examples

```
=> SELECT SET_SPREAD_OPTION( 'TokenTimeout', '35000');
NOTICE 9003:  Spread has been notified about the change
                SET_SPREAD_OPTION
-------------------------------------------------------
 Spread option 'TokenTimeout' has been set to '35000'.

(1 row)

=> SELECT * FROM V_MONITOR.SPREAD_STATE;
   node_name     | token_timeout
-----------------+---------------
 v_vmart_node0001 |        35000
 v_vmart_node0002 |        35000
 v_vmart_node0003 |        35000
(3 rows);
```

See also

- Adjusting Spread Daemon timeouts for virtual environments
- SPREAD_STATE

SHUTDOWN

Shuts down a Vertica database. By default, the shutdown fails if any users are connected. You can check the status of the shutdown operation in the vertica.log file.

In Eon Mode, you can call SHUTDOWN_WITH_DRAIN to perform a graceful shutdown that drains client connections and then shuts down the database.

> **Tip**
> Before calling SHUTDOWN, you can close all current user connections and prevent further connection attempts as follows:
>
> 1. Temporarily set configuration parameter MaxClientSessions to 0.
> 2. Call CLOSE_ALL_SESSIONS to close all non-dbamin connections.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SHUTDOWN ( [ 'false' | 'true' ] )
```

Parameters

false

Default, returns a message if users are connected and aborts the shutdown.

true

Forces the database to shut down, disallowing further connections.

Privileges

Superuser

Examples

The following command attempts to shut down the database. Because users are connected, the command fails:

```
=> SELECT SHUTDOWN('false');
NOTICE:  Cannot shut down while users are connected
        SHUTDOWN
-----------------------------
 Shutdown: aborting shutdown
(1 row)
```

See also

SESSIONS

Eon Mode functions

The following functions are meant to be used in Eon Mode.

In this section

- ALTER_LOCATION_SIZE
- BACKGROUND_DEPOT_WARMING
- CANCEL_DEPOT_WARMING
- CANCEL_DRAIN_SUBCLUSTER
- CLEAN_COMMUNAL_STORAGE
- CLEAR_DATA_DEPOT
- CLEAR_DEPOT_PIN_POLICY_PARTITION
- CLEAR_DEPOT_PIN_POLICY_PROJECTION
- CLEAR_DEPOT_PIN_POLICY_TABLE
- CLEAR_FETCH_QUEUE
- DEMOTE_SUBCLUSTER_TO_SECONDARY
- FINISH_FETCHING_FILES
- FLUSH_REAPER_QUEUE
- MIGRATE_ENTERPRISE_TO_EON
- PROMOTE_SUBCLUSTER_TO_PRIMARY
- REBALANCE_SHARDS
- RESHARD_DATABASE
- SANDBOX_SUBCLUSTER
- SET_DEPOT_PIN_POLICY_PARTITION
- SET_DEPOT_PIN_POLICY_PROJECTION
- SET_DEPOT_PIN_POLICY_TABLE
- SHUTDOWN_SUBCLUSTER
- SHUTDOWN_WITH_DRAIN
- START_DRAIN_SUBCLUSTER
- START_REAPING_FILES
- SYNC_CATALOG
- UNSANDBOX_SUBCLUSTER

ALTER_LOCATION_SIZE

Eon Mode only

Resizes the depot on one node, all nodes in a subcluster, or all nodes in the database.

Important

Reducing the size of the depot is liable to increase contention over depot usage and require frequent evictions. This behavior can increase the number of queries and load operations that are routed to communal storage for processing, which can incur slower performance and increased access charges.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Immutable

Syntax

```
ALTER_LOCATION_SIZE( 'location', '[target]', 'size')
```

Parameters

location

Specifies the location to resize, one of the following:

- depot : Resizes the node's current depot.
- The depot's absolute path in the Linux filesystem. If you change the depot size on multiple nodes and specify a path, the path must be identical on all affected nodes . By default, this is not the case, as the node's name is typically this path. For example, the default depot path for node 1 in the verticadb database is /vertica/data/verticadb/v_verticadb_node0001_depot .

target

The node or nodes on which to change the depot, one of the following:

- Node name: Resize the specified node.
- Subcluster name: Resize depots of all nodes in the specified subcluster.
- Empty string: Resize all depots in the database.

size

Valid only if the storage location usage type is set to DEPOT , specifies the maximum amount of disk space that the depot can allocate from the storage location's file system.
You can specify *size* in two ways:

- *integer* % : Percentage of storage location disk size.
- *integer* {K|M|G|T} : Amount of storage location disk size in kilobytes, megabytes, gigabytes, or terabytes.

Important
The depot size cannot exceed 80 percent of the file system disk space where the depot is stored. If you specify a value that is too large, Vertica issues a warning and automatically changes the value to 80 percent of the file system size.

Privileges

Superuser

Examples

Increase depot size on all nodes to 80 percent of file system:

```
=> SELECT node_name, location_label, location_path, max_size, disk_percent FROM storage_locations WHERE location_usage = 'DEPOT' ORDER BY
node_name;
    node_name     | location_label |      location_path      |  max_size   | disk_percent
------------------+----------------+-------------------------+-------------+--------------
 v_vmart_node0001 | auto-data-depot | /home/dbadmin/verticadb | 36060108800 | 70%
 v_vmart_node0002 | auto-data-depot | /home/dbadmin/verticadb | 36059377664 | 70%
 v_vmart_node0003 | auto-data-depot | /home/dbadmin/verticadb | 36060108800 | 70%
(3 rows)

=> SELECT alter_location_size('depot', '','80%');
 alter_location_size
---------------------
 depotSize changed.
(1 row)

=> SELECT node_name, location_label, location_path, max_size, disk_percent FROM storage_locations WHERE location_usage = 'DEPOT' ORDER BY
node_name;
    node_name     | location_label |      location_path      |  max_size   | disk_percent
------------------+----------------+-------------------------+-------------+--------------
 v_vmart_node0001 | auto-data-depot | /home/dbadmin/verticadb | 41211552768 | 80%
 v_vmart_node0002 | auto-data-depot | /home/dbadmin/verticadb | 41210717184 | 80%
 v_vmart_node0003 | auto-data-depot | /home/dbadmin/verticadb | 41211552768 | 80%
(3 rows)
```

Change the depot size to 75% of the filesystem size for all nodes in the analytics subcluster:

```
=> SELECT subcluster_name, subclusters.node_name, storage_locations.max_size, storage_locations.disk_percent FROM subclusters INNER JOIN
storage_locations ON subclusters.node_name = storage_locations.node_name WHERE storage_locations.location_usage='DEPOT';
  subcluster_name   |      node_name       |   max_size  | disk_percent
--------------------+----------------------+-------------+--------------
 default_subcluster | v_verticadb_node0001 | 25264737485 | 60%
 default_subcluster | v_verticadb_node0002 | 25264737485 | 60%
 default_subcluster | v_verticadb_node0003 | 25264737485 | 60%
 analytics          | v_verticadb_node0004 | 25264737485 | 60%
 analytics          | v_verticadb_node0005 | 25264737485 | 60%
 analytics          | v_verticadb_node0006 | 25264737485 | 60%
 analytics          | v_verticadb_node0007 | 25264737485 | 60%
 analytics          | v_verticadb_node0008 | 25264737485 | 60%
 analytics          | v_verticadb_node0009 | 25264737485 | 60%
(9 rows)

=> SELECT ALTER_LOCATION_SIZE('depot','analytics','75%');
 ALTER_LOCATION_SIZE
---------------------
 depotSize changed.
(1 row)

=> SELECT subcluster_name, subclusters.node_name, storage_locations.max_size, storage_locations.disk_percent FROM subclusters INNER JOIN
storage_locations ON subclusters.node_name = storage_locations.node_name WHERE storage_locations.location_usage='DEPOT';
  subcluster_name   |      node_name       |   max_size  | disk_percent
--------------------+----------------------+-------------+--------------
 default_subcluster | v_verticadb_node0001 | 25264737485 | 60%
 default_subcluster | v_verticadb_node0002 | 25264737485 | 60%
 default_subcluster | v_verticadb_node0003 | 25264737485 | 60%
 analytics          | v_verticadb_node0004 | 31580921856 | 75%
 analytics          | v_verticadb_node0005 | 31580921856 | 75%
 analytics          | v_verticadb_node0006 | 31580921856 | 75%
 analytics          | v_verticadb_node0007 | 31580921856 | 75%
 analytics          | v_verticadb_node0008 | 31580921856 | 75%
 analytics          | v_verticadb_node0009 | 31580921856 | 75%
(9 rows)
```

See also

Eon Mode architecture
BACKGROUND_DEPOT_WARMING

Eon Mode only

> **Deprecated**
> Vertica version 10.0.0 removes support for foreground depot warming. When enabled, depot warming always happens in the background. Because foreground depot warming no longer exists, this function serves no purpose and has been deprecated. Calling it has no effect.

Forces a node that is warming its depot to start processing queries while continuing to warm its depot in the background. Depot warming only occurs when a node is joining the database and is activating its subscriptions. This function only has an effect if:

- The database is running in Eon Mode.
- The node is currently warming its depot.
- The node is warming its depot from communal storage. This is the case when the UseCommunalStorageForBatchDepotWarming configuration parameter is set to the default value of 1. See Eon Mode parameters for more information about this parameter.

After calling this function, the node warms its depot in the background while taking part in queries.

This function has no effect on a node that is not warming its depot.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
BACKGROUND_DEPOT_WARMING('node-name' [, 'subscription-name'])
```

Arguments

node-name

The name of the node that you want to warm its depot in the background.

subscription-name

The name of a shard that the node subscribes to that you want the node to warm in the background. You can find the names of the shards a node subscribes to in the SHARD_NAME column of the NODE_SUBSCRIPTIONS system table.

> **Note**
> When you supply the name of a specific shard subscription to warm in the background, the node may not immediately begin processing queries. It continues to warm any other shard subscriptions in the foreground if they are not yet warm. The node does not begin taking part in queries until it finishes warming the other subscriptions.

Return value

A message indicating that the node's warming will continue in the background.

Privileges

The user must be a superuser .

Examples

The following example demonstrates having node 6 of the verticadb database warm its depot in the background:

```
=> SELECT BACKGROUND_DEPOT_WARMING('v_verticadb_node0006');
                    BACKGROUND_DEPOT_WARMING
-------------------------------------------------------------------------------
 Depot warming running in background. Check monitoring tables for progress.
(1 row)
```

See also

- CANCEL_DEPOT_WARMING
- ALTER_LOCATION_SIZE
- CANCEL_DRAIN_SUBCLUSTER
- CLEAN_COMMUNAL_STORAGE
- CLEAR_DATA_DEPOT

CANCEL_DEPOT_WARMING

Eon Mode only

Cancels depot warming on a node. Depot warming only occurs when a node is joining the database and is activating its subscriptions. You can choose to cancel all warming on the node, or cancel the warming of a specific shard's subscription. The node finishes whatever data transfers it is currently carrying out to warm its depot and removes pending warming-related transfers from its queue. It keeps any data it has already loaded into its depot. If you cancel warming for a specific subscription, it stops warming its depot if all of its other subscriptions are warmed. If they aren't warmed, the node continues to warm those other subscriptions.

This function only has an effect if:

- The database is running in Eon Mode.
- The node is currently warming its depot.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CANCEL_DEPOT_WARMING('node-name' [, 'subscription-name'])
```

Arguments

' ***node-name*** '

The name of the node whose depot warming you want canceled.

' ***subscription-name*** '

The name of a shard that the node subscribes to that you want the node to stop warming. You can find the names of the shards a node subscribes to in the SHARD_NAME column of the NODE_SUBSCRIPTIONS system table.

Return value

Returns a message indicating warming has been canceled.

Privileges

The user must be a superuser.

Usage considerations

Canceling depot warming can negatively impact the performance of your queries. A node with a cold depot may have to retrieve much of its data from communal storage, which is slower than accessing the depot.

Examples

The following demonstrates canceling the depot warming taking place on node 7:

```
=> SELECT CANCEL_DEPOT_WARMING('v_verticadb_node0007');
  CANCEL_DEPOT_WARMING
--------------------------
 Depot warming cancelled.
(1 row)
```

See also

- BACKGROUND_DEPOT_WARMING
- ALTER_LOCATION_SIZE
- CANCEL_DRAIN_SUBCLUSTER
- CLEAN_COMMUNAL_STORAGE
- CLEAR_DATA_DEPOT

CANCEL_DRAIN_SUBCLUSTER

Eon Mode only

Cancels the draining of a subcluster or subclusters. This function can cancel draining operations that were started by either START_DRAIN_SUBCLUSTER or the draining portion of the SHUTDOWN_WITH_DRAIN function. CANCEL_DRAIN_SUBCLUSTER marks all nodes in the designated subclusters as not draining. The previously draining nodes again accept new client connections and connections redirected from load-balancing.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CANCEL_DRAIN_SUBCLUSTER( 'subcluster-name' )
```

Arguments

subcluster-name

Name of the subcluster whose draining operation to cancel. Enter an empty string to cancel the draining operation on all subclusters.

Privileges

Superuser

Examples

The following example demonstrates how to cancel a draining operation on a subcluster.

First, you can query the DRAINING_STATUS system table to view which subclusters are currently draining:

```
=> SELECT node_name, subcluster_name, is_draining FROM draining_status ORDER BY 1;
node_name          | subcluster_name    | is_draining
-------------------+--------------------+-------
verticadb_node0001 | default_subcluster | f
verticadb_node0002 | default_subcluster | f
verticadb_node0003 | default_subcluster | f
verticadb_node0004 | analytics          | t
verticadb_node0005 | analytics          | t
verticadb_node0006 | analytics          | t
```

The following function call cancels the draining of the analytics subcluster:

```
=> SELECT CANCEL_DRAIN_SUBCLUSTER('analytics');
            CANCEL_DRAIN_SUBCLUSTER
-----------------------------------------------------------
Targeted subcluster: 'analytics'
Action: CANCEL DRAIN

(1 row)
```

To confirm that the subcluster is no longer draining, you can again query the DRAINING_STATUS system table:

```
=> SELECT node_name, subcluster_name, is_draining FROM draining_status ORDER BY 1;
node_name          | subcluster_name    | is_draining
-------------------+--------------------+-------
verticadb_node0001 | default_subcluster | f
verticadb_node0002 | default_subcluster | f
verticadb_node0003 | default_subcluster | f
verticadb_node0004 | analytics          | f
verticadb_node0005 | analytics          | f
verticadb_node0006 | analytics          | f
(6 rows)
```

See also

- Drain client connections
- START_DRAIN_SUBCLUSTER
- DRAINING_STATUS

CLEAN_COMMUNAL_STORAGE

Eon Mode only

Marks for deletion invalid data in communal storage, often data that leaked due to an event where Vertica cleanup mechanisms failed. Events that require calling this function include:

- Node failure
- Interrupted migration of an Enterprise database to Eon
- Restoring objects from backup

> **Tip**
> It is generally good practice to call CLEAN_COMMUNAL_STORAGE soon after completing an Enterprise-to-Eon migration , and reviving the migrated Eon database.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAN_COMMUNAL_STORAGE ( ['actually-delete'] )
```

Parameters

actually-delete

BOOLEAN, specifies whether to queue data files for deletion:

- true (default): Add files to the reaper queue and return immediately. The queued files are removed automatically by the reaper service, or can be removed manually by calling FLUSH_REAPER_QUEUE .
- false : Report information about extra files but do not queue them for deletion.

Privileges

Superuser

Examples

```
=> SELECT CLEAN_COMMUNAL_STORAGE('true')
CLEAN_COMMUNAL_STORAGE
------------------------------------------------------------------------
CLEAN COMMUNAL STORAGE
Task was canceled.
Total leaked files: 9265
Total size: 4236501526
Files have been queued for deletion.
Check communal_cleanup_records for more information.
(1 row)
```

CLEAR_DATA_DEPOT

Eon Mode only

Deletes the specified depot data. You can clear depot data of a single table or all tables, from one subcluster, a single node, or the entire database cluster. Clearing depot data has no effect on communal storage.

> **Note**
> Clearing depot data can incur extra processing time for any subsequent queries that require that data and must now fetch it from communal storage.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAR_DATA_DEPOT( [ '[table-name]' [, '[target-depots]'] ] )
```

Arguments

> **Note**
> To clear all depot data from the database cluster, call this function with no arguments.

table-name

Name of the table to delete from the target depots. If you omit a table name or supply an empty string, data of all tables is deleted from the target depots.

target-depots

Specifies to clear all data from the specified depots, one of the following:

- *subcluster-name* : Clears depot data from the specified subcluster.
- *node-name* : Clears depot data from the specified node. Depot data on other nodes in the same subcluster are unaffected.

This argument optionally qualifies the argument for *table-name* . If you omit this argument or supply an empty string, Vertica clears all depot data from the database cluster.

Privileges

Superuser

Examples

Clear all depot data for table t1 table from the depot of subcluster subcluster_1 :

```
=> SELECT CLEAR_DATA_DEPOT('t1', 'subcluster_1');
 clear_data_depot
------------------
 Depot cleared
(1 row)
```

Clear all depot data from subcluster subcluster_1 :

```
=> SELECT CLEAR_DATA_DEPOT('', 'subcluster_1');
 clear_data_depot
------------------
 Depot cleared
(1 row)
```

Clear all depot data from a single node:

```
=> select clear_data_depot('','v_vmart_node0001');
 clear_data_depot
------------------
 Depot cleared
(1 row)
```

Clear all depot data for table t1 from the database cluster:

```
=> SELECT CLEAR_DATA_DEPOT('t1');
 clear_data_depot
------------------
 Depot cleared
(1 row)
```

Clear all depot data from the database cluster:

```
=> SELECT CLEAR_DATA_DEPOT();
 clear_data_depot
------------------
 Depot cleared
(1 row)
```

CLEAR_DEPOT_PIN_POLICY_PARTITION

Eon Mode only

Clears a depot pinning policy from the specified table or projection partitions. After the object is unpinned, it can be evicted from the depot by any unpinned or pinned object..

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAR_DEPOT_PIN_POLICY_PARTITION( '[[database.]schema.]object-name', 'min-range-value', 'max-range-value' [, subcluster ] )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

object-name

The table or projection with a partition pinning policy to clear.

min-range-value max-range-value

Clears a pinning policy from the specified range of partition keys in *table* , where *min-range-value* must be ≤ *max-range-value* . If the policy applies to a single partition, *min-range-value* and *max-range-value* must be equal.

subcluster

Clears the specified pinning policy from the *subcluster* depot. If you omit this parameter, the policy is cleared from all database depots.

Privileges

Superuser

See also

- Managing depot caching
- SET_DEPOT_PIN_POLICY_PARTITION

CLEAR_DEPOT_PIN_POLICY_PROJECTION

Eon Mode only

Clears a depot pinning policy from the specified projection. After the object is unpinned, it can be evicted from the depot by any unpinned or pinned object.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAR_DEPOT_PIN_POLICY_PROJECTION( '[[database.]schema.]projection' [, 'subcluster' ] )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection

Projection with a pinning policy to clear.

subcluster

Clears the specified pinning policy from the *subcluster* depot. If you omit this parameter, the policy is cleared from all database depots.

Privileges

Superuser

See also

- Managing depot caching
- CLEAR_DEPOT_PIN_POLICY_PARTITION
- SET_DEPOT_PIN_POLICY_TABLE

CLEAR_DEPOT_PIN_POLICY_TABLE

Eon Mode only

Clears a depot pinning policy from the specified table. After the object is unpinned, it can be evicted from the depot by any unpinned or pinned object.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAR_DEPOT_PIN_POLICY_TABLE( '[[database.]schema.]table' [, 'subcluster' ] )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Table with a pinning policy to clear.

subcluster

Clears the specified pinning policy from the *subcluster* depot. If you omit this parameter, the policy is cleared from all database depots.

Privileges

Superuser

See also

- Managing depot caching
- SET_DEPOT_PIN_POLICY_TABLE

CLEAR_FETCH_QUEUE

Eon Mode only

Removes all entries or entries for a specific transaction from the queue of fetch requests of data from the communal storage. You can view the fetch queue by querying the DEPOT_FETCH_QUEUE system table. This function removes all of the queued requests synchronously. It returns after all the fetches have been removed from the queue.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAR_FETCH_QUEUE([transaction_id])
```

Parameters

transaction_id

The id of the transaction whose fetches will be cleared from the queue. If this value is not specified, all fetches are removed from the fetch queue.

Examples

This example clears all of the queued fetches for all transactions.

```
=> SELECT CLEAR_FETCH_QUEUE();

   CLEAR_FETCH_QUEUE

--------------------------

Cleared the fetch queue.

(1 row)
```

This example clears the fetch queue for a specific transaction.

```
=> SELECT node_name,transaction_id FROM depot_fetch_queue;
     node_name      |  transaction_id
--------------------+-------------------
 v_verticadb_node0001 | 45035996273719510
 v_verticadb_node0003 | 45035996273719510
 v_verticadb_node0002 | 45035996273719510
 v_verticadb_node0001 | 45035996273719777
 v_verticadb_node0003 | 45035996273719777
 v_verticadb_node0002 | 45035996273719777

(6 rows)

=> SELECT clear_fetch_queue(45035996273719510);
   clear_fetch_queue
-------------------------
 Cleared the fetch queue.
(1 row)

=> SELECT node_name,transaction_id from depot_fetch_queue;
     node_name      |  transaction_id
--------------------+-------------------
 v_verticadb_node0001 | 45035996273719777
 v_verticadb_node0003 | 45035996273719777
 v_verticadb_node0002 | 45035996273719777

(3 rows)
```

DEMOTE_SUBCLUSTER_TO_SECONDARY

Eon Mode only

Converts a primary subcluster to a secondary subcluster .

Vertica will not allow you to demote a primary subcluster if any of the following are true:

- The subcluster contains a critical node .
- The subcluster is the only primary subcluster in the database. You must have at least one primary subcluster.
- The initiator node is a member of the subcluster you are trying to demote. You must call DEMOTE_SUBCLUSTER_TO_SECONDARY from another subcluster.

Important

This function call can take a long time to complete because all the nodes in the subcluster you are promoting or demoting take a global catalog lock, write a checkpoint, and then commit. This global catalog lock can cause other database tasks to fail with errors.

Schedule calls to this function when other database activity is low.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DEMOTE_SUBCLUSTER_TO_SECONDARY('subcluster-name')
```

Parameters

subcluster-name

The name of the primary subcluster to demote to a secondary subcluster.

Privileges

Superuser

Examples

The following example demotes the subcluster analytics_cluster to a secondary subcluster:

```
=> SELECT DISTINCT subcluster_name, is_primary from subclusters;
  subcluster_name  | is_primary
-------------------+------------
 analytics_cluster | t
 load_subcluster   | t
(2 rows)

=> SELECT DEMOTE_SUBCLUSTER_TO_SECONDARY('analytics_cluster');
 DEMOTE_SUBCLUSTER_TO_SECONDARY
--------------------------------
 DEMOTE SUBCLUSTER TO SECONDARY
(1 row)

=> SELECT DISTINCT subcluster_name, is_primary from subclusters;
  subcluster_name  | is_primary
-------------------+------------
 analytics_cluster | f
 load_subcluster   | t
(2 rows)
```

Attempting to demote the subcluster that contains the initiator node results in an error:

```
=> SELECT node_name FROM sessions WHERE user_name = 'dbadmin'
   AND client_type = 'vsql';
      node_name
----------------------
 v_verticadb_node0004
(1 row)

=> SELECT node_name, is_primary FROM subclusters WHERE subcluster_name = 'analytics';
      node_name       | is_primary
----------------------+------------
 v_verticadb_node0004 | t
 v_verticadb_node0005 | t
 v_verticadb_node0006 | t
(3 rows)

=> SELECT DEMOTE_SUBCLUSTER_TO_SECONDARY('analytics');
ERROR 9204:  Cannot promote or demote subcluster including the initiator node
HINT:  Run this command on another subcluster
```

See also

- PROMOTE_SUBCLUSTER_TO_PRIMARY
- SHUTDOWN_SUBCLUSTER
- Subclusters
- ALTER SUBCLUSTER
- CRITICAL_SUBCLUSTERS

FINISH_FETCHING_FILES

Eon Mode only

Fetches to the depot all files that are queued for download from communal storage.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
FINISH_FETCHING_FILES()
```

Privileges

Superuser

Examples

Get all files queued for download:

```
=> SELECT FINISH_FETCHING_FILES();
     FINISH_FETCHING_FILES
---------------------------------
 Finished fetching all the files
(1 row)
```

See also

Eon Mode concepts
FLUSH_REAPER_QUEUE

Eon Mode only

Deletes all data marked for deletion in the database. Use this function to remove all data marked for deletion before the reaper service deletes disk files.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
FLUSH_REAPER_QUEUE( [sync-catalog] )
```

Parameters

sync-catalog

Specifies to sync metadata in the database catalog on all nodes before the function executes:

- true (default): Sync the database catalog
- false : Run without syncing.

Privileges

Superuser

Examples

Remove all files that are marked for deletion:

```
=> SELECT FLUSH_REAPER_QUEUE();
              FLUSH_REAPER_QUEUE
-----------------------------------------------------
 Sync'd catalog and deleted all files in the reaper queue.
(1 row)
```

See also

CLEAN_COMMUNAL_STORAGE
MIGRATE_ENTERPRISE_TO_EON

Enterprise Mode only

Migrates an Enterprise database to an Eon Mode database. MIGRATE_ENTERPRISE_TO_EON runs in the foreground; until it returns—either with success or an error—it blocks all operations in the same session on the source Enterprise database. If successful, MIGRATE_ENTERPRISE_TO_EON returns with a list of nodes in the migrated database.

If migration is interrupted before the meta-function returns—for example, the client disconnects, or a network outage occurs—the migration returns an error. In this case, call MIGRATE_ENTERPRISE_TO_EON again to restart migration. For details, see Handling Interrupted Migration .

You can repeat migration multiple times to the same communal storage location—for example, to capture changes that occurred in the source database during the previous migration. For details, see Repeating Migration .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
MIGRATE_ENTERPRISE_TO_EON ( 'communal-storage-location', 'depot-location' [, is-dry-run] )
```

communal-storage-location

URI of communal storage location. For URI syntax examples for each supported schema, see File systems and object stores .

depot-location

Path of Eon depot location, typically:

```
/vertica/depot
```

Important

Management Console requires this convention to enable access to depot data and activity.

is-dry-run

Boolean. If set to true, MIGRATE_ENTERPRISE_TO_EON only checks whether the Enterprise source database complies with all migration prerequisites . If the meta-function discovers any compliance issues, it writes these to the migration error log migrate_enterprise_to_eon_error.log in the database directory.

Default: false

Privileges

Superuser

Examples

Migrate an Enterprise database to Eon Mode on AWS:

```
=> SELECT MIGRATE_ENTERPRISE_TO_EON ('s3://verticadbbucket', '/vertica/depot');
                migrate_enterprise_to_eon
-------------------------------------------------------------------------
 v_vmart_node0001,v_vmart_node0002,v_vmart_node0003,v_vmart_node0004
(1 row)
```

See also

Migrating an enterprise database to Eon Mode

PROMOTE_SUBCLUSTER_TO_PRIMARY

Eon Mode only

Converts a secondary subcluster to a primary subcluster . You cannot use this function to promote the subcluster that contains the initiator node . You must call it while connected to a node in another subcluster.

Important

This function call can take a long time to complete because all the nodes in the subcluster you are promoting or demoting take a global catalog lock, write a checkpoint, and then commit. This global catalog lock can cause other database tasks to fail with errors.

Schedule calls to this function when other database activity is low.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
PROMOTE_SUBCLUSTER_TO_PRIMARY('subcluster-name')
```

Parameters

subcluster-name

The name of the secondary cluster to promote to a primary subcluster.

Privileges

Superuser

Examples

The following example promotes the subcluster named analytics_cluster to a primary cluster:

```
=> SELECT DISTINCT subcluster_name, is_primary from subclusters;
 subcluster_name  | is_primary
------------------+------------
 analytics_cluster | f
 load_subcluster   | t
(2 rows)


=> SELECT PROMOTE_SUBCLUSTER_TO_PRIMARY('analytics_cluster');
 PROMOTE_SUBCLUSTER_TO_PRIMARY
-------------------------------
 PROMOTE SUBCLUSTER TO PRIMARY
(1 row)


=> SELECT DISTINCT subcluster_name, is_primary from subclusters;
 subcluster_name  | is_primary
------------------+------------
 analytics_cluster | t
 load_subcluster   | t
(2 rows)
```

See also

- DEMOTE_SUBCLUSTER_TO_SECONDARY
- SHUTDOWN_SUBCLUSTER
- Subclusters
- ALTER SUBCLUSTER
- CRITICAL_SUBCLUSTERS

REBALANCE_SHARDS

Eon Mode only

Rebalances shard assignments in a subcluster or across the entire cluster in Eon Mode. If the current session ends, the operation immediately aborts. The amount of time required to rebalance shards scales in a roughly linear fashion based on the number of objects in your database.

Run REBALANCE_SHARDS after you modify your cluster using ALTER NODE or when you add nodes to a subcluster.

> **Note**
> Vertica rebalances shards in a subcluster automatically when you:
>
> - Remove a node from a subcluster.
> - Add a new subcluster with the admintools command db_add_subcluster with the -s option followed by a list of hosts.

After you rebalance shards, you will no longer be able to restore objects from a backup taken before the rebalancing. (Full backups are always possible.) After you rebalance, make another full backup so you will be able to restore objects from it in the future.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
REBALANCE_SHARDS(['subcluster-name'])
```

Parameters

subcluster-name

The name of the subcluster where shards will be rebalanced. If you do not supply this parameter, all subclusters in the database rebalance their shards.

Privileges

Superuser

Examples

The following shows that the nodes in the in the newly-added analytics subcluster do not yet have shard subscriptions. It then calls REBALANCE_SHARDS to update the node's subscriptions:

```
=> SELECT subcluster_name, n.node_name, shard_name, subscription_state FROM
   v_catalog.nodes n LEFT JOIN v_catalog.node_subscriptions ns ON (n.node_name
   = ns.node_name) ORDER BY 1,2,3;

   subcluster_name    |      node_name       | shard_name  | subscription_state
----------------------+----------------------+-------------+--------------------
 analytics_subcluster | v_verticadb_node0004 |             |
 analytics_subcluster | v_verticadb_node0005 |             |
 analytics_subcluster | v_verticadb_node0006 |             |
 default_subcluster   | v_verticadb_node0001 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0001 | segment0001 | ACTIVE
 default_subcluster   | v_verticadb_node0001 | segment0003 | ACTIVE
 default_subcluster   | v_verticadb_node0002 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0002 | segment0001 | ACTIVE
 default_subcluster   | v_verticadb_node0002 | segment0002 | ACTIVE
 default_subcluster   | v_verticadb_node0003 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0003 | segment0002 | ACTIVE
 default_subcluster   | v_verticadb_node0003 | segment0003 | ACTIVE
(12 rows)
```

```
=> SELECT REBALANCE_SHARDS('analytics_subcluster');
 REBALANCE_SHARDS
-------------------
 REBALANCED SHARDS
(1 row)

=> SELECT subcluster_name, n.node_name, shard_name, subscription_state FROM
   v_catalog.nodes n LEFT JOIN v_catalog.node_subscriptions ns ON (n.node_name
   = ns.node_name) ORDER BY 1,2,3;

   subcluster_name    |      node_name       | shard_name  | subscription_state
----------------------+----------------------+-------------+--------------------
 analytics_subcluster | v_verticadb_node0004 | replica     | ACTIVE
 analytics_subcluster | v_verticadb_node0004 | segment0001 | ACTIVE
 analytics_subcluster | v_verticadb_node0004 | segment0003 | ACTIVE
 analytics_subcluster | v_verticadb_node0005 | replica     | ACTIVE
 analytics_subcluster | v_verticadb_node0005 | segment0001 | ACTIVE
 analytics_subcluster | v_verticadb_node0005 | segment0002 | ACTIVE
 analytics_subcluster | v_verticadb_node0006 | replica     | ACTIVE
 analytics_subcluster | v_verticadb_node0006 | segment0002 | ACTIVE
 analytics_subcluster | v_verticadb_node0006 | segment0003 | ACTIVE
 default_subcluster   | v_verticadb_node0001 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0001 | segment0001 | ACTIVE
 default_subcluster   | v_verticadb_node0001 | segment0003 | ACTIVE
 default_subcluster   | v_verticadb_node0002 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0002 | segment0001 | ACTIVE
 default_subcluster   | v_verticadb_node0002 | segment0002 | ACTIVE
 default_subcluster   | v_verticadb_node0003 | replica     | ACTIVE
 default_subcluster   | v_verticadb_node0003 | segment0002 | ACTIVE
 default_subcluster   | v_verticadb_node0003 | segment0003 | ACTIVE
(18 rows)
```

See also

- Shards and subscriptions
- Eon Mode concepts

RESHARD_DATABASE

Eon Mode only

Changes the number of shards in a database. This function requires a global catalog lock (GCLX) during runtime. The runtime depends on the size of your catalog.

RESHARD_DATABASE does not immediately affect the storage containers in communal storage. After re-sharding, the new shards still point to the existing containers. If you increase the number of shards in your database, multiple shards will point to the same storage containers. Eventually, the Tuple Mover (TM) mergeout tasks will realign the storage containers with the new shard segmentation bounds. If you want the TM to immediately realign storage containers, call DO_TM_TASK to run a 'RESHARDMERGEOUT' task.

This function does not disrupt most queries. However, the global catalog lock might affect data loads and DDL statements.

Important
RESHARD_DATABASE might be rolled back if you call REBALANCE_SHARDS during runtime. In some cases, rollback is caused by down nodes or nodes that fail during the re-shard process.

Syntax

```
RESHARD_DATABASE(shard-count)
```

Arguments

shard-count

A positive integer, the number of shards in the re-sharded database. For information about choosing a suitable *shard-count* , see Choosing the Number of Shards and the Initial Node Count .

Privileges

Superuser

Examples

See Change the number of shards in the database .

See also

- RESHARDING_EVENTS

SANDBOX_SUBCLUSTER

Creates a sandbox for a secondary subcluster.

> **Note**
> Vertica recommends using the admintools sandbox_subcluster command to create sandboxes. This command includes additional sanity checks and validates that the sandboxed nodes are UP after sandbox creation. For details, see Creating sandboxes .

At function runtime, the nodes in the specified subcluster create a checkpoint of the catalog. When these nodes auto-restart in the sandbox, they form a new primary cluster that uses the data and catalog checkpoint from the main cluster. After the nodes successfully restart, the sandboxed subcluster and the main cluster are mutually isolated and can diverge.

While the nodes in the main cluster sync their metadata to */path-to-communal-storage/`metadata`/db_name* , the nodes in the sandbox sync to */path-to-communal-storage/`metadata`/sandbox_name* .

You can perform standard database operations and queries, such as loading data or creating new tables, in either cluster without affecting the other cluster. For example, dropping a table in the sandboxed subcluster does not drop the table in the main cluster, and vice versa.

Because both clusters reference the same data files, neither cluster can delete files that existed at the time of sandbox creation. However, files that are created in the sandbox can be removed. Files in the main cluster can be queued for removal, but they are not processed until all active sandboxes are removed.

You cannot nest sandboxes, but you can have multiple individual sandboxes active at the same time.

This is a meta-function. You must call-meta-functions in a top-level SELECT statement. The function also requires a global catalog lock (GCLX) during runtime.

Behavior type

Volatile

Syntax

```
SANDBOX_SUBCLUSTER( 'sandbox-name', 'subcluster-name', 'options' )
```

Arguments

sandbox-name

Name of the sandbox. The name must conform to the following rules:

- Consist of at most 30 characters, all of which must have an ASCII code between 36 and 126
- Begin with a letter
- Unique among all existing databases and sandboxes

subcluster-name

Name of the secondary subcluster to sandbox. Attempting to sandbox a primary subcluster or a subcluster that is already sandboxed results in an error. The nodes in the subcluster must all have a status of UP and provide full subscription coverage for all shards.

options

Currently, there are no options for this function.

Privileges

Superuser

Examples

The following example sandboxes the sc02 secondary subcluster into a sandbox named sand :

```
=> SELECT SANDBOX_SUBCLUSTER('sand', 'sc_02', '');
                          SANDBOX_SUBCLUSTER
-------------------------------------------------------------------------------------------
Subcluster 'sc_02' has been sandboxed to 'sand'. It is going to auto-restart and re-form.
(1 row)
```

If you query the NODES system table from the main cluster, you can see that the nodes of sc_02 have a status of UNKNOWN and are listed as member of the sand sandbox:

```
=> SELECT node_name, subcluster_name, node_state, sandbox FROM NODES;
     node_name      |  subcluster_name   | node_state | sandbox
--------------------+--------------------+------------+---------
 v_verticadb_node0001 | default_subcluster | UP         |
 v_verticadb_node0002 | default_subcluster | UP         |
 v_verticadb_node0003 | default_subcluster | UP         |
 v_verticadb_node0004 | sc_02              | UNKNOWN    | sand
 v_verticadb_node0005 | sc_02              | UNKNOWN    | sand
 v_verticadb_node0006 | sc_02              | UNKNOWN    | sand
(6 rows)
```

When you issue the same query on one of the sandboxed nodes, the table shows that the sandboxed nodes are UP and the nodes from the main cluster are UNKNOWN, confirming that the cluster is successfully sandboxed:

```
=> SELECT node_name, subcluster_name, node_state, sandbox FROM NODES;
     node_name      |  subcluster_name   | node_state | sandbox
--------------------+--------------------+------------+---------
 v_verticadb_node0001 | default_subcluster | UNKNOWN    |
 v_verticadb_node0002 | default_subcluster | UNKNOWN    |
 v_verticadb_node0003 | default_subcluster | UNKNOWN    |
 v_verticadb_node0004 | sc_02              | UP         | sand
 v_verticadb_node0005 | sc_02              | UP         | sand
 v_verticadb_node0006 | sc_02              | UP         | sand
(6 rows)
```

You can now perform standard database operations in either cluster without impacting the other cluster. For instance, if you create a machine learning dataset named train_data in the sandboxed subcluster, the new table does not propagate to the main cluster:

```
--In the sandboxed subcluster
=> CREATE TABLE train_data(time timestamp, Temperature float);
CREATE TABLE

=> COPY train_data FROM LOCAL 'daily-min-temperatures.csv' DELIMITER ',';
 Rows Loaded
-------------
   3650
(1 row)

=> SELECT * FROM train_data LIMIT 5;
       time         | Temperature
--------------------+-------------
 1981-01-27 00:00:00 |        19.4
 1981-02-20 00:00:00 |        15.7
 1981-02-27 00:00:00 |        17.5
 1981-03-04 00:00:00 |          16
 1981-04-24 00:00:00 |        11.5
(5 rows)

--In the main cluster
=> SELECT * FROM train_data LIMIT 5;
ERROR 4566:  Relation "train_data" does not exist
```

See also

- UNSANDBOX_SUBCLUSTER
- Subcluster sandboxing
- Creating sandboxes

SET_DEPOT_PIN_POLICY_PARTITION

Eon Mode only

Pins the specified partitions of a table or projection to a subcluster depot, or all database depots, to reduce exposure to depot eviction.

Partition groups can be pinned only if all partitions within the group are pinned individually. If you alter or remove table partitioning, Vertica drops all partition pinning policies for that table. The table's pinning policy, if any, is unaffected.

For details on pinning policies and usage guidelines, see Pinning Depot Objects .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_DEPOT_PIN_POLICY_PARTITION ( '[[database.]schema.]object-name', 'min-range-value', 'max-range-value' [, 'subcluster' ] [, 'download' ] )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

object-name

Table or projection to pin. If you specify a projection, it must store the partition keys.

> **Note**
> After you pin a table or one of its projections to a subcluster, you cannot subsequently pin any of its partitions to that subcluster. Conversely, you can pin one or more table partitions to a subcluster, and then pin the table or one of its projections to that subcluster.

min-range-value

max-range-value

Minimum and maximum value of partition keys in *object-name* to pin, where *min-range-value* must be ≤ *max-range-value* . To specify a single partition, *min-range-value* and *max-range-value* must be equal.

> **Note**
> If partition pinning policies on the same table specify overlapping key ranges, Vertica collates the partition ranges. For example, if you create two partition policies with key ranges of 1-3 and 2-4, Vertica creates a single pinning policy with a key range of 1-4.

subcluster

Sets this pinning policy on the *subcluster* depot. To set this policy on the default subcluster, specify default_subcluster . If you omit this parameter, the policy is set on all database depots.

download

Boolean, if set to true, SET_DEPOT_PIN_POLICY_PARTITION immediately queues the specified partitions for download from communal storage.

Default: false

Privileges

Superuser

Precedence of pinning policies

In general, partition management functions that involve two partitioned tables give precedence to the target table's pinning policy, as follows:

Function	Application of pinnning policy
COPY_PARTITIONS_TO_TABLE	Partition-level pinning is reliable if the source and target tables have pinning policies on the same partition keys. If the two tables have different pinning policies, then the partition pinning policies of the target table apply.
MOVE_PARTITIONS_TO_TABLE	Partition-level pinning policies of the target table apply.
SWAP_PARTITIONS_BETWEEN_TABLES	Partition-level pinning policies of the target table apply.

For example, the following statement copies partitions from table foo to table bar :

```
=> SELECT COPY_PARTITIONS_TO_TABLE('foo', '1', '5', 'bar');
```

In this case, the following logic applies:

- If the two tables have different partition pinning policies, then the pinning policy of target table bar for partition keys 1-5 applies.
- If table bar does not exist, then Vertica creates it from table foo , and copies foo 's policy on partition keys 1-5. Subsequently, if you clear the partition pinning policy from either table, it is also cleared from the other.

See also

- Pinning Depot Objects
- CLEAR_DEPOT_PIN_POLICY_PARTITION

SET_DEPOT_PIN_POLICY_PROJECTION

Eon Mode only

Pins a projection to a subcluster depot, or all database depots, to reduce its exposure to depot eviction. For details on pinning policies and usage guidelines, see Pinning Depot Objects .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_DEPOT_PIN_POLICY_PROJECTION ( '[[database.]schema.]projection' [, 'subcluster' ] [, download ] )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection

Projection to pin.

> **Note**
> After you pin a table to a subcluster, you cannot subsequently pin any of its projections to that subcluster. Conversely, you can pin one or more projections of a table to a subcluster, and then pin the table to that subcluster.

subcluster

Sets this pinning policy on the *subcluster* depot. To set this policy on the default subcluster, specify default_subcluster . If you omit this parameter, the policy is set on all database depots.

download

Boolean, if set to true SET_DEPOT_PIN_POLICY_PROJECTION immediately queues the specified projection for download from communal storage.

Default: false

Privileges

Superuser

See also

- Pinning Depot Objects
- CLEAR_DEPOT_PIN_POLICY_PROJECTION

SET_DEPOT_PIN_POLICY_TABLE

Eon Mode only

Pins a table to a subcluster depot, or all database depots, to reduce its exposure to depot eviction. For details on pinning policies and usage guidelines, see Pinning Depot Objects .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_DEPOT_PIN_POLICY_TABLE ( '[[database.]schema.]table' [, 'subcluster' ] [, download ] )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Table to pin.

> **Note**
> After you pin a table to a subcluster, you cannot subsequently pin any of its partitions or projections to that subcluster. Conversely, you can pin one or more partitions or projections of a table to a subcluster, and then pin the table to that subcluster.

subcluster

Sets this pinning policy on the *subcluster* depot. To set this policy on the default subcluster, specify default_subcluster . If you omit this parameter, the policy is set on all database depots.

download

Boolean, if set to true, SET_DEPOT_PIN_POLICY_TABLE immediately queues the specified table for download from communal storage.

Default: false

Privileges

Superuser

See also

- Pinning Depot Objects
- CLEAR_DEPOT_PIN_POLICY_TABLE

SHUTDOWN_SUBCLUSTER

Eon Mode only

Shuts down a subcluster. This function shuts down the subcluster synchronously, returning when shutdown is complete with the message *Subcluster shutdown* . If the subcluster is already down, the function returns with no error.

Stopping a subcluster does not warn you if there are active user sessions connected to the subcluster. This behavior is the same as stopping an individual node. Before stopping a subcluster, verify that no users are connected to it.

If you want to drain client connections before shutting down a subcluster, you can gracefully shutdown the subcluster using SHUTDOWN_WITH_DRAIN .

> **Caution**
> This function does not test whether the target subcluster is critical (a subcluster whose loss would cause the database to shut down). Using this function to shut down a critical subcluster results in the database shutting down. Always verify that the subcluster you want to shut down is not critical by querying the CRITICAL_SUBCLUSTERS system table before calling this function.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SHUTDOWN_SUBCLUSTER('subcluster-name')
```

Arguments

subcluster-name

Name of the subcluster to shut down.

Privileges

Superuser

Examples

The following example demonstrates shutting down the subcluster analytics :

```
=> SELECT subcluster_name, node_name, node_state FROM nodes order by 1,2;
 subcluster_name   |      node_name       | node_state
-------------------+----------------------+------------
 analytics         | v_verticadb_node0004 | UP
 analytics         | v_verticadb_node0005 | UP
 analytics         | v_verticadb_node0006 | UP
 default_subcluster | v_verticadb_node0001 | UP
 default_subcluster | v_verticadb_node0002 | UP
 default_subcluster | v_verticadb_node0003 | UP
(6 rows)

=> SELECT SHUTDOWN_SUBCLUSTER('analytics');
WARNING 4539:  Received no response from v_verticadb_node0004 in stop subcluster
WARNING 4539:  Received no response from v_verticadb_node0005 in stop subcluster
WARNING 4539:  Received no response from v_verticadb_node0006 in stop subcluster
 SHUTDOWN_SUBCLUSTER
---------------------
 Subcluster shutdown
(1 row)

=> SELECT subcluster_name, node_name, node_state FROM nodes order by 1,2;
 subcluster_name   |      node_name       | node_state
-------------------+----------------------+------------
 analytics         | v_verticadb_node0004 | DOWN
 analytics         | v_verticadb_node0005 | DOWN
 analytics         | v_verticadb_node0006 | DOWN
 default_subcluster | v_verticadb_node0001 | UP
 default_subcluster | v_verticadb_node0002 | UP
 default_subcluster | v_verticadb_node0003 | UP
(6 rows)
```

Note
The "WARNING 4539" messages after calling SHUTDOWN_SUBCLUSTER occur because the nodes are in the process of shutting down. They are expected.

See also

- DEMOTE_SUBCLUSTER_TO_SECONDARY
- PROMOTE_SUBCLUSTER_TO_PRIMARY
- Subclusters
- ALTER SUBCLUSTER
- CRITICAL_SUBCLUSTERS

SHUTDOWN_WITH_DRAIN

Eon Mode only

Gracefully shuts down a subcluster or subclusters. The function drains client connections on the subcluster's nodes and then shuts down the subcluster. This is synchronous function that returns when the shutdown message has been sent to the subcluster.

Work from existing user sessions continues on draining nodes, but the nodes refuse new client connections and are excluded from load-balancing operations. dbadmin can still connect to draining nodes.

The nodes drain until either the existing connections complete their work and close or the user-specified timeout is reached. When one of these conditions is met, the function proceeds to shut down the subcluster.

For more information about the graceful shutdown process, see Graceful Shutdown .

Caution
This function does not test whether the target subcluster is critical (a subcluster whose loss would cause the database to shut down). Using this function to shut down a critical subcluster results in the database shutting down. Always verify that the subcluster you want to shut down is not

critical by querying the CRITICAL_SUBCLUSTERS system table before calling this function.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SHUTDOWN_WITH_DRAIN( 'subcluster-name', timeout-seconds )
```

Arguments

subcluster-name

Name of the subcluster to shutdown. Enter an empty string to shutdown all subclusters in a database.

timeout-seconds

Number of seconds to wait before forcefully closing *subcluster-name* 's client connections and shutting down. The behavior depends on the sign of *timeout-seconds* :

- Positive integer: The function waits until either the runtime reaches *timeout-seconds* or the client connections finish their work and close. As soon as one of these conditions is met, the function immediately proceeds to shut down the subcluster.
- Zero: The function immediately closes any open client connections and shuts down the subcluster.
- Negative integer: The function marks the subcluster as draining and waits indefinitely to shut down the subcluster until all active user sessions disconnect.

Privileges

Superuser

Examples

In the following example, the function marks the subcluster named analytics as draining and then shuts it down as soon as either the existing client connections close or 300 seconds pass:

```
=> SELECT SHUTDOWN_WITH_DRAIN('analytics', 120);
NOTICE 0:  Draining has started on subcluster (analytics)
NOTICE 0:  Begin shutdown of subcluster (analytics)
                        SHUTDOWN_WITH_DRAIN
-----------------------------------------------------------------------------------------------------------
Set subcluster (analytics) to draining state
Waited for 3 nodes to drain
Shutdown message sent to subcluster (analytics)

(1 row)
```

You can query the DC_DRAINING_EVENTS table to see more information about draining and shutdown events, such as whether any user sessions were forcibly closed. This subcluster had one active user session when the shutdown began, but it closed before the timeout was reached:

```
=> SELECT event_type, event_type_name, event_description, event_result, event_result_name FROM dc_draining_events;
 event_type |       event_type_name        |                        event_description                        | event_result | event_result_name
------------+------------------------------+-----------------------------------------------------------------+--------------+-------------------
          0 | START_DRAIN_SUBCLUSTER       | START_DRAIN for SHUTDOWN of subcluster (analytics)              |            0 | SUCCESS
          2 | START_WAIT_FOR_NODE_DRAIN    | Wait timeout is 120 seconds                                     |            4 | INFORMATIONAL
          4 | INTERVAL_WAIT_FOR_NODE_DRAIN | 1 sessions remain after 0 seconds                               |            4 | INFORMATIONAL
          4 | INTERVAL_WAIT_FOR_NODE_DRAIN | 1 sessions remain after 30 seconds                              |            4 | INFORMATIONAL
          3 | END_WAIT_FOR_NODE_DRAIN      | Wait for drain ended with 0 sessions remaining                  |            0 | SUCCESS
          5 | BEGIN_SHUTDOWN_AFTER_DRAIN   | Starting shutdown of subcluster (analytics) following drain     |            4 | INFORMATIONAL
(6 rows)
```

See also

- Graceful Shutdown
- DRAINING_STATUS
- START_DRAIN_SUBCLUSTER
- CANCEL_DRAIN_SUBCLUSTER

START_DRAIN_SUBCLUSTER

Eon Mode only

Drains a subcluster or subclusters. The function marks all nodes in the designated subcluster as draining. Work from existing user sessions continues on draining nodes, but the nodes refuse new client connections and are excluded from load balancing operations. dbadmin can still connect to draining nodes.

To drain connections on a subcluster as part of a graceful shutdown process, you can call SHUTDOWN_WITH_DRAIN . For details, see Graceful Shutdown .

To cancel a draining operation on a subcluster, call CANCEL_DRAIN_SUBCLUSTER . If all draining nodes in a subcluster are stopped, they are marked as not draining upon restart.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
START_DRAIN_SUBCLUSTER( 'subcluster-name' )
```

Arguments

subcluster-name

Name of the subcluster to drain. Enter an empty string to drain all subclusters in the database.

Privileges

Superuser

Examples

The following example demonstrates how to drain a subcluster named analytics:

```
=> SELECT subcluster_name, node_name, node_state FROM nodes;
subcluster_name    |      node_name     | node_state
-------------------+--------------------+------------
default_subcluster | verticadb_node0001 | UP
default_subcluster | verticadb_node0002 | UP
default_subcluster | verticadb_node0003 | UP
analytics          | verticadb_node0004 | UP
analytics          | verticadb_node0005 | UP
analytics          | verticadb_node0006 | UP
(6 rows)

=> SELECT START_DRAIN_SUBCLUSTER('analytics');
              START_DRAIN_SUBCLUSTER
-----------------------------------------------------------
Targeted subcluster: 'analytics'
Action: START DRAIN
(1 row)
```

You can confirm that the subcluster is draining by querying the DRAINING_STATUS system table:

```
=> SELECT node_name, subcluster_name, is_draining FROM draining_status ORDER BY 1;
node_name          | subcluster_name    | is_draining
-------------------+--------------------+-------
verticadb_node0001 | default_subcluster | f
verticadb_node0002 | default_subcluster | f
verticadb_node0003 | default_subcluster | f
verticadb_node0004 | analytics          | t
verticadb_node0005 | analytics          | t
verticadb_node0006 | analytics          | t
```

See also

- Drain client connections
- CANCEL_DRAIN_SUBCLUSTER
- DRAINING_STATUS

START_REAPING_FILES

Eon Mode only

Starts the disk file deletion in the background as an asynchronous function. By default, this meta-function syncs the catalog before beginning deletion. Disk file deletion is handled in the foreground by FLUSH_REAPER_QUEUE .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
START_REAPING_FILES( [sync-catalog] )
```

Parameters

sync-catalog

Specifies to sync metadata in the database catalog on all nodes before the function executes:

- true (default): Sync the database catalog
- false : Run without syncing.

Privileges

Superuser

Examples

Start the reaper service:

```
=> SELECT START_REAPING_FILES();
```

Start the reaper service and skip the initial catalog sync:

```
=> SELECT START_REAPING_FILES(false);
```

SYNC_CATALOG

Eon Mode only

Synchronizes the catalog to communal storage to enable reviving the current catalog version in the case of an imminent crash. Vertica synchronizes all pending checkpoint and transaction logs to communal storage.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SYNC_CATALOG( [ 'node-name' ] )
```

Parameters

node-name

The node to synchronize. If you omit this argument, Vertica synchronizes the catalog on all nodes.

Privileges

Superuser

Examples

Synchronize the catalog on all nodes:

```
=> SELECT SYNC_CATALOG();
```

Synchronize the catalog on one node:

```
=> SELECT SYNC_CATALOG( 'node001' );
```

UNSANDBOX_SUBCLUSTER

Removes a subcluster from a sandbox.

Note
Vertica recommends using the admintools unsandbox_subcluster command to remove sandboxes. This command automatically stops the sandboxed nodes, wipes the node's catalog subdirectories, and restarts the nodes. If you use the UNSANDBOX_SUBCLUSTER function, these steps must be completed manually. For details, see Removing sandboxes .

After stopping the nodes in the sandboxed subcluster, you must run this function in the main cluster from which the sandboxed subcluster was spun-off. The function changes the metadata in the main cluster that designates the specified subcluster as sandboxed, but does not restart the subcluster and rejoin it to the main cluster. To rejoin the subcluster to the main cluster and return the nodes to their normal state, you must complete the following tasks:

1. Wipe the catalog subdirectory from the sandboxed nodes. The main cluster provides the current catalog information on node restart.
2. Restart the nodes. On successful restart, the nodes should rejoin the main cluster.
3. Remove the sandbox metadata prefix from the shared communal storage location. This helps avoid problems that might arise form reusing the same sandbox name.

Note
If you upgraded the Vertica version of the sandboxed subcluster, you must downgrade the version of the subcluster before rejoining it to the main cluster.

If there are no more active sandboxes, you can run CLEAN_COMMUNAL_STORAGE to remove any data created in the sandbox. The main cluster can also resume processing data queued for deletion.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
UNSANDBOX_SUBCLUSTER( 'subcluster-name', 'options' )
```

Arguments

subcluster-name

Identifies the subcluster to unsandbox. This must be a currently sandboxed subcluster.

options

Currently, there are no options for this function.

Privileges

Superuser

Examples

After stopping the nodes in the sandbox, you can unsandbox the subcluster by calling the UNSANDBOX_SUBCLUSTER function in the main cluster. In the following example, the function unsandboxes the sc_02 subcluster from the sand sandbox:

```
=> SELECT UNSANDBOX_SUBCLUSTER('sc_02', '');
                                UNSANDBOX_SUBCLUSTER
-------------------------------------------------------------------------------------------------------------
Subcluster 'sc_02' has been unsandboxed. If wiped out and restarted, it should be able to rejoin the cluster.
(1 row)
```

To rejoin the nodes to the main cluster, you must wipe the local catalog from each of the previously sandboxed nodes—whose catalog location can be found by querying NODES —and then restart the nodes:

```
$ rm -rf paths-to-node-catalogs

$ admintools -t restart_node -s list-of-nodes -p password
```

After the nodes restart, you can query the NODES system table to confirm that the previously sandboxed nodes are UP and are no longer a member of sand :

```
=> SELECT node_name, subcluster_name, node_state, sandbox FROM NODES;
     node_name      |  subcluster_name   | node_state | sandbox
--------------------+--------------------+------------+---------
 v_verticadb_node0001 | default_subcluster | UP         |
 v_verticadb_node0002 | default_subcluster | UP         |
 v_verticadb_node0003 | default_subcluster | UP         |
 v_verticadb_node0004 | sc_02              | UP         |
 v_verticadb_node0005 | sc_02              | UP         |
 v_verticadb_node0006 | sc_02              | UP         |
(6 rows)
```

If there are no more active sandboxes, you can run the CLEAN_COMMUNAL_STORAGE function to remove any data created in the sandbox. You should also remove the sandbox's metadata from the shared communal storage location, which can be found at */path-to-communal-storage/`metadata`/sandbox_name* . The following example removes the sandbox's metadata from an S3 bucket and then calls CLEAN_COMMUNAL_STORAGE to cleanup any data from the sandbox::

```
$ aws s3 rm /path-to-communal/metadata/sandbox_name

=> SELECT CLEAN_COMMUNAL_STORAGE('true');
                CLEAN_COMMUNAL_STORAGE
-----------------------------------------------------------------------
 CLEAN COMMUNAL STORAGE
Total leaked files: 143
Files have been queued for deletion.
Check communal_cleanup_records for more information.

(1 row)
```

See also

- SANDBOX_SUBCLUSTER
- Removing sandboxes

Epoch functions

This section contains the epoch management functions specific to Vertica.

In this section

- ADVANCE_EPOCH
- GET_AHM_EPOCH
- GET_AHM_TIME
- GET_CURRENT_EPOCH
- GET_LAST_GOOD_EPOCH
- MAKE_AHM_NOW
- SET_AHM_EPOCH
- SET_AHM_TIME

ADVANCE_EPOCH

Manually closes the current epoch and begins a new epoch.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ADVANCE_EPOCH ( [ integer ] )
```

Parameters

integer

Specifies the number of epochs to advance.

Privileges

Superuser

Notes

This function is primarily maintained for backward compatibility with earlier versions of Vertica.

Examples

The following command increments the epoch number by 1:

```
=> SELECT ADVANCE_EPOCH(1);
```

GET_AHM_EPOCH

Returns the number of the epoch in which the Ancient History Mark is located. Data deleted up to and including the AHM epoch can be purged from physical storage.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_AHM_EPOCH()
```

> **Note**
> The AHM epoch is 0 (zero) by default (purge is disabled).

Privileges

None

Examples

```
=> SELECT GET_AHM_EPOCH();
   GET_AHM_EPOCH
----------------------
 Current AHM epoch: 0
(1 row)
```

GET_AHM_TIME

Returns a TIMESTAMP value representing the Ancient History Mark . Data deleted up to and including the AHM epoch can be purged from physical storage.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_AHM_TIME()
```

Privileges

None

Examples

```
=> SELECT GET_AHM_TIME();
                GET_AHM_TIME
-----------------------------------------------------
 Current AHM Time: 2010-05-13 12:48:10.532332-04
(1 row)
```

GET_CURRENT_EPOCH

The epoch into which data (COPY, INSERT, UPDATE, and DELETE operations) is currently being written.

Returns the number of the current epoch.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_CURRENT_EPOCH()
```

Privileges

None

Examples

```
=> SELECT GET_CURRENT_EPOCH();
 GET_CURRENT_EPOCH
-------------------
               683
(1 row)
```

GET_LAST_GOOD_EPOCH

Returns the last good epoch number. If the database has no projections, the function returns an error.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_LAST_GOOD_EPOCH()
```

Privileges

None

Examples

```
=> SELECT GET_LAST_GOOD_EPOCH();
 GET_LAST_GOOD_EPOCH
---------------------
                 682
(1 row)
```

MAKE_AHM_NOW

Sets the Ancient History Mark (AHM) to the greatest allowable value. This lets you purge all deleted data.

> **Caution**
> After running this function, you cannot query historical data that precedes the current epoch. Only database administrators should use this function.

MAKE_AHM_NOW performs the following operations:

- Advances the epoch.
- Sets the AHM to the last good epoch (LGE) — at least to the epoch that is current when you execute MAKE_AHM_NOW .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
MAKE_AHM_NOW ( [ true ] )
```

Parameters

true

Allows AHM to advance when one of the following conditions is true:

- One or more nodes are down.
- One projection is being refreshed from another (retentive refresh).

In both cases , you must supply this argument to MAKE_AHM_NOW , otherwise Vertica returns an error. If you execute MAKE_AHM_NOW(true) during retentive refresh, Vertica rolls back the refresh operation and advances the AHM.

> **Caution**
> If the function advances AHM beyond the last good epoch of the down nodes, those nodes must recover all data from scratch.

Privileges

Superuser

Setting AHM when nodes are down

If any node in the cluster is down, you must call MAKE_AHM_NOW with an argument of true; otherwise, the function returns an error.

> **Note**
> This requirement applies only to Enterprise mode; in Eon mode, it is ignored.

In the following example, MAKE_AHM_NOW advances the AHM even though a node is down:

```
=> SELECT MAKE_AHM_NOW(true);
WARNING:  Received no response from v_vmartdb_node0002 in get cluster LGE
WARNING:  Received no response from v_vmartdb_node0002 in get cluster LGE
WARNING:  Received no response from v_vmartdb_node0002 in set AHM
       MAKE_AHM_NOW
------------------------------
 AHM set (New AHM Epoch: 684)
(1 row)
```

See also

- SET_AHM_EPOCH
- SET_AHM_TIME

SET_AHM_EPOCH

Sets the Ancient History Mark (AHM) to the specified epoch. This function allows deleted data up to and including the AHM epoch to be purged from physical storage.

SET_AHM_EPOCH is normally used for testing purposes. Instead, consider using SET_AHM_TIME which is easier to use.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_AHM_EPOCH ( epoch, [ true ] )
```

Parameters

epoch

Specifies one of the following:

- The number of the epoch in which to set the AHM
- Zero (0) (the default) disables PURGE

Important

The number of the specified epoch must be:

- Greater than the current AHM epoch
- Less than the current epoch

Query the SYSTEM table to view current epoch values relative to the AHM.

true

Allows the AHM to advance when nodes are down.

> **Caution**
> If you advance AHM beyond the last good epoch of the down nodes, those nodes must recover all data from scratch.

Privileges

Superuser

Setting AHM when nodes are down

If any node in the cluster is down, you must call SET_AHM_EPOCH with an argument of true; otherwise, the function returns an error.

> **Note**
> This requirement applies only to Enterprise mode; in Eon mode, it is ignored.

Examples

The following command sets the AHM to a specified epoch of 12:

```
=> SELECT SET_AHM_EPOCH(12);
```

The following command sets the AHM to a specified epoch of 2 and allows the AHM to advance despite a failed node:

```
=> SELECT SET_AHM_EPOCH(2, true);
```

See also

- MAKE_AHM_NOW
- SET_AHM_TIME

SET_AHM_TIME

Sets the Ancient History Mark (AHM) to the epoch corresponding to the specified time on the initiator node. This function allows historical data up to and including the AHM epoch to be purged from physical storage. SET_AHM_TIME returns a TIMESTAMPTZ that represents the end point of the AHM epoch.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_AHM_TIME ( time, [ true ] )
```

Parameters

time

A TIMESTAMP/TIMESTAMPTZ value that is automatically converted to the appropriate epoch number.

true

Allows the AHM to advance when nodes are down.

> **Caution**
> If you advance AHM beyond the last good epoch of the down nodes, those nodes must recover all data from scratch.

Privileges

Superuser

Setting AHM when nodes are down

If any node in the cluster is down, you must call SET_AHM_TIME with an argument of true; otherwise, the function returns an error.

> **Note**
> This requirement applies only to Enterprise mode; in Eon mode, it is ignored.

Examples

Epochs depend on a configured epoch advancement interval. If an epoch includes a three-minute range of time, the purge operation is accurate only to within minus three minutes of the specified timestamp:

```
=> SELECT SET_AHM_TIME('2008-02-27 18:13');
        set_ahm_time
-------------------------------------
 AHM set to '2008-02-27 18:11:50-05'
(1 row)
```

> **Note**
> The –05 part of the output string is a time zone value, an offset in hours from UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, or GMT).

In the previous example, the actual AHM epoch ends at 18:11:50, roughly one minute before the specified timestamp. This is because SET_AHM_TIME selects the epoch that ends at or before the specified timestamp. It does not select the epoch that ends after the specified timestamp because that would purge data deleted as much as three minutes after the AHM.

For example, using only hours and minutes, suppose that epoch 9000 runs from 08:50 to 11:50 and epoch 9001 runs from 11:50 to 15:50. SET_AHM_TIME('11:51') chooses epoch 9000 because it ends roughly one minute before the specified timestamp.

In the next example, suppose that a node went down at 11:00:00 AM on January 1st 2017. At noon, you want to advance the AHM to 11:15:00, but the node is still down.

Suppose you try to set the AHM using this command:

```
=> SELECT SET_AHM_TIME('2017-01-01 11:15:00');
```

Then you will receive an error message. Vertica prevents you from moving the AHM past the point where a node went down. Vertica returns this error to prevent the AHM from advancing past the down node's last good epoch. You can force the AHM to advance by supplying the optional second parameter:

```
=> SELECT SET_AHM_TIME('2017-01-01 11:15:00', true);
```

However, if you force the AHM past the last good epoch, the failed node will have to recover from scratch.

See also

- MAKE_AHM_NOW
- SET_AHM_EPOCH
- SET DATESTYLE
- TIMESTAMP/TIMESTAMPTZ

LDAP link functions

This section contains the functions associated with the Vertica LDAP Link service.

In this section

- LDAP_LINK_DRYRUN_CONNECT
- LDAP_LINK_DRYRUN_SEARCH
- LDAP_LINK_DRYRUN_SYNC
- LDAP_LINK_SYNC_CANCEL
- LDAP_LINK_SYNC_START

LDAP_LINK_DRYRUN_CONNECT

Takes a set of LDAP Link connection parameters as arguments and begins a dry run connection between the LDAP server and Vertica.

By providing an empty string for the LDAPLinkBindPswd argument, you can also perform an anonymous bind if your LDAP server allows unauthenticated binds.

The dryrun and LDAP_LINK_SYNC_START functions must be run from the clerk node. To determine the clerk node, query NODE_RESOURCES :

```
=> SELECT node_name, dbclerk FROM node_resources WHERE dbclerk='t';
    node_name     | dbclerk
------------------+---------
 v_vmart_node0001 | t
(1 row)
```

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
LDAP_LINK_DRYRUN_CONNECT (
   'LDAPLinkURL',
   'LDAPLinkBindDN',
   'LDAPLinkBindPswd'
)
```

Privileges

Superuser

Examples

This tests the connection to an LDAP server at ldap://example.dc.com with the DN CN=amir,OU=QA,DC=dc,DC=com .

```
=> SELECT LDAP_LINK_DRYRUN_CONNECT('ldap://example.dc.com','CN=amir,OU=QA,DC=dc,DC=com','password');

                ldap_link_dryrun_connect
-----------------------------------------------------------------------------
Dry Run Connect Completed. Query v_monitor.ldap_link_dryrun_events for results.
```

To check the results of the bind, query the system table LDAP_LINK_DRYRUN_EVENTS.

```
=> SELECT event_timestamp, event_type, entry_name, role_name, link_scope, search_base from LDAP_LINK_DRYRUN_EVENTS;
       event_timestamp       |       event_type       |       entry_name       | link_scope | search_base
-----------------------------+------------------------+------------------------+------------+-------------
2019-12-09 15:41:43.589398-05 | BIND_STARTED           | ---------- | ---------- | ----------
2019-12-09 15:41:43.590504-05 | BIND_FINISHED          | ---------- | ---------- | ----------
```

See also

- LDAP_LINK_DRYRUN_SEARCH
- LDAP_LINK_DRYRUN_SYNC
- Configuring LDAP link with dry runs
- LDAP link parameters

LDAP_LINK_DRYRUN_SEARCH

Takes a set of LDAP Link connection and search parameters as arguments and begins a dry run search for users and groups that would get imported from the LDAP server.

By providing an empty string for the LDAPLinkBindPswd argument, you can also perform an anonymous search if your LDAP server's Access Control List (ACL) is configured to allow unauthenticated searches. The settings for allowing anonymous binds are different from the ACL settings for allowing anonymous searches.

The dryrun and LDAP_LINK_SYNC_START functions must be run from the clerk node. To determine the clerk node, query NODE_RESOURCES :

```
=> SELECT node_name, dbclerk FROM node_resources WHERE dbclerk='t';
    node_name     | dbclerk
------------------+---------
 v_vmart_node0001 | t
(1 row)
```

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
LDAP_LINK_DRYRUN_SEARCH (
   'LDAPLinkURL',
   'LDAPLinkBindDN',
   'LDAPLinkBindPswd',
   'LDAPLinkSearchBase',
   'LDAPLinkScope',
   'LDAPLinkFilterUser',
   'LDAPLinkFilterGroup',
   'LDAPLinkUserName',
   'LDAPLinkGroupName',
   'LDAPLinkGroupMembers',
   [LDAPLinkSearchTimeout],
   ['LDAPLinkJoinAttr']
)
```

Privileges

Superuser

Examples

This searches for users and groups in the LDAP server. In this case, the LDAPLinkSearchBase parameter specifies the dc.com domain and a sub scope, which replicates the entire subtree under the DN.

To further filter results, the function checks for users and groups with the person and group objectClass attributes. It then searches the group attribute cn , identifying members of that group with the member attribute, and then identifying those individual users with the attribute uid .

```
=> SELECT LDAP_LINK_DRYRUN_SEARCH('ldap://example.dc.com','CN=amir,OU=QA,DC=dc,DC=com','$vertica$','dc=DC,dc=com','sub',
'(objectClass=person)','(objectClass=group)','uid','cn','member',10,'dn');

               ldap_link_dryrun_search
-------------------------------------------------------------------------------
Dry Run Search Completed. Query v_monitor.ldap_link_dryrun_events for results.
```

To check the results of the search, query the system table LDAP_LINK_DRYRUN_EVENTS.

```
=> SELECT event_timestamp, event_type, entry_name, ldapurihash, link_scope, search_base from LDAP_LINK_DRYRUN_EVENTS;
       event_timestamp          |    event_type    |       entry_name       | ldapurihash | link_scope | search_base
--------------------------------+------------------+------------------------+-------------+------------+-------------
2020-01-03 21:03:26.411753+05:30 | BIND_STARTED     | ---------------------- |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:26.422188+05:30 | BIND_FINISHED    | ---------------------- |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:26.422223+05:30 | SYNC_STARTED     | ---------------------- |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:26.422229+05:30 | SEARCH_STARTED   | **********             |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:32.043107+05:30 | LDAP_GROUP_FOUND | Account Operators      |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:32.04312+05:30  | LDAP_GROUP_FOUND | Administrators         |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:32.043182+05:30 | LDAP_USER_FOUND  | user1                  |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:32.043186+05:30 | LDAP_USER_FOUND  | user2                  |           0 | sub        | dc=DC,dc=com
2020-01-03 21:03:32.04319+05:30  | SEARCH_FINISHED  | **********             |           0 | sub        | dc=DC,dc=com
```

See also

- LDAP_LINK_DRYRUN_CONNECT
- LDAP_LINK_DRYRUN_SYNC
- Configuring LDAP link with dry runs
- LDAP link parameters

LDAP_LINK_DRYRUN_SYNC

Takes a set of LDAP Link connection and search parameters as arguments and begins a dry run synchronization between the database and the LDAP server, which maps and synchronizes the LDAP server's users and groups with their equivalents in Vertica. This meta-function also dry runs the creation and orphaning of users and roles in Vertica.

The dryrun and LDAP_LINK_SYNC_START functions must be run from the clerk node. To determine the clerk node, query NODE_RESOURCES :

```
=> SELECT node_name, dbclerk FROM node_resources WHERE dbclerk='t';
   node_name     | dbclerk
-----------------+---------
 v_vmart_node0001 | t
(1 row)
```

You can view the results of the dry run in the system table LDAP_LINK_DRYRUN_EVENTS .

To cancel an in-progress synchronization, use LDAP_LINK_SYNC_CANCEL .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
LDAP_LINK_DRYRUN_SYNC (
   'LDAPLinkURL',
   'LDAPLinkBindDN',
   'LDAPLinkBindPswd',
   'LDAPLinkSearchBase',
   'LDAPLinkScope',
   'LDAPLinkFilterUser',
   'LDAPLinkFilterGroup',
   'LDAPLinkUserName',
   'LDAPLinkGroupName',
   'LDAPLinkGroupMembers',
   [LDAPLinkSearchTimeout],
   ['LDAPLinkJoinAttr']
)
```

Privileges

Superuser

Examples

To perform a dry run to map the users and groups returned from LDAP_LINK_DRYRUN_SEARCH, pass the same parameters as arguments to LDAP_LINK_DRYRUN_SYNC.

```
=> SELECT LDAP_LINK_DRYRUN_SYNC('ldap://example.dc.com','CN=amir,OU=QA,DC=dc,DC=com','$vertica$','dc=DC,dc=com','sub',
'(objectClass=person)','(objectClass=group)','uid','cn','member',10,'dn');

                    LDAP_LINK_DRYRUN_SYNC
-------------------------------------------------------------------------------------
Dry Run Connect and Sync Completed. Query v_monitor.ldap_link_dryrun_events for results.
```

To check the results of the sync, query the system table LDAP_LINK_DRYRUN_EVENTS.

```
=> SELECT event_timestamp, event_type, entry_name, ldapurihash, link_scope, search_base from LDAP_LINK_DRYRUN_EVENTS;
        event_timestamp          |     event_type     |       entry_name       | ldapurihash | link_scope | search_base
---------------------------------+--------------------+------------------------+-------------+------------+-------------
2020-01-03 21:08:30.883783+05:30 | BIND_STARTED       | ---------- |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:30.890574+05:30 | BIND_FINISHED      | ---------- |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:30.890602+05:30 | SYNC_STARTED       | ---------- |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:30.890605+05:30 | SEARCH_STARTED     | **********             |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939369+05:30 | LDAP_GROUP_FOUND   | Account Operators      |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939395+05:30 | LDAP_GROUP_FOUND   | Administrators         |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939461+05:30 | LDAP_USER_FOUND    | user1                  |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939463+05:30 | LDAP_USER_FOUND    | user2                  |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939468+05:30 | SEARCH_FINISHED    | **********             |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939718+05:30 | PROCESSING_STARTED | **********             |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939887+05:30 | USER_CREATED       | user1                  |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939895+05:30 | USER_CREATED       | user2                  |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939949+05:30 | ROLE_CREATED       | Account Operators      |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.939959+05:30 | ROLE_CREATED       | Administrators         |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.940603+05:30 | PROCESSING_FINISHED | **********            |          0 | sub        | dc=DC,dc=com
2020-01-03 21:08:31.940613+05:30 | SYNC_FINISHED      | ---------- |          0 | sub        | dc=DC,dc=com
```

See also

- LDAP_LINK_DRYRUN_CONNECT
- LDAP_LINK_DRYRUN_SEARCH
- Configuring LDAP link with dry runs
- LDAP link parameters

LDAP_LINK_SYNC_CANCEL

Cancels in-progress LDAP Link synchronizations (including those started by LDAP_LINK_DRYRUN_SYNC) between the LDAP server and Vertica.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ldap_link_sync_cancel()
```

Privileges

Superuser

Examples

```
=> SELECT ldap_link_sync_cancel();
```

See also

- LDAP_LINK_SYNC_START
- Configuring LDAP link with dry runs

LDAP_LINK_SYNC_START

Begins the synchronization between the LDAP server and Vertica immediately rather than waiting for the interval set in LDAPLinkInterval.

The dryrun and LDAP_LINK_SYNC_START functions must be run from the clerk node. To determine the clerk node, query NODE_RESOURCES :

```
=> SELECT node_name, dbclerk FROM node_resources WHERE dbclerk='t';
   node_name     | dbclerk
-----------------+---------
 v_vmart_node0001 | t
(1 row)
```

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ldap_link_sync_start()
```

Privileges

Superuser

Examples

```
=> SELECT ldap_link_sync_start();
```

See also

LDAP link parameters

License functions

This section contains functions that monitor Vertica license status and compliance.

In this section

- AUDIT
- AUDIT_FLEX
- AUDIT_LICENSE_SIZE
- AUDIT_LICENSE_TERM
- DISPLAY_LICENSE
- GET_AUDIT_TIME
- GET_COMPLIANCE_STATUS
- SET_AUDIT_TIME

AUDIT

Returns the raw data size (in bytes) of a database, schema, or table as it is counted in an audit of the database size. Unless you specify zero error tolerance and 100 percent confidence level, AUDIT returns only approximate results that can vary over multiple iterations.

Important

The data size returned by AUDIT should not be compared with the compressed data size of objects reported in the USED_BYTES column of system tables like STORAGE_CONTAINERS and PROJECTION_STORAGE .

AUDIT estimates the size for data in Vertica tables using the same data sampling method that Vertica uses to determine if a database complies with the licensed database size allowance. Vertica does not use these results to determine whether the size of the database complies with the Vertica license's data allowance. For details, see Auditing database size .

For data stored in external tables based on ORC or Parquet format, AUDIT uses the total size of the data files. This value is never estimated—it is read from the file system storing the ORC or Parquet files (either the Vertica node's local file system, S3, or HDFS).

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
AUDIT('[[[database.]schema.]scope ]'[, 'granularity'] [, error-tolerance[, confidence-level]] )
```

Parameters

[database .] schema

Database and schema . The default schema is public . If you specify a database, it must be the current database.

scope

Specifies the extent of the audit:

- Empty string ('') audits the entire database.
- The name of the schema or table to audit.

The schema or table to audit. To audit the database, set this parameter to an empty string.

granularity

The level at which the audit reports its results, one of the following strings:

- database
- schema
- table

The level of granularity must be equal to or less than the granularity of *scope* . If you omit this parameter, granularity is set to the same level as *scope* . Thus, if online_sales is a schema, the following statements are identical:

```
AUDIT('online_sales', 'schema');
AUDIT('online_sales');
```

If AUDIT sets granularity to a level lower than the target object, it returns with a message that refers you to system table USER_AUDITS . For details, see Querying V_CATALOG.USER_AUDITS , below.

error-tolerance

Specifies the percentage margin of error allowed in the audit estimate. Enter the tolerance value as a decimal number, between 0 and 100. The default value is 5, for a 5% margin of error.

This argument has no effect on audits of external tables based on ORC or Parquet files. Audits of these tables always returns the actual size of the underlying data files.
Setting this value to 0 results in a full database audit, which is very resource intensive, as AUDIT analyzes the entire database. A full database audit significantly impacts performance, so Vertica does not recommend it for a production database.

> **Caution**
> Due to the iterative sampling that the auditing process uses, setting the error tolerance to a small fraction of a percent (for example, 0.00001) can cause AUDIT to run for a longer period than a full database audit. The lower you specify this value, the more resources the audit uses, as it performs more data sampling.

confidence-level

Specifies the statistical confidence level percentage of the estimate. Enter the confidence value as a decimal number, between 0 and 100. The default value is 99, indicating a confidence level of 99%.

This argument has no effect on audits of external tables based on ORC or Parquet files. Audits of these tables always returns the actual size of the underlying data files.
The higher the confidence value, the more resources the function uses, as it performs more data sampling. Setting this value to 100 results in a full audit of the database, which is very resource intensive, as the function analyzes all of the database. A full database audit significantly impacts performance, so Vertica does not recommend it for a production database.

Privileges

Superuser, or the following privileges:

- SELECT privilege on the target tables
- USAGE privilege on the target schemas

> **Note**
> If you audit a schema or the database, Vertica only returns the size of all objects that you have privileges to access within the audited object, as described above.

Querying V_CATALOG.USER_AUDITS

If AUDIT sets granularity to a level lower than the target object, it returns with a message that refers you to system table USER_AUDITS . To obtain audit data on objects of the specified granularity, query this table. For example, the following query seeks to audit all tables in the store schema:

```
=> SELECT AUDIT('store', 'table');
                AUDIT
--------------------------------------
See table sizes in v_catalog.user_audits for schema store
(1 row)
```

The next query queries USER_AUDITS and obtains the latest audits on those tables:

```
=> SELECT object_name, AVG(size_bytes)::int size_bytes, MAX(audit_start_timestamp::date) audit_start
    FROM user_audits WHERE object_schema='store'
    GROUP BY rollup(object_name) HAVING GROUPING_ID(object_name) < 1 ORDER BY GROUPING_ID();
  object_name    | size_bytes | audit_start
-----------------+------------+------------
store_dimension  |      22067 | 2017-10-26
store_orders_fact |   27201312 | 2017-10-26
store_sales_fact |  301260170 | 2017-10-26
(3 rows)
```

Examples

See Auditing database size .

AUDIT_FLEX

Returns the estimated ROS size of __raw__ columns, equivalent to the export size of the flex data in the audited objects. You can audit all flex data in the database, or narrow the audit scope to a specific flex table, projection, or schema. Vertica stores the audit results in system table USER_AUDITS .

The audit excludes the following:

- Flex keys
- Other columns in the audited tables.
- Temporary flex tables

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
AUDIT_FLEX ('[scope]')
```

Parameters

scope

Specifies the extent of the audit:

- Empty string ('') audits all flexible tables in the database.
- The name of a schema, projection, or flex table.

Privileges

Superuser, or the following privileges:

- SELECT privilege on the target tables
- USAGE privilege on the target schemas

> **Note**
> If you audit a schema or the database, Vertica only returns the size of all objects that you have privileges to access within the audited object, as described above.

Examples

Audit all flex tables in the current database:

```
dbs=> select audit_flex('');
 audit_flex
------------
 8567679
(1 row)
```

Audit the flex tables in schema public :

```
dbs=> select audit_flex('public');
audit_flex
------------
8567679
(1 row)
```

Audit the flex data in projection bakery_b0 :

```
dbs=> select audit_flex('bakery_b0');
 audit_flex
------------
 8566723
(1 row)
```

Audit flex table bakery :

```
dbs=> select audit_flex('bakery');
 audit_flex
------------
 8566723
(1 row)
```

To report the results of all audits saved in the USER_AUDITS , the following shows part of an extended display from the system table showing an audit run on a schema called test , and the entire database, dbs :

```
dbs=> \x
Expanded display is on.

dbs=> select * from user_audits;
-[ RECORD 1 ]-------------------------+------------------------------
size_bytes                            | 0
user_id                               | 45035996273704962
user_name                             | release
object_id                             | 45035996273736664
object_type                           | SCHEMA
object_schema                         |
object_name                           | test
audit_start_timestamp                 | 2014-02-04 14:52:15.126592-05
audit_end_timestamp                   | 2014-02-04 14:52:15.139475-05
confidence_level_percent              | 99
error_tolerance_percent               | 5
used_sampling                         | f
confidence_interval_lower_bound_bytes | 0
confidence_interval_upper_bound_bytes | 0
sample_count                          | 0
cell_count                            | 0
-[ RECORD 2 ]-------------------------+------------------------------
size_bytes                            | 38051
user_id                               | 45035996273704962
user_name                             | release
object_id                             | 45035996273704974
object_type                           | DATABASE
object_schema                         |
object_name                           | dbs
audit_start_timestamp                 | 2014-02-05 13:44:41.11926-05
audit_end_timestamp                   | 2014-02-05 13:44:41.227035-05
confidence_level_percent              | 99
error_tolerance_percent               | 5
used_sampling                         | f
confidence_interval_lower_bound_bytes | 38051
confidence_interval_upper_bound_bytes | 38051
sample_count                          | 0
cell_count                            | 0
-[ RECORD 3 ]-------------------------+------------------------------
...
```

AUDIT_LICENSE_SIZE

Triggers an immediate audit of the database size to determine if it is in compliance with the raw data storage allowance included in your Vertica licenses.

If you use ORC or Parquet data stored in HDFS, results are only accurate if you run this function as a user who has access to all HDFS data. Either run the query with a principal that has read access to all such data, or use a Hadoop delegation token that grants this access. For more information about using delegation tokens, see Accessing kerberized HDFS data .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
AUDIT_LICENSE_SIZE()
```

Privileges

Superuser

Examples

```
=> SELECT audit_license_size();
 audit_license_size
--------------------
Raw Data Size: 0.00TB +/- 0.00TB
License Size : 10.00TB
Utilization  : 0%
Audit Time   : 2015-09-24 12:19:15.425486-04
Compliance Status : The database is in compliance with respect to raw data size.

License End Date: 2015-11-23 00:00:00 Days Remaining: 60.53
(1 row)
```

AUDIT_LICENSE_TERM

Triggers an immediate audit to determine if the Vertica license has expired.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
AUDIT_LICENSE_TERM()
```

Privileges

Superuser

Examples

```
=> SELECT audit_license_term();
 audit_license_term
--------------------
Raw Data Size: 0.00TB +/- 0.00TB
License Size : 10.00TB
Utilization  : 0%
Audit Time   : 2015-09-24 12:19:15.425486-04
Compliance Status : The database is in compliance with respect to raw data size.

License End Date: 2015-11-23 00:00:00 Days Remaining: 60.53
(1 row)
```

DISPLAY_LICENSE

Returns the terms of your Vertica license. The information this function displays is:

- The start and end dates for which the license is valid (or "Perpetual" if the license has no expiration).
- The number of days you are allowed to use Vertica after your license term expires (the grace period)
- The amount of data your database can store, if your license includes a data allowance.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DISPLAY_LICENSE()
```

Privileges

None

Examples

```
=> SELECT DISPLAY_LICENSE();
              DISPLAY_LICENSE
-------------------------------------------------------
 Vertica Systems, Inc.
2007-08-03
Perpetual
500GB

(1 row)
```

GET_AUDIT_TIME

Reports the time when the automatic audit of database size occurs. Vertica performs this audit if your Vertica license includes a data size allowance. For details of this audit, see Managing licenses in the Administrator's Guide. To change the time the audit runs, use the SET_AUDIT_TIME function.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_AUDIT_TIME()
```

Privileges

None

Examples

```
=> SELECT get_audit_time();
get_audit_time
-------------------------------------------------------
 The audit is scheduled to run at 11:59 PM each day.
(1 row)
```

GET_COMPLIANCE_STATUS

Displays whether your database is in compliance with your Vertica license agreement. This information includes the results of Vertica's most recent audit of the database size (if your license has a data allowance as part of its terms), the license term (if your license has an end date), and the number of nodes (if your license has a node limit).

GET_COMPLIANCE_STATUS measures data allowance by TBs (where a TB equals 1024^4 bytes).

The information displayed by GET_COMPLIANCE_STATUS includes:

- The estimated size of the database (see Auditing database size for an explanation of the size estimate).
- The raw data size allowed by your Vertica license.
- The percentage of your allowance that your database is currently using.
- The number of nodes and license limit.
- The date and time of the last audit.
- Whether your database complies with the data allowance terms of your license agreement.
- The end date of your license.
- How many days remain until your license expires.

> **Note**
> If your license does not have a data allowance, end date, or node limit, some of the values might not appear in the output for GET_COMPLIANCE_STATUS .

If the audit shows your license is not in compliance with your data allowance, you should either delete data to bring the size of the database under the licensed amount, or upgrade your license. If your license term has expired, you should contact Vertica immediately to renew your license. See Managing licenses for further details.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_COMPLIANCE_STATUS()
```

Privileges

None

Examples

```
=> SELECT GET_COMPLIANCE_STATUS();
 get_compliance_status
---------------------
Raw Data Size: 0.00TB +/- 0.00TB
License Size : 10.00TB
Utilization  : 0%
Audit Time   : 2015-09-24 12:19:15.425486-04
Compliance Status : The database is in compliance with respect to raw data size.

License End Date: 2015-11-23 00:00:00 Days Remaining: 60.53
(1 row)
```

The following example shows output for a Vertica for SQL on Apache Hadoop cluster.

```
=> SELECT GET_COMPLIANCE_STATUS();
 get_compliance_status
---------------------
Node count : 4
License Node limit : 5
No size-compliance concerns for an Unlimited license

No expiration date for a Perpetual license
(1 row)
```

SET_AUDIT_TIME

Sets the time that Vertica performs automatic database size audit to determine if the size of the database is compliant with the raw data allowance in your Vertica license. Use this function if the audits are currently scheduled to occur during your database's peak activity time. This is normally not a concern, since the automatic audit has little impact on database performance.

Audits are scheduled by the preceding audit, so changing the audit time does not affect the next scheduled audit. For example, if your next audit is scheduled to take place at 11:59PM and you use SET_AUDIT_TIME to change the audit schedule 3AM, the previously scheduled 11:59PM audit still runs. As that audit finishes, it schedules the next audit to occur at 3AM.

Vertica always performs the next scheduled audit even where you have changed the audit time using SET_AUDIT_TIME and then triggered an automatic audit by issuing the statement, SELECT AUDIT_LICENSE_SIZE . Only after the next scheduled audit does Vertica begin auditing at the new time you set using SET_AUDIT_TIME. Thereafter, Vertica audits at the new time.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_AUDIT_TIME(time)
```

time

A string containing the time in 'HH:MM AM/PM' format (for example, '1:00 AM') when the audit should run daily.

Privileges

Superuser

Examples

```
=> SELECT SET_AUDIT_TIME('3:00 AM');
                        SET_AUDIT_TIME
-------------------------------------------------------------------------
 The scheduled audit time will be set to 3:00 AM after the next audit.
(1 row)
```

Notifier functions

This section contains functions for using and managing the notifier.

In this section

- GET_DATA_COLLECTOR_NOTIFY_POLICY
- NOTIFY
- SET_DATA_COLLECTOR_NOTIFY_POLICY

GET_DATA_COLLECTOR_NOTIFY_POLICY

Lists any notification policies set on a Data collector component.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_DATA_COLLECTOR_NOTIFY_POLICY('component')
```

component

Name of the Data Collector component to check for notification policies.

Query system table DATA_COLLECTOR for component names. For example:

```
=> SELECT DISTINCT component, description FROM data_collector WHERE component ilike '%Depot%' ORDER BY component;
  component    |          description
---------------+-------------------------------
 DepotEvictions | Files evicted from the Depot
 DepotFetches   | Files fetched to the Depot
 DepotUploads   | Files Uploaded from the Depot
(3 rows)
```

Examples

```
=> SELECT GET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures');
                GET_DATA_COLLECTOR_NOTIFY_POLICY
-------------------------------------------------------------------------
 Notifiable;  Notifier: vertica_stats; Channel: vertica_notifications
(1 row)
```

The following example shows the output from the function when there is no notification policy for the component:

```
=> SELECT GET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures');
 GET_DATA_COLLECTOR_NOTIFY_POLICY
----------------------------------
 Not notifiable;
(1 row)
```

See also

- SET_DATA_COLLECTOR_NOTIFY_POLICY
- Producing Kafka messages using notifiers

NOTIFY

Sends a specified message to a NOTIFIER .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
NOTIFY ( 'message', 'notifier', 'target-topic' )
```

Parameters

message

The message to send to the endpoint.

notifier

The name of the NOTIFIER .

target-topic

String that specifies one of the following based on the *notifier* type:

- Kafka: The name of an existing destination Kafka topic for the message.

> **Note**
> If the topic doesn't already exist, you can configure your Kafka broker to automatically create the specified topic. For more information, see the Kafka documentataion .

- Syslog: The ProblemDescription subject and channel value.
- SNS: The topic ARN .

Privileges

Superuser

Examples

Send a message to confirm that an ETL job is complete:

```
=> SELECT NOTIFY('ETL Done!', 'my_notifier', 'DB_activity_topic');
```

SET_DATA_COLLECTOR_NOTIFY_POLICY

Creates/enables notification policies for a Data collector component. Notification policies automatically send messages to the specified NOTIFIER when certain events occur.

To view existing notification policies on a Data Collector component, see GET_DATA_COLLECTOR_NOTIFY_POLICY .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_DATA_COLLECTOR_NOTIFY_POLICY('component','notifier', 'topic', enabled)
```

component

Name of the component whose change will be reported via the notifier.

Query system table DATA_COLLECTOR for component names. For example:

```
=> SELECT DISTINCT component, description FROM data_collector WHERE component ilike '%Depot%' ORDER BY component;
  component   |          description
--------------+-------------------------------
 DepotEvictions | Files evicted from the Depot
 DepotFetches   | Files fetched to the Depot
 DepotUploads   | Files Uploaded from the Depot
(3 rows)
```

notifier

Name of the notifier that will send the message.

topic

One of the following:

- Kafka: The name of the Kafka topic that will receive the notification message.

> **Note**
> If the topic doesn't already exist, you can configure your Kafka broker to automatically create the specified topic. For more information, see the Kafka documentataion .

- Syslog: The subject of the field ProblemDescription .
- SNS: The topic ARN .

enabled

Boolean value that specifies whether this policy is enabled. Set to TRUE to enable reporting component changes. Set to FALSE to disable the notifier.

Examples

SNS notifier

The following example creates an SNS topic, subscribes to it with an SQS queue, and then configures an SNS notifier for the DC component LoginFailures :

1. Create an SNS topic .
2. Create an SQS queue .
3. Subscribe the SQS queue to the SNS topic.
4. Set SNSAuth with your AWS credentials:
```
=> ALTER DATABASE DEFAULT SET SNSAuth='VNDDNVOPIUQF917O5PDB:+mcnVONVIbjOnf1ekNis7nm3mE83u9fjdwmlq36Z';
```
5. Set SNSRegion:
```
=> ALTER DATABASE DEFAULT SET SNSRegion='us-east-1'
```
6. Enable HTTPS:
```
=> ALTER DATABASE DEFAULT SET SNSEnableHttps=1;
```
7. Create an SNS notifier:
```
=> CREATE NOTIFIER v_sns_notifier ACTION 'sns' MAXPAYLOAD '256K' MAXMEMORYSIZE '10M' CHECK COMMITTED;
```
8. Verify that the SNS notifier, SNS topic, and SQS queue are properly configured:
   1. Manually send a message from the notifier to the SNS topic with NOTIFY :
   ```
   => SELECT NOTIFY('test message', 'v_sns_notifier', 'arn:aws:sns:us-east-1:123456789012:MyTopic')
   ```
   2. Poll the SQS queue for your message.
9. Attach the SNS notifier to the LoginFailures component with SET_DATA_COLLECTOR_NOTIFY_POLICY :
```
=> SELECT SET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures', 'v_sns_notifier', 'Login failed!', true)
```

Kafka notifier

To be notified of failed login attempts, you can create a notifier that sends a notification when the DC component LoginFailures updates. The TLSMODE 'verify-ca' verifies that the server's certificate is signed by a trusted CA.

```
=> CREATE NOTIFIER vertica_stats ACTION 'kafka://kafka01.example.com:9092' MAXMEMORYSIZE '10M' TLSMODE 'verify-ca';
CREATE NOTIFIER
=> SELECT SET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures','vertica_stats', 'vertica_notifications', true);
SET_DATA_COLLECTOR_NOTIFY_POLICY
---------------------------------
 SET
(1 row)
```

The following example shows how to disable the policy created in the previous example:

```
=> SELECT SET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures','vertica_stats', 'vertica_notifications', false);
 SET_DATA_COLLECTOR_NOTIFY_POLICY
---------------------------------
 SET
(1 row)
```

```
=> SELECT GET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures');
 GET_DATA_COLLECTOR_NOTIFY_POLICY
----------------------------------
 Not notifiable;
(1 row)
```

Syslog notifier

The following example creates a notifier that writes a message to syslog when the Data collector (DC) component LoginFailures updates:

1. Enable syslog notifiers for the current database:
   ```
   => ALTER DATABASE DEFAULT SET SyslogEnabled = 1;
   ```
2. Create and enable a syslog notifier v_syslog_notifier :
   ```
   => CREATE NOTIFIER v_syslog_notifier ACTION 'syslog'
      ENABLE
      MAXMEMORYSIZE '10M'
      IDENTIFIED BY 'f8b0278a-3282-4e1a-9c86-e0f3f042a971'
      PARAMETERS 'eventSeverity = 5';
   ```
3. Configure the syslog notifier v_syslog_notifier for updates to the LoginFailures DC component with SET_DATA_COLLECTOR_NOTIFY_POLICY :
   ```
   => SELECT SET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures','v_syslog_notifier', 'Login failed!', true);
   ```

   This notifier writes the following message to syslog (default location: /var/log/messages) when a user fails to authenticate as the user Bob :

   ```
   Apr 25 16:04:58
   vertica_host_01
   vertica:
      Event Posted:
         Event Code:21
         Event Id:0
         Event Severity: Notice [5]
         PostedTimestamp: 2022-04-25 16:04:58.083063
         ExpirationTimestamp: 2022-04-25 16:04:58.083063
         EventCodeDescription: Notifier
         ProblemDescription: (Login failed!)
      {
         "_db":"VMart",
         "_schema":"v_internal",
         "_table":"dc_login_failures",
         "_uuid":"f8b0278a-3282-4e1a-9c86-e0f3f042a971",
         "authentication_method":"Reject",
         "client_authentication_name":"default: Reject",
         "client_hostname":"::1",
         "client_label":"",
         "client_os_user_name":"dbadmin",
         "client_pid":523418,
         "client_version":"",
         "database_name":"dbadmin",
         "effective_protocol":"3.8",
         "node_name":"v_vmart_node0001",
         "reason":"REJECT",
         "requested_protocol":"3.8",
         "ssl_client_fingerprint":"",
         "ssl_client_subject":"",
         "time":"2022-04-25 16:04:58.082568-05",
         "user_name":"Bob"
      }#012
      DatabaseName: VMart
      Hostname: vertica_host_01
   ```

See also

- Configuring reporting for syslog

- Producing Kafka messages using notifiers
- Configuring reporting for the simple notification service (SNS)

Partition functions

This section contains partition management functions specific to Vertica.

In this section

- CALENDAR_HIERARCHY_DAY
- COPY_PARTITIONS_TO_TABLE
- DROP_PARTITIONS
- DUMP_PROJECTION_PARTITION_KEYS
- DUMP_TABLE_PARTITION_KEYS
- MOVE_PARTITIONS_TO_TABLE
- PARTITION_PROJECTION
- PARTITION_TABLE
- PURGE_PARTITION
- SWAP_PARTITIONS_BETWEEN_TABLES

CALENDAR_HIERARCHY_DAY

Specifies to group DATE partition keys into a hierarchy of years, months, and days. The Vertica Tuple Mover regularly evaluates partition keys against the current date, and merges partitions as needed into the appropriate year and month partition groups.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CALENDAR_HIERARCHY_DAY( partition-expression[, active-months[, active-years] ] )
```

Parameters

partition-expression

The DATE expression on which to group partition keys, which must be identical to the table's PARTITION BY expression.

active-months

An integer ≥ 0 that specifies how many months preceding MONTH(CURRENT_DATE) to store unique partition keys in separate partitions.

If you specify 1, only partition keys of the current month are stored in separate partitions.
If you specify 0, all partition keys of the current month are merged into a partition group for that month.

For details, see Hierarchical partitioning .

Default: 2

active-years

An integer ≥ 0, specifies how many years preceding YEAR(CURRENT_DATE) to partition group keys by month in separate partitions.

If you specify 1, only partition keys of the current year are stored in month partition groups.
If you specify 0, all partition keys of the current and previous years are merged into year partition groups.

For details, see Hierarchical partitioning .

Default: 2

Important

The CALENDAR_HIERARCHY_DAY algorithm assumes that most table activity is focused on recent dates. Setting *active-years* and *active-months* to a low number ≥ 2 serves to isolate most merge activity to date-specific containers, and incurs minimal overhead. Vertica recommends that you use the default setting of 2 for *active-years* and *active-months* . For most users, these settings achieve an optimal balance between ROS storage and performance.

As a best practice, never set *active-years* and *active-months* to 0.

Usage

Specify this function in a table partition clause , as its GROUP BY expression:

```
PARTITION BY partition-expression
  GROUP BY CALENDAR_HIERARCHY_DAY(
    group-expression
     [, active-months[, active-years] ] )
```

For example:

```
=> CREATE TABLE public.store_orders
(
    order_no int,
    order_date timestamp NOT NULL,
    shipper varchar(20),
    ship_date date
);
...
=> ALTER TABLE public.store_orders
     PARTITION BY order_date::DATE
     GROUP BY CALENDAR_HIERARCHY_DAY(order_date::DATE, 3, 2) REORGANIZE;
```

Examples

See Hierarchical partitioning .

COPY_PARTITIONS_TO_TABLE

Copies partitions from one table to another. This lightweight partition copy increases performance by initially sharing the same storage between two tables. After the copy operation is complete, the tables are independent of each other. Users can perform operations on one table without impacting the other. These operations can increase the overall storage required for both tables.

> **Note**
> Although they share storage space, Vertica considers the partitions as discrete objects for license capacity purposes. For example, copying a one TB partition would only consume one TB of space. Your Vertica license, however, considers them as separate objects consuming two TB of space.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
COPY_PARTITIONS_TO_TABLE (
   '[[database.]schema.]source-table',
   'min-range-value',
   'max-range-value',
   '[[database.]schema.]target-table'
    [, 'force-split']
)
```

Parameters

[database .] schema

Database and schema . The default schema is public . If you specify a database, it must be the current database.

source-table

The source table of the partitions to copy.

min-range-value max-range-value

The minimum and maximum value of partition keys to copy, where *min-range-value* must be ≤ *max-range-value* . To copy one partition, *min-range-value* and *max-range-value* must be equal.

target-table

The target table of the partitions to copy. If the table does not exist, Vertica creates a table from the source table's definition, by calling CREATE TABLE with LIKE and INCLUDING PROJECTIONS clause. The new table inherits ownership from the source table. For details, see Replicating a

table .

force-split

Optional Boolean argument, specifies whether to split ROS containers if the range of partition keys spans multiple containers or part of a single container:

- true : Split ROS containers as needed.
- false (default): Return with an error if ROS containers must be split to implement this operation.

Privileges

Non-superuser, one of the following:

- Owner of source and target tables
- TRUNCATE (if force-split is true) and SELECT on the source table, INSERT on the target table

If the target table does not exist, you must also have CREATE privileges on the target schema to enable table creation.

Table attribute requirements

The following attributes of both tables must be identical:

- Column definitions, including NULL/NOT NULL constraints
- Segmentation
- Partition clause
- Number of projections
- Projection sort order
- Primary and unique key constraints. However, the key constraints do not have to be identically enabled. For more information on constraints, see Constraints .

> **Note**
> If the target table has primary or unique key constraints enabled and copying or moving the partitions will insert duplicate key values into the target table, Vertica rolls back the operation.

- Check constraints. For MOVE_PARTITIONS_TO_TABLE and COPY_PARTITIONS_TO_TABLE , Vertica enforces enabled check constraints on the target table only. For SWAP_PARTITIONS_BETWEEN_TABLES , Vertica enforces enabled check constraints on both tables. If there is a violation of an enabled check constraint, Vertica rolls back the operation.
- Number and definitions of text indices.

Additionally, If access policies exist on the source table, the following must be true:

- Access policies on both tables must be identical.
- One of the following must be true:
  - The executing user owns the source table.
  - AccessPolicyManagementSuperuserOnly is set to true. See Managing access policies for details.

Table restrictions

The following restrictions apply to the source and target tables:

- If the source and target partitions are in different storage tiers, Vertica returns a warning but the operation proceeds. The partitions remain in their existing storage tier.
- The target table cannot be immutable .
- The following tables cannot be used as sources or targets:
  - Temporary tables
  - Virtual tables
  - System tables
  - External tables

Examples

If you call COPY_PARTITIONS_TO_TABLE and the target table does not exist, the function creates the table automatically. In the following example, the target table partn_backup.tradfes_200801 does not exist. COPY_PARTITIONS_TO_TABLE creates the table and replicates the partition. Vertica also copies all the constraints associated with the source table except foreign key constraints.

```
=> SELECT COPY_PARTITIONS_TO_TABLE (
        'prod_trades',
        '200801',
        '200801',
        'partn_backup.trades_200801');
COPY_PARTITIONS_TO_TABLE
-------------------------------------------------
 1 distinct partition values copied at epoch 15.
(1 row)
```

See also

Archiving partitions
DROP_PARTITIONS

> **Note**
> This function supersedes meta-function DROP_PARTITION, which was deprecated in Vertica 9.0.

Drops the specified table partition keys.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DROP_PARTITIONS (
   '[[database.]schema.]table-name',
   'min-range-value',
   'max-range-value'
   [, 'force-split']
)
```

Parameters

[database .] schema

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table-name

The target table. The table cannot be used as a dimension table in a pre-join projection and cannot have out-of-date (unrefreshed) projections.

min-range-value max-range-value

The minimum and maximum value of partition keys to drop, where *min-range-value* must be ≤ *max-range-value* . To drop one partition key, *min-range-value* and *max-range-value* must be equal.

force-split

Optional Boolean argument, specifies whether to split ROS containers if the range of partition keys spans multiple containers or part of a single container:

- true : Split ROS containers as needed.
- false (default): Return with an error if ROS containers must be split to implement this operation.

> **Note**
> In rare cases, DROP_PARTITIONS executes at the same time as a mergeout operation on the same ROS container. As a result, the function cannot split the container as specified and returns with an error. When this happens, call DROP_PARTITIONS again.

Privileges

One of the following:

- DBADMIN
- Table owner
- USAGE privileges on the table schema and TRUNCATE privileges on the table

Examples

See Dropping partitions .

See also

PARTITION_TABLE

DUMP_PROJECTION_PARTITION_KEYS

Dumps the partition keys of the specified projection.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DUMP_PROJECTION_PARTITION_KEYS( '[[database.]schema.]projection-name')
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection-name

Projection name

Privileges

Non-superuser: TRUNCATE on anchor table

Examples

The following statements create the table and projection online_sales.online_sales_fact and online_sales.online_sales_fact_rep , respectively, and partitions table data by the column call_center_key :

```
=> CREATE TABLE online_sales.online_sales_fact
(
    sale_date_key int NOT NULL,
    ship_date_key int NOT NULL,
    product_key int NOT NULL,
    product_version int NOT NULL,
    customer_key int NOT NULL,
    call_center_key int NOT NULL,
    online_page_key int NOT NULL,
    shipping_key int NOT NULL,
    warehouse_key int NOT NULL,
    promotion_key int NOT NULL,
    pos_transaction_number int NOT NULL,
    sales_quantity int,
    sales_dollar_amount float,
    ship_dollar_amount float,
    net_dollar_amount float,
    cost_dollar_amount float,
    gross_profit_dollar_amount float,
    transaction_type varchar(16)
)
PARTITION BY (online_sales_fact.call_center_key);

=> CREATE PROJECTION online_sales.online_sales_fact_rep AS SELECT * from online_sales.online_sales_fact unsegmented all nodes;
```

The following DUMP_PROJECTION_PARTITION_KEYS statement dumps the partition key from the projection online_sales.online_sales_fact_rep :

```
=> SELECT DUMP_PROJECTION_PARTITION_KEYS('online_sales.online_sales_fact_rep');

Partition keys on node v_vmart_node0001
  Projection 'online_sales_fact_rep'
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: 200
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: 199
   ...
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: 2
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: 1

 Partition keys on node v_vmart_node0002
  Projection 'online_sales_fact_rep'
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: 200
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: 199
...
(1 row)
```

See also

- Partitioning tables
- DUMP_PARTITION_KEYS
- DUMP_TABLE_PARTITION_KEYS
- PARTITION_PROJECTION
- PARTITION_TABLE

DUMP_TABLE_PARTITION_KEYS

Dumps the partition keys of all projections for the specified table.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DUMP_TABLE_PARTITION_KEYS ( '[[database.]schema.]table-name' )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table-name

Name of the table

Privileges

Non-superuser: TRUNCATE on table

Examples

The following example creates a simple table called states and partitions the data by state:

```
=> CREATE TABLE states (year INTEGER NOT NULL,
      state VARCHAR NOT NULL)
      PARTITION BY state;
=> CREATE PROJECTION states_p (state, year) AS
      SELECT * FROM states
      ORDER BY state, year UNSEGMENTED ALL NODES;
```

Now dump the partition keys of all projections anchored on table states :

```
=> SELECT DUMP_TABLE_PARTITION_KEYS( 'states' );
     DUMP_TABLE_PARTITION_KEYS                                    ----------------------------------------------------------------------------------------------
 Partition keys on node v_vmart_node0001
  Projection 'states_p'
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: VT
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: PA
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: NY
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: MA

 Partition keys on node v_vmart_node0002
...
(1 row)
```

See also

- DUMP_PROJECTION_PARTITION_KEYS
- DUMP_TABLE_PARTITION_KEYS
- PARTITION_PROJECTION
- PARTITION_TABLE
- Partitioning tables

MOVE_PARTITIONS_TO_TABLE

Moves partitions from one table to another.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
MOVE_PARTITIONS_TO_TABLE (
   '[[database.]schema.]source-table',
   'min-range-value',
   'max-range-value',
   '[[database.]schema.]target-table'
    [, force-split]
)
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

source-table

The source table of the partitions to move.

min-range-value max-range-value

The minimum and maximum value of partition keys to move, where *min-range-value* must be ≤ *max-range-value* . To move one partition, *min-range-value* and *max-range-value* must be equal.

target-table

The target table of the partitions to move. If the table does not exist, Vertica creates a table from the source table's definition, by calling CREATE TABLE with LIKE and INCLUDING PROJECTIONS clause. The new table inherits ownership from the source table. For details, see Replicating a table .

force-split

Optional Boolean argument, specifies whether to split ROS containers if the range of partition keys spans multiple containers or part of a single container:

- true : Split ROS containers as needed.
- false (default): Return with an error if ROS containers must be split to implement this operation.

Privileges

Non-superuser, one of the following:

- Owner of source and target tables
- SELECT, TRUNCATE on the source table, INSERT on the target table

If the target table does not exist, you must also have CREATE privileges on the target schema to enable table creation.

Table attribute requirements

The following attributes of both tables must be identical:

- Column definitions, including NULL/NOT NULL constraints
- Segmentation
- Partition clause
- Number of projections
- Projection sort order
- Primary and unique key constraints. However, the key constraints do not have to be identically enabled. For more information on constraints, see Constraints .

> **Note**
> If the target table has primary or unique key constraints enabled and copying or moving the partitions will insert duplicate key values into the target table, Vertica rolls back the operation.

- Check constraints. For MOVE_PARTITIONS_TO_TABLE and COPY_PARTITIONS_TO_TABLE , Vertica enforces enabled check constraints on the target table only. For SWAP_PARTITIONS_BETWEEN_TABLES , Vertica enforces enabled check constraints on both tables. If there is a violation of an enabled check constraint, Vertica rolls back the operation.
- Number and definitions of text indices.

Additionally, If access policies exist on the source table, the following must be true:

- Access policies on both tables must be identical.
- One of the following must be true:
  - The executing user owns the source table.
  - AccessPolicyManagementSuperuserOnly is set to true. See Managing access policies for details.

Table restrictions

The following restrictions apply to the source and target tables:

- If the source and target partitions are in different storage tiers, Vertica returns a warning but the operation proceeds. The partitions remain in their existing storage tier.
- The target table cannot be immutable .
- The following tables cannot be used as sources or targets:
  - Temporary tables
  - Virtual tables
  - System tables
  - External tables

Examples

See Archiving partitions .

See also

- COPY_PARTITIONS_TO_TABLE
- SWAP_PARTITIONS_BETWEEN_TABLES

PARTITION_PROJECTION

Splits ROS containers for a specified projection. PARTITION_PROJECTION also purges data while partitioning ROS containers if deletes were applied before the AHM epoch.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
PARTITION_PROJECTION ( '[[database.]schema.]projection')
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection ``

The projection to partition.

Privileges

- Table owner
- USAGE privilege on schema

Examples

In this example, PARTITION_PROJECTION forces a split of ROS containers on the states_p projection:

```
=> SELECT PARTITION_PROJECTION ('states_p');
 PARTITION_PROJECTION
-------------------------
 Projection partitioned
(1 row)
```

See also

- PARTITION_TABLE
- Partitioning tables

PARTITION_TABLE

Invokes the Tuple Mover to reorganize ROS storage containers as needed to conform with the current partitioning policy.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
PARTITION_TABLE ( '[schema.]table-name')
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table-name

The table to partition.

Privileges

- Table owner
- USAGE privilege on schema

Restrictions

- You cannot run PARTITION_TABLE on a table that is an anchor table for a live aggregate projection or a Top-K projection.
- To reorganize storage to conform to a new policy, run PARTITION_TABLE after changing the partition GROUP BY expression.

See also

- PARTITION_PROJECTION
- Partitioning existing table data

PURGE_PARTITION

Purges a table partition of deleted rows. Similar to PURGE and PURGE_PROJECTION , this function removes deleted data from physical storage so you can reuse the disk space. PURGE_PARTITION removes data only from the AHM epoch and earlier.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
PURGE_PARTITION ( '[[database.]schema.]table', partition-key )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

The partitioned table to purge.

partition-key

The key of the partition to purge.

Privileges

- Table owner
- USAGE privilege on schema

Examples

The following example lists the count of deleted rows for each partition in a table, then calls PURGE_PARTITION() to purge the deleted rows from the data.

```
=> SELECT partition_key,table_schema,projection_name,sum(deleted_row_count)
   AS deleted_row_count FROM partitions
   GROUP BY partition_key,table_schema,projection_name
   ORDER BY partition_key;

 partition_key | table_schema | projection_name | deleted_row_count
---------------+--------------+-----------------+-------------------
 0             | public       | t_super         |                 2
 1             | public       | t_super         |                 2
 2             | public       | t_super         |                 2
 3             | public       | t_super         |                 2
 4             | public       | t_super         |                 2
 5             | public       | t_super         |                 2
 6             | public       | t_super         |                 2
 7             | public       | t_super         |                 2
 8             | public       | t_super         |                 2
 9             | public       | t_super         |                 1
(10 rows)
=> SELECT PURGE_PARTITION('t',5); -- Purge partition with key 5.
                        purge_partition
-------------------------------------------------------------------------
 Task: merge partitions
(Table: public.t) (Projection: public.t_super)
(1 row)

=> SELECT partition_key,table_schema,projection_name,sum(deleted_row_count)
   AS deleted_row_count FROM partitions
   GROUP BY partition_key,table_schema,projection_name
   ORDER BY partition_key;


 partition_key | table_schema | projection_name | deleted_row_count
---------------+--------------+-----------------+-------------------
 0             | public       | t_super         |                 2
 1             | public       | t_super         |                 2
 2             | public       | t_super         |                 2
 3             | public       | t_super         |                 2
 4             | public       | t_super         |                 2
 5             | public       | t_super         |                 0
 6             | public       | t_super         |                 2
 7             | public       | t_super         |                 2
 8             | public       | t_super         |                 2
 9             | public       | t_super         |                 1
(10 rows)
```

See also

- PURGE
- PURGE_PROJECTION
- PURGE_TABLE
- STORAGE_CONTAINERS

SWAP_PARTITIONS_BETWEEN_TABLES

Swaps partitions between two tables.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SWAP_PARTITIONS_BETWEEN_TABLES (
   '[[database.]schema.]staging-table',
   'min-range-value',
   'max-range-value',
   '[[database.]schema.]target-table'
    [, force-split]
)
```

Arguments

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

staging-table

The staging table from which to swap partitions.

min-range-value max-range-value

The minimum and maximum value of partition keys to swap, where *min-range-value* must be ≤ *max-range-value* . To swap one partition, *min-range-value* and *max-range-value* must be equal.

target-table

The table to which the partitions are to be swapped. The target table cannot be the same as the staging table.

force-split

Optional Boolean argument, specifies whether to split ROS containers if the range of partition keys spans multiple containers or part of a single container:

- true : Split ROS containers as needed.
- false (default): Return with an error if ROS containers must be split to implement this operation.

Privileges

Non-superuser, one of the following:

- Owner of source and target tables
- Target and source tables: TRUNCATE, INSERT, SELECT

Requirements

The following attributes of both tables must be identical:

- Column definitions, including NULL/NOT NULL constraints
- Segmentation
- Partition clause
- Number of projections
- Projection sort order
- Primary and unique key constraints. However, the key constraints do not have to be identically enabled. For more information on constraints, see Constraints .

> **Note**
> If the target table has primary or unique key constraints enabled and copying or moving the partitions will insert duplicate key values into the target table, Vertica rolls back the operation.

- Check constraints. For MOVE_PARTITIONS_TO_TABLE and COPY_PARTITIONS_TO_TABLE , Vertica enforces enabled check constraints on the target table only. For SWAP_PARTITIONS_BETWEEN_TABLES , Vertica enforces enabled check constraints on both tables. If there is a violation of an enabled check constraint, Vertica rolls back the operation.
- Number and definitions of text indices.

Additionally, If access policies exist on the source table, the following must be true:

- Access policies on both tables must be identical.
- One of the following must be true:
  - The executing user owns the target table.
  - AccessPolicyManagementSuperuserOnly is set to true.

Restrictions

The following restrictions apply to the source and target tables:

- If the source and target partitions are in different storage tiers, Vertica returns a warning but the operation proceeds. The partitions remain in their existing storage tier.
- The target table cannot be immutable .
- The following tables cannot be used as sources or targets:
  - Temporary tables
  - Virtual tables
  - System tables
  - External tables

Examples

See Swapping partitions .

Privileges and access functions

This section contains functions for managing user and role privileges, and access policies.

In this section

- ENABLED_ROLE
- GET_PRIVILEGES_DESCRIPTION
- HAS_ROLE
- RELEASE_SYSTEM_TABLES_ACCESS
- RESTRICT_SYSTEM_TABLES_ACCESS

ENABLED_ROLE

Checks whether a Vertica user role is enabled, and returns true or false. This function is typically used when you create access policies on database roles.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ENABLED_ROLE ( 'role' )
```

Parameters

role

The role to evaluate.

Privileges

None

Examples

See:

- Creating column access policies
- Creating row access policies

See also

CREATE ACCESS POLICY

GET_PRIVILEGES_DESCRIPTION

Returns the effective privileges the current user has on an object, including explicit, implicit , inherited , and role-based privileges.

Because this meta-function only returns effective privileges , GET_PRIVILEGES_DESCRIPTION only returns privileges with fully-satisfied prerequisites. For a list of prerequisites for common operations, see Privileges required for common database operations .

For example, a user must have the following privileges to query a table:

- Schema: USAGE
- Table: SELECT

If user Brooke has SELECT privileges on table s1.t1 but lacks USAGE privileges on schema s1 , Brooke cannot query the table, and GET_PRIVILEGES_DESCRIPTION does not return SELECT as a privilege for the table.

> **Note**
> Inherited privileges are not displayed if privilege inheritance is disabled at the database level.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_PRIVILEGES_DESCRIPTION( 'type', '[[database.]schema.]name' );
```

Parameters

type

Specifies an object type, one of the following:

- database
- table
- schema
- view
- sequence
- model
- library
- resource pool

[*database* .] *schema*

Specifies a database and schema, by default the current database and public , respectively.

name

Name of the target object

Privileges

None

Examples

In the following example, user Glenn has set the REPORTER role and wants to check his effective privileges on schema s1 and table s1.articles .

- Table s1.articles inherits privileges from its schema (s1).
- The REPORTER role has the following privileges:
  - SELECT on schema s1
  - INSERT WITH GRANT OPTION on table s1.articles
- User Glenn has the following privileges:
  - UPDATE and USAGE on schema s1 .
  - DELETE on table s1.articles .

GET_PRIVILEGES_DESCRIPTION returns the following effective privileges for Glenn on schema s1 :

```
=> SELECT GET_PRIVILEGES_DESCRIPTION('schema', 's1');
  GET_PRIVILEGES_DESCRIPTION
---------------------------------
 SELECT, UPDATE, USAGE
(1 row)
```

GET_PRIVILEGES_DESCRIPTION returns the following effective privileges for Glenn on table s1.articles :

```
=> SELECT GET_PRIVILEGES_DESCRIPTION('table', 's1.articles');
  GET_PRIVILEGES_DESCRIPTION
--------------------------------
 INSERT*, SELECT, UPDATE, DELETE
(1 row)
```

See also

- Database users and privileges
- Database roles
- Granting and revoking privileges

HAS_ROLE

Checks whether a Vertica user role is granted to the specified user or role, and returns true or false.

You can also query system tables ROLES , GRANTS , and USERS to obtain information on users and their role assignments. For details, see Viewing user roles .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Stable

Syntax

```
HAS_ROLE( [ 'grantee' ,] 'verify-role' );
```

Parameters

grantee

Valid only for superusers, specifies the name of a user or role to look up. If this argument is omitted, the function uses the current user name (CURRENT_USER). If you specify a role, Vertica checks whether this role is granted to the role specified in *verify-role* .

Important

If a non-superuser supplies this argument, Vertica returns an error.

verify-role

Name of the role to verify for *grantee* .

Privileges

None

Examples

In the following example, a dbadmin user checks whether user MikeL is assigned the admnistrator role:

```
=> \c
You are now connected as user "dbadmin".
=> SELECT HAS_ROLE('MikeL', 'administrator');
 HAS_ROLE
----------
 t
(1 row)
```

User MikeL checks whether he has the regional_manager role:

```
=> \c - MikeL
You are now connected as user "MikeL".
=> SELECT HAS_ROLE('regional_manager');
 HAS_ROLE
----------
 f
(1 row)
```

The dbadmin grants the regional_manager role to the administrator role. On checking again, MikeL verifies that he now has the regional_manager role:

```
dbadmin=> \c
You are now connected as user "dbadmin".
dbadmin=> GRANT regional_manager to administrator;
GRANT ROLE
dbadmin=> \c - MikeL
You are now connected as user "MikeL".
dbadmin=> SELECT HAS_ROLE('regional_manager');
 HAS_ROLE
----------
 t
(1 row)
```

See also

- GRANTS
- ROLES
- USERS
- Database users and privileges

RELEASE_SYSTEM_TABLES_ACCESS

Allows non-superusers to access all non- SUPERUSER_ONLY system tables. After you call this function, Vertica ignores the IS_ACCESSIBLE_DURING_LOCKDOWN setting in table SYSTEM_TABLES . To restrict non-superusers access to system tables, call RESTRICT_SYSTEM_TABLES_ACCESS .

By default, the database behaves as though RELEASE_SYSTEM_TABLES_ACCESS() was called. That is, non-superusers have access to all non-SUPERUSER_ONLY system tables.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RELEASE_SYSTEM_TABLES_ACCESS()
```

Privileges

Superuser

Examples

By default, non-superuser Alice has access to client_auth and disk_storage . She also has access to replication_status because she was granted the privilege by the dbadmin:

```
=> SELECT table_name, is_superuser_only, is_accessible_during_lockdown FROM system_tables WHERE table_name='disk_storage' OR table_name='dat
    table_name      | is_superuser_only | is_accessible_during_lockdown
--------------------+-------------------+-------------------------------
client_auth         | f                 | t
disk_storage        | f                 | f
database_backups    | t                 | f
replication_status | t                 | t
(4 rows)
```

The dbadmin calls RESTRICT_SYSTEM_TABLES_ACCESS :

```
=> SELECT RESTRICT_SYSTEM_TABLES_ACCESS();
                  RESTRICT_SYSTEM_TABLES_ACCESS
-----------------------------------------------------------------
Dropped grants to public on non-accessible during lockdown system tables.

(1 row)
```

Alice loses access to disk_storage , but she retains access to client_auth and replication_status because their IS_ACCESSIBLE_DURING_LOCKDOWN fields are true:

```
=> SELECT storage_status FROM disk_storage;
ERROR 4367:  Permission denied for relation disk_storage
```

The dbadmin calls RELEASE_SYSTEM_TABLES_ACCESS(), restoring Alice's access to disk_storage :

```
=> SELECT RELEASE_SYSTEM_TABLES_ACCESS();
          RELEASE_SYSTEM_TABLES_ACCESS
-----------------------------------------------------
Granted SELECT privileges on system tables to public.

(1 row)
```

RESTRICT_SYSTEM_TABLES_ACCESS

Prevents non-superusers from accessing tables that have the IS_ACCESSIBLE_DURING_LOCKDOWN flag set to false.

To enable non-superuser access to system tables restricted by this function, call RELEASE_SYSTEM_TABLES_ACCESS .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RESTRICT_SYSTEM_TABLES_ACCESS()
```

Privileges

Superuser

Examples

By default, client_auth and disk_storage tables are accessible to all users, but only the former is accessible after RESTRICT_SYSTEM_TABLES_ACCESS() is called. Non-superusers never have access to database_backups and replication_status unless explicitly granted the privilege by the dbadmin:

```
=> SELECT table_name, is_superuser_only, is_accessible_during_lockdown FROM system_tables WHERE table_name='disk_storage' OR table_name='dat
    table_name     | is_superuser_only | is_accessible_during_lockdown
-------------------+-------------------+------------------------------
client_auth        | f                 | t
disk_storage       | f                 | f
database_backups   | t                 | f
replication_status | t                 | t
(4 rows)
```

The dbadmin then calls RESTRICT_SYSTEM_TABLES_ACCESS():

```
=> SELECT RESTRICT_SYSTEM_TABLES_ACCESS();
                  RESTRICT_SYSTEM_TABLES_ACCESS
-------------------------------------------------------------------------
Dropped grants to public on non-accessible during lockdown system tables.

(1 row)
```

Bob loses access to disk_storage , but retains access to client_auth because its IS_ACCESSIBLE_DURING_LOCKDOWN field is true:

```
=> SELECT storage_status FROM disk_storage;
ERROR 4367:  Permission denied for relation disk_storage

=> SELECT auth_oid FROM client_auth;
    auth_oid
-----------------
 45035996273705106
 45035996273705110
 45035996273705114
(3 rows)
```

Projection functions

This section contains projection management functions specific to Vertica.

See also

- V_CATALOG.PROJECTIONS
- V_CATALOG.PROJECTION_COLUMNS
- V_MONITOR.PROJECTION_REFRESHES
- V_MONITOR.PROJECTION_STORAGE

In this section

- CLEAR_PROJECTION_REFRESHES
- EVALUATE_DELETE_PERFORMANCE
- GET_PROJECTION_SORT_ORDER
- GET_PROJECTION_STATUS
- GET_PROJECTIONS
- PURGE_PROJECTION
- REFRESH
- REFRESH_COLUMNS
- START_REFRESH

CLEAR_PROJECTION_REFRESHES

Clears information projection refresh history from system table PROJECTION_REFRESHES .

System table PROJECTION_REFRESHES records information about refresh operations , successful and unsuccessful. PROJECTION_REFRESHES retains projection refresh data until one of the following events occurs:

- Another refresh operation starts on a given projection.
- CLEAR_PROJECTION_REFRESHES is called and clears data on all projections.
- The table's storage quota is exceeded.

CLEAR_PROJECTION_REFRESHES checks PROJECTION_REFRESHES Boolean column IS_EXECUTING to determine whether refresh operations are still running or are complete. The function only removes information for refresh operations that are complete.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAR_PROJECTION_REFRESHES()
```

Privileges

Superuser

Examples

```
=> SELECT CLEAR_PROJECTION_REFRESHES();
 CLEAR_PROJECTION_REFRESHES
----------------------------
 CLEAR
(1 row)
```

See also

- REFRESH
- START_REFRESH
- Clearing projection refresh history

EVALUATE_DELETE_PERFORMANCE

Evaluates projections for potential DELETE and UPDATE performance issues. If Vertica finds any issues, it issues a warning message. When evaluating multiple projections, EVALUATE_DELETE_PERFORMANCE returns up to ten projections with issues, and the name of a table that lists all issues that it found.

> **Note**
> EVALUATE_DELETE_PERFORMANCE returns messages that specifically reference delete performance. Keep in mind, however, that delete and update operations benefit equally from the same optimizations.

For information on resolving delete and update performance issues, see Optimizing DELETE and UPDATE .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
EVALUATE_DELETE_PERFORMANCE ( ['[[database.]schema.]scope'] )
```

Parameters

`[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

scope

Specifies the projections to evaluate, one of the following:

- [*table* .] *projection*
  Evaluate *projection* . For example:
  ```
  SELECT EVALUATE_DELETE_PERFORMANCE('store.store_orders_fact.store_orders_fact_b1');
  ```
- *table*
  Specifies to evaluate all projections of *table* . For example:
  ```
  SELECT EVALUATE_DELETE_PERFORMANCE('store.store_orders_fact');
  ```

If you supply no arguments, EVALUATE_DELETE_PERFORMANCE evaluates all projections that you can access. Depending on the size of your database, this can incur considerable overhead.

Privileges

Non-superuser: SELECT privilege on the anchor table

Examples

EVALUATE_DELETE_PERFORMANCE evaluates all projections of table exampl e for potential DELETE and UPDATE performance issues.

```
=> create table example (A int, B int,C int);
CREATE TABLE
=> create projection one_sort (A,B,C) as (select A,B,C from example) order by A;
CREATE PROJECTION
=> create projection two_sort (A,B,C) as (select A,B,C from example) order by A,B;
CREATE PROJECTION
=> select evaluate_delete_performance('example');
        evaluate_delete_performance
-----------------------------------------------------
 No projection delete performance concerns found.
(1 row)
```

The previous example show that the two projections one_sort and two_sort have no inherent structural issues that might cause poor DELETE performance. However, the data contained within the projection can create potential delete issues if the sorted columns do not uniquely identify a row or small number of rows.

In the following example, Perl is used to populate the table with data using a nested series of loops:

- The inner loop populates column C.
- The middle loop populates column B .
- The outer loop populates column A .

The result is column A contains only three distinct values (0, 1, and 2), while column B slowly varies between 20 and 0 and column C changes in each row:

```
=> \! perl -e 'for ($i=0; $i<3; $i++) { for ($j=0; $j<21; $j++) { for ($k=0; $k<19; $k++) { printf "%d,%d,%d\n", $i,$j,$k;}}}' | /opt/vertica/bin/vsql -c "copy example
from stdin delimiter ',' direct;"
Password:
=> select * from example;
 A | B  | C
---+----+----
 0 | 20 | 18
 0 | 20 | 17
 0 | 20 | 16
 0 | 20 | 15
 0 | 20 | 14
 0 | 20 | 13
 0 | 20 | 12
 0 | 20 | 11
 0 | 20 | 10
 0 | 20 |  9
 0 | 20 |  8
 0 | 20 |  7
 0 | 20 |  6
 0 | 20 |  5
 0 | 20 |  4
 0 | 20 |  3
 0 | 20 |  2
 0 | 20 |  1
 0 | 20 |  0
 0 | 19 | 18
 ...
 2 |  1 |  0
 2 |  0 | 18
 2 |  0 | 17
 2 |  0 | 16
 2 |  0 | 15
 2 |  0 | 14
 2 |  0 | 13
 2 |  0 | 12
 2 |  0 | 11
 2 |  0 | 10
 2 |  0 |  9
 2 |  0 |  8
 2 |  0 |  7
 2 |  0 |  6
 2 |  0 |  5
 2 |  0 |  4
 2 |  0 |  3
 2 |  0 |  2
 2 |  0 |  1
 2 |  0 |  0
=> SELECT COUNT (*) FROM example;
 COUNT
-------
  1197
(1 row)
=> SELECT COUNT (DISTINCT A) FROM example;
 COUNT
-------
     3
(1 row)
```

EVALUATE_DELETE_PERFORMANCE is run against the projections again to determine whether the data within the projections causes any potential DELETE performance issues. Projection one_sort has potential delete issues as it only sorts on column A which has few distinct values. Each value in the sort column corresponds to many rows in the projection, which can adversely impact DELETE performance. In contrast, projection two_sort is

sorted on columns A and B , where each combination of values in the two sort columns identifies just a few rows, so deletes can be performed faster:

```
=> select evaluate_delete_performance('example');
           evaluate_delete_performance
-----------------------------------------------------
 The following projections exhibit delete performance concerns:
       "public"."one_sort_b1"
       "public"."one_sort_b0"
See v_catalog.projection_delete_concerns for more details.

=> \x
Expanded display is on.
dbadmin=> select * from projection_delete_concerns;
-[ RECORD 1 ]------+-----------------------------------------------------------------------------------------------------------------------------------------------------------
projection_id      | 45035996273878562
projection_schema  | public
projection_name    | one_sort_b1
creation_time      | 2019-06-17 13:59:03.777085-04
last_modified_time | 2019-06-17 14:00:27.702223-04
comment            | The squared number of rows matching each sort key is about 159201 on average.
-[ RECORD 2 ]------+-----------------------------------------------------------------------------------------------------------------------------------------------------------
projection_id      | 45035996273878548
projection_schema  | public
projection_name    | one_sort_b0
creation_time      | 2019-06-17 13:59:03.777279-04
last_modified_time | 2019-06-17 13:59:03.777279-04
comment            | The squared number of rows matching each sort key is about 159201 on average.
```

If you omit supplying an argument to EVALUATE_DELETE_PERFORMANCE, it evaluates all projections that you can access:

```
=> select evaluate_delete_performance();
                       evaluate_delete_performance
-------------------------------------------------------------------------
 The following projections exhibit delete performance concerns:
       "public"."one_sort_b0"
       "public"."one_sort_b1"
See v_catalog.projection_delete_concerns for more details.
(1 row)
```

GET_PROJECTION_SORT_ORDER

Returns the order of columns in a projection's ORDER BY clause.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_PROJECTION_SORT_ORDER( '[[database.]schema.]projection' );
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection

The target projection.

Privileges

Non-superuser: SELECT privilege on the anchor table

Examples

```
=> SELECT get_projection_sort_order ('store_orders_super');
                              get_projection_sort_order
-----------------------------------------------------------------------------------------------
 public.store_orders_super [Sort Cols: "order_no", "order_date", "shipper", "ship_date"]

(1 row)
```

GET_PROJECTION_STATUS

Returns information relevant to the status of a projection :

- The current K-safety status of the database
- The number of nodes in the database
- Whether the projection is segmented
- The number and names of buddy projections
- Whether the projection is safe
- Whether the projection is up to date
- Whether statistics have been computed for the projection

Use GET_PROJECTION_STATUS to monitor the progress of a projection data refresh.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_PROJECTION_STATUS ( '[[database.]schema.]projection' );
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection

The projection for which to display status.

Examples

```
=> SELECT GET_PROJECTION_STATUS('public.customer_dimension_site01');
                               GET_PROJECTION_STATUS
-----------------------------------------------------------------------------------------------
 Current system K is 1.
# of Nodes: 4.
public.customer_dimension_site01 [Segmented: No] [Seg Cols: ] [K: 3] [public.customer_dimension_site04, public.customer_dimension_site03,
public.customer_dimension_site02]
[Safe: Yes] [UptoDate: Yes][Stats: Yes]
```

GET_PROJECTIONS

Returns contextual and projection information about projections of the specified anchor table.

Contextual information

- Database K-safety
- Number of database nodes
- Number of projections for this table

Projection data

For each projection, specifies:

- All buddy projections
- Whether it is segmented
- Whether it is safe
- Whether it is up-to-date.

You can also use GET_PROJECTIONS to monitor the progress of a projection data refresh .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_PROJECTIONS ( '[[database.]schema-name.]table' )
```

Parameters

`[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Anchor table of the projections to list.

Privileges

None

Examples

The following example gets information about projections for VMart table store.store_dimension :

```
=> SELECT GET_PROJECTIONS('store.store_dimension');
-[ RECORD 1 ]---+
GET_PROJECTIONS | Current system K is 1.
# of Nodes: 3.
Table store.store_dimension has 2 projections.

Projection Name: [Segmented] [Seg Cols] [# of Buddies] [Buddy Projections] [Safe] [UptoDate] [Stats]
-------------------------------------------------------------------------------------------------
store.store_dimension_b1 [Segmented: Yes] [Seg Cols: "store.store_dimension.store_key"] [K: 1] [store.store_dimension_b0] [Safe: Yes] [UptoDate: Yes]
[Stats: RowCounts]
store.store_dimension_b0 [Segmented: Yes] [Seg Cols: "store.store_dimension.store_key"] [K: 1] [store.store_dimension_b1] [Safe: Yes] [UptoDate: Yes]
[Stats: RowCounts]
```

PURGE_PROJECTION

Permanently removes deleted data from physical storage so disk space can be reused. You can purge historical data up to and including the Ancient History Mark epoch.

> **Caution**
> PURGE_PROJECTION can use significant disk space while purging the data.

See PURGE for details about purge operations.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
PURGE_PROJECTION ( '[[database.]schema.]projection' )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection

The projection to purge.

Privileges

- Table owner
- USAGE privilege on schema

Examples

The following example purges all historical data in projection tbl_p that precedes the Ancient History Mark epoch.

```
=> CREATE TABLE tbl (x int, y int);
CREATE TABLE
=> INSERT INTO tbl VALUES(1,2);
 OUTPUT
--------
      1
(1 row)

=> INSERT INTO tbl VALUES(3,4);
 OUTPUT
--------
      1
(1 row)

dbadmin=> COMMIT;
COMMIT
=> CREATE PROJECTION tbl_p AS SELECT x FROM tbl UNSEGMENTED ALL NODES;
WARNING 4468: Projection <public.tbl_p> is not available for query processing.
Execute the select start_refresh() function to copy data into this projection.
The projection must have a sufficient number of buddy projections and all nodes must be up before starting a refresh
CREATE PROJECTION
=> SELECT START_REFRESH();
             START_REFRESH
-----------------------------------------
 Starting refresh background process.
=> DELETE FROM tbl WHERE x=1;
 OUTPUT
--------
      1
(1 row)

=> COMMIT;
COMMIT
=> SELECT MAKE_AHM_NOW();
         MAKE_AHM_NOW
-------------------------------
 AHM set (New AHM Epoch: 9066)
(1 row)

=> SELECT PURGE_PROJECTION ('tbl_p');
 PURGE_PROJECTION
-------------------
 Projection purged
(1 row)
```

See also

- PURGE_TABLE
- STORAGE_CONTAINERS
- Purging deleted data .

REFRESH

Synchronously refreshes one or more table projections in the foreground, and updates the PROJECTION_REFRESHES system table. If you run REFRESH with no arguments, it refreshes all projections that contain stale data.

To understand projection refreshing in detail, see Refreshing projections .

If a refresh would violate a table or schema disk quota, the operation fails. For more information, see Disk quotas .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
REFRESH ( [ '[[database.]schema.]table[,...]' ] )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

The anchor table of the projections to refresh. If you specify multiple tables, REFRESH attempts to refresh them in parallel. Such calls are part of the Database Designer deployment (and deployment script).

Returns

> **Note**
> If REFRESH does not refresh any projections, it returns a header string with no results.

Column	Returns
Projection Name	The projection targeted for refresh.
Anchor Table	The projection's associated anchor table.
Status	Projections' refresh status: • queued : Queued for refresh. • refreshing : Refresh is in process. • refreshed : Refresh successfully completed. • failed : Refresh did not successfully complete.
Refresh Method	Method used to refresh the projection.
Error Count	Number of times a refresh failed for the projection.
Duration (sec)	How long (in seconds) the projection refresh ran.

Privileges

- Superuser
- Owner of the specified tables

Refresh methods

Vertica can refresh a projection from one of its buddies, if one is available. In this case, the target projection gets the source buddy's historical data. Otherwise, the projection is refreshed from scratch with data of the latest epoch at the time of the refresh operation. In this case, the projection cannot participate in historical queries on any epoch that precedes the refresh operation.

To determine the method used to refresh a given projection, query the REFRESH_METHOD column from the PROJECTION_REFRESHES system table.

Examples

The following example refreshes the projections in two tables:

```
=> SELECT REFRESH('t1, t2');
                                        REFRESH
-------------------------------------------------------------------------------------------
Refresh completed with the following outcomes:

Projection Name: [Anchor Table] [Status] [Refresh Method] [Error Count] [Duration (sec)]
-------------------------------------------------------------------------------------------

"public"."t1_p": [t1] [refreshed] [scratch] [0] [0]"public"."t2_p": [t2] [refreshed] [scratch] [0] [0]
```

In the following example, only the projection on one table was refreshed:

```
=> SELECT REFRESH('allow, public.deny, t');
                                        REFRESH
-------------------------------------------------------------------------------------------

Refresh completed with the following outcomes:

Projection Name: [Anchor Table] [Status] [Refresh Method] [Error Count] [Duration (sec)]
-------------------------------------------------------------------------------------------
"n/a"."n/a": [n/a] [failed: insufficient permissions on table "allow"] [] [1] [0]
"n/a"."n/a": [n/a] [failed: insufficient permissions on table "public.deny"] [] [1] [0]
"public"."t_p1": [t] [refreshed] [scratch] [0] [0]
```

See also

- CLEAR_PROJECTION_REFRESHES
- START_REFRESH

REFRESH_COLUMNS

Refreshes table columns that are defined with the constraint SET USING or DEFAULT USING . All refresh operations associated with a call to REFRESH_COLUMNS belong to the same transaction. Thus, all tables and columns specified by REFRESH_COLUMNS must be refreshed; otherwise, the entire operation is rolled back.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
REFRESH_COLUMNS ( 'table-list', '[column-list]'
  [, '[refresh-mode]' [, min-partition-key, max-partition-key [, force-split] ]
)
```

Parameters

table-list

A comma-delimited list of the tables to refresh:

[[*database* .] *schema.*] *table* [,...]

Important

If you specify multiple tables, parameter *refresh-mode* must be set to REBUILD.

column-list

A comma-delimited list of columns to refresh, specified as follows:

- [[[*database* .] *schema.*] *table* .] *column* [,...]
- [[*database* .] *schema.*] *table* . *

where asterisk (*) specifies to refresh all SET USING/DEFAULT USING columns in *table* . For example:

```
SELECT REFRESH_COLUMNS ('t1, t2', 't1.*, t2.b', 'REBUILD');
```

If *column-list* is set to an empty string (''), REFRESH_COLUMNS refreshes all SET USING/DEFAULT USING columns in the specified tables.

The following requirements apply:

- All specified columns must have a SET USING or DEFAULT USING constraint.
- If REFRESH_COLUMNS specifies multiple tables, all column names must be qualified by their table names. If the target tables span multiple schemas, all column names must be fully qualified by their schema and table names. For example:

```
SELECT REFRESH_COLUMNS ('t1, t2', 't1.a, t2.b', 'REBUILD');
```

If you specify a database, it must be the current database.

refresh-mode

Specifies how to refresh SET USING columns:

- UPDATE : Marks original rows as deleted and replaces them with new rows. In order to save these updates, you must issue a COMMIT statement.
- REBUILD : Replaces all data in the specified columns. The rebuild operation is auto-committed.

If set to an empty string or omitted, REFRESH_COLUMNS executes in UPDATE mode. If you specify multiple tables, you must explicitly specify REBUILD mode.
In both cases, REFRESH_COLUMNS returns an error if any SET USING column is defined as a primary or unique key in a table that enforces those constraints .

See REBUILD Mode Restrictions for limitations on using the REBUILD option.

min-partition-key
max-partition-key

Qualifies REBUILD mode, limiting the rebuild operation to one or more partitions. To specify a range of partitions, *max-partition-key* must be greater than *min-partition-key* . To update one partition, the two arguments must be equal.

The following requirements apply:

- The function can specify only one table to refresh.
- The table must be partitioned on the specified keys.

You can use these arguments to refresh columns with recently loaded data—that is, data in the latest partitions. Using this option regularly can significantly minimize the overhead otherwise incurred by rebuilding entire columns in a large table.

See Partition-based REBUILD below for details.

force-split

Boolean argument, specifies whether to split ROS containers if the range of partition keys spans multiple containers or part of a single container:

- *true* (default): Split ROS containers as needed.
- *false* : Return with an error if ROS containers must be split to implement this operation:

Privileges

- Schemas of queried and flattened tables: USAGE
- Queried table: SELECT
- Flattened table: SELECT, UPDATE

UPDATE versus REBUILD modes

In general, UPDATE mode is a better choice when changes to SET USING column data are confined to a relatively small number of rows. Use REBUILD mode when a significant amount of SET USING column data is stale and must be updated. It is generally good practice to call REFRESH_COLUMNS with REBUILD on any new SET USING column—for example, to populate a SET USING column after adding it with ALTER TABLE...ADD COLUMN.

REBUILD mode restrictions

If you call REFRESH_COLUMNS on a SET USING column and specify the refresh mode as REBUILD, Vertica returns an error if the column is specified in any of the following:

- Table's partition key
- Unsegmented projection
- Projection with expressions , or any live aggregate projection that invokes a user-defined transform function (UDTF)
- Sort order or segmentation of any projection
- Any projection that omits an anchor table column that is referenced in the column's SET USING expression
- GROUPED clause of any projection

Partition-based REBUILD operations

If a flattened table is partitioned, you can reduce the overhead of calling REFRESH_COLUMNS in REBUILD mode, by specifying one or more partition keys. Doing so limits the rebuild operation to the specified partitions. For example, table public.orderFact is defined with SET USING column cust_name . This table is partitioned on column order_date , where the partition clause invokes Vertica function CALENDAR_HIERARCHY_DAY . Thus, you can call REFRESH_COLUMNS on specific time-delimited partitions of this table—in this case, on orders over the last two months:

```
=> SELECT REFRESH_COLUMNS ('public.orderFact',
                  'cust_name',
                  'REBUILD',
                  TO_CHAR(ADD_MONTHS(current_date, -2),'YYYY-MM')||'-01',
                  TO_CHAR(LAST_DAY(ADD_MONTHS(current_date, -1))));
    REFRESH_COLUMNS
---------------------------
 refresh_columns completed
(1 row)
```

Rewriting SET USING queries

When you call REFRESH_COLUMNS on a flattened table 's SET USING (or DEFAULT USING) column, it executes the SET USING query by joining the target and source tables. By default, the source table is always the inner table of the join. In most cases, cardinality of the source table is less than the target table, so REFRESH_COLUMNS executes the join efficiently.

Occasionally—notably, when you call REFRESH_COLUMNS on a partitioned table—the source table can be larger than the target table. In this case, performance of the join operation can be suboptimal.

You can address this issue by enabling configuration parameter RewriteQueryForLargeDim . When enabled (1), Vertica rewrites the query, by reversing the inner and outer join between the target and source tables.

Important
Enable this parameter only if the SET USING source data is in a table that is larger than the target table. If the source data is in a table smaller than the target table, then enabling RewriteQueryForLargeDim can adversely affect refresh performance.

Examples

See Flattened table example and DEFAULT versus SET USING .

START_REFRESH

Refreshes projections in the current schema with the latest data of their respective anchor tables . START_REFRESH runs asynchronously in the background, and updates the PROJECTION_REFRESHES system table. This function has no effect if a refresh is already running.

To refresh only projections of a specific table, use REFRESH . When you deploy a design through Database Designer, it automatically refreshes its projections.

If a refresh would violate a table or schema disk quota, the operation fails. For more information, see Disk quotas .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
START_REFRESH()
```

Privileges

None

Requirements

All nodes must be up.

Refresh methods

Vertica can refresh a projection from one of its buddies, if one is available. In this case, the target projection gets the source buddy's historical data. Otherwise, the projection is refreshed from scratch with data of the latest epoch at the time of the refresh operation. In this case, the projection cannot participate in historical queries on any epoch that precedes the refresh operation.

To determine the method used to refresh a given projection, query the REFRESH_METHOD column from the PROJECTION_REFRESHES system table.

Examples

```
=> SELECT START_REFRESH();
         START_REFRESH
----------------------------------------
 Starting refresh background process.
(1 row)
```

See also

- Refreshing projections
- CLEAR_PROJECTION_REFRESHES

Session functions

This section contains session management functions specific to Vertica.

See also the SQL system table V_MONITOR.SESSIONS .

In this section

- CANCEL_REFRESH
- CLOSE_ALL_SESSIONS
- CLOSE_SESSION
- CLOSE_USER_SESSIONS
- GET_NUM_ACCEPTED_ROWS
- GET_NUM_REJECTED_ROWS
- INTERRUPT_STATEMENT
- RELEASE_ALL_JVM_MEMORY
- RELEASE_JVM_MEMORY
- RESERVE_SESSION_RESOURCE
- RESET_SESSION

CANCEL_REFRESH

Cancels refresh-related internal operations initiated by START_REFRESH and REFRESH .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CANCEL_REFRESH()
```

Privileges

None

Notes

- Refresh tasks run in a background thread in an internal session, so you cannot use INTERRUPT_STATEMENT to cancel those statements. Instead, use CANCEL_REFRESH to cancel statements that are run by refresh-related internal sessions.
- Run CANCEL_REFRESH() on the same node on which START_REFRESH() was initiated.
- CANCEL_REFRESH() cancels the refresh operation running on a node, waits for the cancelation to complete, and returns SUCCESS.
- Only one set of refresh operations runs on a node at any time.

Examples

Cancel a refresh operation executing in the background.

```
=> SELECT START_REFRESH();
         START_REFRESH
----------------------------------------
Starting refresh background process.
(1 row)
=> SELECT CANCEL_REFRESH();
          CANCEL_REFRESH
----------------------------------------
Stopping background refresh process.
(1 row)
```

See also

- INTERRUPT_STATEMENT
- SESSIONS
- START_REFRESH
- PROJECTION_REFRESHES

CLOSE_ALL_SESSIONS

Closes all external sessions except the one that issues this function. Call this function before shutting down the Vertica database.

Vertica closes sessions asynchronously, so another session can open before this function returns. In this case, reissue this function. To view the status of all open sessions, query system table SESSIONS .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLOSE_ALL_SESSIONS()
```

Privileges

Non-superuser: None to close your own session

Examples

Two user sessions are open on separate nodes:

```
=> SELECT * FROM sessions;
-[ RECORD 1 ]----------+-----------------------------------------------------
node_name              | v_vmartdb_node0001
user_name              | dbadmin
client_hostname        | 127.0.0.1:52110
client_pid             | 4554
login_timestamp        | 2011-01-03 14:05:40.252625-05
session_id             | stress04-4325:0x14
client_label           |
transaction_start      | 2011-01-03 14:05:44.325781
transaction_id         | 45035996273728326
transaction_description | user dbadmin (select * from sessions;)
statement_start        | 2011-01-03 15:36:13.896288
statement_id           | 10
last_statement_duration_us | 14978
current_statement      | select * from sessions;
ssl_state              | None
authentication_method  | Trust
-[ RECORD 2 ]----------+-----------------------------------------------------
node_name              | v_vmartdb_node0002
user_name              | dbadmin
client_hostname        | 127.0.0.1:57174
client_pid             | 30117
login_timestamp        | 2011-01-03 15:33:00.842021-05
session_id             | stress05-27944:0xc1a
client_label           |
transaction_start      | 2011-01-03 15:34:46.538102
transaction_id         | -1
transaction_description | user dbadmin (COPY Mart_Fact FROM '/data/mart_Fact.tbl'
                       DELIMITER '|' NULL '\\n';)
statement_start        | 2011-01-03 15:34:46.538862
statement_id           |
last_statement_duration_us | 26250
current_statement      | COPY Mart_Fact FROM '/data/Mart_Fact.tbl' DELIMITER '|'
                       NULL '\\n';
ssl_state              | None
authentication_method  | Trust
-[ RECORD 3 ]----------+-----------------------------------------------------
node_name              | v_vmartdb_node0003
user_name              | dbadmin
client_hostname        | 127.0.0.1:56367
client_pid             | 1191
login_timestamp        | 2011-01-03 15:31:44.939302-05
session_id             | stress06-25663:0xbec
client_label           |
transaction_start      | 2011-01-03 15:34:51.05939
transaction_id         | 54043195528458775
transaction_description | user dbadmin (COPY Mart_Fact FROM '/data/Mart_Fact.tbl'
                       DELIMITER '|' NULL '\\n' DIRECT;)
statement_start        | 2011-01-03 15:35:46.436748
statement_id           |
last_statement_duration_us | 1591403
current_statement      | COPY Mart_Fact FROM '/data/Mart_Fact.tbl' DELIMITER '|'
                       NULL '\\n' DIRECT;
ssl_state              | None
authentication_method  | Trust
```

Close all sessions:

```
=> \x
Expanded display is off.
=> SELECT CLOSE_ALL_SESSIONS();
                        CLOSE_ALL_SESSIONS
-------------------------------------------------------------------------
 Close all sessions command sent. Check v_monitor.sessions for progress.
(1 row)
```

Session contents after issuing CLOSE_ALL_SESSIONS :

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1 ]----------+---------------------------------------
node_name              | v_vmartdb_node0001
user_name              | dbadmin
client_hostname        | 127.0.0.1:52110
client_pid             | 4554
login_timestamp        | 2011-01-03 14:05:40.252625-05
session_id             | stress04-4325:0x14
client_label           |
transaction_start      | 2011-01-03 14:05:44.325781
transaction_id         | 45035996273728326
transaction_description | user dbadmin (SELECT * FROM sessions;)
statement_start        | 2011-01-03 16:19:56.720071
statement_id           | 25
last_statement_duration_us | 15605
current_statement      | SELECT * FROM SESSIONS;
ssl_state              | None
authentication_method  | Trust
```

See also

- CLOSE_SESSION
- CLOSE_USER_SESSIONS
- SHUTDOWN
- Managing sessions

CLOSE_SESSION

Interrupts the specified external session, rolls back the current transaction if any, and closes the socket. You can only close your own session.

It might take some time before a session is closed. To view the status of all open sessions, query the system table SESSIONS .

For detailed information about session management options, see Managing sessions .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLOSE_SESSION ( 'sessionid')
```

Parameters

sessionid

A string that specifies the session to close. This identifier is unique within the cluster at any point in time but can be reused when the session closes.

Privileges

None

Examples

User session opened. Record 2 shows the user session running a COPY DIRECT statement.

```
=> SELECT * FROM sessions;
-[ RECORD 1 ]----------+-------------------------------------------------
node_name              | v_vmartdb_node0001
user_name              | dbadmin
client_hostname        | 127.0.0.1:52110
client_pid             | 4554
login_timestamp        | 2011-01-03 14:05:40.252625-05
session_id             | stress04-4325:0x14
client_label           |
transaction_start      | 2011-01-03 14:05:44.325781
transaction_id         | 45035996273728326
transaction_description | user dbadmin (SELECT * FROM sessions;)
statement_start        | 2011-01-03 15:36:13.896288
statement_id           | 10
last_statement_duration_us | 14978
current_statement      | select * from sessions;
ssl_state              | None
authentication_method  | Trust
-[ RECORD 2 ]----------+-------------------------------------------------
node_name              | v_vmartdb_node0002
user_name              | dbadmin
client_hostname        | 127.0.0.1:57174
client_pid             | 30117
login_timestamp        | 2011-01-03 15:33:00.842021-05
session_id             | stress05-27944:0xc1a
client_label           |
transaction_start      | 2011-01-03 15:34:46.538102
transaction_id         | -1
transaction_description | user dbadmin (COPY ClickStream_Fact FROM
                       '/data/clickstream/1g/ClickStream_Fact.tbl'
                       DELIMITER '|' NULL '\\n' DIRECT;)
statement_start        | 2011-01-03 15:34:46.538862
statement_id           |
last_statement_duration_us | 26250
current_statement      | COPY ClickStream_Fact FROM '/data/clickstream
                       /1g/ClickStream_Fact.tbl' DELIMITER '|' NULL
                       '\\n' DIRECT;
ssl_state              | None
authentication_method  | Trust
```

Close user session stress05-27944:0xc1a

```
=> \x
Expanded display is off.
=> SELECT CLOSE_SESSION('stress05-27944:0xc1a');
                    CLOSE_SESSION
-------------------------------------------------------------------------
 Session close command sent. Check v_monitor.sessions for progress.
(1 row)
```

Query the sessions table again for current status, and you can see that the second session has been closed:

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1 ]----------+-------------------------------------------
node_name              | v_vmartdb_node0001
user_name              | dbadmin
client_hostname        | 127.0.0.1:52110
client_pid             | 4554
login_timestamp        | 2011-01-03 14:05:40.252625-05
session_id             | stress04-4325:0x14
client_label           |
transaction_start      | 2011-01-03 14:05:44.325781
transaction_id         | 45035996273728326
transaction_description | user dbadmin (select * from SESSIONS;)
statement_start        | 2011-01-03 16:12:07.841298
statement_id           | 20
last_statement_duration_us | 2099
current_statement      | SELECT * FROM SESSIONS;
ssl_state              | None
authentication_method  | Trust
```

See also

- CLOSE_ALL_SESSIONS
- SHUTDOWN

CLOSE_USER_SESSIONS

Stops the session for a user, rolls back any transaction currently running, and closes the connection. To determine the status of the sessions to close, query the SESSIONS table.

> **Note**
> Running this function on your own sessions leaves one session running.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLOSE_USER_SESSIONS ( 'user-name' )
```

Parameters

user-name

Specifies the user whose sessions are to be closed. If you specify your own user name, Vertica closes all sessions except the one in which you issue this function.

Privileges

DBADMIN

Examples

This example closes all active session for user u1 :

```
=> SELECT close_user_sessions('u1');
```

See also

- CLOSE_ALL_SESSIONS
- CLOSE_SESSION
- SHUTDOWN

GET_NUM_ACCEPTED_ROWS

Returns the number of rows loaded into the database for the last completed load for the current session. GET_NUM_ACCEPTED_ROWS is a meta-function . Do not use it as a value in an INSERT query.

The number of accepted rows is not available for a load that is currently in process. Check the LOAD_STREAMS system table for its status.

This meta-function supports loads from STDIN, COPY LOCAL from a Vertica client, or a single file on the initiator. You cannot use GET_NUM_ACCEPTED_ROWS for multi-node loads.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_NUM_ACCEPTED_ROWS();
```

Privileges

None

> **Note**
> The data regarding accepted rows from the last load during the current session does not persist, and is lost when you initiate a new load.

Examples

This examples shows the number of accepted rows from the vmart_load_data.sql meta-command.

```
=> \i vmart_load_data.sql;
=> SELECT GET_NUM_ACCEPTED_ROWS ();
GET_NUM_ACCEPTED_ROWS
----------------------
300000
(1 row)
```

See also

- GET_NUM_REJECTED_ROWS

GET_NUM_REJECTED_ROWS

Returns the number of rows that were rejected during the last completed load for the current session. GET_NUM_REJECTED_ROWS is a meta-function . Do not use it as a value in an INSERT query.

Rejected row information is unavailable for a load that is currently running. The number of rejected rows is not available for a load that is currently in process. Check the LOAD_STREAMS system table for its status.

This meta-function supports loads from STDIN, COPY LOCAL from a Vertica client, or a single file on the initiator. You cannot use GET_NUM_REJECTED_ROWS for multi-node loads.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
GET_NUM_REJECTED_ROWS();
```

Privileges

None

> **Note**
> The data regarding rejected rows from the last load during the current session does not persist, and is dropped when you initiate a new load.

Examples

This example shows the number of rejected rows from the vmart_load_data.sql meta-command.

```
=>  \i vmart_load_data.sql
=> SELECT GET_NUM_REJECTED_ROWS ();
GET_NUM_REJECTED_ROWS
----------------------
0
(1 row)
```

See also

- GET_NUM_ACCEPTED_ROWS

INTERRUPT_STATEMENT

Interrupts the specified statement in a user session, rolls back the current transaction, and writes a success or failure message to the log file.

Sessions can be interrupted during statement execution. Only statements run by user sessions can be interrupted.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
INTERRUPT_STATEMENT( 'session-id', statement-id)
```

Parameters

session-id

Identifies the session to interrupt. This identifier is unique within the cluster at any point in time.

statement-id

Identifies the statement to interrupt. If the *statement-id* is valid, the statement can be interrupted and INTERRUPT_STATEMENT returns a success message. Otherwise the system returns an error.

Privileges

Superuser

Messages

The following list describes messages you might encounter:

Message	Meaning
Statement interrupt sent. Check SESSIONS for progress.	This message indicates success.
Session <id> could not be successfully interrupted: session not found.	The session ID argument to the interrupt command does not match a running session.
Session <id> could not be successfully interrupted: statement not found.	The statement ID does not match (or no longer matches) the ID of a running statement (if any).
No interruptible statement running	The statement is DDL or otherwise non-interruptible.
Internal (system) sessions cannot be interrupted.	The session is internal, and only statements run by external sessions can be interrupted.

Examples

Two user sessions are open. RECORD 1 shows user session running SELECT FROM SESSION , and RECORD 2 shows user session running COPY DIRECT :

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1 ]----------+-----------------------------------------------------
node_name              | v_vmartdb_node0001
user_name              | dbadmin
client_hostname        | 127.0.0.1:52110
client_pid             | 4554
login_timestamp        | 2011-01-03 14:05:40.252625-05
session_id             | stress04-4325:0x14
client_label           |
transaction_start      | 2011-01-03 14:05:44.325781
transaction_id         | 45035996273728326
transaction_description | user dbadmin (select * from sessions;)
statement_start        | 2011-01-03 15:36:13.896288
statement_id           | 10
last_statement_duration_us | 14978
current_statement      | select * from sessions;
ssl_state              | None
authentication_method  | Trust
-[ RECORD 2 ]----------+-----------------------------------------------------
node_name              | v_vmartdb_node0003
user_name              | dbadmin
client_hostname        | 127.0.0.1:56367
client_pid             | 1191
login_timestamp        | 2011-01-03 15:31:44.939302-05
session_id             | stress06-25663:0xbec
client_label           |
transaction_start      | 2011-01-03 15:34:51.05939
transaction_id         | 54043195528458775
transaction_description | user dbadmin (COPY Mart_Fact FROM '/data/Mart_Fact.tbl'
                       DELIMITER '|' NULL '\\n' DIRECT;)
statement_start        | 2011-01-03 15:35:46.436748
statement_id           | 5
last_statement_duration_us | 1591403
current_statement      | COPY Mart_Fact FROM '/data/Mart_Fact.tbl' DELIMITER '|'
                       NULL '\\n' DIRECT;
ssl_state              | None
authentication_method  | Trust
```

Interrupt the COPY DIRECT statement running in session stress06-25663:0xbec :

```
=> \x
Expanded display is off.
=> SELECT INTERRUPT_STATEMENT('stress06-25663:0x1537', 5);
                    interrupt_statement
-----------------------------------------------------------------------
 Statement interrupt sent. Check v_monitor.sessions for progress.
(1 row)
```

Verify that the interrupted statement is no longer active by looking at the current_statement column in the SESSIONS system table. This column becomes blank when the statement is interrupted:

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1 ]----------+-----------------------------------------------------
node_name              | v_vmartdb_node0001
user_name              | dbadmin
client_hostname        | 127.0.0.1:52110
client_pid             | 4554
login_timestamp        | 2011-01-03 14:05:40.252625-05
session_id             | stress04-4325:0x14
client_label           |
transaction_start      | 2011-01-03 14:05:44.325781
transaction_id         | 45035996273728326
transaction_description | user dbadmin (select * from sessions;)
statement_start        | 2011-01-03 15:36:13.896288
statement_id           | 10
last_statement_duration_us | 14978
current_statement      | select * from sessions;
ssl_state              | None
authentication_method  | Trust
-[ RECORD 2 ]----------+-----------------------------------------------------
node_name              | v_vmartdb_node0003
user_name              | dbadmin
client_hostname        | 127.0.0.1:56367
client_pid             | 1191
login_timestamp        | 2011-01-03 15:31:44.939302-05
session_id             | stress06-25663:0xbec
client_label           |
transaction_start      | 2011-01-03 15:34:51.05939
transaction_id         | 54043195528458775
transaction_description | user dbadmin (COPY Mart_Fact FROM '/data/Mart_Fact.tbl'
                       DELIMITER '|' NULL '\\n' DIRECT;)
statement_start        | 2011-01-03 15:35:46.436748
statement_id           | 5
last_statement_duration_us | 1591403
current_statement      |
ssl_state              | None
authentication_method  | Trust
```

See also

- SESSIONS
- Managing sessions

RELEASE_ALL_JVM_MEMORY

Forces all sessions to release the memory consumed by their Java Virtual Machines (JVM).

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RELEASE_ALL_JVM_MEMORY();
```

Privileges

Must be a superuser .

Examples

The following example demonstrates viewing the JVM memory use in all open sessions, then calling RELEASE_ALL_JVM_MEMORY() to release the memory:

```
=> select user_name,external_memory_kb FROM V_MONITOR.SESSIONS;
 user_name | external_memory_kb
-----------+--------------------
 dbadmin   |              79705
(1 row)

=> SELECT RELEASE_ALL_JVM_MEMORY();
                        RELEASE_ALL_JVM_MEMORY
-------------------------------------------------------------------------------
 Close all JVM sessions command sent. Check v_monitor.sessions for progress.
(1 row)

=> SELECT user_name,external_memory_kb FROM V_MONITOR.SESSIONS;
 user_name | external_memory_kb
-----------+--------------------
 dbadmin   |                  0
(1 row)
```

See also

- RELEASE_JVM_MEMORY

RELEASE_JVM_MEMORY

Terminates a Java Virtual Machine (JVM), making available the memory the JVM was using.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RELEASE_JVM_MEMORY();
```

Privileges

None.

Examples

User session opened. RECORD 2 shows the user session running COPY DIRECT statement.

```
=> SELECT RELEASE_JVM_MEMORY();
          release_jvm_memory
-----------------------------------------
Java process killed and memory released
(1 row)
```

See also

- RELEASE_ALL_JVM_MEMORY

RESERVE_SESSION_RESOURCE

Reserves memory resources from the general resource pool for the exclusive use of the Vertica backup and restore process. No other Vertica process can access reserved resources. If insufficient resources are available, Vertica queues the reservation request.

This meta-function is a session level reservation. When a session ends Vertica automatically releases any resources reserved in that session. Because the meta-function operates at the session level, the resource name does not need to be unique across multiple sessions.

You can view reserved resources by querying the SESSIONS table.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RESERVE_SESSION_RESOURCE ( 'name', memory)
```

Parameters

name

The name of the resource to reserve.

memory

The amount of memory in kilobytes to allocate to the resource.

Privileges

None

Examples

Reserve 1024 kilobytes of memory for the backup and restore process:

```
=> SELECT reserve_session_resource('VBR_RESERVE',1024);
  -[ RECORD 1 ]------------+----------------
  reserve_session_resource | Grant succeed
```

RESET_SESSION

Applies your default connection string configuration settings to your current session.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RESET_SESSION()
```

Examples

The following example shows how you use RESET_SESSION.

Resets the current client connection string to the default connection string settings:

```
=> SELECT RESET_SESSION();
   RESET_SESSION
-----------------------
 Reset session: done.
(1 row)
```

Storage functions

This section contains storage management functions specific to Vertica.

In this section

- ALTER_LOCATION_LABEL
- ALTER_LOCATION_USE
- CLEAR_CACHES
- CLEAR_OBJECT_STORAGE_POLICY
- DO_TM_TASK
- DROP_LOCATION
- ENFORCE_OBJECT_STORAGE_POLICY
- MEASURE_LOCATION_PERFORMANCE
- MOVE_RETIRED_LOCATION_DATA
- RESTORE_LOCATION
- RETIRE_LOCATION
- SET_LOCATION_PERFORMANCE
- SET_OBJECT_STORAGE_POLICY

ALTER_LOCATION_LABEL

Adds a label to a storage location, or changes or removes an existing label. You can change a location label if it is not specified by any storage policy.

Caution

If you label a storage location that contains data, Vertica moves the data to an unlabeled location, if one exists. To prevent data movement between storage locations, labels should be applied either to all storage locations or none.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ALTER_LOCATION_LABEL ( 'path' , '[node]' , '[location-label]' )
```

Parameters

path

The storage location path.

node

The node where the label change is applied. If you supply an empty string, Vertica applies the change across all cluster nodes.

location-label

The label to assign to the specified storage location.

If you supply an empty string, Vertica removes that storage location's label.
You can remove a location label only if the following conditions are both true:

- No database object has a storage policy that specifies this label.
- The labeled location is not the last available storage for the objects associated with it.

Privileges

Superuser

Examples

The following ALTER_LOCATION_LABEL statement applies across all cluster nodes the label SSD to the storage location /home/dbadmin/SSD/tables :

```
=> SELECT ALTER_LOCATION_LABEL('/home/dbadmin/SSD/tables','', 'SSD');
        ALTER_LOCATION_LABEL
-----------------------------------------
 /home/dbadmin/SSD/tables label changed.
(1 row)
```

See also

- Altering location labels
- CLEAR_OBJECT_STORAGE_POLICY
- SET_OBJECT_STORAGE_POLICY

ALTER_LOCATION_USE

Alters the type of data that a storage location holds.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ALTER_LOCATION_USE ( 'path' , '[node]' , 'usage' )
```

Arguments

path

Where the storage location is mounted.

node

The Vertica node on which to alter the storage location. To alter the location on all cluster nodes in a single transaction, use an empty string ('').
If the usage is SHARED TEMP or SHARED USER, you must alter it on all nodes.

usage

One of the following:

- DATA : The storage location stores only data files.
- TEMP : The location stores only temporary files that are created during loads or queries.
- DATA,TEMP : The location can store both types of files.

Privileges

Superuser

Restrictions

You cannot change a storage location from a USER usage type if you created the location that way, or to a USER type if you did not. You can change a USER storage location to specify DATA (storing TEMP files is not supported). However, doing so does not affect the primary objective of a USER storage location, to be accessible by non-dbadmin users with assigned privileges.

You cannot change a storage location from SHARED TEMP or SHARED USER to SHARED DATA or the reverse.

Monitoring storage locations

For information about the disk storage used on each node, query the DISK_STORAGE system table.

Examples

The following example alters a storage location across all cluster nodes to store only data:

```
=> SELECT ALTER_LOCATION_USE ('/thirdSL/' , '' , 'DATA');
```

See also

- Altering location use
- RETIRE_LOCATION
- GRANT (storage location)
- REVOKE (storage location)

CLEAR_CACHES

Clears the Vertica internal cache files.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAR_CACHES ( )
```

Privileges

Superuser

Notes

If you want to run benchmark tests for your queries, in addition to clearing the internal Vertica cache files, clear the Linux file system cache. The kernel uses unallocated memory as a cache to hold clean disk blocks. If you are running version 2.6.16 or later of Linux and you have root access, you can clear the kernel file system cache as follows:

1. Make sure that all data in the cache is written to disk:
   ```
   # sync
   ```
2. Writing to the drop_caches file causes the kernel to drop clean caches, entries, and inodes from memory, causing that memory to become free, as follows:
   - To clear the page cache:
     ```
     # echo 1 > /proc/sys/vm/drop_caches
     ```
   - To clear the entries and inodes:
     ```
     # echo 2 > /proc/sys/vm/drop_caches
     ```
   - To clear the page cache, entries, and inodes:
     ```
     # echo 3 > /proc/sys/vm/drop_caches
     ```

Examples

The following example clears the Vertica internal cache files:

```
=> SELECT CLEAR_CACHES();
 CLEAR_CACHES
--------------
 Cleared
(1 row)
```

CLEAR_OBJECT_STORAGE_POLICY

Removes a user-defined storage policy from the specified database, schema or table. Storage containers at the previous policy's labeled location are moved to the default location. By default, this move occurs after all pending mergeout tasks return.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAR_OBJECT_STORAGE_POLICY ( 'object-name' [,'key-min', 'key-max'] [, 'enforce-storage-move' ] )
```

Parameters

object-name

The object to clear, one of the following:

- *database* : Clears *database* of its storage policy.
- [*database* .] *schema* : Clears *schema* of its storage policy.
- [[*database* .] *schema* .] *table* : Clears *table* of its storage policy. If *table* is in any schema other than public , you must supply the schema name.

In all cases, *database* must be the name of the current database.

key-min

key-max

Valid only if *object-name* is a table, specifies the range of table partition key values stored at the labeled location.

enforce-storage-move

Specifies when the Tuple Mover moves all existing storage containers for the specified object to its default storage location:

- false (default): Move storage containers only after all pending mergeout tasks return.
- true : Immediately move all storage containers to the new location.

> **Tip**
> You can also enforce all storage policies immediately by calling Vertica meta-function ENFORCE_OBJECT_STORAGE_POLICY .

Privileges

Superuser

Examples

This following statement clears the storage policy for table store.store_orders_fact . The true argument specifies to implement the move immediately:

```
=> SELECT CLEAR_OBJECT_STORAGE_POLICY ('store.store_orders_fact', 'true');
                    CLEAR_OBJECT_STORAGE_POLICY
-------------------------------------------------------------------------------
 Object storage policy cleared.
Task: moving storages
(Table: store.store_orders_fact) (Projection: store.store_orders_fact_b0)
(Table: store.store_orders_fact) (Projection: store.store_orders_fact_b1)

(1 row)
```

See also

- Clearing storage policies
- ALTER_LOCATION_LABEL
- SET_OBJECT_STORAGE_POLICY
- ENFORCE_OBJECT_STORAGE_POLICY

DO_TM_TASK

Runs a Tuple Mover (TM) operation and commits current transactions. You can limit this operation to a specific table or projection. When started using this function, the TM uses the GENERAL resource pool instead of the TM resource pool.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DO_TM_TASK('task'[, '[[database.]schema.]{ table | projection}]' )
```

Parameters

task

Specifies one of the following tuple mover operations:

- mergeout : Consolidates ROS containers and purge s deleted records. For details, see Mergeout .
- reshardmergeout : Realigns storage containers to the shard definitions created by a RESHARD_DATABASE call. Specify a table or projection and a range of partition values to limit the scope of the reshardmergeout operations.
- analyze_row_count : Collects a minimal set of statistics and aggregate row counts for the specified projections, and saves it in the database catalog. Collects the number of rows in the specified projection. If you specify a table name, DO_TM_TASK returns the row counts for all projections of that table. For details, see Analyzing row counts .
- update_storage_catalog (recommended only for Eon Mode): Updates the catalog with metadata on bundled table data. For details, see Writing bundle metadata to the catalog .

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table **|** ***projection***

Applies *task* to the specified table or projection. If you specify a projection and it is not found, DO_TM_TASK looks for a table with that name and, if found, applies the task to it and all projections associated with it.

If you specify no table or projection, the task is applied to all database tables and their projections.

Privileges

- Schema: USAGE
- Table: One of INSERT, UPDATE, or DELETE

Examples

The following example performs a mergeout on all projections in a table:

```
=> SELECT DO_TM_TASK('mergeout', 't1');
```

You can perform a re-shard mergeout task on a range of partitions of a table:

```
=> SELECT DO_TM_TASK('reshardmergeout', 'store_orders', '2001', '2005');
```

DROP_LOCATION

Permanently removes a retired storage location. This operation cannot be undone. You must first retire a storage location with RETIRE_LOCATION before dropping it; you cannot drop a storage location that is in use.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DROP_LOCATION ( 'path', 'node' )
```

Arguments

path

Where the storage location to drop is mounted.

node

The Vertica node on which to drop the location. To perform this operation on all nodes, use an empty string (''). If the storage location is SHARED, you must perform this operation on all nodes.

Privileges

Superuser

Storage locations with temp and data files

If you use a storage location to store data and then alter it to store only temp files, the location can still contain data files. Vertica does not let you drop a storage location containing data files. You can use the MOVE_RETIRED_LOCATION_DATA function to manually merge out the data files from the storage location, or you can drop partitions. Deleting data files does not work.

Examples

The following example shows how to drop a previously retired storage location on v_vmart_node0003 :

```
=> SELECT DROP_LOCATION('/data', 'v_vmart_node0003');
```

See also

- Dropping storage locations
- ALTER_LOCATION_USE
- RESTORE_LOCATION
- RETIRE_LOCATION

ENFORCE_OBJECT_STORAGE_POLICY

Enterprise Mode only

Applies storage policies of the specified object immediately. By default, the Tuple Mover enforces object storage policies after all pending mergeout operations are complete. Calling this function is equivalent to setting the *enforce* argument when using RETIRE_LOCATION . You typically use this function as the last step before dropping a storage location.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ENFORCE_OBJECT_STORAGE_POLICY ( 'object-name' ['key-min', 'key-max'] )
```

Arguments

object-name

The database object whose storage policies are to be applied, one of the following:

- *database* : Applies *database* storage policies.
- [*database* .] *schema* : Applies *schema* storage policies.
- [[*database* .] *schema* .] *table* : Applies *table* storage policies. If *table* is in any schema other than public , you must supply the schema name.

In all cases, *database* must be the name of the current database.

key-min* , *key-max

Valid only if *object-name* is a table, specifies the range of table partition key values on which to perform the move.

Privileges

One of the following:

- Superuser
- Object owner and access to its storage location.

Examples

Apply storage policy updates to the test table:

```
=> SELECT ENFORCE_OBJECT_STORAGE_POLICY ('test');
```

See also

- CLEAR_OBJECT_STORAGE_POLICY
- RETIRE_LOCATION
- DROP_LOCATION

- Managing storage locations

MEASURE_LOCATION_PERFORMANCE

Measures a storage location's disk performance.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
MEASURE_LOCATION_PERFORMANCE ( 'path', 'node' )
```

Parameters

path

Specifies where the storage location to measure is mounted.

node

The Vertica node where the location to be measured is available. To obtain a list of all node names on the cluster, query system table DISK_STORAGE .

Privileges

Superuser

Notes

- If you intend to create a tiered disk architecture in which projections, columns, and partitions are stored on different disks based on predicted or measured access patterns, you need to measure storage location performance for each location in which data is stored. You do not need to measure storage location performance for temp data storage locations because temporary files are stored based on available space.
- The method of measuring storage location performance applies only to configured clusters. If you want to measure a disk before configuring a cluster see Measuring storage performance .
- Storage location performance equates to the amount of time it takes to read and write 1MB of data from the disk. This time equates to:

  $IO\text{-}time = (time\text{-}to\text{-}read\text{-}write\text{-}1MB + time\text{-}to\text{-}seek) = (1/throughput + 1/latency)$

  Throughput is the average throughput of sequential reads/writes (units in MB per second).
  Latency is for random reads only in seeks (units in seeks per second)

  > **Note**
  > The IO time of a faster storage location is less than a slower storage location.

Examples

The following example measures the performance of a storage location on v_vmartdb_node0004:

```
=> SELECT MEASURE_LOCATION_PERFORMANCE('/secondVerticaStorageLocation/' , 'v_vmartdb_node0004');
WARNING:  measure_location_performance can take a long time. Please check logs for progress
        measure_location_performance
-----------------------------------------------------
 Throughput : 122 MB/sec. Latency : 140 seeks/sec
```

See also

- CREATE LOCATION
- ALTER_LOCATION_USE
- RESTORE_LOCATION
- RETIRE_LOCATION
- Measuring storage performance

MOVE_RETIRED_LOCATION_DATA

Moves all data from the specified retired storage location or from all retired storage locations in the database. MOVE_RETIRED_LOCATION_DATA migrates the data to non-retired storage locations according to the storage policies of the objects whose data is stored in the location. This function returns only after it completes migration of all affected storage location data.

Note
The Tuple Mover migrates data of retired storage locations when it consolidates data into larger ROS containers.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
MOVE_RETIRED_LOCATION_DATA( ['location-path'] [, 'node'] )
```

Arguments

location-path

The path of the storage location as specified in the LOCATION_PATH column of system table STORAGE_LOCATIONS . This storage location must be marked as retired.

If you omit this argument, MOVE_RETIRED_LOCATION_DATA moves data from all retired storage locations.

node

The node on which to move data of the retired storage location. If *location-path* is undefined on *node* , this function returns an error.

If you omit this argument, MOVE_RETIRED_LOCATION_DATA moves data from* location-path * on all nodes.

Privileges

Superuser

Examples

1. Query system table STORAGE_LOCATIONS to show which storage locations are retired:

```
=> SELECT node_name, location_path, location_label, is_retired FROM STORAGE_LOCATIONS
  WHERE is_retired = 't';
   node_name     |    location_path     | location_label | is_retired
-----------------+----------------------+----------------+------------
 v_vmart_node0001 | /home/dbadmin/SSDLoc | ssd            | t
 v_vmart_node0002 | /home/dbadmin/SSDLoc | ssd            | t
 v_vmart_node0003 | /home/dbadmin/SSDLoc | ssd            | t
(3 rows)
```

2. Query system table STORAGE_LOCATIONS for the location of the messages table, which is currently stored in retired storage location ssd :

```
=> SELECT node_name, total_row_count, location_label FROM STORAGE_CONTAINERS
  WHERE projection_name ILIKE 'messages%';
   node_name     | total_row_count | location_label
-----------------+-----------------+----------------
 v_vmart_node0001 |          333514 | ssd
 v_vmart_node0001 |          333255 | ssd
 v_vmart_node0002 |          333255 | ssd
 v_vmart_node0002 |          333231 | ssd
 v_vmart_node0003 |          333231 | ssd
 v_vmart_node0003 |          333514 | ssd
(6 rows)
```

3. Call MOVE_RETIRED_LOCATION_DATA to move the data off the ssd storage location.

```
=> SELECT MOVE_RETIRED_LOCATION_DATA('/home/dbadmin/SSDLoc');
        MOVE_RETIRED_LOCATION_DATA
-----------------------------------------------
 Move data off retired storage locations done

(1 row)
```

4. Repeat the previous query to verify the storage location of the messages table:

```
=> SELECT node_name, total_row_count, storage_type, location_label FROM storage_containers
   WHERE projection_name ILIKE 'messages%';
    node_name     | total_row_count | location_label
------------------+-----------------+----------------
 v_vmart_node0001 |          333255 | base
 v_vmart_node0001 |          333514 | base
 v_vmart_node0003 |          333514 | base
 v_vmart_node0003 |          333231 | base
 v_vmart_node0002 |          333231 | base
 v_vmart_node0002 |          333255 | base
(6 rows)
```

See also

- RETIRE_LOCATION
- RESTORE_LOCATION
- Managing storage locations

RESTORE_LOCATION

Restores a storage location that was previously retired with RETIRE_LOCATION .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RESTORE_LOCATION ( 'path', 'node' )
```

Arguments

path

Where to mount the retired storage location.

node

The Vertica node on which to restore the location. To perform this operation on all nodes, use an empty string (''). If the storage location is SHARED, you must perform this operation on all nodes.

The operation fails if you dropped any locations.

Privileges

Superuser

Effects of restoring a previously retired location

After restoring a storage location, Vertica re-ranks all of the cluster storage locations. It uses the newly restored location to process queries as determined by its rank.

Monitoring storage locations

For information about the disk storage used on each node, query the DISK_STORAGE system table.

Examples

Restore a retired storage location on node4 :

```
=> SELECT RESTORE_LOCATION ('/thirdSL/' , 'v_vmartdb_node0004');
```

See also

- ALTER_LOCATION_USE
- DROP_LOCATION

RETIRE_LOCATION

Deactivates the specified storage location. To obtain a list of all existing storage locations, query the STORAGE_LOCATIONS system table.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
RETIRE_LOCATION ( 'path', 'node' [, enforce ] )
```

Arguments

path

Where the storage location to retire is mounted.

node

The Vertica node on which to retire the location. To perform this operation on all nodes, use an empty string (''). If the storage location is SHARED, you must perform this operation on all nodes.

enforce

If true , the location label is set to an empty string and the data is moved elsewhere. The location can then be dropped without errors or warnings. Use this argument to expedite dropping a location.

Privileges

Superuser

Effects of retiring a storage location

RETIRE_LOCATION checks that the location is not the only storage for data and temp files. At least one location must exist on each node to store data and temp files. However, you can store both sorts of files in either the same location or separate locations.

If a location is the last available storage for its associated objects, you can retire it only if you set *enforce* to true .

When you retire a storage location:

- No new data is stored at the retired location, unless you first restore it using RESTORE_LOCATION .
- By default, if the storage location being retired contains stored data, the data is not moved. Thus, you cannot drop the storage location. Instead, Vertica removes the stored data through one or more mergeouts. To drop the location immediately after retiring it, set *enforce* to true.
- If the storage location being retired is used only for temp files or you use *enforce* , you can drop the location. See Dropping storage locations and DROP_LOCATION .

Monitoring storage locations

For information about the disk storage used on each node, query the DISK_STORAGE system table.

Examples

The following examples show two approaches to retiring a storage location.

You can retire a storage location and its data will be moved out automatically at a future time:

```
=> SELECT RETIRE_LOCATION ('/data' , 'v_vmartdb_node0004');
```

You can specify that data in the storage location be moved immediately, so that you can then drop the location without waiting:

```
=> SELECT RETIRE_LOCATION ('/data' , 'v_vmartdb_node0004', true);
```

See also

- Retiring storage locations
- CREATE LOCATION
- DROP_LOCATION
- RESTORE_LOCATION

SET_LOCATION_PERFORMANCE

Sets disk performance for a storage location.

> **Note**
> Before calling this function, call MEASURE_LOCATION_PERFORMANCE to obtain the location's throughput and average latency .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_LOCATION_PERFORMANCE ( 'path', 'node' , 'throughput', 'average-latency')
```

Parameters

path

Specifies where the storage location to set is mounted.

node

Specifies the Vertica node where the location to set is available.

throughput

Specifies the throughput for the location, set to a value ≥1.

average-latency

Specifies the average latency for the location, set to a value ≥1.

Privileges

Superuser

Examples

The following example sets the performance of a storage location on node2 to a throughput of 122 megabytes per second and a latency of 140 seeks per second.

```
=> SELECT SET_LOCATION_PERFORMANCE('/secondVerticaStorageLocation/','node2','122','140');
```

See also

- CREATE LOCATION
- Measuring storage performance
- Setting storage performance

SET_OBJECT_STORAGE_POLICY

Creates or changes the storage policy of a database object by assigning it a labeled storage location. The Tuple Mover uses this location to store new and existing data for this object. If the object already has an active storage policy, calling SET_OBJECT_STORAGE_POLICY sets this object's default storage to the new labeled location. Existing data for the object is moved to the new location.

> **Note**
> You cannot create a storage policy on a USER type storage location.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SET_OBJECT_STORAGE_POLICY (
 '[[database.]schema.]object-name', 'location-label'
  [,'key-min', 'key-max'] [, 'enforce-storage-move' ] )
```

Parameters

[database .] schema

Database and schema . The default schema is public . If you specify a database, it must be the current database.

object-name

Identifies the database object assigned to a labeled storage location. The *object-name* can resolve to a database, schema, or table.

location-label

The label of *object-name* 's storage location.

key-min

key-max

Valid only if *object-name* is a table, specifies the range of table partition key values to store at the labeled location.

enforce-storage-move

Specifies when the Tuple Mover moves all existing storage containers for *object-name* to the labeled storage location:

- false (default): Move storage containers only after all pending mergeout tasks return.
- true : Immediately move all storage containers to the new location.

> **Tip**
> You can also enforce all storage policies immediately by calling Vertica meta-function ENFORCE_OBJECT_STORAGE_POLICY

Privileges

One of the following:

- Superuser
- Object owner and access to its storage location.

Examples

See Clearing storage policies

See also

- Creating storage policies
- Moving data storage locations
- ALTER_LOCATION_LABEL
- CLEAR_OBJECT_STORAGE_POLICY

Table functions

This section contains functions for managing tables and constraints.

See also the V_CATALOG.TABLE_CONSTRAINTS system table.

In this section

- ANALYZE_CONSTRAINTS
- ANALYZE_CORRELATIONS
- COPY_TABLE
- DISABLE_DUPLICATE_KEY_ERROR
- INFER_EXTERNAL_TABLE_DDL
- INFER_TABLE_DDL
- LAST_INSERT_ID
- PURGE_TABLE
- REBALANCE_TABLE
- REENABLE_DUPLICATE_KEY_ERROR

ANALYZE_CONSTRAINTS

Analyzes and reports on constraint violations within the specified scope

You can enable automatic enforcement of primary key, unique key, and check constraints when INSERT , UPDATE , MERGE , or COPY statements execute. Alternatively, you can use ANALYZE_CONSTRAINTS to validate constraints after issuing these statements. Refer to Constraint enforcement for more information.

ANALYZE_CONSTRAINTS performs a lock in the same way that SELECT * FROM t1 holds a lock on table t1 . See LOCKS for additional information.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ANALYZE_CONSTRAINTS ('[[[database.]schema.]table ]' [, 'column[,...]'] )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Identifies the table to analyze. If you omit specifying a schema, Vertica uses the current schema search path. If set to an empty string, Vertica analyzes all tables in the current schema.

column

The column in *table* to analyze. You can specify multiple comma-delimited columns. Vertica narrows the scope of the analysis to the specified columns. If you omit specifying a column, Vertica analyzes all columns in *table* .

Privileges

- Schema: USAGE
- Table: SELECT

Detecting constraint violations during a load process

Vertica checks for constraint violations when queries are run, not when data is loaded. To detect constraint violations as part of the load process, use a COPY statement with the NO COMMIT option. By loading data without committing it, you can run a post-load check of your data using the ANALYZE_CONSTRAINTS function. If the function finds constraint violations, you can roll back the load because you have not committed it.

If ANALYZE_CONSTRAINTS finds violations, such as when you insert a duplicate value into a primary key, you can correct errors using the following functions. Effects last until the end of the session only:

- DISABLE_DUPLICATE_KEY_ERROR
- REENABLE_DUPLICATE_KEY_ERROR

Important

If a check constraint SQL expression evaluates to an unknown for a given row because a column within the expression contains a null, the row passes the constraint condition.

Return values

ANALYZE_CONSTRAINTS returns results in a structured set (see table below) that lists the schema name, table name, column name, constraint name, constraint type, and the column values that caused the violation.

If the result set is empty, then no constraint violations exist; for example:

```
> SELECT ANALYZE_CONSTRAINTS ('public.product_dimension', 'product_key');
Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
(0 rows)
```

The following result set shows a primary key violation, along with the value that caused the violation ('10') :

```
=> SELECT ANALYZE_CONSTRAINTS ('');
Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
store       t1           c1             pk_t1             PRIMARY           ('10')
(1 row)
```

The result set columns are described in further detail in the following table:

Column Name	Data Type	Description
Schema Name	VARCHAR	The name of the schema.
Table Name	VARCHAR	The name of the table, if specified.
Column Names	VARCHAR	A list of comma-delimited columns that contain constraints.
Constraint Name	VARCHAR	The given name of the primary key, foreign key, unique, check, or not null constraint, if specified.

Constraint Type	VARCHAR	Identified by one of the following strings: • PRIMARY KEY • FOREIGN KEY • UNIQUE • CHECK • NOT NULL
Column Values	VARCHAR	Value of the constraint column, in the same order in which Column Names contains the value of that column in the violating row. When interpreted as SQL, the value of this column forms a list of values of the same type as the columns in Column Names ; for example: ('1'), ('1', 'z')

Examples

See Detecting constraint violations .

ANALYZE_CORRELATIONS

> **Deprecated**
> This function is deprecated and will be removed in a future release.

Analyzes the specified tables for pairs of columns that are strongly correlated. ANALYZE_CORRELATIONS stores the 20 pairs with the strongest correlation. ANALYZE_CORRELATIONS also analyzes statistics.

ANALYZE_CORRELATIONS analyzes only pairwise single-column correlations.

For example, state name and country name columns are strongly correlated because the city name usually, but perhaps not always, identifies the state name. The city of Conshohoken is uniquely associated with Pennsylvania, while the city of Boston exists in Georgia, Indiana, Kentucky, New York, Virginia, and Massachusetts. In this case, city name is strongly correlated with state name.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Stable

Syntax

```
ANALYZE_CORRELATIONS ('[[[database.]schema.]table ]' [, 'recalculate'] )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table-name

Identifies the table to analyze. If you omit specifying a schema, Vertica uses the current schema search path. If set to an empty string, Vertica analyzes all tables in the current schema.

recalculate

Boolean that specifies whether to analyze correlated columns that were previously analyzed.

> **Note**
> Column correlation analysis typically needs to be done only once.

Default: false

Privileges

One of the following:

- Superuser
- User with USAGE privilege on the design schema

Examples

In the following example, ANALYZE_CORRELATIONS analyzes column correlations for all tables in the public schema, even if they currently exist:

```
=> SELECT ANALYZE_CORRELATIONS ('public.*', 'true');
 ANALYZE_CORRELATIONS
----------------------
                    0
(1 row)
```

COPY_TABLE

Copies one table to another. This lightweight, in-memory function copies the DDL and all user-created projections from the source table. Projection statistics for the source table are also copied. Thus, the source and target tables initially have identical definitions and share the same storage.

> **Note**
> Although they share storage space, Vertica regards the tables as discrete objects for license capacity purposes. For example, a single-terabyte table and its copy initially consume only one TB of space. However, your Vertica license regards them as separate objects that consume two TB of space.

After the copy operation is complete, the source and copy tables are independent of each other, so you can perform DML operations on one table without impacting the other. These operations can increase the overall storage required for both tables.

> **Caution**
> If you create multiple copies of the same table concurrently, one or more of the copy operations is liable to fail. Instead, copy tables sequentially.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
COPY_TABLE (
   '[[database.]schema.]source-table',
   '[[database.]schema.]target-table'
)
```

Parameters

`[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

source-table

The source table to copy. Vertica copies all data from this table to the target table.

target-table

The target table of the source table. If the target table already exists, Vertica appends the source to the existing table.

If the table does not exist, Vertica creates a table from the source table's definition, by calling CREATE TABLE with LIKE and INCLUDING PROJECTIONS clause. The new table inherits ownership from the source table. For details, see Replicating a table .

Privileges

Non-superuser:

- Source table: SELECT
- Target schema/table (new): CREATE
- Target table (existing): INSERT

Table attribute requirements

The following attributes of both tables must be identical:

- Column definitions, including NULL/NOT NULL constraints
- Segmentation
- Partitioning expression
- Number of projections
- Projection sort order
- Primary and unique key constraints. However, the key constraints do not have to be identically enabled.

> **Note**
> If the target table has primary or unique key constraints enabled and moving the partitions will insert duplicate key values into the target table, Vertica rolls back the operation. Enforcing constraints requires disk reads and can slow the copy process.

- Number and definitions of text indices.
- If the destination table already exists, the source and destination tables must have identical access policies.

Additionally, If access policies exist on the source table, the following must be true:

- Access policies on both tables must be identical.
- One of the following must be true:
  - The executing user owns the source table.
  - AccessPolicyManagementSuperuserOnly is set to true. See Managing access policies for details.

Table restrictions

The following restrictions apply to the source and target tables:

- If the source and target partitions are in different storage tiers, Vertica returns a warning but the operation proceeds. The partitions remain in their existing storage tier.
- If the source table contains a sequence, Vertica converts the sequence to an integer before copying it to the target table. If the target table contains IDENTITY or named sequence columns, Vertica cancels the copy and displays an error message.
- The following tables cannot be used as sources or targets:
  - Temporary tables
  - Virtual tables
  - System tables
  - External tables

Examples

If you call COPY_TABLE and the target table does not exist, the function creates the table automatically. In the following example, COPY_TABLE creates the target table public.newtable . Vertica also copies all the constraints associated with the source table public.product_dimension except foreign key constraints:

```
=> SELECT COPY_TABLE ( 'public.product_dimension', 'public.newtable');
-[ RECORD 1 ]-------------------------------------------------
copy_table | Created table public.newtable.
Copied table public.product_dimension to public.newtable
```

See also

Creating a table from other tables

DISABLE_DUPLICATE_KEY_ERROR

Disables error messaging when Vertica finds duplicate primary or unique key values at run time (for use with key constraints that are not automatically enabled). Queries execute as though no constraints are defined on the schema. Effects are session scoped.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DISABLE_DUPLICATE_KEY_ERROR();
```

Privileges

Superuser

Examples

When you call DISABLE_DUPLICATE_KEY_ERROR , Vertica issues warnings letting you know that duplicate values will be ignored, and incorrect results are possible. DISABLE_DUPLICATE_KEY_ERROR is for use only for key constraints that are not automatically enabled.

```
=> select DISABLE_DUPLICATE_KEY_ERROR();
WARNING 3152:  Duplicate values in columns marked as UNIQUE will now be ignored for the remainder of your session or until
reenable_duplicate_key_error() is called
WARNING 3539:  Incorrect results are possible. Please contact Vertica Support if unsure
 disable_duplicate_key_error
------------------------------
 Duplicate key error disabled
(1 row)
```

See also

ANALYZE_CONSTRAINTS
INFER_EXTERNAL_TABLE_DDL

> **Deprecated**
> This function is deprecated and will be removed in a future release. Instead, use INFER_TABLE_DDL .

Inspects a file in Parquet, ORC, or Avro format and returns a CREATE EXTERNAL TABLE AS COPY statement that can be used to read the file. This statement might be incomplete. It could also contain more columns or columns with longer names than what Vertica supports; this function does not enforce Vertica system limits . Always inspect the output and address any issues before using it to create a table.

This function supports partition columns for the Parquet, ORC, and Avro formats, inferred from the input path. Because partitioning is done through the directory structure, there might not be enough information to infer the type of partition columns. In this case, this function shows these columns with a data type of UNKNOWN and emits a warning.

The function handles most data types, including complex types. If an input type is not supported in Vertica, the function emits a warning.

By default, the function uses strong typing for complex types. You can instead treat the column as a flexible complex type by setting the vertica_type_for_complex_type parameter to LONG VARBINARY .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
INFER_EXTERNAL_TABLE_DDL( path USING PARAMETERS param=value[,...] )
```

Arguments

path

Path to a file or directory. Any path that is valid for COPY and uses a file format supported by this function is valid.

Parameters

format

Input format (string), one of 'Parquet', 'ORC', or 'Avro'. This parameter is required.

table_name

The name of the external table to create. This parameter is required.

Do not include a schema name as part of the table name; use the table_schema parameter.

table_schema

The schema in which to create the external table. If omitted, the function does not include a schema in the output.

vertica_type_for_complex_type

Type used to represent all columns of complex types, if you do not want to expand them fully. The only supported value is LONG VARBINARY . For more information, see Flexible complex types .

Privileges

Non-superuser: READ privileges on the USER-accessible storage location .

Examples

In the following example, the input file contains data for a table with two integer columns. The table definition can be fully inferred, and you can use the returned SQL statement as-is.

```
=> SELECT INFER_EXTERNAL_TABLE_DDL('/data/orders/*.orc'
       USING PARAMETERS format = 'orc', table_name = 'orders');

               INFER_EXTERNAL_TABLE_DDL
-----------------------------------------------------------------------------------------------
create external table "orders" (
  "id" int,
  "quantity" int
) as copy from '/data/orders/*.orc' orc;
(1 row)
```

To create a table in a schema, use the table_schema parameter. Do not add it to the table name; the function treats it as a name with a period in it, not a schema.

The following example shows output with complex types. You can use the definition as-is or modify the VARCHAR sizes:

```
=> SELECT INFER_EXTERNAL_TABLE_DDL('/data/people/*.parquet'
       USING PARAMETERS format = 'parquet', table_name = 'employees');
WARNING 9311:  This generated statement contains one or more varchar/varbinary columns which default to length 80
                  INFER_EXTERNAL_TABLE_DDL
-------------------------------------------------------------------------
 create external table "employees"(
  "employeeID" int,
  "personal" Row(
    "name" varchar,
    "address" Row(
      "street" varchar,
      "city" varchar,
      "zipcode" int
    ),
    "taxID" int
  ),
  "department" varchar
 ) as copy from '/data/people/*.parquet' parquet;
(1 row)
```

In the following example, the input file contains a map in the "prods" column. You can read a map as an array of rows:

```
=> SELECT INFER_EXTERNAL_TABLE_DDL('/data/orders.parquet'
    USING PARAMETERS format='parquet', table_name='orders');
WARNING 9311:  This generated statement contains one or more varchar/varbinary columns which default to length 80
               INFER_EXTERNAL_TABLE_DDL
-----------------------------------------------------------------------
 create external table "orders"(
  "orderkey" int,
  "custkey" int,
  "prods" Array[Row(
    "key" varchar,
    "value" numeric(12,2)
  )],
  "orderdate" date
 ) as copy from '/data/orders.parquet' parquet;
(1 row)
```

In the following example, the data is partitioned by region. The function was not able to infer the data type and reports UNKNOWN:

```
=> SELECT INFER_EXTERNAL_TABLE_DDL('/data/sales/*/*
      USING PARAMETERS format = 'parquet', table_name = 'sales');
WARNING 9262: This generated statement is incomplete because of one or more unknown column types.
Fix these data types before creating the table
               INFER_EXTERNAL_TABLE_DDL
-------------------------------------------------------------------------
 create external table "sales"(
  "tx_id" int,
  "date" date,
  "region" UNKNOWN
) as copy from '/data/sales/*/*' PARTITION COLUMNS region parquet;
(1 row)
```

For VARCHAR and VARBINARY columns, this function does not specify a length. The Vertica default length for these types is 80 bytes. If the data values are longer, using this table definition unmodified could cause data to be truncated. Always review VARCHAR and VARBINARY columns to determine if you need to specify a length. This function emits a warning if the input file contains columns of these types:

```
WARNING 9311: This generated statement contains one or more varchar/varbinary columns which default to length 80
```

INFER_TABLE_DDL

Inspects a file in Parquet, ORC, JSON, or Avro format and returns a CREATE TABLE or CREATE EXTERNAL TABLE statement based on its contents.

The returned statement might be incomplete if the input data contains ambiguous or unknown data types. It could also contain more columns or columns with longer names than what Vertica supports; this function does not enforce Vertica system limits . Always inspect the output and address any issues before using it to create a table.

This function supports partition columns, inferred from the input path. Because partitioning is done through the directory structure, there might not be enough information to infer the type of partition columns. In this case, this function shows these columns with a data type of UNKNOWN and emits a warning.

The function handles most data types, including complex types. If an input type is not supported in Vertica, the function emits a warning.

For VARCHAR and VARBINARY columns, this function does not specify a length. The Vertica default length for these types is 80 bytes. If the data values are longer, using the returned table definition unmodified could cause data to be truncated. Always review VARCHAR and VARBINARY columns to determine if you need to specify a length. This function emits a warning if the input file contains columns of these types:

```
WARNING 9311: This generated statement contains one or more varchar/varbinary columns which default to length 80
```

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
INFER_TABLE_DDL( path USING PARAMETERS param=value[,...] )
```

Arguments

path

Path to a file or glob. Any path that is valid for COPY and uses a file format supported by this function is valid. For all formats except JSON, if a glob specifies more than one file, this function reads a single, arbitrarily-chosen file. For JSON, the function might read more than one file. See JSON .

Parameters

format

Input format (string), one of 'Parquet', 'ORC', 'Avro', or 'JSON'. This parameter is required.

table_name

The name of the table to create. This parameter is required.

Do not include a schema name as part of the table name; use the table_schema parameter.

table_schema

The schema in which to create the table. If omitted, the function does not include a schema in the output.

table_type

The type of table to create, either 'native' or 'external'.

Default: 'native'

with_copy_statement

For native tables, whether to include a COPY statement in addition to the CREATE TABLE statement.

Default: false

one_line_result

Whether to return the DDL as a single line instead of pretty-printing. The single-line format might be easier to copy into SQL scripts.

Default: false (pretty-print)

max_files

(JSON only.) Maximum number of files in *path* to inspect, if *path* is a glob. Use this parameter to increase the amount of data the function considers, for example if you suspect variation among files. Files are chosen arbitrarily from the glob. For details, see JSON .

Default: 1

max_candidates

(JSON only.) Number of candidate table definitions to show. The function generates only one candidate per file, so if you increase max_candidates , also increase max_files . For details, see JSON .

Default: 1

Privileges

Non-superuser: READ privileges on the USER-accessible storage location .

JSON

JSON, unlike the other supported formats, does not embed a schema in data files. This function infers JSON table DDL by instead inspecting the raw data. Because raw data can be ambiguous or inconsistent, the function takes a different approach for this format.

For each input file, the function iterates through records to develop a candidate table definition. A top-level field that appears in any record is included as a column, even if not all records use it. If the same field appears in the file with different types, the function chooses a type that is consistent with all observed occurrences.

Consider a file with data about restaurants:

```
{
   "name" : "Pizza House",
   "cuisine" : "Italian",
   "location_city" : [],
   "chain" : true,
   "hours" : [],
   "menu" : [{"item" : "cheese pizza", "price" : 7.99},
             {"item" : "spinach pizza", "price" : 8.99},
             {"item" : "garlic bread", "price" : 4.99}]
}
{
   "name" : "Sushi World",
   "cuisine" : "Asian",
   "location_city" : ["Pittsburgh"],
   "chain" : false,
   "menu" : [{"item" : "maki platter", "price" : "21.95"},
             {"item" : "tuna roll", "price" : "4.95"}]
}
```

The first record contains two empty arrays, so there is not enough information to determine the element types. The second record has a string value for one of them, so the function can infer a type of VARCHAR for it. The other array element type remains unknown.

In the first record menu prices are numbers, but in the second they are strings. Both FLOAT and the string can be coerced to NUMERIC, so the function returns NUMERIC:

```
=> SELECT INFER_TABLE_DDL ('/data/restaurants.json'
   USING PARAMETERS table_name='restaurants', format='json');
WARNING 0:  This generated statement contains one or more varchar/varbinary types which default to length 80

               INFER_TABLE_DDL
-------------------------------------------------------------------------
 Candidate matched 1/1 of total files(s):
 create table "restaurants"(
   "chain" bool,
   "cuisine" varchar,
   "hours" Array[UNKNWON],
   "location_city" Array[varchar],
   "menu" Array[Row(
     "item" varchar,
     "price" numeric
   )],
   "name" varchar
 );

(1 row)
```

All scalar types can be coerced to VARCHAR, so if a conflict cannot be resolved more specifically (as in the NUMERIC example), the function can still return a type. Complex types, however, cannot always be resolved in this way. In the following example, records in a file have conflicting definitions of the hours field:

```
{
   "name" : "Sushi World",
   "cuisine" : "Asian",
   "location_city" : ["Pittsburgh"],
   "chain" : false,
   "hours" : {"open" : "11:00", "close" : "22:00" }
}
{
   "name" : "Greasy Spoon",
   "cuisine" : "American",
   "location_city" : [],
   "chain" : "false",
   "hours" : {"open" : ["11:00","12:00"], "close" : ["21:00","22:00"] },
}
```

In the first record the value is a ROW with two TIME fields. In the second record the value is a ROW with two ARRAY[TIME] fields (representing weekday and weekend hours). These types are incompatible, so the function suggests a flexible complex type by using LONG VARBINARY:

```
=> SELECT INFER_TABLE_DDL ('/data/restaurants.json'
   USING PARAMETERS table_name='restaurants', format='json');
WARNING 0:  This generated statement contains one or more varchar/varbinary types which default to length 80

               INFER_TABLE_DDL
-------------------------------------------------------------------------
 Candidate matched 1/1 of total files(s):
 create table "restaurants"(
   "chain" bool,
   "cuisine" varchar,
   "hours" long varbinary,
   "location_city" Array[varchar],
   "name" varchar
 );

(1 row)
```

If you call the function with a glob, by default it reads one file. Set max_files to a higher number to inspect more data. The function calculates one candidate table definition per file and returns the definition that covers the largest number of files.

Increasing the number of files does not, by itself, increase the number of candidates the function returns. With more files the function can consider more candidates, but by default it returns the single candidate that represents the largest number of files. To see more than one possible table definition, also set max_candidates . There is no benefit to setting max_candidates to a larger number than max_files .

In the following example, the glob contains two files that differ in the structure of the menu column. In the first file, the menu field has two fields:

```
{
   "name" : "Bob's pizzeria",
   "cuisine" : "Italian",
   "location_city" : ["Cambridge", "Pittsburgh"],
   "menu" : [{"item" : "cheese pizza", "price" : 8.25},
            {"item" : "spinach pizza", "price" : 10.50}]
}
```

In the second file, the menu has different offerings at different times of day:

```
{
   "name" : "Greasy Spoon",
   "cuisine" : "American",
   "location_city" : [],
   "menu" : [{"time" : "breakfast",
          "items" :
          [{"item" : "scrambled eggs", "price" : "3.99"}]
         },
          {"time" : "lunch",
           "items" :
          [{"item" : "grilled cheese", "price" : "3.95"},
       {"item" : "tuna melt", "price" : "5.95"},
       {"item" : "french fries", "price" : "1.99"}]}]
}
```

To see both candidates, raise both max_files and max_candidates :

```
=> SELECT INFER_TABLE_DDL ('/data/*.json'
    USING PARAMETERS table_name='restaurants', format='json',
max_files=3, max_candidates=3);
WARNING 0:  This generated statement contains one or more float types which might lose precision
WARNING 0:  This generated statement contains one or more varchar/varbinary types which default to length 80

                INFER_TABLE_DDL
-------------------------------------------------------------------------------
 Candidate matched 1/2 of total files(s):
  create table "restaurants"(
    "cuisine" varchar,
    "location_city" Array[varchar],
    "menu" Array[Row(
      "item" varchar,
      "price" float
    )],
    "name" varchar
  );
Candidate matched 1/2 of total files(s):
  create table "restaurants"(
    "cuisine" varchar,
    "location_city" Array[varchar],
    "menu" Array[Row(
      "items" Array[Row(
        "item" varchar,
        "price" numeric
      )],
      "time" varchar
    )],
    "name" varchar
  );

(1 row)
```

Examples

In the following example, the input path contains data for a table with two integer columns. The external table definition can be fully inferred, and you can use the returned SQL statement as-is. The function reads one file from the input path:

```
=> SELECT INFER_TABLE_DDL('/data/orders/*.orc'
    USING PARAMETERS format = 'orc', table_name = 'orders', table_type = 'external');

                INFER_TABLE_DDL
-------------------------------------------------------------------------------
create external table "orders" (
  "id" int,
  "quantity" int
) as copy from '/data/orders/*.orc' orc;
(1 row)
```

To create a table in a schema, use the table_schema parameter. Do not add it to the table name; the function treats it as a name with a period in it, not a schema.

The following example shows output with complex types. You can use the definition as-is or modify the VARCHAR sizes:

```
=> SELECT INFER_TABLE_DDL('/data/people/*.parquet'
    USING PARAMETERS format = 'parquet', table_name = 'employees');
WARNING 9311:  This generated statement contains one or more varchar/varbinary columns which default to length 80

                INFER_TABLE_DDL
-------------------------------------------------------------------------------
 create table "employees"(
  "employeeID" int,
  "personal" Row(
    "name" varchar,
    "address" Row(
      "street" varchar,
      "city" varchar,
      "zipcode" int
    ),
    "taxID" int
  ),
  "department" varchar
 );
(1 row)
```

In the following example, the input file contains a map in the "prods" column. You can read a map as an array of rows:

```
=> SELECT INFER_TABLE_DDL('/data/orders.parquet'
    USING PARAMETERS format='parquet', table_name='orders');
WARNING 9311:  This generated statement contains one or more varchar/varbinary columns which default to length 80
                INFER_TABLE_DDL
-------------------------------------------------------------------------------
 create table "orders"(
  "orderkey" int,
  "custkey" int,
  "prods" Array[Row(
    "key" varchar,
    "value" numeric(12,2)
  )],
  "orderdate" date
 );
(1 row)
```

The following example returns the definition of a native table and the COPY statement, putting the table definition on a single line to simplify cutting and pasting into a script:

```
=> SELECT INFER_TABLE_DDL('/data/orders/*.orc'
    USING PARAMETERS format = 'orc', table_name = 'orders',
                     table_type = 'native', with_copy_statement = true, one_line_result=true);

                INFER_TABLE_DDL
-------------------------------------------------------------------------------
create table "orders" ("id" int, "quantity" int);
copy "orders" from '/data/orders/*.orc' orc;
(1 row)
```

In the following example, the data is partitioned by region. The function was not able to infer the data type and reports UNKNOWN:

```
=> SELECT INFER_TABLE_DDL('/data/sales/*/*
    USING PARAMETERS format = 'orc', table_name = 'sales', table_type = 'external');
WARNING 9262:  This generated statement is incomplete because of one or more unknown column types. Fix these data types before creating the table
WARNING 9311:  This generated statement contains one or more varchar/varbinary columns which default to length 80

                INFER_TABLE_DDL
-------------------------------------------------------------------------
 create external table "sales"(
  "orderkey" int,
  "custkey" int,
  "prodkey" Array[varchar],
  "orderprices" Array[numeric(12,2)],
  "orderdate" date,
  "region" UNKNOWN
 ) as copy from '/data/sales/*/*' PARTITION COLUMNS region orc;
(1 row)
```

In the following example, the function reads multiple JSON files and they differ in how they represent the menu column:

```
=> SELECT INFER_TABLE_DDL ('/data/*.json'
    USING PARAMETERS table_name='restaurants', format='json',
max_files=3, max_candidates=3);
WARNING 0:  This generated statement contains one or more float types which might lose precision
WARNING 0:  This generated statement contains one or more varchar/varbinary types which default to length 80

                INFER_TABLE_DDL
-------------------------------------------------------------------------
 Candidate matched 1/2 of total files(s):
  create table "restaurants"(
    "cuisine" varchar,
    "location_city" Array[varchar],
    "menu" Array[Row(
      "item" varchar,
      "price" float
    )],
    "name" varchar
  );
Candidate matched 1/2 of total files(s):
  create table "restaurants"(
    "cuisine" varchar,
    "location_city" Array[varchar],
    "menu" Array[Row(
      "items" Array[Row(
        "item" varchar,
        "price" numeric
      )],
      "time" varchar
    )],
    "name" varchar
  );

(1 row)
```

LAST_INSERT_ID

Returns the last value of an IDENTITY column. If multiple sessions concurrently load the same table with an IDENTITY column, the function returns the last value generated for that column.

> **Note**
> This function works only with IDENTITY columns. It does not work with named sequences .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
LAST_INSERT_ID()
```

Privileges

- Table owner
- USAGE privileges on the table schema

Examples

See IDENTITY sequences .

PURGE_TABLE

> **Note**
> This function was formerly named PURGE_TABLE_PROJECTIONS(). Vertica still supports the former function name.

Permanently removes deleted data from physical storage so disk space can be reused. You can purge historical data up to and including the Ancient History Mark epoch.

Purges all projections of the specified table. You cannot use this function to purge temporary tables.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
PURGE_TABLE ( '[[database.]schema.]table' )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

The table to purge.

Privileges

- Table owner
- USAGE privilege on schema

> **Caution**
> PURGE_TABLE could temporarily take up significant disk space while the data is being purged.

Examples

The following example purges all projections for the store sales fact table located in the Vmart schema:

```
=> SELECT PURGE_TABLE('store.store_sales_fact');
```

See also

- PURGE
- PURGE_TABLE
- STORAGE_CONTAINERS
- Purging deleted data

REBALANCE_TABLE

Synchronously rebalances data in the specified table.

A rebalance operation performs the following tasks:

- Distributes data based on:
  - User-defined fault groups , if specified
  - Large cluster automatic fault groups
- Redistributes database projection data across all nodes.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
REBALANCE_TABLE('[[database.]schema.]table-name')
```

Parameters

schema

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table-name

The table to rebalance.

Privileges

Superuser

When to rebalance

Rebalancing is useful or even necessary after you perform the following tasks:

- Mark one or more nodes as ephemeral in preparation of removing them from the cluster.
- Add one or more nodes to the cluster so that Vertica can populate the empty nodes with data.
- Change the scaling factor of an elastic cluster, which determines the number of storage containers used to store a projection across the database.
- Set the control node size or realign control nodes on a large cluster layout
- Add nodes to or remove nodes from a fault group .

> **Tip**
> By default, before performing a rebalance, Vertica queries system tables to compute the size of all projections involved in the rebalance task. This query can add significant overhead to the rebalance operation. To disable this query, set projection configuration parameter RebalanceQueryStorageContainers to 0.

Examples

The following command shows how to rebalance data on the specified table.

```
=> SELECT REBALANCE_TABLE('online_sales.online_sales_fact');
REBALANCE_TABLE
-------------------
 REBALANCED
(1 row)
```

See also

- REBALANCE_CLUSTER
- Rebalancing Data Across Nodes
- NODES

REENABLE_DUPLICATE_KEY_ERROR

Restores the default behavior of error reporting by reversing the effects of DISABLE_DUPLICATE_KEY_ERROR . Effects are session-scoped.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
REENABLE_DUPLICATE_KEY_ERROR();
```

Privileges

Superuser

Examples

```
=> SELECT REENABLE_DUPLICATE_KEY_ERROR();
 REENABLE_DUPLICATE_KEY_ERROR
------------------------------
 Duplicate key error enabled
(1 row)
```

See also

ANALYZE_CONSTRAINTS

Match and search functions

This section contains functions for text search and regular expressions, and functions used in the MATCH clause.

In this section

- MATCH clause functions
- Regular expression functions
- Text search functions

MATCH clause functions

Used with the MATCH clause , the functions in this section return additional data about the patterns found or returned. For example, you can use these functions to return values representing the name of the event or pattern that matched the input row, the sequential number of the match, or a partition-wide unique identifier for the instance of the pattern that matched.

Pattern matching is particularly useful for clickstream analysis where you might want to identify users' actions based on their Web browsing behavior (page clicks). A typical online clickstream funnel is:

Company home page -> product home page -> search -> results -> purchase online

Using the above clickstream funnel, you can search for a match on the user's sequence of web clicks and identify that the user:

- Landed on the company home page.
- Navigated to the product page.
- Ran a search.
- Clicked a link from the search results.
- Made a purchase.

For examples that use this clickstream model, see Event series pattern matching .

> **Note**
> GROUP BY and PARTITION BY expressions do not support window functions.

In this section

- EVENT_NAME
- MATCH_ID
- PATTERN_ID

EVENT_NAME

Returns a VARCHAR value representing the name of the event that matched the row.

Syntax

```
EVENT_NAME()
```

Notes

Pattern matching functions must be used in MATCH clause syntax; for example, if you call EVENT_NAME() on its own, Vertica returns the following error message:

```
=> SELECT event_name();
ERROR:  query with pattern matching function event_name must include a MATCH clause
```

Examples

> **Note**
> This example uses the schema defined in Event series pattern matching .

The following statement analyzes users' browsing history on website2.com and identifies patterns where the user landed on website2.com from another Web site (Entry) and browsed to any number of other pages (Onsite) before making a purchase (Purchase). The query also outputs the values for EVENT_NAME(), which is the name of the event that matched the row.

```
SELECT uid,
       sid,
       ts,
       refurl,
       pageurl,
       action,
       event_name()
FROM clickstream_log
MATCH
  (PARTITION BY uid, sid ORDER BY ts
   DEFINE
     Entry    AS RefURL  NOT ILIKE '%website2.com%' AND PageURL ILIKE '%website2.com%',
     Onsite   AS PageURL ILIKE     '%website2.com%' AND Action='V',
     Purchase AS PageURL ILIKE     '%website2.com%' AND Action = 'P'
   PATTERN
     P AS (Entry Onsite* Purchase)
   ROWS MATCH FIRST EVENT);
 uid | sid |    ts    |        refurl        |       pageurl        | action | event_name
-----+-----+----------+----------------------+----------------------+--------+------------
   1 | 100 | 12:00:00 | website1.com         | website2.com/home    | V      | Entry
   1 | 100 | 12:01:00 | website2.com/home    | website2.com/floby   | V      | Onsite
   1 | 100 | 12:02:00 | website2.com/floby   | website2.com/shamwow | V      | Onsite
   1 | 100 | 12:03:00 | website2.com/shamwow | website2.com/buy     | P      | Purchase
   2 | 100 | 12:10:00 | website1.com         | website2.com/home    | V      | Entry
   2 | 100 | 12:11:00 | website2.com/home    | website2.com/forks   | V      | Onsite
   2 | 100 | 12:13:00 | website2.com/forks   | website2.com/buy     | P      | Purchase
(7 rows)
```

See also

- MATCH clause
- MATCH_ID
- PATTERN_ID
- Event series pattern matching

MATCH_ID

Returns a successful pattern match as an INTEGER value. The returned value is the ordinal position of a match within a partition.

Syntax

```
MATCH_ID()
```

Notes

Pattern matching functions must be used in MATCH clause syntax; for example, if you call MATCH_ID() on its own, Vertica returns the following error message:

```
=> SELECT match_id();
ERROR:  query with pattern matching function match_id must include a MATCH clause
```

Examples

> **Note**
> This example uses the schema defined in Event series pattern matching .

The following statement analyzes users' browsing history on a site called website2.com and identifies patterns where the user reached website2.com from another Web site (Entry in the MATCH clause) and browsed to any number of other pages (Onsite) before making a purchase (Purchase). The query also outputs values for the MATCH_ID(), which represents a sequential number of the match.

```
SELECT uid,
       sid,
       ts,
       refurl,
       pageurl,
       action,
       match_id()
FROM clickstream_log
MATCH
  (PARTITION BY uid, sid ORDER BY ts
   DEFINE
     Entry    AS RefURL  NOT ILIKE '%website2.com%' AND PageURL ILIKE '%website2.com%',
     Onsite   AS PageURL ILIKE     '%website2.com%' AND Action='V',
     Purchase AS PageURL ILIKE     '%website2.com%' AND Action = 'P'
   PATTERN
     P AS (Entry Onsite* Purchase)
   ROWS MATCH FIRST EVENT);

uid | sid |    ts    |       refurl        |       pageurl        | action | match_id
----+-----+----------+---------------------+----------------------+--------+----------
1   | 100 | 12:00:00 | website1.com        | website2.com/home    | V      |        1
1   | 100 | 12:01:00 | website2.com/home   | website2.com/floby   | V      |        2
1   | 100 | 12:02:00 | website2.com/floby  | website2.com/shamwow | V      |        3
1   | 100 | 12:03:00 | website2.com/shamwow | website2.com/buy    | P      |        4
2   | 100 | 12:10:00 | website1.com        | website2.com/home    | V      |        1
2   | 100 | 12:11:00 | website2.com/home   | website2.com/forks   | V      |        2
2   | 100 | 12:13:00 | website2.com/forks  | website2.com/buy     | P      |        3
(7 rows)
```

See also

- MATCH clause
- EVENT_NAME
- PATTERN_ID
- Event series pattern matching

PATTERN_ID

Returns an integer value that is a partition-wide unique identifier for the instance of the pattern that matched.

Syntax

```
PATTERN_ID()
```

Notes

Pattern matching functions must be used in MATCH clause syntax; for example, if call PATTERN_ID() on its own, Vertica returns the following error message:

```
=> SELECT pattern_id();
ERROR:  query with pattern matching function pattern_id must include a MATCH clause
```

Examples

Note
This example uses the schema defined in Event series pattern matching .

The following statement analyzes users' browsing history on website2.com and identifies patterns where the user landed on website2.com from another Web site (Entry) and browsed to any number of other pages (Onsite) before making a purchase (Purchase). The query also outputs values for PATTERN_ID(), which represents the partition-wide identifier for the instance of the pattern that matched.

```
SELECT uid,
       sid,
       ts,
       refurl,
       pageurl,
       action,
       pattern_id()
FROM clickstream_log
MATCH
  (PARTITION BY uid, sid ORDER BY ts
   DEFINE
     Entry    AS RefURL  NOT ILIKE '%website2.com%' AND PageURL ILIKE '%website2.com%',
     Onsite   AS PageURL ILIKE     '%website2.com%' AND Action='V',
     Purchase AS PageURL ILIKE     '%website2.com%' AND Action = 'P'
   PATTERN
     P AS (Entry Onsite* Purchase)
   ROWS MATCH FIRST EVENT);


uid | sid |    ts    |        refurl        |       pageurl        | action | pattern_id
----+-----+----------+----------------------+----------------------+--------+------------
1   | 100 | 12:00:00 | website1.com         | website2.com/home    | V      |          1
1   | 100 | 12:01:00 | website2.com/home    | website2.com/floby   | V      |          1
1   | 100 | 12:02:00 | website2.com/floby   | website2.com/shamwow | V      |          1
1   | 100 | 12:03:00 | website2.com/shamwow | website2.com/buy     | P      |          1
2   | 100 | 12:10:00 | website1.com         | website2.com/home    | V      |          1
2   | 100 | 12:11:00 | website2.com/home    | website2.com/forks   | V      |          1
2   | 100 | 12:13:00 | website2.com/forks   | website2.com/buy     | P      |          1
(7 rows)
```

See also

- MATCH clause
- EVENT_NAME
- MATCH_ID
- Event series pattern matching

Regular expression functions

A regular expression lets you perform pattern matching on strings of characters. The regular expression syntax allows you to precisely define the pattern used to match strings, giving you much greater control than wildcard matching used in the LIKE predicate. The Vertica regular expression functions let you perform tasks such as determining if a string value matches a pattern, extracting a portion of a string that matches a pattern, or counting the number of times a pattern occurs within a string.

Vertica uses the Perl Compatible Regular Expression (PCRE) library to evaluate regular expressions. As its name implies, PCRE's regular expression syntax is compatible with the syntax used by the Perl 5 programming language. You can read PCRE's documentation about its library. However, if you are unfamiliar with using regular expressions, the Perl Regular Expressions Documentation is a good introduction.

Note
The regular expression functions only operate on valid UTF-8 strings. If you try using a regular expression function on a string that is not valid UTF-8, the query fails with an error. To prevent an error from occurring, use the ISUTF8 function as an initial clause to ensure the strings you pass to the regular expression functions are valid UTF-8 strings. Alternatively, or you can use the 'b' argument to treat the strings as binary octets, rather than UTF-8 encoded strings.

In this section

- MATCH_COLUMNS
- REGEXP_COUNT
- REGEXP_ILIKE
- REGEXP_INSTR
- REGEXP_LIKE
- REGEXP_NOT_ILIKE
- REGEXP_NOT_LIKE
- REGEXP_REPLACE
- REGEXP_SUBSTR

MATCH_COLUMNS

Specified as an element in a SELECT list, returns all columns in queried tables that match the specified pattern. For example:

```
=> SELECT MATCH_COLUMNS ('%order%') FROM store.store_orders_fact LIMIT 3;
 order_number | date_ordered | quantity_ordered | total_order_cost | reorder_level
--------------+--------------+------------------+------------------+---------------
       191119 | 2003-03-09   |               15 |             4021 |            23
        89985 | 2003-05-04   |               19 |             2692 |            23
       246962 | 2007-06-01   |               77 |             4419 |            42
(3 rows)
```

Syntax

```
MATCH_COLUMNS ('pattern')
```

Arguments

pattern

The pattern to match against all column names in the queried tables, where *pattern* typically contains one or both of the following wildcard characters:

- _ (underscore): Match any single character.
- % (percent sign): Match any string of zero or more characters.

The pattern can also include backslash (\ ) characters to escape reserved characters that are embedded in column names: _ (underscore), % (percent sign), and backslash (\ ) itself.

Privileges

None

DDL usage

You can use MATCH_COLUMNS to define database objects—for example, specify it in CREATE PROJECTION to identify projection columns, or in CREATE TABLE...AS to identify columns in the new table. In all cases, Vertica expands the MATCH_COLUMNS output before it stores the object DDL. Subsequent changes to the original source table have no effect on the derived object definitions.

Restrictions

In general, MATCH_COLUMNS is specified as an element in a SELECT list. For example, CREATE PROJECTION can call MATCH_COLUMNS to specify the columns to include in a projection. However, attempts to specify columns in the projection's segmentation clause return with an error:

```
=> CREATE PROJECTION p_store_orders AS SELECT
   MATCH_COLUMNS('%product%'),
   MATCH_COLUMNS('%store%'),
   order_number FROM store.store_orders_fact SEGMENTED BY MATCH_COLUMNS('products%') ALL NODES;
ERROR 0:  MATCH_COLUMNS() function can only be specified as an element in a SELECT list
=> CREATE PROJECTION p_store_orders AS SELECT
   MATCH_COLUMNS('%product%'),
   MATCH_COLUMNS('%store%'),
   order_number FROM store.store_orders_fact;
WARNING 4468:  Projection <store.p_store_orders_b0> is not available for query processing. Execute the select start_refresh() function to copy data into this projection.
        The projection must have a sufficient number of buddy projections and all nodes must be up before starting a refresh
WARNING 4468:  Projection <store.p_store_orders_b1> is not available for query processing. Execute the select start_refresh() function to copy data into this projection.
        The projection must have a sufficient number of buddy projections and all nodes must be up before starting a refresh
CREATE PROJECTION
```

If you call MATCH_COLUMNS from a function that supports a fixed number of arguments, Vertica returns an error. For example, the UPPER function supports only one argument; so calling MATCH_COLUMNS from UPPER as follows returns an error:

```
=> SELECT MATCH_COLUMNS('emp%') FROM employee_dimension LIMIT 1;
-[ RECORD 1 ]----------+--------------------------------
employee_key           | 1
employee_gender        | Male
employee_first_name    | Craig
employee_middle_initial | F
employee_last_name     | Robinson
employee_age           | 22
employee_street_address | 5 Bakers St
employee_city          | Thousand Oaks
employee_state         | CA
employee_region        | West

=> SELECT UPPER (MATCH_COLUMNS('emp%')) FROM employee_dimension;
ERROR 10465:  MATCH_COLUMNS() function can only be specified as an element in a SELECT list
```

In contrast, the HASH function accepts an unlimited number of arguments, so calling MATCH_COLUMNS as an argument succeeds:

```
=> select HASH(MATCH_COLUMNS('emp%')) FROM employee_dimension LIMIT 10;
       HASH
---------------------
 2047284364908178817
 1421997332260827278
 7981613309330877388
  792898558199431621
 5275639269069980417
 7892790768178152349
  184601038712735208
 3020263228621856381
 7056305566297085916
 3328422577712931057
(10 rows)
```

Other constraints

The following usages of MATCH_COLUMNS are invalid and return with an error:

- Including MATCH_COLUMNS in the non-recursive (base) term query of a RECURSIVE WITH clause
- Concatenating the results of MATCH_COLUMNS calls:

  ```
  => SELECT MATCH_COLUMNS ('%store%')||MATCH_COLUMNS('%store%') FROM store.store_orders_fact;
  ERROR 0:  MATCH_COLUMNS() function can only be specified as an element in a SELECT list
  ```

- Setting an alias on MATCH_COLUMNS

Examples

The following CREATE PROJECTION statement uses MATCH_COLUMNS to specify table columns in the new projection:

```
=> CREATE PROJECTION p_store_orders AS SELECT
    MATCH_COLUMNS('%product%'),
    MATCH_COLUMNS('%store%'),
    order_number FROM store.store_orders_fact;
WARNING 4468:  Projection <store.p_store_orders_b0> is not available for query processing. Execute the select start_refresh() function to copy data into this projection.
        The projection must have a sufficient number of buddy projections and all nodes must be up before starting a refresh
WARNING 4468:  Projection <store.p_store_orders_b1> is not available for query processing. Execute the select start_refresh() function to copy data into this projection.
        The projection must have a sufficient number of buddy projections and all nodes must be up before starting a refresh
CREATE PROJECTION

=> SELECT export_objects('', 'store.p_store_orders_b0');
...

CREATE PROJECTION store.p_store_orders_b0 /*+basename(p_store_orders)*/
(
 product_key,
 product_version,
 store_key,
 order_number
)
AS
 SELECT store_orders_fact.product_key,
        store_orders_fact.product_version,
        store_orders_fact.store_key,
        store_orders_fact.order_number
 FROM store.store_orders_fact
 ORDER BY store_orders_fact.product_key,
          store_orders_fact.product_version,
          store_orders_fact.store_key,
          store_orders_fact.order_number
SEGMENTED BY hash(store_orders_fact.product_key, store_orders_fact.product_version, store_orders_fact.store_key, store_orders_fact.order_number)
ALL NODES OFFSET 0;

SELECT MARK_DESIGN_KSAFE(1);

(1 row)
```

As shown in the EXPORT_OBJECTS output, Vertica stores the result sets of the two MATCH_COLUMNS calls in the new projection's DDL. Later changes in the anchor table DDL have no effect on this projection.

REGEXP_COUNT

Returns the number times a regular expression matches a string.

This function operates on UTF-8 strings using the default locale, even if the locale is set otherwise.

Important

If you port a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while Vertica does not.

Syntax

```
REGEXP_COUNT ( string-expession, pattern [, position [, regexp-modifier ]... ] )
```

Parameters

string-expression

The VARCHAR or LONG VARCHAR expression to evaluate for matches with the regular expression specified in *pattern* . If *string-expression* is in

the __raw__ column of a flex or columnar table, cast the string to a LONG VARCHAR before searching for *pattern* .

pattern

The regular expression to match against *string-expression* . The regular expression must conform with Perl regular expression syntax .

position

The number of characters from the start of the string where the function should start searching for matches. By default, the function begins searching for a match at the first (leftmost) character. Setting this parameter to a value greater than 1 begins searching for a match at the * n *th character you specify.

Default: 1

regexp-modifier

One or more single-character flags that modify how the regular expression *pattern* is matched to *string-expression* :

- b : Treat strings as binary octets, rather than UTF-8 characters.
- c (default): Force the match to be case sensitive.
- i : Force the match to be case insensitive.
- m : Treat the string to match as multiple lines. Using this modifier, the start of line (^) and end of line ($) regular expression operators match line breaks (\n) within the string. Without the m modifier, the start and end of line operators match only the start and end of the string.
- n : Match the regular expression operator (.) to a newline (\n). By default, the . operator matches any character except a newline.
- x : Add comments to regular expressions. The x modifier causes the function to ignore all un-escaped space characters and comments in the regular expression. Comments start with hash (#) and end with a newline (\n). All spaces in the regular expression to be matched in strings must be escaped with a backslash (\ ).

Examples

Count the number of occurrences of the substring an in the specified string (a man, a plan, a canal: Panama):

```
=> SELECT REGEXP_COUNT('a man, a plan, a canal: Panama', 'an');
 REGEXP_COUNT
--------------
            4
(1 row)
```

Find the number of occurrences of the substring an , starting with the fifth character.

```
=> SELECT REGEXP_COUNT('a man, a plan, a canal: Panama', 'an',5);
 REGEXP_COUNT
--------------
            3
(1 row)
```

Find the number of occurrences of a substring containing a lower-case character followed by an :

```
=> SELECT REGEXP_COUNT('a man, a plan, a canal: Panama', '[a-z]an');
 REGEXP_COUNT
--------------
            3
(1 row
```

REGEXP_COUNT specifies the i modifier, so it ignores case:

```
=> SELECT REGEXP_COUNT('a man, a plan, a canal: Panama', '[a-z]an', 1, 'i');

 REGEXP_COUNT
--------------
            4
```

REGEXP_ILIKE

Returns true if the string contains a match for the regular expression. REGEXP_ILIKE is similar to the LIKE , except that it uses a case insensitive regular expression, rather than simple wildcard character matching.

This function operates on UTF-8 strings using the default locale, even if the locale is set otherwise.

Important
If you port a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while Vertica does not.

Syntax

```
REGEXP_ILIKE ( string-expression, pattern )
```

Parameters

string-expression ``
The VARCHAR or LONG VARCHAR expression to evaluate for matches with the regular expression specified in *pattern* . If *string-expression* is in the __raw__ column of a flex or columnar table, cast the string to a LONG VARCHAR before searching for *pattern* .

pattern ``
The regular expression to match against *string-expression* . The regular expression must conform with Perl regular expression syntax .

Examples

This example creates a table containing several strings to demonstrate regular expressions.

1. Create table longvc with a single, long varchar column body , and insert data with distinct characters:

```
=> CREATE table longvc(body long varchar (1048576));
CREATE TABLE

=> insert into longvc values ('На бepery пустынных волн');
=> insert into longvc values ('Voin syödä lasia, se ei vahingoita minua');
=> insert into longvc values ('私はガラスを食べられます。それは私を傷つけません。');
=> insert into longvc values ('Je peux manger du verre, ça ne me fait pas mal.');
=> insert into longvc values ('zésbaésbaa');
=> insert into longvc values ('Out of the frying pan, he landed immediately in the fire');

=> SELECT * FROM longvc;
                    body
-------------------------------------------------
 На бepery пустынных волн
 Voin syödä lasia, se ei vahingoita minua
 私はガラスを食べられます。それは私を傷つけません。
 Je peux manger du verre, ça ne me fait pas mal.
 zésbaésbaa
 Out of the frying pan, he landed immediately in the fire
(6 rows)
```

2. Pattern match table rows containing the character ç :

```
=> SELECT * FROM longvc where regexp_ilike(body, 'ç');
                    body
-------------------------------------------------
 Je peux manger du verre, ça ne me fait pas mal.
(1 row)
```

3. Select all rows that contain the characters A / a :

```
=> SELECT * FROM longvc where regexp_ilike(body, 'A');
                    body
-------------------------------------------------
 Je peux manger du verre, ça ne me fait pas mal.
 Voin syödä lasia, se ei vahingoita minua
 zésbaésbaa
(3 rows)
```

4. Select all rows that contain the characters O / o :

```
=> SELECT * FROM longvc where regexp_ilike(body, 'O');
                        body
-------------------------------------------------------------
 Voin syödä lasia, se ei vahingoita minua
 Out of the frying pan, he landed immediately in the fire
(2 rows)
```

REGEXP_INSTR

Returns the starting or ending position in a string where a regular expression matches. REGEXP_INSTR returns 0 if no match for the regular expression is found in the string.

This function operates on UTF-8 strings using the default locale, even if the locale is set otherwise.

Important
If you port a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while Vertica does not.

Syntax

```
REGEXP_INSTR ( string-expression, pattern
  [, position [, occurrence [, return-position [, regexp-modifier ]... [, captured-subexp ]]]] )
```

Parameters

string-expression

The VARCHAR or LONG VARCHAR expression to evaluate for matches with the regular expression specified in *pattern* . If *string-expression* is in the __raw__ column of a flex or columnar table, cast the string to a LONG VARCHAR before searching for *pattern* .

pattern

The regular expression to match against *string-expression* . The regular expression must conform with Perl regular expression syntax .

position

The number of characters from the start of the string where the function should start searching for matches. By default, the function begins searching for a match at the first (leftmost) character. Setting this parameter to a value greater than 1 begins searching for a match at the * n *th character you specify.

Default: 1

occurrence

Controls which occurrence of a pattern match in the string to return. By default, the function returns the position of the first matching substring. Use this parameter to find the position of subsequent matching substrings. For example, setting this parameter to 3 returns the position of the third substring that matches the pattern.

Default: 1

return-position

Sets the position within the string to return. Using the default position (0), the function returns the string position of the first character of the substring that matches the pattern. If you set *return-position* to 1, the function returns the position of the first character after the end of the matching substring.

Default: 0

regexp-modifier

One or more single-character flags that modify how the regular expression *pattern* is matched to *string-expression* :

- b : Treat strings as binary octets, rather than UTF-8 characters.
- c (default): Force the match to be case sensitive.
- i : Force the match to be case insensitive.
- m : Treat the string to match as multiple lines. Using this modifier, the start of line (^) and end of line ($) regular expression operators match line breaks (\n) within the string. Without the m modifier, the start and end of line operators match only the start and end of the string.
- n : Match the regular expression operator (.) to a newline (\n). By default, the . operator matches any character except a newline.
- x : Add comments to regular expressions. The x modifier causes the function to ignore all un-escaped space characters and comments in the regular expression. Comments start with hash (#) and end with a newline (\n). All spaces in the regular expression to be matched in

strings must be escaped with a backslash (\ ).

captured-subexp

The captured subexpression whose position to return. By default, the function returns the position of the first character in *string* that matches the regular expression. If you set this value from 1 – 9, the function returns the subexpression captured by the corresponding set of parentheses in the regular expression. For example, setting this value to 3 returns the substring captured by the third set of parentheses in the regular expression.

Default: 0

> **Note**
> The subexpressions are numbered left to right, based on the appearance of opening parenthesis, so nested regular expressions . For example, in the regular expression \s*(\w+\s+(\w+)) , subexpression 1 is the one that captures everything but any leading whitespaces.

Examples

Find the first occurrence of a sequence of letters starting with the letter e and ending with the letter y in the specified string (easy come, easy go).

```
=> SELECT REGEXP_INSTR('easy come, easy go','e\w*y');
 REGEXP_INSTR
--------------
            1
(1 row)
```

Starting at the second character (2), find the first sequence of letters starting with the letter e and ending with the letter y :

```
=> SELECT REGEXP_INSTR('easy come, easy go','e\w*y',2);
 REGEXP_INSTR
--------------
           12
(1 row)
```

Starting at the first character (1), find the second sequence of letters starting with the letter e and ending with the letter y :

```
=> SELECT REGEXP_INSTR('easy come, easy go','e\w*y',1,2);
 REGEXP_INSTR
--------------
           12
(1 row)
```

Find the position of the first character after the first whitespace:

```
=> SELECT REGEXP_INSTR('easy come, easy go','\s',1,1,1);
 REGEXP_INSTR
--------------
            6
(1 row)
```

Find the position of the start of the third word in a string by capturing each word as a subexpression, and returning the third subexpression's start position.

```
=> SELECT REGEXP_INSTR('one two three','(\w+)\s+(\w+)\s+(\w+)', 1,1,0,'',3);
REGEXP_INSTR
--------------
            9
(1 row)
```

REGEXP_LIKE

Returns true if the string matches the regular expression. REGEXP_LIKE is similar to the LIKE , except that it uses regular expressions rather than simple wildcard character matching.

This function operates on UTF-8 strings using the default locale, even if the locale is set otherwise.

Important

If you port a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while Vertica does not.

Syntax

```
REGEXP_LIKE ( string-expression, pattern [, regexp-modifier ]... )
```

Parameters

string-expression

The VARCHAR or LONG VARCHAR expression to evaluate for matches with the regular expression specified in *pattern* . If *string-expression* is in the __raw__ column of a flex or columnar table, cast the string to a LONG VARCHAR before searching for *pattern* .

pattern

The regular expression to match against *string-expression* . The regular expression must conform with Perl regular expression syntax .

regexp-modifier

One or more single-character flags that modify how the regular expression *pattern* is matched to *string-expression* :

- b : Treat strings as binary octets, rather than UTF-8 characters.
- c (default): Force the match to be case sensitive.
- i : Force the match to be case insensitive.
- m : Treat the string to match as multiple lines. Using this modifier, the start of line (^) and end of line ($) regular expression operators match line breaks (\n) within the string. Without the m modifier, the start and end of line operators match only the start and end of the string.
- n : Match the regular expression operator (.) to a newline (\n). By default, the . operator matches any character except a newline.
- x : Add comments to regular expressions. The x modifier causes the function to ignore all un-escaped space characters and comments in the regular expression. Comments start with hash (#) and end with a newline (\n). All spaces in the regular expression to be matched in strings must be escaped with a backslash (\ ).

Examples

Create a table that contains several strings:

```
=> CREATE TABLE t (v VARCHAR);
CREATE TABLE
=> CREATE PROJECTION t1 AS SELECT * FROM t;
CREATE PROJECTION
=> COPY t FROM stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> aaa
>> Aaa
>> abc
>> abc1
>> 123
>> \.
=> SELECT * FROM t;
  v
-------
 aaa
 Aaa
 abc
 abc1
 123
(5 rows)
```

Select all records from table t that contain the letter a :

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'a');
 v
------
 Aaa
 aaa
 abc
 abc1
(4 rows)
```

Select all rows from table t that start with the letter a :

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'^a');
 v
------
 aaa
 abc
 abc1
(3 rows)
```

Select all rows that contain the substring aa :

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'aa');
 v
-----
 Aaa
 aaa
(2 rows)
```

Select all rows that contain a digit.

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'\d');
 v
------
 123
 abc1
(2 rows)
```

Select all rows that contain the substring aaa .

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'aaa');
 v
-----
 aaa
(1 row)
```

Select all rows that contain the substring aaa using case-insensitive matching.

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'aaa', 'i');
 v
-----
 Aaa
 aaa
(2 rows)
```

Select rows that contain the substring a b c .

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'a b c');
 v
---
(0 rows)
```

Select rows that contain the substring a b c , ignoring space within the regular expression.

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'a b c','x');
 v
------
 abc
 abc1
(2 rows)
```

Add multi-line rows to table t :

```
=> COPY t FROM stdin RECORD TERMINATOR '!';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> Record 1 line 1
>> Record 1 line 2
>> Record 1 line 3!
>> Record 2 line 1
>> Record 2 line 2
>> Record 2 line 3!
>> \.
```

Select rows from table t that start with the substring Record and end with the substring line 2 .

```
=> SELECT v from t WHERE REGEXP_LIKE(v,'^Record.*line 2$');
 v
---
(0 rows)
```

Select rows that start with the substring Record and end with the substring line 2 , treating multiple lines as separate strings.

```
=> SELECT v from t WHERE REGEXP_LIKE(v,'^Record.*line 2$','m');
                     v
-----------------------------------------------------

Record 2 line 1
Record 2 line 2
Record 2 line 3
Record 1 line 1
Record 1 line 2
Record 1 line 3
(2 rows)
```

REGEXP_NOT_ILIKE

Returns true if the string does not match the case-insensitive regular expression.

This function operates on UTF-8 strings using the default locale, even if the locale is set otherwise.

Important
If you port a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while Vertica does not.

Syntax

```
REGEXP_NOT_ILIKE ( string-expression, pattern )
```

Parameters

string-expression ``

The VARCHAR or LONG VARCHAR expression to evaluate for matches with the regular expression specified in *pattern* . If *string-expression* is in the __raw__ column of a flex or columnar table, cast the string to a LONG VARCHAR before searching for *pattern* .

pattern ``

The regular expression to match against *string-expression* . The regular expression must conform with Perl regular expression syntax .

Examples

1. Create a table (longvc) with a single, long varchar column (body). Then, insert data with some distinct characters, and query the table contents:

```
=> CREATE table longvc(body long varchar (1048576));
CREATE TABLE

=> insert into longvc values ('На бepery пустынных волн');
=> insert into longvc values ('Voin syödä lasia, se ei vahingoita minua');
=> insert into longvc values ('私はガラスを食べられます。それは私を傷つけません。');
=> insert into longvc values ('Je peux manger du verre, ça ne me fait pas mal.');
=> insert into longvc values ('zésbaésbaa');

=> SELECT * FROM longvc;
                    body
-----------------------------------------------------
 На бepery пустынных волн
 Voin syödä lasia, se ei vahingoita minua
 私はガラスを食べられます。それは私を傷つけません。
 Je peux manger du verre, ça ne me fait pas mal.
 zésbaésbaa
(5 rows)
```

2. Find all rows that do not contain the character ç :

```
=> SELECT * FROM longvc where regexp_not_ilike(body, 'ç');
                     body
-----------------------------------------------------
 Voin syödä lasia, se ei vahingoita minua
 zésbaésbaa
 На бepery пустынных волн
 私はガラスを食べられます。それは私を傷つけません。
(4 rows)
```

3. Find all rows that do not contain the substring a :

```
=> SELECT * FROM longvc where regexp_not_ilike(body, 'a');
                     body
-----------------------------------------------------
 На бepery пустынных волн
 私はガラスを食べられます。それは私を傷つけません。
(2 rows)
```

REGEXP_NOT_LIKE

Returns true if the string does not contain a match for the regular expression. REGEXP_NOT_LIKE is a case sensitive regular expression.

This function operates on UTF-8 strings using the default locale, even if the locale is set otherwise.

Important
If you port a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while Vertica does not.

Syntax

```
REGEXP_NOT_LIKE ( string-expression, pattern )
```

Parameters

string-expression ``
The VARCHAR or LONG VARCHAR expression to evaluate for matches with the regular expression specified in *pattern* . If *string-expression* is in the __raw__ column of a flex or columnar table, cast the string to a LONG VARCHAR before searching for *pattern* .

pattern ``
The regular expression to match against *string-expression* . The regular expression must conform with Perl regular expression syntax .

Examples

1. Create a table (longvc) with the LONG VARCHAR column body . Then, insert data with some distinct characters and query the table contents:

```
=> CREATE table longvc(body long varchar (1048576));
CREATE TABLE

=> insert into longvc values ('На бepery пустынных волн');
=> insert into longvc values ('Voin syödä lasia, se ei vahingoita minua');
=> insert into longvc values ('私はガラスを食べられます。それは私を傷つけません。');
=> insert into longvc values ('Je peux manger du verre, ça ne me fait pas mal.');
=> insert into longvc values ('zésbaésbaa');

=> SELECT * FROM longvc;
                  body
-------------------------------------------------
 На бepery пустынных волн
 Voin syödä lasia, se ei vahingoita minua
 私はガラスを食べられます。それは私を傷つけません。
 Je peux manger du verre, ça ne me fait pas mal.
 zésbaésbaa
(5 rows)
```

2. Use REGEXP_NOT_LIKE to return rows that do not contain the character ç :

```
=> SELECT * FROM longvc where regexp_not_like(body, 'ç');
                   body
-----------------------------------------------------
 Voin syödä lasia, se ei vahingoita minua
 zésbaésbaa
 На бepery пустынных волн
 私はガラスを食べられます。それは私を傷つけません。
(4 rows)
```

3. Return all rows that do not contain the characters *ö and *ä :

```
=> SELECT * FROM longvc where regexp_not_like(body, '.*ö.*ä');
                   body
-----------------------------------------------------
 Je peux manger du verre, ça ne me fait pas mal.
 zésbaésbaa
 На бepery пустынных волн
 私はガラスを食べられます。それは私を傷つけません。
(4 rows)
```

4. Pattern match all rows that do not contain the characters z and *ésbaa :

```
=> SELECT * FROM longvc where regexp_not_like(body, 'z.*ésbaa');
                   body
-----------------------------------------------------
 Je peux manger du verre, ça ne me fait pas mal.
 Voin syödä lasia, se ei vahingoita minua
 zésbaésbaa
 На бepery пустынных волн
 私はガラスを食べられます。それは私を傷つけません。
(5 rows)
```

REGEXP_REPLACE

Replaces all occurrences of a substring that match a regular expression with another substring. REGEXP_REPLACE is similar to the REPLACE function, except it uses a regular expression to select the substring to be replaced.

This function operates on UTF-8 strings using the default locale, even if the locale is set otherwise.

Important
If you port a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while Vertica does not.

Syntax

```
REGEXP_REPLACE ( string-expression, target
  [, replacement [, position [, occurrence[...] [, regexp-modifier]]]] )
```

Parameters

string-expression

The VARCHAR or LONG VARCHAR expression to evaluate for matches with the regular expression specified in *pattern* . If *string-expression* is in the __raw__ column of a flex or columnar table, cast the string to a LONG VARCHAR before searching for *pattern* .

pattern

The regular expression to match against *string-expression* . The regular expression must conform with Perl regular expression syntax .

replacement

The string to replace matched substrings. If you do not supply a *replacement* , the function deletes matched substrings. The replacement string can contain backreferences for substrings captured by the regular expression. The first captured substring is inserted into the replacement string using \1 , the second \2 , and so on.

position

The number of characters from the start of the string where the function should start searching for matches. By default, the function begins searching for a match at the first (leftmost) character. Setting this parameter to a value greater than 1 begins searching for a match at the *n* -th character you specify.

Default: 1

occurrence

Controls which occurrence of a pattern match in the string to replace. By default, the function replaces all matching substrings. For example, setting this parameter to 3 replaces the third matching instance.

Default: 1

regexp-modifier

One or more single-character flags that modify how the regular expression *pattern* is matched to *string-expression* :

- b : Treat strings as binary octets, rather than UTF-8 characters.
- c (default): Force the match to be case sensitive.
- i : Force the match to be case insensitive.
- m : Treat the string to match as multiple lines. Using this modifier, the start of line (^) and end of line ($) regular expression operators match line breaks (\n) within the string. Without the m modifier, the start and end of line operators match only the start and end of the string.
- n : Match the regular expression operator (.) to a newline (\n). By default, the . operator matches any character except a newline.
- x : Add comments to regular expressions. The x modifier causes the function to ignore all un-escaped space characters and comments in the regular expression. Comments start with hash (#) and end with a newline (\n). All spaces in the regular expression to be matched in strings must be escaped with a backslash (\ ).

How Oracle handles subexpressions

Unlike Oracle, Vertica can handle an unlimited number of captured subexpressions, while Oracle is limited to nine.

In Vertica, you can use \10 in the replacement pattern to access the substring captured by the tenth set of parentheses in the regular expression. In Oracle, \10 is treated as the substring captured by the first set of parentheses, followed by a zero. To force this Oracle behavior in Vertica, use the \g back reference and enclose the number of the captured subexpression in curly braces. For example, \g{1}0 is the substring captured by the first set of parentheses followed by a zero.

You can also name captured subexpressions to make your regular expressions less ambiguous. See the PCRE documentation for details.

Examples

Find groups of word characters—letters, numbers and underscore—that end with thy in the string healthy, wealthy, and wise , and replace them with nothing.

```
=> SELECT REGEXP_REPLACE('healthy, wealthy, and wise','\w+thy');
 REGEXP_REPLACE
----------------
 , , and wise
(1 row)
```

Find groups of word characters ending with thy and replace with the string something .

```
=> SELECT REGEXP_REPLACE('healthy, wealthy, and wise','\w+thy', 'something');
        REGEXP_REPLACE
---------------------------------
 something, something, and wise
(1 row)
```

Find groups of word characters ending with thy and replace with the string something starting at the third character in the string.

```
=> SELECT REGEXP_REPLACE('healthy, wealthy, and wise','\w+thy', 'something', 3);
         REGEXP_REPLACE
-----------------------------------
 hesomething, something, and wise
(1 row)
```

Replace the second group of word characters ending with thy with the string something .

```
=> SELECT REGEXP_REPLACE('healthy, wealthy, and wise','\w+thy', 'something', 1, 2);
       REGEXP_REPLACE
-----------------------------
 healthy, something, and wise
(1 row)
```

Find groups of word characters ending with thy capturing the letters before the thy , and replace with the captured letters plus the letters ish .

```
=> SELECT REGEXP_REPLACE('healthy, wealthy, and wise','(\w+)thy', '\1ish');
      REGEXP_REPLACE
---------------------------
 healish, wealish, and wise
(1 row)
```

Create a table to demonstrate replacing strings in a query.

```
=> CREATE TABLE customers (name varchar(50), phone varchar(11));
CREATE TABLE
=> CREATE PROJECTION customers1 AS SELECT * FROM customers;
CREATE PROJECTION
=> COPY customers FROM stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> Able, Adam|17815551234
>> Baker,Bob|18005551111
>> Chu,Cindy|16175559876
>> Dodd,Dinara|15083452121
>> \.
```

Query the customers, using REGEXP_REPLACE to format phone numbers.

```
=> SELECT name, REGEXP_REPLACE(phone, '(\d)(\d{3})(\d{3})(\d{4})',
'\1-(\2) \3-\4') as phone FROM customers;
    name     |      phone
-------------+------------------
 Able, Adam  | 1-(781) 555-1234
 Baker,Bob   | 1-(800) 555-1111
 Chu,Cindy   | 1-(617) 555-9876
 Dodd,Dinara | 1-(508) 345-2121
(4 rows)
```

REGEXP_SUBSTR

Returns the substring that matches a regular expression within a string. If no matches are found, REGEXP_SUBSTR returns NULL. This is different from an empty string, which the function can return if the regular expression matches a zero-length string.

This function operates on UTF-8 strings using the default locale, even if the locale is set otherwise.

Important
If you port a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while Vertica does not.

Syntax

```
REGEXP_SUBSTR ( string-expression, pattern
  [, position [,  occurrence [, regexp-modifier [, captured-subexp ]]... ]] )
```

Parameters

string-expression

The VARCHAR or LONG VARCHAR expression to evaluate for matches with the regular expression specified in *pattern* . If *string-expression* is in the __raw__ column of a flex or columnar table, cast the string to a LONG VARCHAR before searching for *pattern* .

pattern

The regular expression to match against *string-expression* . The regular expression must conform with Perl regular expression syntax .

position

The number of characters from the start of the string where the function should start searching for matches. By default, the function begins searching for a match at the first (leftmost) character. Setting this parameter to a value greater than 1 begins searching for a match at the *n* -th character you specify.

Default: 1

occurrence

Controls which occurrence of a pattern match in the string to return. By default, the function returns the first matching substring. For example, setting this parameter to 3 returns the third matching instance.

Default: 1

regexp-modifier

One or more single-character flags that modify how the regular expression *pattern* is matched to *string-expression* :

- b : Treat strings as binary octets, rather than UTF-8 characters.
- c (default): Force the match to be case sensitive.
- i : Force the match to be case insensitive.
- m : Treat the string to match as multiple lines. Using this modifier, the start of line (^) and end of line ($) regular expression operators match line breaks (\n) within the string. Without the m modifier, the start and end of line operators match only the start and end of the string.
- n : Match the regular expression operator (.) to a newline (\n). By default, the . operator matches any character except a newline.
- x : Add comments to regular expressions. The x modifier causes the function to ignore all un-escaped space characters and comments in the regular expression. Comments start with hash (#) and end with a newline (\n). All spaces in the regular expression to be matched in strings must be escaped with a backslash (\ ).

captured-subexp

The group to return. By default, the function returns all matching groups. For example, setting this value to 3 returns the substring captured by the third set of parentheses in the regular expression.

Default: 0

> **Note**
> The subexpressions are numbered left to right, based on the appearance of opening parenthesis, so nested regular expressions . For example, in the regular expression \s*(\w+\s+(\w+)) , subexpression 1 is the one that captures everything but any leading whitespaces.

Examples

Select the first substring of letters that end with thy .

```
=> SELECT REGEXP_SUBSTR('healthy, wealthy, and wise','\w+thy');
 REGEXP_SUBSTR
---------------
 healthy
(1 row)
```

Select the first substring of letters that ends with thy starting at the second character in the string.

```
=> SELECT REGEXP_SUBSTR('healthy, wealthy, and wise','\w+thy',2);
 REGEXP_SUBSTR
---------------
 ealthy
(1 row)
```

Select the second substring of letters that ends with thy .

```
=> SELECT REGEXP_SUBSTR('healthy, wealthy, and wise','\w+thy',1,2);
 REGEXP_SUBSTR
---------------
 wealthy
(1 row)
```

Return the contents of the third captured subexpression, which captures the third word in the string.

```
=> SELECT REGEXP_SUBSTR('one two three', '(\w+)\s+(\w+)\s+(\w+)', 1, 1, '', 3);
 REGEXP_SUBSTR
---------------
 three
(1 row)
```

Text search functions

This section contains text search functions specific to Vertica.

In this section

- DELETE_TOKENIZER_CONFIG_FILE
- GET_TOKENIZER_PARAMETER
- READ_CONFIG_FILE
- SET_TOKENIZER_PARAMETER

DELETE_TOKENIZER_CONFIG_FILE

Deletes a tokenizer configuration file.

Syntax

```
SELECT v_txtindex.DELETE_TOKENIZER_CONFIG_FILE (USING PARAMETERS proc_oid='proc_oid', confirm={true | false });
```

Parameters

confirm = [true | false]

Boolean flag. Indicates that the configuration file should be removed even if the tokenizer is still in use.

True — Force deletion of the tokenizer when the used parameter value is True.
False — Delete tokenizer if the used parameter value is False.

Default: False

proc_oid

A unique identifier assigned to a tokenizer when it is created. Users must query the system table vs_procedures to get the proc_oid for a given tokenizer name. See Configuring a tokenizer for more information.

Examples

The following example shows how you can use DELETE_TOKENIZER_CONFIG_FILE to delete the tokenizer configuration file:

```
=> SELECT v_txtindex.DELETE_TOKENIZER_CONFIG_FILE (USING PARAMETERS proc_oid='45035996274126984');
 DELETE_TOKENIZER_CONFIG_FILE
------------------------------
 t
(1 row)
```

GET_TOKENIZER_PARAMETER

Returns the configuration parameter for a given tokenizer.

Syntax

```
SELECT v_txtindex.GET_TOKENIZER_PARAMETER(parameter_name USING PARAMETERS proc_oid='proc_oid');
```

Parameters

parameter_name

Name of the parameter to be returned.

One of the following:

- stopWordsCaseInsensitive
- minorSeparators
- majorSeparators
- minLength
- maxLength
- ngramsSize
- used

proc_oid

A unique identifier assigned to a tokenizer when it is created. Users must query the system table vs_procedures to get the proc_oid for a given tokenizer name. See Configuring a tokenizer for more information.

Examples

The following examples show how you can use GET_TOKENIZER_PARAMETER.

Return the stop words used in a tokenizer:

```
=> SELECT v_txtindex.GET_TOKENIZER_PARAMETER('stopwordscaseinsensitive' USING PARAMETERS proc_oid='45035996274126984');
 getTokenizerParameter
-----------------------
 devil,TODAY,the,fox
(1 row)
```

Return the major separators used in a tokenizer:

```
=> SELECT v_txtindex.GET_TOKENIZER_PARAMETER('majorseparators' USING PARAMETERS proc_oid='45035996274126984');
 getTokenizerParameter
-----------------------
 {}()&[]
(1 row)
```

READ_CONFIG_FILE

Reads and returns the key-value pairs of all the parameters of a given tokenizer.

You must use the OVER() clause with this function.

Syntax

```
SELECT v_txtindex.READ_CONFIG_FILE(USING PARAMETERS proc_oid='proc_oid') OVER ()
```

Parameters

proc_oid

A unique identifier assigned to a tokenizer when it is created. Users must query the system table vs_procedures to get the proc_oid for a given tokenizer name. See Configuring a tokenizer for more information.

Examples

The following example shows how you can use READ_CONFIG_FILE to return the parameters associated with a tokenizer:

```
=> SELECT v_txtindex.READ_CONFIG_FILE(USING PARAMETERS proc_oid='45035996274126984') OVER();
             config_key | config_value
 -------------------------+---------------------
  majorseparators         | {}()&[]
  stopwordscaseinsensitive | devil,TODAY,the,fox
(2 rows)
```

SET_TOKENIZER_PARAMETER

Configures the tokenizer parameters.

Important

\n, \t,\r must be entered as Unicode using Vertica notation, U&'\000D' , or using Vertica escaping notation, E'\r' . Otherwise, they are taken literally as two separate characters. For example, "\" & "r" .

Syntax

```
SELECT v_txtindex.SET_TOKENIZER_PARAMETER (parameter_name, parameter_value USING PARAMETERS proc_oid='proc_oid')
```

Parameters

parameter_name

Name of the parameter to be configured.

Use one of the following:

- stopwordsCaseInsensitive : List of stop words. All the tokens that belong to the list are ignored. Vertica supports separators and stop words up to the first 256 Unicode characters.
  If you want to define a stop word that contains a comma or a backslash, then it needs to be escaped.
  For example: "Dear Jack\," "Dear Jack\\"
  Default: '' (empty list)
- majorSeparators :List of major separators. Enclose in quotes with no spaces between.
  Default: E' []<>(){}|!;,'"*&?+\r\n\t'
- minorSeparators : List of minor separators. Enclose in quotes with no spaces between.
  Default: E'/:=@.-$#%_'
- minLength — Minimum length a token can have, type Integer. Must be greater than 0.
  Default: '2'
- maxLength : Maximum length a token can be. Type Integer. Cannot be greater than 1024 bytes. For information about increasing the token size, see Text search parameters .
  Default: '128'
- ngramsSize : Integer value greater than zero. Use only with ngram tokenizers.
  Default: '3'
- used : Indicates when a tokenizer configuration cannot be changed. Type Boolean. After you set used to True , any calls to setTokenizerParameter fail.
  You must set the parameter used to True before using the configured tokenizer. Doing so prevents the configuration from being modified after being used to create a text index.
  Default: False

parameter_value

The value of a configuration parameter.

If you want to disable minorSeperators or stopWordsCaseInsensitive, then set their values to '' .

proc_oid

A unique identifier assigned to a tokenizer when it is created. Users must query the system table vs_procedures to get the proc_oid for a given tokenizer name. See Configuring a tokenizer for more information.

Examples

The following examples show how you can use SET_TOKENIZER_PARAMETER to configure stop words and separators.

Configure the stop words of a tokenizer:

```
=> SELECT v_txtindex.SET_TOKENIZER_PARAMETER('stopwordsCaseInsensitive', 'devil,TODAY,the,fox' USING PARAMETERS
proc_oid='45035996274126984');
 SET_TOKENIZER_PARAMETER
-------------------------
 t
(1 row)
```

Configure the major separators of a tokenizer:

```
=> SELECT v_txtindex.SET_TOKENIZER_PARAMETER('majorSeparators',E'{}()&[]' USING PARAMETERS proc_oid='45035996274126984');
 SET_TOKENIZER_PARAMETER
-------------------------
 t
(1 row)
```

Mathematical functions

Some of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with DOUBLE PRECISION data could vary in accuracy and behavior in boundary cases depending on the host system.

In this section

- ABS
- ACOS
- ACOSH
- ASIN
- ASINH
- ATAN
- ATAN2
- ATANH
- CBRT
- CEILING
- COS
- COSH
- COT
- DEGREES
- DISTANCE
- DISTANCEV
- EXP
- FLOOR
- HASH
- LN
- LOG
- LOG10
- MOD
- PI
- POWER
- RADIANS
- RANDOM
- RANDOMINT
- RANDOMINT_CRYPTO
- ROUND
- SIGN
- SIN
- SINH
- SQRT
- TAN
- TANH
- TRUNC
- WIDTH_BUCKET

ABS

Returns the absolute value of the argument. The return value has the same data type as the argument..

Behavior type

Immutable

Syntax

```
ABS ( expression )
```

Arguments

expression

Resolves to a value of type INTEGER or DOUBLE PRECISION.

Examples

```
SELECT ABS(-28.7);
 abs
------
 28.7
(1 row)
```

ACOS

Returns a DOUBLE PRECISION value representing the trigonometric inverse cosine of the argument.

Behavior type

Immutable

Syntax

```
ACOS ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
SELECT ACOS (1);
 acos
------
    0
(1 row)
```

ACOSH

Returns a DOUBLE PRECISION value that represents the inverse (arc) hyperbolic cosine of the function argument.

Behavior type

Immutable

Syntax

```
ACOSH ( expression )
```

Arguments

expression

Resolves to a value of type INTEGER or DOUBLE PRECISION ≥ 1.0, otherwise returns NaN.

Examples

```
=> SELECT acosh(4);
      acosh
------------------
 2.06343706889556
(1 row)
```

ASIN

Returns a DOUBLE PRECISION value representing the trigonometric inverse sine of the argument.

Behavior type

Immutable

Syntax

```
ASIN ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
SELECT ASIN(1);
      asin
-----------------
 1.5707963267949
(1 row)
```

ASINH

Returns a DOUBLE PRECISION value that represents the inverse (arc) hyperbolic sine of the function argument.

Behavior type

Immutable

Syntax

```
ASINH ( expression )
```

Arguments

expression

Resolves to a value of type INTEGER or DOUBLE PRECISION.

Examples

```
=> SELECT asinh(2.85);
      asinh
------------------
 1.76991385902105
(1 row)
```

ATAN

Returns a DOUBLE PRECISION value representing the trigonometric inverse tangent of the argument.

Behavior type

Immutable

Syntax

```
ATAN ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
SELECT ATAN(1);
       atan
-------------------
 0.785398163397448
(1 row)
```

ATAN2

Returns a DOUBLE PRECISION value representing the trigonometric inverse tangent of the arithmetic dividend of the arguments.

Behavior type

Immutable

Syntax

```
ATAN2 ( quotient, divisor )
```

Arguments

quotient

Resolves to a value of type DOUBLE PRECISION representing the quotient.

divisor

Resolves to a value of type DOUBLE PRECISION representing the divisor.

Examples

```
SELECT ATAN2(2,1);
     ATAN2
------------------
 1.10714871779409
(1 row)
```

ATANH

Returns a DOUBLE PRECISION value that represents the inverse hyperbolic tangent of the function argument.

Behavior type

Immutable

Syntax

```
ATANH ( expression )
```

Arguments

expression

Resolves to a value of type INTEGER or DOUBLE PRECISION between -1.0 and +1.0, inclusive, otherwise returns NaN.

Examples

```
=> SELECT atanh(-0.875);
      atanh
--------------------
 -1.35402510055111
(1 row)
```

CBRT

Returns the cube root of the argument. The return value has the type DOUBLE PRECISION.

Behavior type

Immutable

Syntax

```
CBRT ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
SELECT CBRT(27.0);
 cbrt
------
    3
(1 row)
```

CEILING

Rounds up the returned value up to the next whole number. For example, given arguments of 5.01 and 5.99, CEILING returns 6. CEILING is the opposite of FLOOR , which rounds down the returned value.

Behavior type

Immutable

Syntax

```
CEIL[ING] ( expression )
```

Arguments

expression

Resolves to an INTEGER or DOUBLE PRECISION value.

Examples

```
=> SELECT CEIL(-42.8);
 CEIL
------
  -42
(1 row)
SELECT CEIL(48.01);
 CEIL
------
   49
(1 row)
```

COS

Returns a DOUBLE PRECISION value tat represents the trigonometric cosine of the passed parameter.

Behavior type

Immutable

Syntax

```
COS ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
SELECT COS(-1);
       COS
------------------
 0.54030230586814
(1 row)
```

COSH

Returns a DOUBLE PRECISION value that represents the hyperbolic cosine of the passed parameter.

Behavior type

Immutable

Syntax

```
COSH ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
=> SELECT COSH(-1);
       COSH
------------------
 1.54308063481524
```

COT

Returns a DOUBLE PRECISION value representing the trigonometric cotangent of the argument.

Behavior type

Immutable

Syntax

```
COT ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
SELECT COT(1);
       cot
-------------------
 0.642092615934331
(1 row)
```

DEGREES

Converts an expression from radians to fractional degrees, or from degrees, minutes, and seconds to fractional degrees. The return value has the type DOUBLE PRECISION.

Behavior type

Immutable

Syntax

```
DEGREES ( { radians | degrees, minutes, seconds } )
```

Arguments

radians

Unit of angular measure. 2* π * radians is equal to a full rotation.

degrees

Unit of angular measure, equal to 1/360 of a full rotation.

minutes

Unit of angular measurement, representing 1/60 of a degree.

seconds

Unit of angular measurement, representing 1/60 of a minute.

Examples

```
SELECT DEGREES(0.5);
    DEGREES
------------------
 28.6478897565412
(1 row)

SELECT DEGREES(1,2,3);
    DEGREES
------------------
 1.03416666666667
(1 row)
```

DISTANCE

Returns the distance (in kilometers) between two points. You specify the latitude and longitude of the starting point and the ending point. You can also specify the radius of curvature for greater accuracy when using an ellipsoidal model.

Behavior type

Immutable

Syntax

```
DISTANCE ( lat0, lon0, lat1, lon1 [, radius-of-curvature ] )
```

Arguments

lat0

Starting point latitude.

lon0

Starting point longitude.

lat1

Ending point latitude

lon1

Ending point longitude.

radius-of-curvature

Radius of the earth's curvature at the midpoint between the starting and ending points. This argument allows for greater accuracy when using an ellipsoidal earth model. If you omit this argument, DISTANCE uses the WGS-84 average r1 radius, about 6371.009 km.

Examples

This example finds the distance in kilometers for 1 degree of longitude at latitude 45 degrees, assuming earth is spherical.

```
SELECT DISTANCE(45,0,45,1);
    DISTANCE
----------------------
 78.6262959272162
(1 row)
```

DISTANCEV

Returns the distance (in kilometers) between two points using the Vincenty formula. Because the Vincenty formula includes the parameters of the WGS-84 ellipsoid model, you need not specify a radius of curvature. You specify the latitude and longitude of both the starting point and the ending point. This function is more accurate, but will be slower, than the DISTANCE function.

Behavior type

Immutable

Syntax

```
DISTANCEV (lat0, lon0, lat1, lon1);
```

Arguments

lat0

Specifies the latitude of the starting point.

lon0

Specifies the longitude of the starting point.

lat1

Specifies the latitude of the ending point.

lon1

Specifies the longitude of the ending point.

Examples

This example finds the distance in kilometers for 1 degree of longitude at latitude 45 degrees, assuming earth is ellipsoidal.

```
SELECT DISTANCEV(45,0, 45,1);
   distanceV
------------------
 78.8463347095916
(1 row)
```

EXP

Returns the exponential function, e to the power of a number. The return value has the same data type as the argument.

Behavior type

Immutable

Syntax

```
EXP ( exponent )
```

Arguments

exponent

Resolves to a value of type INTEGER or DOUBLE PRECISION.

Examples

```
SELECT EXP(1.0);
       exp
------------------
 2.71828182845905
(1 row)
```

FLOOR

Rounds down the returned value to the previous whole number. For example, given arguments of 5.01 and 5.99, FLOOR returns 5. FLOOR is the opposite of CEILING , which rounds up the returned value.

Behavior type

Immutable

Syntax

```
FLOOR ( expression )
```

Arguments

expression

Resolves to an INTEGER or DOUBLE PRECISION value.

Examples

```
=> SELECT FLOOR((TIMESTAMP '2005-01-17 10:00' - TIMESTAMP '2005-01-01') / INTERVAL '7');
 FLOOR
-------
     2
(1 row)

=> SELECT FLOOR(-42.8);
 FLOOR
-------
   -43
(1 row)

=> SELECT FLOOR(42.8);
 FLOOR
-------
    42
(1 row)
```

Although the following example looks like an INTEGER, the number on the left is 2^49 as an INTEGER, while the number on the right is a FLOAT:

```
=> SELECT 1<<49, FLOOR(1 << 49);
    ?column?     |      floor
-----------------+-----------------
 562949953421312 | 562949953421312
(1 row)
```

Compare the previous example to:

```
=> SELECT 1<<50, FLOOR(1 << 50);
     ?column?     |        floor
------------------+----------------------
 1125899906842624 | 1.12589990684262e+15
(1 row)
```

HASH

Calculates a hash value over the function arguments, producing a value in the range $0 <= x < 2^{63}$.

The HASH function is typically used to segment a projection over a set of cluster nodes. The function selects a specific node for each row based on the values of the row columns. The HASH function distributes data evenly across the cluster, which facilitates optimal query execution.

Behavior type

Immutable

Syntax

```
HASH ( { * | expression[,...] } )
```

Arguments

*** | *expression* [,...]**

One of the following:

- * (asterisk)
  Specifies to hash all columns in the queried table.
- *expression*
  An expression of any data type. Functions that are included in *expression* must be deterministic. If specified in a projection's hash segmentation clause , each expression typically resolves to a column reference .

Examples

```
=> SELECT HASH(product_price, product_cost) FROM product_dimension
   WHERE product_price = '11';
      hash
---------------------
 4157497907121511878
 1799398249227328285
 3250220637492749639
(3 rows)
```

See also

Hash segmentation clause

LN

Returns the natural logarithm of the argument. The return data type is the same as the argument.

Behavior type

Immutable

Syntax

```
LN ( expression )
```

Arguments

expression

Resolves to a value of type INTEGER or DOUBLE PRECISION.

Examples

```
SELECT LN(2);
        ln
-------------------
 0.693147180559945
(1 row)
```

LOG

Returns the logarithm to the specified base of the argument. The data type of the return value is the same data type as the passed parameter.

Behavior type

Immutable

Syntax

```
LOG ( [ base, ] expression )
```

Arguments

base

Specifies the base (default is base 10)

expression

Resolves to a value of type INTEGER or DOUBLE PRECISION.

Examples

```
=> SELECT LOG(2.0, 64);
 LOG
-----
   6
(1 row)
SELECT LOG(100);
 LOG
-----
   2
(1 row)
```

LOG10

Returns the base 10 logarithm of the argument, also known as the *common logarithm* . The data type of the return value is the same as the data type of the passed parameter.

Behavior type

Immutable

Syntax

```
LOG10 ( expression )
```

Arguments

expression

Resolves to a value of type INTEGER or DOUBLE PRECISION.

Examples

```
=> SELECT LOG10(30);
      LOG10
------------------
 1.47712125471966
(1 row)
```

MOD

Returns the remainder of a division operation.

Behavior type

Immutable

Syntax

```
MOD( expression1, expression2 )
```

Arguments

expression1

Resolves to a numeric data type that specifies the dividend.

expression2

Resolves to a numeric data type that specifies the divisor.

Computation rules

When computing MOD(*expression1* , *expression2*), the following rules apply:

- If either *expression1* or *expression2* is the null value, then the result is the null value.
- If *expression2* is zero, then an exception condition is raised: data exception — division by zero.
- Otherwise, the result is the unique exact numeric value *R* with scale 0 (zero) such that all of the following are true:
  - *R* has the same sign as *expression2* .
  - The absolute value of *R* is less than the absolute value of *expression1* .
  - *expression2* = *expression1* * *K* + *R* for some exact numeric value *K* with scale 0 (zero).

Examples

```
SELECT MOD(9,4);
 mod
-----
   1
(1 row)

SELECT MOD(10,3);
 mod
-----
   1
(1 row)

SELECT MOD(-10,3);
 mod
-----
  -1
(1 row)

SELECT MOD(-10,-3);
 mod
-----
  -1
(1 row)

SELECT MOD(10,-3);
 mod
-----
   1
(1 row)

=> SELECT MOD(6.2,0);
ERROR 3117:  Division by zero
```

PI

Returns the constant pi (P), the ratio of any circle's circumference to its diameter in Euclidean geometry The return type is DOUBLE PRECISION.

Behavior type

Immutable

Syntax

```
PI()
```

Examples

```
SELECT PI();
        pi
-------------------
 3.14159265358979
(1 row)
```

POWER

Returns a DOUBLE PRECISION value representing one number raised to the power of another number.

Behavior type

Immutable

Syntax

```
POW[ER] ( expression1, expression2 )
```

Arguments

expression1

Resolves to a DOUBLE PRECISION value that represents the base.

expression2

Resolves to a DOUBLE PRECISION value that represents the exponent.

Examples

```
SELECT POWER(9.0, 3.0);
 power
-------
   729
(1 row)
```

RADIANS

Returns a DOUBLE PRECISION value representing an angle expressed in radians. You can express the input angle in DEGREES , and optionally include minutes and seconds.

Behavior type

Immutable

Syntax

```
RADIANS (degrees [, minutes, seconds])
```

Arguments

degrees

Unit of angular measurement, representing 1/360 of a full rotation.

minutes

Unit of angular measurement, representing 1/60 of a degree.

seconds

Unit of angular measurement, representing 1/60 of a minute.

Examples

```
SELECT RADIANS(45);
     RADIANS
--------------------
 0.785398163397448
(1 row)

SELECT RADIANS (1,2,3);
     RADIANS
--------------------
 0.018049613347708
(1 row)
```

RANDOM

Returns a uniformly-distributed random DOUBLE PRECISION value x , where $0 <= x < 1$.

Typical pseudo-random generators accept a seed, which is set to generate a reproducible pseudo-random sequence. Vertica, however, distributes SQL processing over a cluster of nodes, where each node generates its own independent random sequence.

Results depending on RANDOM are not reproducible because the work might be divided differently across nodes. Therefore, Vertica automatically generates truly random seeds for each node each time a request is executed and does not provide a mechanism for forcing a specific seed.

Behavior type

Volatile

Syntax

```
RANDOM()
```

Examples

In the following example, RANDOM returns a float ≥ 0 and < 1.0:

```
SELECT RANDOM();
     random
-------------------
 0.211625560652465
(1 row)
```

RANDOMINT

Accepts and returns an integer between 0 and the integer argument *expression* -1.

Typical pseudo-random generators accept a seed, which is set to generate a reproducible pseudo-random sequence. Vertica, however, distributes SQL processing over a cluster of nodes, where each node generates its own independent random sequence.

Results depending on RANDOM are not reproducible because the work might be divided differently across nodes. Therefore, Vertica automatically generates truly random seeds for each node each time a request is executed and does not provide a mechanism for forcing a specific seed.

Behavior type

Volatile

Syntax

```
RANDOMINT ( expression )
```

Arguments

expression

Resolves to a positive INTEGER between 1 and $2^{63} - 1$, inclusive. If you supply a negative value or *expression* > 1, Vertica returns an error.

Examples

In the following example, the result is an INTEGER ≥ 0 and < *expression* , randomly chosen from the set {0,1,2,3,4}.

```
=> SELECT RANDOMINT(5);
RANDOMINT
----------
       3
(1 row)
```

RANDOMINT_CRYPTO

Accepts and returns an INTEGER value from a set of values between 0 and the specified function argument -1. For this cryptographic random number generator, Vertica uses RAND_bytes to provide the random value.

Behavior type

Volatile

Syntax

```
RANDOMINT_CRYPTO ( expression )
```

Arguments

expression

Resolves to a positive integer between 1 and $2^{63} - 1$, inclusive.

Examples

In the following example, RANDOMINT_CRYPTO returns an INTEGER >= 0 and less than the specified argument 5 , randomly chosen from the set {0,1,2,3,4} .

```
=> SELECT RANDOMINT_crypto(5);
RANDOMINT_crypto
-----------------
               3
(1 row)
```

ROUND

Rounds a value to a specified number of decimal places, retaining the original precision and scale. Fractions greater than or equal to .5 are rounded up. Fractions less than .5 are rounded down (truncated).

Behavior type

Immutable

Syntax

```
ROUND ( expression [, places ] )
```

Arguments

expression

Resolves to a value of type NUMERIC or DOUBLE PRECISION (FLOAT) .

places

An INTEGER value. When *places* is a positive integer, Vertica rounds the value to the right of the decimal point using the specified number of places. When *places* is a negative integer, Vertica rounds the value on the left side of the decimal point using the specified number of places.

Notes

Using ROUND with a NUMERIC datatype returns NUMERIC , retaining the original precision and scale.

```
=> SELECT ROUND(3.5);
 ROUND
-------
   4.0
(1 row)
```

Examples

```
=> SELECT ROUND(2.0, 1.0) FROM dual;
 ROUND
-------
   2.0
(1 row)
=> SELECT ROUND(12.345, 2.0);
 ROUND
--------
 12.350
(1 row)
=> SELECT ROUND(3.444444444444444);
       ROUND
--------------------
 3.000000000000000
(1 row)
=> SELECT ROUND(3.14159, 3);
  ROUND
---------
 3.14200
(1 row)
=> SELECT ROUND(1234567, -3);
  ROUND
---------
 1235000
(1 row)
=> SELECT ROUND(3.4999, -1);
 ROUND
--------
 0.0000
(1 row)
```

The following example creates a table with two columns, adds one row of values, and shows sample rounding to the left and right of a decimal point.

```
=> CREATE TABLE sampleround (roundcol1 NUMERIC, roundcol2 NUMERIC);
CREATE TABLE

=> INSERT INTO sampleround VALUES (1234567, .1234567);
 OUTPUT
--------
      1
(1 row)

=> SELECT ROUND(roundcol1,-3) AS pn3, ROUND(roundcol1,-4) AS pn4, ROUND(roundcol1,-5) AS pn5 FROM sampleround;

          pn3          |          pn4          |          pn5
-----------------------+-----------------------+-----------------------
 1235000.000000000000000 | 1230000.000000000000000 | 1200000.000000000000000
(1 row)

=> SELECT ROUND(roundcol2,3) AS p3, ROUND(roundcol2,4) AS p4, ROUND(roundcol2,5) AS p5 FROM sampleround;

       p3        |       p4        |       p5
-----------------+-----------------+-----------------
 0.123000000000000 | 0.123500000000000 | 0.123460000000000
(1 row)
```

SIGN

Returns a DOUBLE PRECISION value of -1, 0, or 1 representing the arithmetic sign of the argument.

Behavior type

Immutable

Syntax

```
SIGN ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
SELECT SIGN(-8.4);
 sign
------
   -1
(1 row)
```

SIN

Returns a DOUBLE PRECISION value that represents the trigonometric sine of the passed parameter.

Behavior type

Immutable

Syntax

```
SIN ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
SELECT SIN(30 * 2 * 3.14159 / 360);
       SIN
-------------------
 0.499999616987256
(1 row)
```

SINH

Returns a DOUBLE PRECISION value that represents the hyperbolic sine of the passed parameter.

Behavior type

Immutable

Syntax

```
SINH ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
=> SELECT SINH(30 * 2 * 3.14159 / 360);
      SINH
-------------------
 0.547852969600632
```

SQRT

Returns a DOUBLE PRECISION value representing the arithmetic square root of the argument.

Behavior type

Immutable

Syntax

```
SQRT ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
SELECT SQRT(2);
     sqrt
-----------------
 1.4142135623731
(1 row)
```

TAN

Returns a DOUBLE PRECISION value that represents the trigonometric tangent of the passed parameter.

Behavior type

Immutable

Syntax

```
TAN ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
=> SELECT TAN(30);
       TAN
-------------------
 -6.40533119664628
(1 row)
```

TANH

Returns a DOUBLE PRECISION value that represents the hyperbolic tangent of the passed parameter.

Behavior type

Immutable

Syntax

```
TANH ( expression )
```

Arguments

expression

Resolves to a value of type DOUBLE PRECISION.

Examples

```
=> SELECT TANH(-1);
      TANH
-------------------
 -0.761594155955765
```

TRUNC

Returns the *expression* value fully truncated (toward zero). Supplying a *places* argument truncates the expression to the number of decimal places you indicate.

Behavior type

Immutable

Syntax

```
TRUNC ( expression [, places ] )
```

Arguments

expression

Resolves to a value of type NUMERIC or DOUBLE PRECISION (FLOAT) .

places

INTEGER value:

- Positive: Vertica truncates the value to the right of the decimal point.
- Negative: Vertica truncates the value on the left side of the decimal point.

Notes

Using TRUNC with a NUMERIC datatype returns NUMERIC , retaining the original precision and scale.

```
=> SELECT TRUNC(3.5);
 TRUNC
-------
   3.0
(1 row)
```

Examples

```
=> SELECT TRUNC(42.8);
 TRUNC
-------
  42.0
(1 row)
=> SELECT TRUNC(42.4382, 2);
  TRUNC
---------
 42.4300
(1 row)
```

The following example creates a table with two columns, adds one row of values, and shows sample truncating to the left and right of a decimal point.

```
=> CREATE TABLE sampletrunc (truncol1 NUMERIC, truncol2 NUMERIC);
CREATE TABLE

=> INSERT INTO sampletrunc VALUES (1234567, .1234567);
 OUTPUT
--------
      1
(1 row)

=> SELECT TRUNC(truncol1,-3) AS p3, TRUNC(truncol1,-4) AS p4, TRUNC(truncol1,-5) AS p5 FROM sampletrunc;

          p3           |          p4           |          p5
-----------------------+-----------------------+-----------------------
 1234000.000000000000000 | 1230000.000000000000000 | 1200000.000000000000000
(1 row)

=> SELECT TRUNC(truncol2,3) AS p3, TRUNC(truncol2,4) AS p4, TRUNC(truncol2,5) AS p5 FROM sampletrunc;

       p3        |       p4        |       p5
-----------------+-----------------+-----------------
 0.123000000000000 | 0.123400000000000 | 0.123450000000000
(1 row)
```

WIDTH_BUCKET

Constructs equiwidth histograms, in which the histogram range is divided into intervals (buckets) of identical sizes. In addition, values below the low bucket return 0, and values above the high bucket return *bucket-count* +1. Returns an integer value.

Behavior type

Immutable

Syntax

```
WIDTH_BUCKET ( expression, hist-min, hist-max, bucket-count )
```

Arguments

expression

The expression for which the histogram is created. This expression must resolve to a numeric or datetime value or a value that can be implicitly converted to a numeric or datetime value. If * expression *evaluates to null, then the * expression *returns null.

hist-min

Resolves to the low boundary of *bucket-count* , a non-null numeric or datetime value.

hist-max

Resolves to the high boundary of *bucket-count* , a non-null numeric or datetime value.

bucket-count

Resolves to an INTEGER constant that indicates the number of buckets.

Notes

- WIDTH_BUCKET divides a data set into buckets of equal width. For example, Age = 0–20, 20–40, 40–60, 60–80. This is known as an equiwidth histogram.
- When using WIDTH_BUCKET pay attention to the minimum and maximum boundary values. Each bucket contains values equal to or greater than the base value of that bucket, so that age ranges of 0–20, 20–40, and so on, are actually 0–19.99 and 20–39.999.
- WIDTH_BUCKET accepts the following data types: (FLOAT and/or INTEGER), (TIMESTAMP and/or DATE and/or TIMESTAMPTZ), or (INTERVAL and/or TIME).

Examples

The following example returns five possible values and has three buckets: 0 [Up to 100), 1 [100–300), 2 [300–500), 3 [500–700), and 4 [700 and up):

```
SELECT product_description, product_cost, WIDTH_BUCKET(product_cost, 100, 700, 3);
```

The following example creates a nine-bucket histogram on the annual_income column for customers in Connecticut who are female doctors. The results return the bucket number to an Income column, divided into eleven buckets, including an underflow and an overflow. Note that if customers had annual incomes greater than the maximum value, they would be assigned to an overflow bucket, 10:

```
SELECT customer_name, annual_income, WIDTH_BUCKET (annual_income, 100000, 1000000, 9) AS "Income"
FROM public.customer_dimension WHERE customer_state='CT'
AND title='Dr.' AND customer_gender='Female' AND household_id < '1000'
ORDER BY "Income";
```

In the following result set, the reason there is a bucket 0 is because buckets are numbered from 1 to bucket_count . Anything less than the given value of hist_min goes in bucket 0, and anything greater than the given value of hist_max goes in the bucket bucket_count+1 . In this example, bucket 9 is empty, and there is no overflow. The value 12,283 is less than 100,000, so it goes into the underflow bucket.

```
  customer_name    | annual_income | Income
-------------------+---------------+--------
 Joanna A. Nguyen  |         12283 |      0
 Amy I. Nguyen     |        109806 |      1
 Juanita L. Taylor |        219002 |      2
 Carla E. Brown    |        240872 |      2
 Kim U. Overstreet |        284011 |      2
 Tiffany N. Reyes  |        323213 |      3
 Rebecca V. Martin |        324493 |      3
 Betty . Roy       |        476055 |      4
 Midori B. Young   |        462587 |      4
 Martha T. Brown   |        687810 |      6
 Julie D. Miller   |        616509 |      6
 Julie Y. Nielson  |        894910 |      8
 Sarah B. Weaver   |        896260 |      8
 Jessica C. Nielson |       861066 |      8
(14 rows)
```

See also

- NTILE [analytic]

NULL-handling functions

NULL-handling functions take arguments of any type, and their return type is based on their argument types.

In this section

- COALESCE
- IFNULL
- ISNULL
- NULLIF
- NULLIFZERO
- NVL
- NVL2
- ZEROIFNULL

COALESCE

Returns the value of the first non-null expression in the list. If all expressions evaluate to null, then COALESCE returns null.

COALESCE conforms to the ANSI SQL-92 standard.

Behavior type

Immutable

Syntax

```
COALESCE ( { * | expression[,...] } )
```

Arguments

*** | *expression* [,...]**

One of the following:

- * (asterisk)
  Evaluates all columns in the queried table.
- *expression*
  An expression of any data type. Functions that are included in *expression* must be deterministic.

Examples

COALESCE returns the first non-null value in each row that is queried from table lead_vocalists . Note that in the first row, COALESCE returns an empty string.

```
=> SELECT quote_nullable(fname)fname, quote_nullable(lname)lname,
    quote_nullable(coalesce (fname, lname)) "1st non-null value" FROM lead_vocalists ORDER BY fname;
 fname |   lname   | 1st non-null value
---------+-----------+--------------------
 ''      | 'Sting'   | ''
 'Diana' | 'Ross'    | 'Diana'
 'Grace' | 'Slick'   | 'Grace'
 'Mick'  | 'Jagger'  | 'Mick'
 'Steve' | 'Winwood' | 'Steve'
 NULL    | 'Cher'    | 'Cher'
(6 rows)
```

See also

- CASE expressions
- ISNULL

IFNULL

Returns the value of the first non-null expression in the list.

IFNULL is an alias of NVL .

Behavior type

Immutable

Syntax

```
IFNULL ( expression1 , expression2 );
```

Parameters

- If * expression1 *is null, then IFNULL returns *expression2.*
- If * expression1 *is not null, then IFNULL returns *expression1.*

Notes

- COALESCE is the more standard, more general function.
- IFNULL is equivalent to ISNULL.
- IFNULL is equivalent to COALESCE except that IFNULL is called with only two arguments.
- ISNULL(a,b) is different from x IS NULL .
- The arguments can have any data type supported by Vertica.
- Implementation is equivalent to the CASE expression. For example:

  ```
  CASE WHEN expression1 IS NULL THEN expression2
  ELSE expression1 END;
  ```

- The following statement returns the value 140:

  ```
  SELECT IFNULL(NULL, 140) FROM employee_dimension;
  ```

- The following statement returns the value 60:

  ```
  SELECT IFNULL(60, 90) FROM employee_dimension;
  ```

Examples

```
=> SELECT IFNULL (SCORE, 0.0) FROM TESTING;
 IFNULL
--------
  100.0
   87.0
     .0
     .0
     .0
(5 rows)
```

See also

- CASE expressions
- COALESCE
- NVL
- ISNULL

ISNULL

Returns the value of the first non-null expression in the list.

ISNULL is an alias of NVL .

Behavior type

Immutable

Syntax

```
ISNULL ( expression1 , expression2 );
```

Parameters

- If * expression1 *is null, then ISNULL returns *expression2.*
- If * expression1 *is not null, then ISNULL returns *expression1.*

Notes

- COALESCE is the more standard, more general function.
- ISNULL is equivalent to COALESCE except that ISNULL is called with only two arguments.
- ISNULL(a,b) is different from x IS NULL .
- The arguments can have any data type supported by Vertica.
- Implementation is equivalent to the CASE expression. For example:

```
CASE WHEN expression1 IS NULL THEN expression2
ELSE expression1 END;
```

- The following statement returns the value 140:

```
SELECT ISNULL(NULL, 140) FROM employee_dimension;
```

- The following statement returns the value 60:

```
SELECT ISNULL(60, 90) FROM employee_dimension;
```

Examples

```
SELECT product_description, product_price,
ISNULL(product_cost, 0.0) AS cost
FROM product_dimension;
     product_description       | product_price | cost
-------------------------------+---------------+------
 Brand #59957 wheat bread      |           405 |  207
 Brand #59052 blueberry muffins |          211 |  140
 Brand #59004 english muffins  |           399 |  240
 Brand #53222 wheat bread      |           323 |   94
 Brand #52951 croissants       |           367 |  121
 Brand #50658 croissants       |           100 |   94
 Brand #49398 white bread      |           318 |   25
 Brand #46099 wheat bread      |           242 |    3
 Brand #45283 wheat bread      |           111 |  105
 Brand #43503 jelly donuts     |           259 |   19
(10 rows)
```

See also

- CASE expressions
- COALESCE
- NVL

NULLIF

Compares two expressions. If the expressions are not equal, the function returns the first expression (expression1). If the expressions are equal, the function returns null.

Behavior type

Immutable

Syntax

```
NULLIF( expression1, expression2 )
```

Parameters

expression1

Is a value of any data type.

expression2

Must have the same data type as * expr1 *or a type that can be implicitly cast to match *expression1* . The result has the same type as *expression1* .

Examples

The following series of statements illustrates one simple use of the NULLIF function.

Creates a single-column table t and insert some values :

```
CREATE TABLE t (x TIMESTAMPTZ);
INSERT INTO t VALUES('2009-09-04 09:14:00-04');
INSERT INTO t VALUES('2010-09-04 09:14:00-04');
```

Issue a select statement:

```
SELECT x, NULLIF(x, '2009-09-04 09:14:00 EDT') FROM t;
          x           |         nullif
------------------------+------------------------
 2009-09-04 09:14:00-04 |
 2010-09-04 09:14:00-04 | 2010-09-04 09:14:00-04
SELECT NULLIF(1, 2);
 NULLIF
--------
      1
(1 row)
SELECT NULLIF(1, 1);
 NULLIF
--------
(1 row)
SELECT NULLIF(20.45, 50.80);
 NULLIF
--------
  20.45
(1 row)
```

NULLIFZERO

Evaluates to NULL if the value in the column is 0.

Syntax

```
NULLIFZERO(expression)
```

Parameters

expression

(INTEGER, DOUBLE PRECISION, INTERVAL, or NUMERIC) Is the string to evaluate for 0 values.

Examples

The TESTING table below shows the test scores for 5 students. Note that test scores are missing for S. Robinson and K. Johnson (NULL values appear in the Score column.)

```
=> SELECT * FROM TESTING;
   Name     | Score
------------+-------
 J. Doe     |   100
 R. Smith   |    87
 L. White   |     0
 S. Robinson |
 K. Johnson  |
(5 rows)
```

The SELECT statement below specifies that Vertica should return any 0 values in the Score column as Null. In the results, you can see that Vertica returns L. White's 0 score as Null.

```
=> SELECT Name, NULLIFZERO(Score) FROM TESTING;
   Name     | NULLIFZERO
------------+------------
 J. Doe     |        100
 R. Smith   |         87
 L. White   |
 S. Robinson |
 K. Johnson  |
(5 rows)
```

NVL

Returns the value of the first non-null expression in the list.

Behavior type

Immutable

Syntax

```
NVL ( expression1 , expression2 );
```

Parameters

- If * expression1 *is null, then NVL returns *expression2.*
- If * expression1 *is not null, then NVL returns *expression1.*

Notes

- COALESCE is the more standard, more general function.
- NVL is equivalent to COALESCE except that NVL is called with only two arguments.
- The arguments can have any data type supported by Vertica.
- Implementation is equivalent to the CASE expression:

```
CASE WHEN expression1 IS NULL THEN expression2
   ELSE expression1 END;
```

Examples

expression1 is not null, so NVL returns expression1:

```
SELECT NVL('fast', 'database');
nvl
------
 fast
(1 row)
```

expression1 is null, so NVL returns expression2:

```
SELECT NVL(null, 'database');
nvl
----------
 database
(1 row)
```

expression2 is null, so NVL returns expression1:

```
SELECT NVL('fast', null);
nvl
------
 fast
(1 row)
```

In the following example, expression1 (title) contains nulls, so NVL returns expression2 and substitutes 'Withheld' for the unknown values:

```
SELECT customer_name, NVL(title, 'Withheld') as title
FROM customer_dimension
ORDER BY title;
    customer_name     | title
------------------------+-------
 Alexander I. Lang     | Dr.
 Steve S. Harris       | Dr.
 Daniel R. King        | Dr.
 Luigi I. Sanchez      | Dr.
 Duncan U. Carcetti    | Dr.
 Meghan K. Li          | Dr.
 Laura B. Perkins      | Dr.
 Samantha V. Robinson  | Dr.
 Joseph P. Wilson      | Mr.
 Kevin R. Miller       | Mr.
 Lauren D. Nguyen      | Mrs.
 Emily E. Goldberg     | Mrs.
 Darlene K. Harris     | Ms.
 Meghan J. Farmer      | Ms.
 Bettercare            | Withheld
 Ameristar             | Withheld
 Initech               | Withheld
(17 rows)
```

See also

- CASE expressions
- COALESCE
- ISNULL
- NVL2

NVL2

Takes three arguments. If the first argument is not NULL, it returns the second argument, otherwise it returns the third argument. The data types of the second and third arguments are implicitly cast to a common type if they don't agree, similar to COALESCE .

Behavior type

Immutable

Syntax

```
NVL2 ( expression1 , expression2 , expression3 );
```

Parameters

- If *expression1* is not null, then NVL2 returns *expression2* .
- If *expression1* is null, then NVL2 returns *expression3* .

Notes

Arguments two and three can have any data type supported by Vertica.

Implementation is equivalent to the CASE expression:

```
CASE WHEN expression1 IS NOT NULL THEN expression2 ELSE expression3 END;
```

Examples

In this example, expression1 is not null, so NVL2 returns expression2:

```
SELECT NVL2('very', 'fast', 'database');
nvl2
------
 fast
(1 row)
```

In this example, expression1 is null, so NVL2 returns expression3:

```
SELECT NVL2(null, 'fast', 'database');
nvl2
----------
 database
(1 row)
```

In the following example, expression1 (title) contains nulls, so NVL2 returns expression3 ('Withheld') and also substitutes the non-null values with the expression 'Known':

```
SELECT customer_name, NVL2(title, 'Known', 'Withheld')
as title
FROM customer_dimension
ORDER BY title;
     customer_name      | title
------------------------+-------
 Alexander I. Lang      | Known
 Steve S. Harris        | Known
 Daniel R. King         | Known
 Luigi I. Sanchez       | Known
 Duncan U. Carcetti     | Known
 Meghan K. Li           | Known
 Laura B. Perkins       | Known
 Samantha V. Robinson   | Known
 Joseph P. Wilson       | Known
 Kevin R. Miller        | Known
 Lauren D. Nguyen       | Known
 Emily E. Goldberg      | Known
 Darlene K. Harris      | Known
 Meghan J. Farmer       | Known
 Bettercare             | Withheld
 Ameristar              | Withheld
 Initech                | Withheld
(17 rows)
```

See also

- CASE expressions
- COALESCE
- COALESCE

ZEROIFNULL

Evaluates to 0 if the column is NULL.

Syntax

```
ZEROIFNULL(expression)
```

Parameters

expression

String to evaluate for NULL values, one of the following data types:

- INTEGER
- DOUBLE PRECISION
- INTERVAL
- NUMERIC

Examples

The following query returns scores for five students from table test_results , where Score is set to 0 for L. White, and null for S. Robinson and K. Johnson:

```
=> SELECT Name, Score FROM test_results;
   Name     | Score
------------+-------
 J. Doe     |   100
 R. Smith   |    87
 L. White   |     0
 S. Robinson |
 K. Johnson  |
(5 rows)
```

The next query invokes ZEROIFNULL on column Score , so Vertica returns 0 for for S. Robinson and K. Johnson:

```
=> SELECT Name, ZEROIFNULL (Score) FROM test_results;
   Name     | ZEROIFNULL
------------+------------
 J. Doe     |        100
 R. Smith   |         87
 L. White   |          0
 S. Robinson |         0
 K. Johnson  |         0
(5 rows)
```

You can also use ZEROIFNULL in PARTITION BY expressions , which must always resolve to a non-null value. For example:

```
CREATE TABLE t1 (a int, b int) PARTITION BY (ZEROIFNULL(a));
CREATE TABLE
```

Vertica invokes this function when it partitions table t1 , typically during a load operation. During the load, the function checks the data of the PARTITION BY expression—in this case, column a —for null values. If encounters a null value in a given row, it sets the partition key to 0, instead of returning with an error.

Performance analysis functions

The functions in this section support profiling and analyzing database and query performance.

In this section

- Profiling functions
- Statistics management functions
- Workload management functions

Profiling functions

This section contains profiling functions specific to Vertica.

In this section

- CLEAR_PROFILING
- DISABLE_PROFILING
- ENABLE_PROFILING
- SHOW_PROFILING_CONFIG

CLEAR_PROFILING

Clears from memory data for the specified profiling type.

> **Note**
> Vertica stores profiled data in memory, so profiling can be memory intensive depending on how much data you collect.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CLEAR_PROFILING( 'profiling-type' [, 'scope'] )
```

Parameters

profiling-type

The type of profiling data to clear:

- session : Clear profiling for basic session parameters and lock time out data.
- query : Clear profiling for general information about queries that ran, such as the query strings used and the duration of queries.
- ee : Clear profiling for information about the execution run of each query.

scope

Specifies at what scope to clear profiling on the specified data, one of the following:

- local : Clear profiling data for the current session.
- global : Clear profiling data across all database sessions.

Examples

The following statement clears profiled data for queries:

```
=> SELECT CLEAR_PROFILING('query');
```

See also

- DISABLE_PROFILING
- ENABLE_PROFILING
- SHOW_PROFILING_CONFIG
- Profiling database performance

DISABLE_PROFILING

Disables for the current session collection of profiling data of the specified type. For detailed information, see Enabling profiling .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DISABLE_PROFILING( 'profiling-type' )
```

Parameters

profiling-type

The type of profiling data to disable:

- session : Disables profiling for basic session parameters and lock time out data.
- query : Disables profiling for general information about queries that ran, such as the query strings used and the duration of queries.
- ee : Disables profiling for information about the execution run of each query.

Examples

The following statement disables profiling on query execution runs:

```
=> SELECT DISABLE_PROFILING('ee');
  DISABLE_PROFILING
----------------------
 EE Profiling Disabled
(1 row)
```

See also

- CLEAR_PROFILING
- ENABLE_PROFILING
- SHOW_PROFILING_CONFIG

ENABLE_PROFILING

Enables collection of profiling data of the specified type for the current session. For detailed information, see Enabling profiling .

> **Note**
> Vertica stores session and query profiling data in memory, so profiling can be memory intensive, depending on how much data you collect.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ENABLE_PROFILING( 'profiling-type' )
```

Parameters

profiling-type

The type of profiling data to enable:

- session : Enable profiling for basic session parameters and lock time out data.
- query : Enable profiling for general information about queries that ran, such as the query strings used and the duration of queries.
- ee : Enable profiling for information about the execution run of each query.

Examples

The following statement enables profiling on query execution runs:

```
=> SELECT ENABLE_PROFILING('ee');
   ENABLE_PROFILING
----------------------
 EE Profiling Enabled
(1 row)
```

See also

- CLEAR_PROFILING
- DISABLE_PROFILING
- SHOW_PROFILING_CONFIG

SHOW_PROFILING_CONFIG

Shows whether profiling is enabled.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Stable

Syntax

```
SHOW_PROFILING_CONFIG ()
```

Examples

The following statement shows that profiling is enabled globally for all profiling types (session, execution engine, and query):

```
=> SELECT SHOW_PROFILING_CONFIG();
SHOW_PROFILING_CONFIG
-----------------------------------------
 Session Profiling: Session off, Global on
 EE Profiling:      Session off, Global on
 Query Profiling:   Session off, Global on
(1 row)
```

See also

- CLEAR_PROFILING
- DISABLE_PROFILING
- ENABLE_PROFILING
- Profiling database performance

Statistics management functions

This section contains Vertica functions for collecting and managing table data statistics.

In this section

- ANALYZE_EXTERNAL_ROW_COUNT
- ANALYZE_STATISTICS
- ANALYZE_STATISTICS_PARTITION
- DROP_EXTERNAL_ROW_COUNT
- DROP_STATISTICS
- DROP_STATISTICS_PARTITION
- EXPORT_STATISTICS
- EXPORT_STATISTICS_PARTITION
- IMPORT_STATISTICS
- VALIDATE_STATISTICS

ANALYZE_EXTERNAL_ROW_COUNT

Calculates the exact number of rows in an external table. ANALYZE_EXTERNAL_ROW_COUNT runs in the background.

> **Note**
> You cannot calculate row counts on external tables with DO_TM_TASK .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ANALYZE_EXTERNAL_ROW_COUNT ('[[[database.]schema.]table-name ]')
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table-name

Specifies the name of the external table for which to calculate the exact row count. If you supply an empty string, Vertica calculate the exact number of rows for all external tables.

Privileges

Any INSERT/UPDATE/DELETE privilege on the external table

Examples

Calculate the exact row count for all external tables:

```
=> SELECT ANALYZE_EXTERNAL_ROW_COUNT('');
```

Calculate the exact row count for table loader_rejects :

```
=> SELECT ANALYZE_EXTERNAL_ROW_COUNT('loader_rejects');
```

See also

- Collecting database statistics
- DROP_EXTERNAL_ROW_COUNT

ANALYZE_STATISTICS

Collects and aggregates data samples and storage information from all nodes that store projections associated with the specified table.The function skips columns of complex data types . By default, Vertica analyzes multiple columns in a single-query execution plan, depending on resource limits. Such multi-column analysis facilitates the following objectives:

- Reduce plan execution latency.

- Speed up analysis of relatively small tables with many columns.

Vertica writes statistics to the database catalog. The query optimizer uses this collected data to create query plans. Without this data, the query optimizer assumes uniform distribution of data values and equal storage usage for all projections.

You can cancel statistics collection with CTRL+C or by calling INTERRUPT_STATEMENT .

ANALYZE_STATISTICS is an alias of the function ANALYZE_HISTOGRAM , which is no longer documented.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ANALYZE_STATISTICS ('[[[database.]schema.]table]' [, 'column-list' [, percent ]]  )
```

Returns

0—Success

If an error occurs, refer to vertica.log for details.

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Table on which to collect data. If set to an empty string, Vertica collects statistics for all database tables and their projections.

column-list

Comma-delimited list of columns in *table* , typically predicate columns. Vertica narrows the scope of the data collection to the specified columns. Columns of complex types are not supported.

If you alter a table to add a column and populate its contents with either default or other values, call ANALYZE_STATISTICS on this column to get the most current statistics.

percent

A float value between 0 and 100 that specifies what percentage of data to read from disk (not the amount of data to analyze). If you omit this argument, Vertica sets the percentage to 10.

Analyzing more than 10 percent disk space takes proportionally longer to process, but produces a higher level of sampling accuracy.

Privileges

Non-superuser:

- Schema: USAGE
- Table: One of INSERT, DELETE, or UPDATE

Restrictions

- Vertica supports ANALYZE_STATISTICS on local and global temporary tables . In both cases, you can obtain statistics only on tables that are created with the option ON COMMIT PRESERVE ROWS . Otherwise, Vertica deletes table content on committing the current transaction, so no table data is available for analysis. Vertica collects no statistics from the following projections: * Live aggregate and Top-K projections * Projections that are defined to include an SQL function within an expression
- Vertica collects no statistics on columns of ARRAY, SET, or ROW types.

Examples

See Collecting table statistics .

See also

ANALYZE_STATISTICS_PARTITION

ANALYZE_STATISTICS_PARTITION

Collects and aggregates data samples and storage information for a range of partitions in the specified table. Vertica writes the collected statistics to the database catalog.

You can cancel statistics collection with CTRL+C or meta-function INTERRUPT_STATEMENT .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ANALYZE_STATISTICS_PARTITION ('[[database.]schema.]table', 'min-range-value','max-range-value' [, 'column-list' [, percent ]] )
```

Returns

0: Success

If an error occurs, refer to vertica.log for details.

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Table on which to collect data.

min-range-value

max-range-value

Minimum and maximum value of partition keys to analyze, where *min-range-value* must be ≤ *max-range-value* . To analyze one partition, *min-range-value* and *max-range-value* must be equal.

column-list

Comma-delimited list of columns in *table* , typically a predicate column. Vertica narrows the scope of the data collection to the specified columns.

percent

Float value between 0 and 100 that specifies what percentage of data to read from disk (not the amount of data to analyze). If you omit this argument, Vertica sets the percentage to 10.

Analyzing more than 10 percent disk space takes proportionally longer to process, but produces a higher level of sampling accuracy.

Privileges

Non-superuser:

- Schema: USAGE
- Table: One of INSERT, DELETE, or UPDATE

Requirements and restrictions

The following requirements and restrictions apply to ANALYZE_STATISTICS_PARTITION:

- The table must be partitioned and cannot contain unpartitioned data.
- The table partition expression must specify a single column. The following expressions are supported:
  - Expressions that specify only the column—that is, partition on all column values. For example:
    ```
    PARTITION BY ship_date GROUP BY CALENDAR_HIERARCHY_DAY(ship_date, 2, 2)
    ```
  - If the column is a DATE or TIMESTAMP/TIMESTAMPTZ , the partition expression can specify a supported date/time function that returns that column or any portion of it, such as month or year. For example, the following partition expression specifies to partition on the year portion of column order_date :
    ```
    PARTITION BY YEAR(order_date)
    ```
  - Expressions that perform addition or subtraction on the column. For example:
    ```
    PARTITION BY YEAR(order_date) -1
    ```
- The table partition expression cannot coerce the specified column to another data type.
- Vertica collects no statistics from the following projections:
  - Live aggregate and Top-K projections
  - Projections that are defined to include an SQL function within an expression

Examples

See Collecting partition statistics .

DROP_EXTERNAL_ROW_COUNT

Removes external table row count statistics compiled by ANALYZE_EXTERNAL_ROW_COUNT . DROP_EXTERNAL_ROW_COUNT runs in the background.

> **Caution**
> Statistics can be time consuming to regenerate.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DROP_EXTERNAL_ROW_COUNT ('[[[database.]schema.]table-name ]');
```

Parameters

schema

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table-name

The external table for which to remove the exact row count. If you specify an empty string, Vertica drops the exact row count statistic for all external tables.

Privileges

- INSERT/UPDATE/DELETE privilege on table
- USAGE privilege on schema that contains the table

Examples

Drop row count statistics for external table loader_rejects :

```
=> SELECT DROP_EXTERNAL_ROW_COUNT('loader_rejects');
```

See also

Collecting database statistics

DROP_STATISTICS

Removes statistical data on database projections previously generated by ANALYZE_STATISTICS . When you drop this data, the Vertica optimizer creates query plans using default statistics.

> **Caution**
> Regenerating statistics can incur significant overhead.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DROP_STATISTICS ('[[[database.]schema.]table]' [, 'category' [, '[column-list]'] )
```

Parameters

[***database*** **.]** ***schema***

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Table on which to drop statistics. If set to an empty string, Vertica drops statistics for all database tables and their projections.

category

Category of statistics to drop, one of the following:

- ALL (default): Drop all statistics, including histograms and row counts.
- HISTOGRAMS : Drop only histograms. Row count statistics remain.

column-list

Comma-delimited list of columns in *table* , typically predicate columns. Vertica narrows the scope of dropped statistics to the specified columns. If you omit this parameter or supply an empty string, Vertica drops statistics on all columns.

Privileges

Non-superuser:

- Schema: USAGE
- Table: One of INSERT, DELETE, or UPDATE

Examples

Drop all base statistics for the table store.store_sales_fact :

```
=> SELECT DROP_STATISTICS('store.store_sales_fact');
 DROP_STATISTICS
-----------------
               0
(1 row)
```

Drop statistics for all table projections:

```
=> SELECT DROP_STATISTICS ('');
 DROP_STATISTICS
-----------------
               0
(1 row)
```

See also

DROP_STATISTICS_PARTITION

DROP_STATISTICS_PARTITION

Removes statistical data on database projections previously generated by ANALYZE_STATISTICS_PARTITION . When you drop this data, the Vertica optimizer creates query plans using table-level statistics, if available, or default statistics.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
DROP_STATISTICS_PARTITION ('[[database.]schema.]table', '[min-range-value]', '[max-range-value]' [, category [, '[column-list]'] )
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Table on which to drop statistics.

min-range-value max-range-value

The minimum and maximum value of partition keys on which to drop statistics, where *min-range-value* must be ≤ *max-range-value* . If you supply empty strings for both parameters, Vertica drops all partition-level statistics for this table or the specified columns.

Important

The range of keys to drop must be equal to, or a superset of, the full range of partitions previously analyzed by ANALYZE_STATISTICS_PARTITION . If the range omits any analyzed partition, DROP_STATISTICS_PARTITION drops no statistics.

category

The category of statistics to drop, one of the following:

- BASE (default): Drop histograms and row counts (min/max column values, histogram).
- HISTOGRAMS : Drop only histograms. Row count statistics remain.
- ALL : Drop all statistics.

column-list

A comma-delimited list of columns in *table* , typically predicate columns. Vertica narrows the scope of dropped statistics to the specified columns. If you omit this parameter or supply an empty string, Vertica drops statistics on all columns.

Privileges

Non-superuser:

- Schema: USAGE
- Table: One of INSERT, DELETE, or UPDATE

See also

DROP_STATISTICS

EXPORT_STATISTICS

Generates statistics in XML format from data previously collected by ANALYZE_STATISTICS . Before you export statistics, collect the latest data by calling ANALYZE_STATISTICS .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Stable

Syntax

```
EXPORT_STATISTICS ('[ filename ]' [,'table-spec' [,'column[,...]']])
```

Arguments

filename

Specifies where to write the generated XML. If *filename* already exists, EXPORT_STATISTICS overwrites it. If you supply an empty string, EXPORT_STATISTICS writes the XML to standard output.

table-spec

Specifies the table on which to export projection statistics:

```
[[database.]schema.]table
```

The default schema is public . If you specify a database, it must be the current database.

If *table-spec* is omitted or set to an empty string, Vertica exports all statistics for the database.

column

The name of a column in *table-spec* , typically a predicate column. You can specify multiple comma-delimited columns. Vertica narrows the scope of exported statistics to the specified columns.

Privileges

Superuser

Restrictions

EXPORT_STATISTICS does not export statistics for LONG data type columns.

Examples

The following statement exports statistics on the VMart example database to a file:

```
=> SELECT EXPORT_STATISTICS('/opt/vertica/examples/VMart_Schema/vmart_stats.xml');
      EXPORT_STATISTICS
-----------------------------------
Statistics exported successfully
(1 row)
```

The next statement exports statistics on a single column (price) from a table named food:

```
=> SELECT EXPORT_STATISTICS('/opt/vertica/examples/VMart_Schema/price.xml', 'food.price');
      EXPORT_STATISTICS
-----------------------------------
Statistics exported successfully
(1 row)
```

See also

- EXPORT_STATISTICS_PARTITION
- DROP_STATISTICS
- IMPORT_STATISTICS

- VALIDATE_STATISTICS
- Collecting database statistics
- Best practices for statistics collection

EXPORT_STATISTICS_PARTITION

Generates partition-level statistics in XML format from data previously collected by ANALYZE_STATISTICS_PARTITION .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Stable

Syntax

```
EXPORT_STATISTICS_PARTITION ('[ filename ]', 'table-spec', 'min-range-value','max-range-value' [, 'column[,...]' )
```

Arguments

filename

Specifies where to write the generated XML. If *filename* already exists, EXPORT_STATISTICS_PARTITION overwrites it. If you supply an empty string, the function writes to standard output.

table-spec

Specifies the table on which to export partition statistics:

```
[[database.]schema.]table
```

The default schema is public . If you specify a database, it must be the current database.

min-range-value* , *max-range-value

The minimum and maximum value of partition keys on which to export statistics, where *min-range-value* must be ≤ *max-range-value* .

Important

The range of keys to export must be equal to, or a superset of, the full range of partitions previously analyzed by ANALYZE_STATISTICS_PARTITION. If the range omits any analyzed partition, EXPORT_STATISTICS_PARTITION exports no statistics.

column

The name of a column in *table* , typically a predicate column. You can specify multiple comma-delimited columns. Vertica narrows the scope of exported statistics to the specified columns.

Privileges

Superuser

Restrictions

EXPORT_STATISTICS_PARTITION does not export statistics for LONG data type columns.

See also

EXPORT_STATISTICS

IMPORT_STATISTICS

Imports statistics from the XML file that was generated by EXPORT_STATISTICS . Imported statistics override existing statistics for the projections that are referenced in the XML file.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Stable

Syntax

```
IMPORT_STATISTICS ( 'filename' )
```

Parameters

filename

The path and name of an XML input file that was generated by EXPORT_STATISTICS .

Privileges

Superuser

Restrictions

- IMPORT_STATISTICS imports only valid statistics. If the source XML file has invalid statistics for a specific column, those statistics are not imported and Vertica throws a warning. If the statistics file has an invalid structure, the import operation fails. To check a statistics file for validity, run VALIDATE_STATISTICS .
- IMPORT_STATISTICS returns warnings for LONG data type columns, as the source XML file generated by EXPORT_STATISTICS contains no statistics for columns of that type.

Examples

Import the statistics for the VMart database from an XML file previously created by EXPORT_STATISTICS :

```
=> SELECT IMPORT_STATISTICS('/opt/vertica/examples/VMart_Schema/vmart_stats.xml');
                IMPORT_STATISTICS
-------------------------------------------------------------------------------
Importing statistics for projection date_dimension_super column date_key failure (stats did not contain row counts)
Importing statistics for projection date_dimension_super column date failure (stats did not contain row counts)
Importing statistics for projection date_dimension_super column full_date_description failure (stats did not contain row counts)
...
(1 row)
```

See also

- ANALYZE_STATISTICS
- EXPORT_STATISTICS

VALIDATE_STATISTICS

Validates statistics in the XML file generated by EXPORT_STATISTICS .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Stable

Syntax

```
VALIDATE_STATISTICS ( 'XML-file' )
```

Parameters

XML-file

the path and name of the XML file that contains the statistics to validate.

Privileges

Superuser

Reporting valid statistics

The following example shows the results when the statistics are valid:

```
=> SELECT EXPORT_STATISTICS('cust_dim_stats.xml','customer_dimension');
   EXPORT_STATISTICS
-----------------------------------
 Statistics exported successfully
(1 row)

=> SELECT VALIDATE_STATISTICS('cust_dim_stats.xml');
 VALIDATE_STATISTICS
---------------------
(1 row)
```

Identifying invalid statistics

If VALIDATE_STATISTICS is unable to read a document's XML, it throws this error:

```
=> SELECT VALIDATE_STATISTICS('/home/dbadmin/stats.xml');
                    VALIDATE_STATISTICS
-------------------------------------------------------------------------------
Error validating statistics file: At line 1:1. Invalid document structure
(1 row)
```

If some table statistics are invalid, VALIDATE_STATISTICS returns a report that identifies them. In the following example, the function reports that attributes distinct , buckets , rows , count , and distinctCount cannot be negative numbers.

```
=> SELECT VALIDATE_STATISTICS('/stats.xml');
WARNING 0:  Invalid value '-1' for attribute 'distinct' under column 'public.t.x'.
   Please use a positive value.
WARNING 0:  Invalid value '-1' for attribute 'buckets' under column 'public.t.x'.
   Please use a positive value.
WARNING 0:  Invalid value '-1' for attribute 'rows' under column 'public.t.x'.
   Please use a positive value.
WARNING 0:  Invalid value '-1' for attribute 'count' under bound '1', column 'public.t.x'.
   Please use a positive value.
WARNING 0:  Invalid value '-1' for attribute 'distinctCount' under bound '1', column 'public.t.x'.
   Please use a positive value.
 VALIDATE_STATISTICS
---------------------
 (1 row)
```

In this case, run ANALYZE_STATISTICS on the table again to create valid statistics.

See also

- ANALYZE_STATISTICS
- EXPORT_STATISTICS

Workload management functions

This section contains workload management functions specific to Vertica.

In this section

- ANALYZE_WORKLOAD
- CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY
- CHANGE_RUNTIME_PRIORITY
- MOVE_STATEMENT_TO_RESOURCE_POOL
- SLEEP

ANALYZE_WORKLOAD

Runs Workload Analyzer, a utility that analyzes system information held in system tables .

Workload Analyzer intelligently monitors the performance of SQL queries and workload history, resources, and configurations to identify the root causes for poor query performance. ANALYZE_WORKLOAD returns tuning recommendations for all events within the scope and time that you specify, from system table TUNING_RECOMMENDATIONS .

Tuning recommendations are based on a combination of statistics , system and data collector events, and database-table-projection design. Workload Analyzer recommendations can help you quickly and easily tune query performance.

See Workload analyzer recommendations for the common triggering conditions and recommendations.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ANALYZE_WORKLOAD ( '[ scope ]' [, 'since-time' | save-data ] );
```

Parameters

scope

Specifies the catalog objects to analyze, as follows:

```
[[database.]schema.]table
```

If set to an empty string, Vertica returns recommendations for all database objects.
If you specify a database, it must be the current database.

since-time

Specifies the start time for the analysis time span, which continues up to the current system status, inclusive. If you omit this parameter, ANALYZE_WORKLOAD returns recommendations on events since the last time you called this function.

> **Note**
> You must explicitly cast strings to TIMESTAMP or TIMESTAMPTZ . For example:
>
> ```
> SELECT ANALYZE_WORKLOAD('T1', '2010-10-04 11:18:15'::TIMESTAMPTZ);
> SELECT ANALYZE_WORKLOAD('T1', TIMESTAMPTZ '2010-10-04 11:18:15');
> ```

save-data

Specifies whether to save returned values from ANALYZE_WORKLOAD :

- false (default): Results are discarded.
- true : Saves the results returned by ANALYZE_WORKLOAD . Subsequent calls to ANALYZE_WORKLOAD return results that start from the last invocation when results were saved. Object events preceding that invocation are ignored.

Return values

Returns aggregated tuning recommendations from TUNING_RECOMMENDATIONS .

Privileges

Superuser

Examples

See Getting tuning recommendations .

See also

- Analyzing workloads
- Workload analyzer recommendations

CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY

Changes the run-time priority of an active query.

> **Note**
> This function replaces deprecated function CHANGE_RUNTIME_PRIORITY .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY(transaction-id, 'value')
```

Parameters

transaction-id

Identifies the transaction, obtained from the system table SESSIONS .

value

The RUNTIMEPRIORITY value: HIGH , MEDIUM , or LOW .

Privileges

- Superuser : None

- Non-superusers can only change the runtime priority of their own queries, and cannot raise the runtime priority of a query to a level higher than that of the resource pool.

Examples

See Changing runtime priority of a running query .

CHANGE_RUNTIME_PRIORITY

Changes the run-time priority of a query that is actively running. Note that, while this function is still valid, you should instead use CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY to change run-time priority. CHANGE_RUNTIME_PRIORITY will be deprecated in a future release of Vertica.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
CHANGE_RUNTIME_PRIORITY(TRANSACTION_ID,STATEMENT_ID, 'value')
```

Parameters

TRANSACTION_ID

An identifier for the transaction within the session.

TRANSACTION_ID cannot be NULL.
You can find the transaction ID in the Sessions table.

STATEMENT_ID

A unique numeric ID assigned by the Vertica catalog, which identifies the currently executing statement.

You can find the statement ID in the Sessions table.
You can specify NULL to change the run-time priority of the currently running query within the transaction.

'value'

The RUNTIMEPRIORITY value. Can be HIGH, MEDIUM, or LOW.

Privileges

No special privileges required. However, non-superusers can change the run-time priority of their own queries only. In addition, non-superusers can never raise the run-time priority of a query to a level higher than that of the resource pool.

Examples

```
=> SELECT CHANGE_RUNTIME_PRIORITY(45035996273705748, NULL, 'low');
```

MOVE_STATEMENT_TO_RESOURCE_POOL

Attempts to move the specified query to the specified target pool.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
MOVE_STATEMENT_TO_RESOURCE_POOL (session_id , transaction_id, statement_id, target_resource_pool_name)
```

Parameters

session_id

Identifier for the session where the query you want to move is currently executing.

transaction_id

Identifier for the transaction within the session.

statement_id

Unique numeric ID for the statement you want to move.

target_resource_pool_name

Name of the existing resource pool to which you want to move the specified query.

Outputs

The function may return the following results:

MOV_REPLAN: Target pool does not have sufficient resources. See v_monitor.resource_pool_move for details. Vertica will attempt to replan the statement on target pool.
MOV_REPLAN: Target pool has priority HOLD. Vertica will attempt to replan the statement on target pool.
MOV_FAILED: Statement not found.
MOV_NO_OP: Statement already on target pool.
MOV_REPLAN: Statement is in queue. Vertica will attempt to replan the statement on target pool.
MOV_SUCC: Statement successfully moved to target pool.

Privileges

Superuser

Examples

The following example shows how you can move a specific statement to a resource pool called my_target_pool:

```
=> SELECT MOVE_STATEMENT_TO_RESOURCE_POOL ('v_vmart_node0001.example.-31427:0x82fbm', 45035996273711993, 1, 'my_target_pool');
```

See also:

- Manually moving queries to different resource pools
- RESOURCE_POOL_MOVE

SLEEP

Waits a specified number of seconds before executing another statement or command.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
SLEEP( seconds )
```

Parameters

seconds

The wait time, specified in one or more seconds (0 or higher) expressed as a positive integer. Single quotes are optional; for example, SLEEP(3) is the same as SLEEP('3') .

Notes

- This function returns value 0 when successful; otherwise it returns an error message due to syntax errors.
- You cannot cancel a sleep operation.
- Be cautious when using SLEEP() in an environment with shared resources, such as in combination with transactions that take exclusive locks.

Examples

The following command suspends execution for 100 seconds:

```
=> SELECT SLEEP(100);
 sleep
-------
     0
(1 row)
```

Stored procedure functions

This section contains functions for managing stored procedures .

In this section

- ACTIVE_SCHEDULER_NODE
- ENABLE_SCHEDULE
- ENABLE_TRIGGER
- EXECUTE_TRIGGER

ACTIVE_SCHEDULER_NODE

Returns the active scheduler node . A schedule must be associated with a trigger to be enabled.

To view existing schedules, see USER_SCHEDULES .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ACTIVE_SCHEDULER_NODE()
```

Privileges

Superuser

Examples

To return the active scheduler node:

```
=> SELECT active_scheduler_node();

 active_scheduler_node
-----------------------
 initiator
(1 row)
```

ENABLE_SCHEDULE

Enables or disables a schedule. A schedule can only be enabled if a trigger is attached to it.

To view existing schedules, see USER_SCHEDULES .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ENABLE_SCHEDULE ( '[[database.]schema.]schedule', enabled )
```

Arguments

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

schedule

The schedule to enable or disable.

enabled

Boolean, whether to enable the trigger.

Privileges

Superuser

Examples

To enable a schedule:

```
=> SELECT enable_schedule('vmart.management.daily_1am', true);
```

To disable a schedule:

```
=> SELECT enable_schedule('vmart.management.daily_1am', false);
```

If you leave the database and schema empty, the default is *current_database* .public:

```
=> SELECT enable_schedule('biannual_22_noon_gmt', true);
```

ENABLE_TRIGGER

Enables or disables a trigger.

To view existing triggers, see STORED_PROC_TRIGGERS .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
ENABLE_TRIGGER ( '[[database.]schema.]trigger', enabled )
```

Arguments

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

trigger

The trigger to enable or disable.

enabled

Boolean, whether to enable the trigger.

Privileges

Superuser

Examples

To enable a trigger:

```
=> SELECT enable_trigger('vmart.management.log_user_actions', true);
```

To disable a trigger:

```
=> SELECT enable_trigger('vmart.management.log_user_actions', false);
```

If you leave the database and schema empty, the default is *current_database* .public:

```
=> SELECT enable_trigger('revoke_log_privileges', true);
```

EXECUTE_TRIGGER

Manually executes the stored procedure attached to a trigger. This is generally used for testing the trigger.

To view existing triggers, see STORED_PROC_TRIGGERS .

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Syntax

```
EXECUTE_TRIGGER ( '[[database.]schema.]trigger' )
```

Arguments

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

trigger

The trigger to execute.

Privileges

Superuser

Examples

To execute a trigger:

```
=> SELECT execute_trigger('vmart.management.log_user_actions');
```

If you leave the database and schema empty, the default is *current_database* .public:

```
=> SELECT execute_trigger('revoke_log_privileges');
```

System information functions

These functions provide information about the current system state. A superuser has unrestricted access to all system information, but users can view only information about their own, current sessions.

In this section

- CURRENT_DATABASE
- CURRENT_LOAD_SOURCE
- CURRENT_SCHEMA
- CURRENT_SESSION
- CURRENT_TRANS_ID
- CURRENT_USER
- DBNAME (function)
- HAS_TABLE_PRIVILEGE
- LIST_ENABLED_CIPHERS
- SESSION_USER
- USER
- USERNAME
- VERSION

CURRENT_DATABASE

Returns the name of the current database, equivalent to DBNAME .

Behavior type

Stable

Syntax

> **Note**
> Parentheses are optional.

```
CURRENT_DATABASE()
```

Examples

```
=> SELECT CURRENT_DATABASE;
 CURRENT_DATABASE
------------------
 VMart
(1 row)
```

CURRENT_LOAD_SOURCE

When called within the scope of a COPY statement, returns the file name used for the load. The following exceptions apply:

- If the function is called outside of the context of a COPY statement, it returns NULL.
- If the function is called by a UDL that does not set the source, it returns the string <unknown> .

This function is not supported for COPY LOCAL .

Behavior type

Stable

Syntax

```
CURRENT_LOAD_SOURCE()
```

Examples

Create a table and populate one of its columns with the names of two separate files as they are loaded:

```
=> CREATE TABLE t (c1 integer, c2 varchar(50), c3 varchar(200));
CREATE TABLE

=> COPY t (c1, c2, c3 AS CURRENT_LOAD_SOURCE())
   FROM '/home/load_file_1' ON exampledb_node02,
        '/home/load_file_2' ON exampledb_node03 DELIMITER ',';
Rows Loaded
-------------
5
(1 row)

=> SELECT * FROM t;
c1  |      c2      |          c3
----+--------------+-----------------------
2   |  dogs        | /home/load_file_1
1   |  cats        | /home/load_file_1
4   |  superheroes | /home/load_file_2
3   |  birds       | /home/load_file_1
5   |  whales      | /home/load_file_2
(5 rows)
```

CURRENT_SCHEMA

Returns the name of the current schema.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Stable

Syntax

```
CURRENT_SCHEMA()
```

> **Note**
> You can call this function without parentheses.

Privileges

None

Examples

The following command returns the name of the current schema:

```
=> SELECT CURRENT_SCHEMA();
 current_schema
----------------
 public
(1 row)
```

The following command returns the same results without the parentheses:

```
=> SELECT CURRENT_SCHEMA;
 current_schema
----------------
 public
(1 row)
```

The following command shows the current schema, listed after the current user , in the search path:

```
=> SHOW SEARCH_PATH;
   name    |                   setting
-----------+-------------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
(1 row)
```

See also

- SET SEARCH_PATH

CURRENT_SESSION

Returns the ID of the current client session.

Many system tables have a SESSION_ID column. You can use the CURRENT_SESSION function in queries of these tables.

Behavior type

Stable

Syntax

```
CURRENT_SESSION()
```

Examples

Each new session has a new session ID:

```
$ vsql
Welcome to vsql, the Vertica Analytic Database interactive terminal.

=> SELECT CURRENT_SESSION();
   CURRENT_SESSION
-----------------------
 initiator-24897:0x1f7
(1 row)
=> \q

$ vsql
Welcome to vsql, the Vertica Analytic Database interactive terminal.

=> SELECT CURRENT_SESSION();
   CURRENT_SESSION
-----------------------
 initiator-24897:0x200
(1 row)
```

CURRENT_TRANS_ID

Returns the ID of the transaction currently in progress.

Many system tables have a TRANSACTION_ID column. You can use the CURRENT_TRANS_ID function in queries of these tables.

Behavior type

Stable

Syntax

```
CURRENT_TRANS_ID()
```

Examples

Even a new session has a transaction ID:

```
$ vsql
Welcome to vsql, the Vertica Analytic Database interactive terminal.
=> SELECT CURRENT_TRANS_ID();
 current_trans_id
------------------
 45035996273705927
(1 row)
```

This function can be used in queries of certain system tables. In the following example, a load operation is in progress:

```
=> SELECT key, SUM(num_instances) FROM v_monitor.UDX_EVENTS
   WHERE event_type = 'UNMATCHED_KEY'
   AND transaction_id=CURRENT_TRANS_ID()
   GROUP BY key;
         key          | SUM
----------------------+-----
 chain                |   1
 menu.elements.calories |   7
(2 rows)
```

CURRENT_USER

Returns a VARCHAR containing the name of the user who initiated the current database connection.

Behavior type

Stable

Syntax

```
CURRENT_USER()
```

Notes

- The CURRENT_USER function does not require parentheses.
- This function is useful for permission checking.
- CURRENT_USER is equivalent to SESSION_USER , USER , and USERNAME .

Examples

```
=> SELECT CURRENT_USER();
 CURRENT_USER
--------------
 dbadmin
(1 row)
```

The following command returns the same results without the parentheses:

```
=> SELECT CURRENT_USER;
 CURRENT_USER
--------------
 dbadmin
(1 row)
```

DBNAME (function)

Returns the name of the current database, equivalent to CURRENT_DATABASE .

Behavior type

Immutable

Syntax

```
DBNAME()
```

Examples

```
=> SELECT DBNAME();
    dbname
------------------
 VMart
(1 row)
```

HAS_TABLE_PRIVILEGE

Returns true or false to verify whether a user has the specified privilege on a table.

This is a meta-function. You must call meta-functions in a top-level SELECT statement.

Behavior type

Volatile

Behavior type

Stable

Syntax

```
HAS_TABLE_PRIVILEGE ( [ user, ] '[[database.]schema.]table', 'privilege' )
```

Parameters

user

Name or OID of a database user. If omitted, Vertica checks privileges for the current user.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Name or OID of the table to check.

privilege

A table privilege , one of the following:

- SELECT: Quer y tables. SELECT privileges are granted by default to the PUBLIC role.
- INSERT: Insert table rows with INSERT , and load data with COPY .

> **Note**
> COPY FROM STDIN is allowed for users with INSERT privileges, while COPY FROM *file* requires admin privileges.

- UPDATE: Update table rows.
- DELETE: Delete table rows.
- REFERENCES: Create foreign key constraints on this table. This privilege must be set on both referencing and referenced tables.
- TRUNCATE: Truncate table contents. Non-owners of tables can also execute the following partition operations on them:
  - DROP_PARTITIONS
  - SWAP_PARTITIONS_BETWEEN_TABLES
  - MOVE_PARTITIONS_TO_TABLE
- ALTER: Modify a table's DDL with ALTER TABLE .
- DROP: Drop a table .

Privileges

Non-superuser, one of the following:

- Table owner
- USAGE privilege on the table schema and one or more privileges on the table

Examples

```
=> SELECT HAS_TABLE_PRIVILEGE('store.store_dimension', 'SELECT');
 HAS_TABLE_PRIVILEGE
---------------------
 t
(1 row)

=> SELECT HAS_TABLE_PRIVILEGE('release', 'store.store_dimension', 'INSERT');
 HAS_TABLE_PRIVILEGE
---------------------
 t
(1 row)

=> SELECT HAS_TABLE_PRIVILEGE(45035996273711159, 45035996273711160, 'select');
 HAS_TABLE_PRIVILEGE
---------------------
t
(1 row)
```

LIST_ENABLED_CIPHERS

Returns a list of enabled cipher suites, which are sets of algorithms used to secure TLS/SSL connections.

By default, Vertica uses OpenSSL's default cipher suites. For more information, see the OpenSSL man page .

Syntax

```
LIST_ENABLED_CIPHERS()
```

Examples

```
=> SELECT LIST_ENABLED_CIPHERS();
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA
TLS_RSA_WITH_AES_128_CBC_SHA
```

See also

- TLS protocol
- Security parameters

SESSION_USER

Returns a VARCHAR containing the name of the user who initiated the current database session.

Behavior type

Stable

Syntax

```
SESSION_USER()
```

Notes

- The SESSION_USER function does not require parentheses.
- SESSION_USER is equivalent to CURRENT_USER , USER , and USERNAME .

Examples

```
=> SELECT SESSION_USER();
 session_user
--------------
 dbadmin
(1 row)
```

The following command returns the same results without the parentheses:

```
=> SELECT SESSION_USER;
 session_user
--------------
 dbadmin
(1 row)
```

USER

Returns a VARCHAR containing the name of the user who initiated the current database connection.

Behavior type

Stable

Syntax

```
USER()
```

Notes

- The USER function does not require parentheses.
- USER is equivalent to CURRENT_USER , SESSION_USER , and USERNAME .

Examples

```
=> SELECT USER();
 current_user
--------------
 dbadmin
(1 row)
```

The following command returns the same results without the parentheses:

```
=> SELECT USER;
 current_user
--------------
 dbadmin
(1 row)
```

USERNAME

Returns a VARCHAR containing the name of the user who initiated the current database connection.

Behavior type

Stable

Syntax

```
USERNAME()
```

Notes

- This function is useful for permission checking.
- USERNAME is equivalent to CURRENT_USER , SESSION_USER and USER .

Examples

```
=> SELECT USERNAME();
 username
--------------
 dbadmin
(1 row)
```

VERSION

Returns a VARCHAR containing a Vertica node's version information.

Behavior type

Stable

Syntax

```
VERSION()
```

> **Note**
> The parentheses are required.

Examples

```
=> SELECT VERSION();
                VERSION
-------------------------------------------
Vertica Analytic Database v10.0.0-0
(1 row)
```

Statements

The primary structure of a SQL query is its statement. Whether a statement stands on its own, or is part of a multi-statement query, each statement must end with a semicolon. The following example contains four common SQL statements—CREATE TABLE, INSERT, SELECT, and COMMIT:

```
=> CREATE TABLE comments (id INT, comment VARCHAR);
CREATE TABLE
=> INSERT INTO comments VALUES (1, 'Hello World');
OUTPUT
--------
1
(1 row)

=> SELECT * FROM comments;
 id |   comment
----+-------------
  1 | Hello World
(1 row)

=> COMMIT;
COMMIT
=>
```

In this section

- ACTIVATE DIRECTED QUERY
- ALTER statements
- BEGIN
- CALL
- COMMENT ON statements
- COMMIT
- CONNECT TO VERTICA
- COPY
- COPY FROM VERTICA
- COPY LOCAL
- CREATE statements
- DEACTIVATE DIRECTED QUERY
- DELETE
- DISCONNECT
- DO
- DROP statements
- END
- EXPLAIN
- EXPORT TO DELIMITED
- EXPORT TO JSON
- EXPORT TO ORC
- EXPORT TO PARQUET
- EXPORT TO VERTICA
- GET DIRECTED QUERY
- GRANT statements

- INSERT
- LOCK TABLE
- MERGE
- PROFILE
- RELEASE SAVEPOINT
- REVOKE statements
- ROLLBACK
- ROLLBACK TO SAVEPOINT
- SAVE QUERY
- SAVEPOINT
- SELECT
- SET statements
- SHOW
- SHOW CURRENT
- SHOW DATABASE
- SHOW NODE
- SHOW SESSION
- SHOW USER
- START TRANSACTION
- TRUNCATE TABLE
- UPDATE

ACTIVATE DIRECTED QUERY

Activates a directed query and makes it available to the query optimizer across all sessions.

Syntax

```
ACTIVATE DIRECTED QUERY { query-name | where-clause }
```

Arguments

query-name

Name of the directed query to activate, as stored in the DIRECTED_QUERIES column query_name . You can also use GET DIRECTED QUERY to obtain names of all directed queries that map to an input query.

where-clause

Resolves to one or more directed queries that are filtered from system table DIRECTED_QUERIES . For example, the following statement activates all directed queries with the same save_plans_version identifier:

```
=> ACTIVATE DIRECTED QUERY WHERE save_plans_version = 21;
```

Privileges

Superuser

Activation life cycle

After you activate a directed query, it remains active until it is explicitly deactivated by DEACTIVATE DIRECTED QUERY or removed from storage by DROP DIRECTED QUERY . If a directed query is active at the time of database shutdown, Vertica automatically reactivates it when you restart the database.

Examples

See Activating and deactivating directed queries .

ALTER statements

ALTER statements let you change existing database objects.

In this section

- ALTER ACCESS POLICY
- ALTER AUTHENTICATION
- ALTER CA BUNDLE
- ALTER DATABASE
- ALTER FAULT GROUP
- ALTER FUNCTION statements
- ALTER HCATALOG SCHEMA
- ALTER LIBRARY

- ALTER LOAD BALANCE GROUP
- ALTER MODEL
- ALTER NETWORK ADDRESS
- ALTER NETWORK INTERFACE
- ALTER NODE
- ALTER NOTIFIER
- ALTER PROCEDURE (stored)
- ALTER PROFILE
- ALTER PROFILE RENAME
- ALTER PROJECTION
- ALTER RESOURCE POOL
- ALTER ROLE
- ALTER ROUTING RULE
- ALTER SCHEDULE
- ALTER SCHEMA
- ALTER SEQUENCE
- ALTER SESSION
- ALTER SUBCLUSTER
- ALTER SUBNET
- ALTER TABLE
- ALTER TLS CONFIGURATION
- ALTER TRIGGER
- ALTER USER
- ALTER VIEW

ALTER ACCESS POLICY

Performs one of the following actions on existing access policies:

- Modify an access policy by changing its expression, and by enabling/disabling the policy.
- Copy an access policy from one table to another.

Syntax

Modify policy:

```
ALTER ACCESS POLICY ON [[database.]schema.]table
  { FOR COLUMN column [ expression ] | FOR ROWS [ WHERE expression ] } { GRANT TRUSTED } { ENABLE | DISABLE }
```

Copy policy:

```
ALTER ACCESS POLICY ON [[database.]schema.]table
  { FOR COLUMN column | FOR ROWS } COPY TO TABLE table;
```

Parameters

`[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

The name of the table that contains the access policy you want to enable, disable, or copy.

FOR COLUMN *column* [*expression*]

Replaces the access policy expression that was previously set for this column. Omit *expression* from the FOR COLUMN clause in order to enable or disable this policy only, or copy it to another table.

FOR ROWS [WHERE *expression*]

Replaces the row access policy expression that was previously set for this table. Omit WHERE *expression* from the FOR ROWS clause in order to enable or disable this policy only, or copy it to another table.

GRANT TRUSTED

Specifies that GRANT statements take precedence over the access policy in determining whether users can perform DML operations on the target table. If omitted, users can only modify table data if the access policy allows them to see the stored data in its original, unaltered state. For more information, see Access policies and DML operations .

Important

GRANT TRUSTED only affects DML operations and does not enable users to see data that the access policy would otherwise mask. Specifying this option may allow users with certain grants to update data that they cannot see.

ENABLE | DISABLE

Indicates whether to enable or disable the access policy at the table level.

COPY TO TABLE *tablename*

Copies the existing access policy to the specified table. The copied access policy includes its enabled/disabled and GRANT TRUSTED statuses.

The following requirements apply:

- Copying a column access policy:
  - The target table must have a column of the same name and compatible data type.
  - The target colum must not have an access policy.
- Copying a row access policy: The target table must not have an access policy.

Privileges

Modify access policy

Non-superuser: Ownership of the table

Copy access policy

Non-superuser: Ownership of the source and destination tables

Examples

See Managing access policies

See also

CREATE ACCESS POLICY

ALTER AUTHENTICATION

Modifies the settings for a specified authentication method.

Syntax

```
ALTER AUTHENTICATION auth_record {
  | { ENABLE | DISABLE }
  | { LOCAL | HOST [ { TLS | NO TLS } ] host_ip_address }
  | RENAME TO new_auth_record_name
  | METHOD value
  | SET param=value[,...]
  | PRIORITY value
  | [ [ NO ] FALLTHROUGH ]
}
```

Parameters

Parameter Name	Description
auth_record	Name of the authentication method to alter. **Type:** VARCHAR
ENABLE | DISABLE	Enable or disable the specified authentication method. **Default:** Enabled When you perform an upgrade and use Kerberos authentication, you must manually set the authentication to ENABLE as it is disabled by default.

LOCAL | HOST [{ TLS | NO TLS }] *host_ip_address*	Specify that the authentication method applies to local or remote (HOST) connections. For authentication methods that use LDAP, specify whether or not LDAP uses Transport Layer Security (TLS). For remote (HOST) connections, you must specify the IP address of the host from which the user or application is connecting, VARCHAR. Vertica supports IPv4 and IPv6 addresses.
RENAME TO *new_auth_record_name*	Rename the authentication record. **Type:** VARCHAR
METHOD *value*	The authentication method you are altering.
SET *param = value*	Set a parameter name and value for the authentication method that you are creating. This is required for LDAP , Ident , and OAuth authentication methods.
PRIORITY *value*	If the user is associated with multiple authentication methods, the priority value specifies which authentication method Vertica tries first. **Default:** 0 **Type:** INTEGER Greater values indicate higher priorities. For example, a priority of 10 is higher than a priority of 5; priority 0 is the lowest possible value. For details, see Authentication record priority .
[[NO] FALLTHROUGH]	Specifies whether to enable authentication fallthrough. For details, see Client authentication .

Privileges

Superuser

Examples

Enabling and Disabling Authentication Methods

This example uses ALTER AUTHENTICATION to disable the v_ldap authentication method and then enable it again:

```
=> ALTER AUTHENTICATION v_ldap DISABLE;
=> ALTER AUTHENTICATION v_ldap ENABLE;
```

Renaming Authentication Methods

This example renames the v_kerberos authentication method to K5 . All users who have been granted the v_kerberos authentication method now have the K5 method granted instead.

```
=> ALTER AUTHENTICATION v_kerberos RENAME TO K5;
```

Modifying Authentication Parameters

This example sets the system user for ident1 authentication to user1 :

```
=> CREATE AUTHENTICATION ident1 METHOD 'ident' LOCAL;
=> ALTER AUTHENTICATION ident1 SET system_users='user1';
```

When you set or modify LDAP or Ident parameters using ALTER AUTHENTICATION, Vertica validates them.

This example changes the IP address and specifies the parameters for an LDAP authentication method named Ldap1 . Specify the bind parameters for the LDAP server. Vertica connects to the LDAP server, which authenticates the database client. If authentication succeeds, Vertica authenticates any users who have been associated with (granted) the Ldap1 authentication method on the designated LDAP server:

```
=> CREATE AUTHENTICATION Ldap1 METHOD 'ldap' HOST '172.16.65.196';

=> ALTER AUTHENTICATION Ldap1 SET host='ldap://172.16.65.177',
  binddn_prefix='cn=', binddn_suffix=',dc=qa_domain,dc=com';
```

The next example specifies the parameters for an LDAP authentication method named Ldap2 . Specify the LDAP search and bind parameters. Sometimes, Vertica does not have enough information to create the distinguished name (DN) for a user attempting to authenticate. In such cases, you must specify to use LDAP search and bind:

```
=> CREATE AUTHENTICATION Ldap2 METHOD 'ldap' HOST '172.16.65.196';
=> ALTER AUTHENTICATION Ldap2 SET basedn='dc=qa_domain,dc=com',
  binddn='cn=Manager,dc=qa_domain,
  dc=com',search_attribute='cn',bind_password='secret';
```

Changing the Authentication Method

This example changes the localpwd authentication from hash to trust:

```
=> CREATE AUTHENTICATION localpwd METHOD 'hash' LOCAL;
=> ALTER AUTHENTICATION localpwd METHOD 'trust';
```

Set Multiple Realms

This example sets another realm for the authentication method krb_local:

```

=> ALTER AUTHENTICATION krb_local set realm = 'COMPANY.COM';
```

See also

- CREATE AUTHENTICATION
- DROP AUTHENTICATION
- GRANT (authentication)
- REVOKE (authentication)
- CLIENT_AUTH

ALTER CA BUNDLE

> **Deprecated**
> CA bundles are only usable with certain deprecated parameters in Kafka notifiers . You should prefer using TLS configurations and the TLS CONFIGURATION parameter for notifiers instead.

Adds and removes certificates from or changes the owner of a certificate authority (CA) bundle.

Syntax

```
ALTER CA BUNDLE name
    [ADD CERTIFICATES ca_cert[, ca_cert[, ...]]
    [REMOVE CERTIFICATES ca_cert[, ca_cert[, ...]]
    [OWNER TO user]
```

Parameters

name

The name of the CA bundle.

ca_cert

The name of the CA certificate to add or remove from the bundle.

user

The name of a database user.

Privileges

Ownership of the CA bundle.

Examples

See Managing CA bundles .

See also

- CREATE CA BUNDLE
- DROP CA BUNDLE

ALTER DATABASE

Use ALTER DATABASE to perform the following tasks:

- Drop all fault groups and their child fault groups from a database.
- Restore down nodes, and revert active standby nodes to standby status.
- Specify the subnet name of a public network to use for import/export .
- Set and clear database configuration parameters .

To see the current value of a parameter, query system table CONFIGURATION_PARAMETERS or use SHOW DATABASE .

Syntax

```
ALTER DATABASE db-spec {
    DROP ALL FAULT GROUP
    | EXPORT ON { subnet-name | DEFAULT }
    | RESET STANDBY
    | SET [PARAMETER] parameter=value [,...]
    | CLEAR [PARAMETER] parameter[,...]
}
```

Parameters

db-spec

Specifies the database to alter, one of the following:

- The database name
- DEFAULT : The current database

DROP ALL FAULT GROUP

Drops all fault groups defined on the specified database.

EXPORT ON

Specifies the network to use for importing and exporting data, one of the following:

- *subnet-name* : A subnet of the public network.
- DEFAULT : Specifies to use a private network.

For details, see Identify the database or nodes used for import/export , and Changing node export addresses .

RESET STANDBY

Enterprise Mode only, restores all down nodes and reverts their replacement nodes to standby status. If any replaced nodes cannot resume activity, Vertica leaves their standby nodes in place.

SET [PARAMETER]

Sets the specified parameters.

CLEAR [PARAMETER]

Resets the specified parameters to their default values.

Privileges

Superuser

ALTER FAULT GROUP

Modifies an existing fault group. ALTER FAULT GROUP can perform the following tasks:

- Add a node to or drop a node from an existing fault group.
- Add a child fault group to or drop a child fault group from a parent fault group.
- Rename a fault group.

Syntax

```
ALTER FAULT GROUP fault-group-name {
   | ADD NODE node-name
   | DROP NODE node-name
   | ADD FAULT GROUP child-fault-group-name
   | DROP FAULT GROUP child-fault-group-name
   | RENAME TO new-fault-group-name }
```

Parameters

fault-group-name

The existing fault group name you want to modify.

> **Tip**
> For a list of all fault groups defined in the cluster, query the FAULT_GROUPS system table.

node-name

The node name you want to add to or drop from the existing (parent) fault group.

child-fault-group-name

The name of the child fault group you want to add to or remove from an existing parent fault group.

new-fault-group-name

The new name for the fault group you want to rename.

Privileges

Superuser

Examples

This example shows how to rename the parent0 fault group to parent100 :

```
=> ALTER FAULT GROUP parent0 RENAME TO parent100;
ALTER FAULT GROUP
```

Verify the change by querying the FAULT_GROUPS system table:

```
=> SELECT member_name FROM fault_groups;
  member_name
----------------------
v_exampledb_node0003
parent100
mygroup
(3 rows)
```

See also

- CREATE FAULT GROUP
- FAULT_GROUPS
- CLUSTER_LAYOUT
- Fault groups
- High availability with fault groups

ALTER FUNCTION statements

Vertica provides ALTER statements for each type of user-defined extension . Each ALTER statement modifies the metadata of a user-defined function in the Vertica catalog:

ALTER statement	Extension
ALTER FUNCTION (scalar)	User-defined scalar functions (UDSFs)
ALTER AGGREGATE FUNCTION	User-defined aggregate functions (UDAFs)
ALTER ANALYTIC FUNCTION	User-defined analytic functions (UDAnF)

ALTER TRANSFORM FUNCTION	User-defined transform functions (UDTFs)
ALTER statements for user-defined load :	
• ALTER SOURCE	Load source functions
• ALTER FILTER	Load filter functions
• ALTER PARSER	Load parser functions

Vertica also provides ALTER FUNCTION (SQL) , which modifies the metadata of a user-defined SQL function.

In this section

- ALTER AGGREGATE FUNCTION
- ALTER ANALYTIC FUNCTION
- ALTER FILTER
- ALTER FUNCTION (scalar)
- ALTER FUNCTION (SQL)
- ALTER PARSER
- ALTER SOURCE
- ALTER TRANSFORM FUNCTION

ALTER AGGREGATE FUNCTION

Alters a user-defined aggregate function .

Syntax

```
ALTER AGGREGATE FUNCTION [[db-name.]schema.]function-name( [ parameter-list ] ) {
   OWNER TO new-owner
   | RENAME TO new-name
   | SET SCHEMA new-schema
}
```

Parameters

[*db-name.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function-name ``

Name of the SQL function to alter.

arg-list

Comma-delimited list of parameters that are defined for this function. If none, specify an empty list.

> **Note**
> Vertica supports function overloading, and uses the parameter list to identify the function to alter.

OWNER TO *new-owner*

Transfers function ownership to another user.

RENAME TO *new-name*

Renames this function.

SET SCHEMA *new-schema*

Moves the function to another schema.

Privileges

Non-superuser: USAGE on the schema and one of the following:

- Function owner
- ALTER privilege on the function

For certain operations, non-superusers must also have the following schema privileges:

For these operations...	Schema privileges required...
RENAME TO (rename function)	CREATE, USAGE
SET SCHEMA (move function to another schema)	CREATE: destination schema USAGE: current schema

See also

CREATE AGGREGATE FUNCTION
ALTER ANALYTIC FUNCTION

Alters a user-defined analytic function .

Syntax

```
ALTER ANALYTIC FUNCTION [[db-name.]schema.]function-name( [ parameter-list ] ) {
   OWNER TO new-owner
   | RENAME TO new-name
   | SET FENCED boolean-expr
   | SET SCHEMA new-schema
}
```

Parameters

[*db-name.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function-name

Name of the function to alter.

parameter-list

Comma-delimited list of parameters that are defined for this function. If none, specify an empty list.

> **Note**
> Vertica supports function overloading, and uses the parameter list to identify the function to alter.

OWNER TO *new-owner*

Transfers function ownership to another user.

RENAME TO *new-name*

Renames this function.

SET FENCED { true | false }

Specifies whether to enable fenced mode for this function.

SET SCHEMA *new-schema*

Moves the function to another schema.

Privileges

Non-superuser: USAGE on the schema and one of the following:

- Function owner
- ALTER privilege on the function

For certain operations, non-superusers must also have the following schema privileges:

Operation	Schema privileges required
RENAME TO (rename function)	CREATE, USAGE
SET SCHEMA (move function to another schema)	• CREATE: destination schema • USAGE: current schema

See also

CREATE ANALYTIC FUNCTION

ALTER FILTER

Alters a user-defined filter .

Syntax

```
ALTER FILTER [[db-name.]schema.]function-name( [ parameter-list ] ) {
   OWNER TO new-owner
   | RENAME TO new-name
   | SET FENCED boolean-expr
   | SET SCHEMA new-schema
}
```

Parameters

[*db-name.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function-name

Name of the function to alter.

parameter-list

Comma-delimited list of parameters that are defined for this function. If none, specify an empty list.

> **Note**
> Vertica supports function overloading, and uses the parameter list to identify the function to alter.

OWNER TO *new-owner*

Transfers function ownership to another user.

RENAME TO *new-name*

Renames this function.

SET FENCED { true | false }

Specifies whether to enable fenced mode for this function.

SET SCHEMA *new-schema*

Moves the function to another schema.

Privileges

Non-superuser: USAGE on the schema and one of the following:

- Function owner
- ALTER privilege on the function

For certain operations, non-superusers must also have the following schema privileges:

Operation	Schema privileges required
RENAME TO (rename function)	CREATE, USAGE
SET SCHEMA (move function to another schema)	• CREATE: destination schema • USAGE: current schema

See also

CREATE FILTER

ALTER FUNCTION (scalar)

Alters a user-defined scalar function .

Syntax

```
ALTER FUNCTION [[db-name.]schema.]function-name( [ parameter-list] ) {
   OWNER TO new-owner
   | RENAME TO new-name
   | SET FENCED boolean-expr
   | SET SCHEMA new-schema
}
```

Parameters

[*db-name.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function-name

Name of the function to alter.

parameter-list

Comma-delimited list of parameters that are defined for this function. If none, specify an empty list.

> **Note**
> Vertica supports function overloading, and uses the parameter list to identify the function to alter.

OWNER TO *new-owner*

Transfers function ownership to another user.

RENAME TO *new-name*

Renames this function.

SET FENCED { true | false }

Specifies whether to enable fenced mode for this function.

SET SCHEMA *new-schema*

Moves the function to another schema.

Privileges

Non-superuser: USAGE on the schema and one of the following:

- Function owner
- ALTER privilege on the function

For certain operations, non-superusers must also have the following schema privileges:

Operation	Schema privileges required
RENAME TO (rename function)	CREATE, USAGE
SET SCHEMA (move function to another schema)	• CREATE: destination schema • USAGE: current schema

Examples

Rename function UDF_one to UDF_two :

```
=> ALTER FUNCTION UDF_one (int, int) RENAME TO UDF_two;
```

Move function UDF_two to schema macros :

```
=> ALTER FUNCTION UDF_two (int, int) SET SCHEMA macros;
```

Disable fenced mode for function UDF_two :

```
=> ALTER FUNCTION UDF_two (int, int) SET FENCED false;
```

See also

CREATE FUNCTION (scalar)
ALTER FUNCTION (SQL)

Alters a user-defined SQL function.

Syntax

```
ALTER FUNCTION [[db-name.]schema.]function-name( [arg-list] ) {
   OWNER TO new-owner
   | RENAME TO new-name
   | SET SCHEMA new-schema
}
```

Parameters

[*db-name.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function-name

The name of the SQL function to alter.

arg-list

A comma-delimited list of function argument names. If none, specify an empty list.

OWNER TO *new-owner*

Transfers function ownership to another user.

RENAME TO *new-name*

Renames this function.

SET SCHEMA *new-schema*

Moves the function to another schema.

Privileges

Non-superuser: USAGE on the schema and one of the following:

- Function owner
- ALTER privilege on the function

For certain operations, non-superusers must also have the following schema privileges:

For these operations...	Schema privileges required...
RENAME TO (rename function)	CREATE, USAGE
SET SCHEMA (move function to another schema)	CREATE: destination schema USAGE: current schema

Examples

Rename function SQL_one to SQL_two :

```
=> ALTER FUNCTION SQL_one (int, int) RENAME TO SQL_two;
```

Move function SQL_two to schema macros :

```
=> ALTER FUNCTION SQL_two (int, int) SET SCHEMA macros;
```

Reassign ownership of SQL_two :

```
=> ALTER FUNCTION SQL_two (int, int) OWNER TO user1;
```

See also

- CREATE FUNCTION (SQL)
- User-defined SQL functions

ALTER PARSER

Alters a user-defined parser .

Syntax

```
ALTER PARSER [[db-name.]schema.]function-name( [ parameter-list ] ) {
   OWNER TO new-owner
   | RENAME TO new-name
   | SET FENCED boolean-expr
   | SET SCHEMA new-schema
}
```

Parameters

[*db-name.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function-name

Name of the function to alter.

parameter-list

Comma-delimited list of parameters that are defined for this function. If none, specify an empty list.

> **Note**
> Vertica supports function overloading, and uses the parameter list to identify the function to alter.

OWNER TO *new-owner*

Transfers function ownership to another user.

RENAME TO *new-name*

Renames this function.

SET FENCED { true | false }

Specifies whether to enable fenced mode for this function.

SET SCHEMA *new-schema*

Moves the function to another schema.

Privileges

Non-superuser: USAGE on the schema and one of the following:

- Function owner
- ALTER privilege on the function

For certain operations, non-superusers must also have the following schema privileges:

Operation	Schema privileges required
RENAME TO (rename function)	CREATE, USAGE
SET SCHEMA (move function to another schema)	• CREATE: destination schema • USAGE: current schema

See also

CREATE PARSER
ALTER SOURCE

Alters a user-defined load source function.

Syntax

```
ALTER SOURCE [[db-name.]schema.]function-name( [ parameter-list ] ) {
   OWNER TO new-owner
   | RENAME TO new-name
   | SET FENCED boolean-expr
   | SET SCHEMA new-schema
}
```

Parameters

[*db-name.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function-name

Name of the function to alter.

parameter-list

Comma-delimited list of parameters that are defined for this function. If none, specify an empty list.

> **Note**
> Vertica supports function overloading, and uses the parameter list to identify the function to alter.

OWNER TO *new-owner*

Transfers function ownership to another user.

RENAME TO *new-name*

Renames this function.

SET FENCED { true | false }

Specifies whether to enable fenced mode for this function.

SET SCHEMA *new-schema*

Moves the function to another schema.

Privileges

Non-superuser: USAGE on the schema and one of the following:

- Function owner
- ALTER privilege on the function

For certain operations, non-superusers must also have the following schema privileges:

Operation	Schema privileges required
RENAME TO (rename function)	CREATE, USAGE
SET SCHEMA (move function to another schema)	• CREATE: destination schema • USAGE: current schema

See also

CREATE SOURCE
ALTER TRANSFORM FUNCTION

Alters a user-defined transform function .

Syntax

```
ALTER TRANSFORM FUNCTION [[db-name.]schema.]function-name( [ parameter-list ] ) {
   OWNER TO new-owner
   | RENAME TO new-name
   | SET FENCED { true | false }
   | SET SCHEMA new-schema
}
```

Parameters

[*db-name.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function-name

Name of the function to alter.

parameter-list

Comma-delimited list of parameters that are defined for this function. If none, specify an empty list.

> **Note**
> Vertica supports function overloading, and uses the parameter list to identify the function to alter.

OWNER TO ***new-owner***

Transfers function ownership to another user.

RENAME TO ***new-name***

Renames this function.

SET FENCED { true | false }

Specifies whether to enable fenced mode for this function.

SET SCHEMA ***new-schema***

Moves the function to another schema.

Privileges

Non-superuser: USAGE on the schema and one of the following:

- Function owner
- ALTER privilege on the function

For certain operations, non-superusers must also have the following schema privileges:

Operation	Schema privileges required
RENAME TO (rename function)	CREATE, USAGE
SET SCHEMA (move function to another schema)	• CREATE: destination schema • USAGE: current schema

See also

CREATE TRANSFORM FUNCTION

ALTER HCATALOG SCHEMA

Alters parameter values on a schema that was created with CREATE HCATALOG SCHEMA . HCatalog schemas are used by the HCatalog Connector to access data stored in a Hive data warehouse. For more information, see Using the HCatalog Connector .

Some parameters cannot be altered after creation. If you need to change one of those values, delete and recreate the schema instead. You can use ALTER HCATALOG SCHEMA to change the following parameters:

- HOSTNAME
- PORT
- HIVESERVER2_HOSTNAME
- WEBSERVICE_HOSTNAME
- WEBSERVICE_PORT
- WEBHDFS_ADDRESS
- HCATALOG_CONNECTION_TIMEOUT
- HCATALOG_SLOW_TRANSFER_LIMIT
- HCATALOG_SLOW_TRANSFER_TIME
- SSL_CONFIG
- CUSTOM_PARTITIONS

Syntax

```
ALTER HCATALOG SCHEMA schema-name SET [param=value]+;
```

Parameters

Parameter	Description
schema-name	The name of the schema in the Vertica catalog to alter. The tables in the Hive database are available through this schema.

param	The name of the parameter to alter.
value	The new value for the parameter. You must specify a value; this statement does not read default values from configuration files like CREATE HCATALOG SCHEMA .

Privileges

One of the following:

- Superuser
- Schema owner

Examples

The following example shows how to change the Hive metastore hostname and port for the "hcat" schema. In this example, Hive uses High Availability metastore.

```
=> ALTER HCATALOG SCHEMA hcat SET HOSTNAME='thrift://ms1.example.com:9083,thrift://ms2.example.com:9083';
```

The following example shows the error you receive if you try to set an unalterable parameter.

```
=> ALTER HCATALOG SCHEMA hcat SET HCATALOG_USER='admin';
  ERROR 4856: Syntax error at or near "HCATALOG_USER" at character 39
```

ALTER LIBRARY

Replaces the library file that is currently associated with a UDx library in the Vertica catalog. Vertica automatically distributes copies of the updated file to all cluster nodes. UDxs defined in the catalog that reference the updated library automatically start using the updated library file. A UDx is considered to be the same if its name and signature match.

The current and replacement libraries must be written in the same language.

> **Caution**
> If a UDx function that is present in the original library is not present in the updated library, it is automatically dropped. This can result in loss of data if that function is in use, for example if a table depends on it to populate a column.

Syntax

```
ALTER LIBRARY [[database.]schema.]name [DEPENDS 'depends-path'] AS 'path';
```

Arguments

schema

Database and schema . The default schema is public . If you specify a database, it must be the current database.

name

The name of an existing library created with CREATE LIBRARY .

DEPENDS ' *depends-path* '

Files or libraries on which this library depends, one or more files or directories on the initiator node file system or other supported file systems or object stores . For a directory, end the path entry with a slash (/), optionally followed by a wildcard (*). To specify more than one file, separate entries with colons (:).
If any path entry contain colons, such as a URI, place brackets around the entire DEPENDS path and use double quotes for the individual path elements, as in the following example:

```
DEPENDS '["s3://mybucket/gson-2.3.1.jar"]'
```

To specify libraries with multiple directory levels, see Multi-level Library Dependencies .

DEPENDS has no effect for libraries written in R. R packages must be installed locally on each node, including external dependencies.

Important
The performance of CREATE LIBRARY can degrade in Eon Mode, in proportion to the number and depth of dependencies specified by the DEPENDS clause.

AS *path*

The absolute path on the initiator node file system of the replacement library file.

Privileges

Superuser, or UDXDEVELOPER and CREATE on the schema. Non-superusers must explicitly enable the UDXDEVELOPER role. See CREATE LIBRARY for examples.

Multi-level library dependencies

If a DEPENDS clause specifies a library with multiple directory levels, Vertica follows the library path to include all subdirectories of that library. For example, the following CREATE LIBRARY statement enables the UDx library mylib to import all Python packages and modules that it finds in subdirectories of site-packages :

```
=> CREATE LIBRARY mylib AS '/path/to/python_udx' DEPENDS '/path/to/python/site-packages' LANGUAGE 'Python';
```

Important

DEPENDS can specify Java library dependencies that are up to 100 levels deep.

Examples

This example shows how to update an already-defined library named myFunctions with a new file.

```
=> ALTER LIBRARY myFunctions AS '/home/dbadmin/my_new_functions.so';
```

See also

Developing user-defined extensions (UDxs)

ALTER LOAD BALANCE GROUP

Changes the configuration of a load balance group.

Syntax

```
ALTER LOAD BALANCE GROUP group-name {
   RENAME TO new-name |
   SET FILTER TO 'ip-cidr-addr' |
   SET POLICY TO 'policy' |
   ADD {ADDRESS | FAULT GROUP | SUBCLUSTER} add-list |
   DROP  {ADDRESS | FAULT GROUP | SUBCLUSTER} drop-list
}
```

Parameters

group-name

Name of an existing load balance group to change.

RENAME TO *new-name*

Renames the group to *new-name* .

SET FILTER TO ' *ip-cidr-addr* '

An IPv4 or IPv6 CIDR to replace the existing IP address filter that selects which members of a fault group or subcluster to include in the load balance group. This setting is only valid if the load balance group contains fault groups or subclusters.

SET POLICY TO ' *policy* '

Changes the policy the load balance group uses to select the target node for the incoming connection. One of:

- ROUNDROBIN
- RANDOM
- NONE

See CREATE LOAD BALANCE GROUP for details.

ADD {ADDRESS | FAULT GROUP | SUBCLUSTER }

Adds objects of the specified type to the load balance group. Load balance groups can only contain one type of object. For example, if you created the load balance group using a list of addresses, you can only add additional addresses, not fault groups or subclusters.

add-list

A comma-delimited list of objects (addresses, fault groups, or subclusters) to add to the fault group.

DROP {ADDRESS | FAULT GROUP | SUBCLUSTER}

Removes objects of the specified type from the load balance group (addresses, fault groups, or subclusters). The object type must match the type of the objects already in the load balance group.

drop-list

The list of objects to remove from the load balance group.

Privileges

Superuser

Examples

Remove an address from the load balance group named group_2.

```
=> SELECT * FROM LOAD_BALANCE_GROUPS;
  name   |   policy   | filter |        type          | object_name
---------+------------+--------+----------------------+-------------
 group_1 | ROUNDROBIN |        | Network Address Group | node01
 group_1 | ROUNDROBIN |        | Network Address Group | node02
 group_2 | ROUNDROBIN |        | Network Address Group | node03
(3 rows)

=> ALTER LOAD BALANCE GROUP group_2 DROP ADDRESS node03;
ALTER LOAD BALANCE GROUP

=> SELECT * FROM LOAD_BALANCE_GROUPS;
  name   |   policy   | filter |        type          | object_name
---------+------------+--------+----------------------+-------------
 group_1 | ROUNDROBIN |        | Network Address Group | node01
 group_1 | ROUNDROBIN |        | Network Address Group | node02
 group_2 | ROUNDROBIN |        | Empty Group           |
(3 rows)
```

The following example adds three network addresses to the group named group_2:

```
=> ALTER LOAD BALANCE GROUP group_2 ADD ADDRESS node01,node02,node03;
ALTER LOAD BALANCE GROUP
=> SELECT * FROM load_balance_groups WHERE name = 'group_2';
-[ RECORD 1 ]----------------------
name        | group_2
policy      | ROUNDROBIN
filter      |
type        | Network Address Group
object_name | node01
-[ RECORD 2 ]----------------------
name        | group_2
policy      | ROUNDROBIN
filter      |
type        | Network Address Group
object_name | node02
-[ RECORD 3 ]----------------------
name        | group_2
policy      | ROUNDROBIN
filter      |
type        | Network Address Group
object_name | node03
```

See also

- ALTER NETWORK ADDRESS
- ALTER ROUTING RULE
- CREATE LOAD BALANCE GROUP
- LOAD_BALANCE_GROUPS
- NETWORK_ADDRESSES

ALTER MODEL

Allows users to rename an existing model, change ownership, or move it to a another schema.

Syntax

```
ALTER MODEL [[database.]schema.]model
  { OWNER TO owner
    | RENAME TO new-name
    | SET SCHEMA schema
}
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

model

Identifies the model to alter.

OWNER TO *owner*

Reassigns ownership of this model to *owner* . If a non-superuser, you must be the current owner.

RENAME TO

Renames the mode, where *new-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

SET SCHEMA *schema*

Moves the model from one schema to another.

Privileges

Non-superuser: USAGE on the schema and one of the following:

- Model owner
- ALTER privilege on the model

For certain operations, non-superusers must have the following schema privileges:

Schema privileges required...	**For these operations...**
CREATE, USAGE	Rename model
CREATE: destination schema USAGE: current schema	Move model to another schema

Examples

See Altering models .

ALTER NETWORK ADDRESS

Changes the configuration of an existing network address.

Syntax

```
ALTER NETWORK ADDRESS name {
   RENAME TO new-name
  | SET TO 'ip-addr' [PORT port-number]
  | { ENABLE | DISABLE }
  }
```

Parameters

name

Name of an existing network address to change.

RENAME TO *new-name*

Renames the network address to *new-name* . This name change has no effect on the network address's membership in load balance groups.

SET TO ' *ip-addr* '

Changes the IP address assigned to the network address.

PORT *port-number*

Sets the port number for the network address. You must supply a network address when altering the port number.

ENABLE | DISABLE

Enables or disables the network address.

Examples

Rename the network address from test_addr to alt_node1 , then change its IP address to 192.168.1.200 with port number 4000 :

```
=> ALTER NETWORK ADDRESS test_addr RENAME TO alt_node1;
ALTER NETWORK ADDRESS
=> ALTER NETWORK ADDRESS alt_node1 SET TO '192.168.1.200' PORT 4000;
ALTER NETWORK ADDRESS
```

See also

- ALTER LOAD BALANCE GROUP
- ALTER ROUTING RULE
- CREATE LOAD BALANCE GROUP
- LOAD_BALANCE_GROUPS
- NETWORK_ADDRESSES

ALTER NETWORK INTERFACE

> **Deprecated**
> This statement has been deprecated. Instead, use ALTER NETWORK ADDRESS .

Renames a network interface.

Syntax

```
ALTER NETWORK INTERFACE network-interface-name RENAME TO new-network-interface-name
```

Parameters

network-interface-name

The name of the existing network interface.

new-network-interface-name

The new name for the network interface.

Privileges

Superuser

Examples

Rename a network interface:

```
=> ALTER NETWORK INTERFACE myNetwork RENAME TO myNewNetwork;
```

ALTER NODE

Sets and clears node-level configuration parameters on the specified node. ALTER NODE also performs the following management tasks:

- Changes the node type.
- Specifies the network interface of the public network on individual nodes that are used for import and export.
- Replaces a down node.

For information about removing a node, see

- Removing nodes from a database
- Removing hosts from a cluster

Syntax

```
ALTER NODE node-name {
   EXPORT ON { network-interface | DEFAULT }
   | [IS] node-type
   | REPLACE [ WITH standby-node ]
   | RESET
   | SET [PARAMETER] parameter=value[,...]
   | CLEAR [PARAMETER] parameter[,...]
}
```

Parameters

node-name

The name of the node to alter.

[IS] *node-type*

Changes the node type, where *node-type* is one of the following:

- PERMANENT: (default): A node that stores data.
- EPHEMERAL: A node that is in transition from one type to another—typically, from PERMANENT to either STANDBY or EXECUTE.
- STANDBY: A node that is reserved to replace any node when it goes down. A standby node stores no segments or data until it is called to replace a down node. When used as a replacement node, Vertica changes its type to PERMANENT. For more information, see Active standby nodes .
- EXECUTE: A node that is reserved for computation purposes only. An execute node contains no segments or data.

> **Note**
> STANDBY and EXECUTE node types are supported only in Enterprise Mode.

EXPORT ON

Specifies the network to use for importing and exporting data, one of the following:

- *network-interface* : The name of a network interface of the public network.
- DEFAULT : Use the default network interface of the public network, as specified by ALTER DATABASE .

REPLACE [WITH *standby-node*]

Enterprise Mode only, replaces the specified node with an available active standby node. If you omit the WITH clause, Vertica tries to find a replacement node from the same fault group as the down node.

If you specify a node that is not down, Vertica ignores this statement.

RESET

Enterprise Mode only, restores the specified down node and returns its replacement to standby status. If the down node cannot resume activity, Vertica ignores this statement and leaves the standby node in place.

SET [PARAMETER]

Sets one or more configuration parameters to the specified value at the node level.

CLEAR [PARAMETER]

Clears one or more specified configuration parameters.

Privileges

Superuser

Examples

Specify to use the default network interface of public network on v_vmart_node0001 for import/export operations:

```
=> ALTER NODE v_vmart_node0001 EXPORT ON DEFAULT;
```

Replace down node v_vmart_node0001 with an active standby node, then restore it:

```
=> ALTER NODE v_vmart_node0001 REPLACE WITH standby1;
...
=> ALTER NODE v_vmart_node0001 RESET;
```

Set and clear configuration parameter MaxClientSessions :

```
=> ALTER NODE v_vmart_node0001 SET MaxClientSessions = 0;
...
=> ALTER NODE v_vmart_node0001 CLEAR MaxClientSessions;
```

Set the node type as EPHEMERAL :

```
=> ALTER NODE v_vmart_node0001 IS EPHEMERAL;
```

ALTER NOTIFIER

Updates an existing notifier.

> **Note**
> To change the action URL associated with an existing identifier, drop the notifier and recreate it.

Syntax

```
ALTER NOTIFIER notifier-name
   [ ENABLE | DISABLE ]
   [ MAXPAYLOAD 'max-payload-size' ]
   [ MAXMEMORYSIZE 'max-memory-size' ]
   [ TLS CONFIGURATION tls-configuration ]
   [ TLSMODE 'tls-mode' ]
   [ CA BUNDLE bundle-name [ CERTIFICATE certificate-name ] ]
   [ IDENTIFIED BY 'uuid' ]
   [ [NO] CHECK COMMITTED ]
   [ PARAMETERS 'adapter-params' ]
```

Parameters

notifier-name

Specifies the notifier to update.

[NO] CHECK COMMITTED

Specifies to wait for delivery confirmation before sending the next message in the queue. Not all messaging systems support delivery confirmation.

ENABLE | DISABLE

Specifies whether to enable or disable the notifier.

MAXPAYLOAD

The maximum size of the message, up to 2 TB, specified in kilobytes, megabytes, gigabytes, or terabytes as follows:

```
MAXPAYLOAD integer{K|M|G|T}
```

The default setting is adapter-specific—for example, 1 M for Kafka.
Changes to this parameter take effect either after the notifier is disabled and reenabled or after the database restarts.

MAXMEMORYSIZE

The maximum size of the internal notifier, up to 2 TB, specified in kilobytes, megabytes, gigabytes, or terabytes as follows:

```
MAXMEMORYSIZE integer{K|M|G|T}
```

If the queue exceeds this size, the notifier drops excess messages.

TLS CONFIGURATION *tls-configuration*

The TLS CONFIGURATION to use for TLS.
Notifiers support the following TLS modes:

- DISABLE
- TRY_VERIFY (behaves like VERIFY_CA)
- VERIFY_CA
- VERIFY_FULL

If the notifier ACTION is 'syslog' or 'sns' , this parameter has no effect.

To encrypt messages sent to syslog, you must configure syslog for TLS.

To encrypt messages sent to an SNS endpoint, you must set the following configuration parameters:

- SNSCAFile or AWSCAFile
- SNSCAPath or AWSCAPath
- SNSEnableHttps

TLSMODE ' *tls-mode* '

> **Deprecated**
> This parameter has been superseded by the TLS CONFIGURATION parameter. If you use this parameter while the TLS CONFIGURATION parameter is not set, Vertica automatically creates a new TLS Configuration for the notifier uses the same values as the deprecated parameter.

Specifies the type of connection between the notifier and an endpoint, one of the following:

- disable (default): Plaintext connection.
- verify-ca : Encrypted connection, and the server's certificate is verified as being signed by a trusted CA.

If you set this parameter to verify-ca , the generated TLS Configuration will be set to TRY_VERIFY, which has the same behavior as VERIFY_CA.

If the notifier ACTION is 'syslog' or 'sns' , this parameter has no effect.

To encrypt messages sent to syslog, you must configure syslog for TLS.

To encrypt messages sent to an SNS endpoint, you must set the following configuration parameters:

- SNSCAFile or AWSCAFile
- SNSCAPath or AWSCAPath
- SNSEnableHttps

CA BUNDLE *bundle-name*

> **Deprecated**
> This parameter has been superseded by the TLS CONFIGURATION parameter. If you use this parameter while the TLS CONFIGURATION parameter is not set, Vertica automatically creates a new TLS Configuration for the notifier uses the same values as the deprecated parameter.

Specifies a CA bundle . The certificates inside the bundle are used to validate the Kafka server's certificate if the TLSMODE requires it.
If a CA bundle is specified for a notifier that currently uses disable , which doesn't validate the Kafka server's certificate, the bundle will go unused when connecting to the Kafka server. This behavior persists unless the TLSMODE is changed to one that validates server certificates.

Changes to contents of the CA bundle take effect either after the notifier is disabled and re-enabled or after the database restarts. However, changes to which CA bundle the notifier uses takes effect immediately.

If the notifier ACTION is 'syslog' or 'sns' , this parameter has no effect.

To encrypt messages sent to syslog, you must configure syslog for TLS.

To encrypt messages sent to an SNS endpoint, you must set the following configuration parameters:

- SNSCAFile or AWSCAFile
- SNSCAPath or AWSCAPath
- SNSEnableHttps

CERTIFICATE *certificate-name*

> **Deprecated**
> This parameter has been superseded by the TLS CONFIGURATION parameter. If you use this parameter while the TLS CONFIGURATION parameter is not set, Vertica automatically creates a new TLS Configuration for the notifier uses the same values as the deprecated parameter.

Specifies a client certificate for validation by the endpoint.
If the notifier ACTION is 'syslog' or 'sns' , this parameter has no effect.

To encrypt messages sent to syslog, you must configure syslog for TLS.

To encrypt messages sent to an SNS endpoint, you must set the following configuration parameters:

- SNSCAFile or AWSCAFile
- SNSCAPath or AWSCAPath
- SNSEnableHttps

IDENTIFIED BY ' *uuid* '

Specifies the notifier's unique identifier. If set, all the messages published by this notifier have this attribute.

PARAMETERS ' *adapter-params* '

Specifies one or more optional adapter parameters that are passed as a string to the adapter. Adapter parameters apply only to the adapter associated with the notifier.

Changes to this parameter take effect either after the notifier is disabled and reenabled or after the database restarts.
For Kafka notifiers, refer to Kafka and Vertica configuration settings .

Privileges

Superuser

Encrypted notifiers for SASL_SSL Kafka configurations

Follow this procedure to create or alter notifiers for Kafka endpoints that use SASL_SSL. Note that you must repeat this procedure whenever you change the TLSMODE, certificates, or CA bundle for a given notifier.

1. Create a TLS Configuration with the desired TLS mode, certificate, and CA certificates.
2. Use CREATE or ALTER to disable the notifier and set the TLS Configuration:
```
=> ALTER NOTIFIER encrypted_notifier
   DISABLE
   TLS CONFIGURATION kafka_tls_config;
```
3. ALTER the notifier and set the proper rdkafka adapter parameters for SASL_SSL:
```
=> ALTER NOTIFIER encrypted_notifier PARAMETERS
   'sasl.username=user;sasl.password=password;sasl.mechanism=PLAIN;security.protocol=SASL_SSL';
```
4. Enable the notifier:
```
=> ALTER NOTIFIER encrypted_notifier ENABLE;
```

Examples

Update the settings on an existing notifier:

```
=> ALTER NOTIFIER my_dc_notifier
   ENABLE
   MAXMEMORYSIZE '2G'
   IDENTIFIED BY 'f8b0278a-3282-4e1a-9c86-e0f3f042a971'
   CHECK COMMITTED;
```

Add a TLS Configuration to a notifier. To create a custom TLS Configuration, see TLS configurations :

```
=> ALTER NOTIFIER my_notifier TLS CONFIGURATION notifier_tls_config
```

See also

- CREATE NOTIFIER
- DROP NOTIFIER
- Monitoring Vertica using notifiers

ALTER PROCEDURE (stored)

Alters a stored procedure , retaining any existing grants .

Syntax

```
ALTER PROCEDURE procedure ( [ [ parameter_mode ] [ parameter ] parameter_type [, ...] ] )
  [ SECURITY { INVOKER | DEFINER }
    | RENAME TO new_procedure_name
    | OWNER TO new_owner
    | SET SCHEMA new_schema
    | SOURCE TO new_source
  ]
```

Parameters

procedure

The procedure to alter.

parameter_mode

The IN parameters of the stored procedure.

parameter

The name of the parameter.

parameter_type

The type of the parameter.

SECURITY { INVOKER | DEFINER }

Specifies whether to execute the procedure with the privileges of the invoker or its definer (owner).

For details, see Executing stored procedures .

RENAME TO *new_procedure_name*

The new name for the procedure.

OWNER TO *new_owner*

The new owner (definer) of the procedure.

SET SCHEMA *new_schema*

The new schema of the procedure.

SOURCE TO *new_source*

The new procedure source code. For details, see Scope and structure .

Privileges

OWNER TO

Superuser

RENAME and SCHEMA TO

Non-superuser:

- CREATE on the procedure's schema
- Ownership of the procedure

Other operations

Non-superuser: Ownership of the procedure

Examples

See Altering stored procedures .

ALTER PROFILE

Changes a profile . All parameters that are not set in a profile inherit their setting from the default profile. You can use ALTER PROFILE to change the default profile.

Syntax

```
ALTER PROFILE name LIMIT [
  PASSWORD_LIFE_TIME setting
  PASSWORD_MIN_LIFE_TIME setting
  PASSWORD_GRACE_TIME setting
  FAILED_LOGIN_ATTEMPTS setting
  PASSWORD_LOCK_TIME setting
  PASSWORD_REUSE_MAX setting
  PASSWORD_REUSE_TIME setting
  PASSWORD_MAX_LENGTH setting
  PASSWORD_MIN_LENGTH setting
  PASSWORD_MIN_LETTERS setting
  PASSWORD_MIN_UPPERCASE_LETTERS setting
  PASSWORD_MIN_LOWERCASE_LETTERS setting
  PASSWORD_MIN_DIGITS setting
  PASSWORD_MIN_SYMBOLS setting
  PASSWORD_MIN_CHAR_CHANGE setting ]
```

Parameters

> **Note**
> To reset a parameter to inherit from the default profile, set its value to default .

Name	Description
name	The name of the profile to create, where * name *conforms to conventions described in Identifiers . To modify the default profile, set *name* to default . For example: ALTER PROFILE DEFAULT LIMIT PASSWORD_MIN_SYMBOLS 1;
PASSWORD_LIFE_TIME	Set to an integer value, one of the following: • ≥ 1: The number of days a password remains valid. • UNLIMITED : Password remains valid indefinitely. After your password's lifetime and grace period expire, you must change your password on your next login, if you have not done so already.
PASSWORD_MIN_LIFE_TIME	Set to an integer value, one of the following: • Default: 0 • ≥ 1: The number of days a password must be set before it can be changed • UNLIMITED : Password can be reset at any time.
PASSWORD_GRACE_TIME	Set to an integer value, one of the following: • ≥ 1: The number of days a password can be used after it expires. • UNLIMITED : No grace period.
FAILED_LOGIN_ATTEMPTS	Set to an integer value, one of the following: • ≥ 1: The number of consecutive failed login attempts Vertica allows before locking your account. • UNLIMITED : Vertica allows an unlimited number of failed login attempts.

PASSWORD_LOCK_TIME	• ≥ 1: The number of days (units configurable with PasswordLockTimeUnit) a user's account is locked after FAILED_LOGIN_ATTEMPTS number of login attempts. The account is automatically unlocked when the lock time elapses. • UNLIMITED : Account remains indefinitely inaccessible until a superuser manually unlocks it.
PASSWORD_REUSE_MAX	Set to an integer value, one of the following: • ≥ 1: The number of times you must change your password before you can reuse an earlier password. • UNLIMITED : You can reuse an earlier password without any intervening changes.
PASSWORD_REUSE_TIME	Set to an integer value, one of the following: • ≥ 1: The number of days that must pass after a password is set before you can reuse it. • UNLIMITED : You can reuse an earlier password immediately.
PASSWORD_MAX_LENGTH	The maximum number of characters allowed in a password, one of the following: • Integer between 8 and 512, inclusive
PASSWORD_MIN_LENGTH	The minimum number of characters required in a password, one of the following: • 0 to PASSWORD_MAX_LENGTH • UNLIMITED : Minimum of PASSWORD_MAX_LENGTH
PASSWORD_MIN_LETTERS	Minimum number of letters (a-z and A-Z) that must be in a password, one of the following: • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)
PASSWORD_MIN_UPPERCASE_LETTERS	Minimum number of uppercase letters (A-Z) that must be in a password, one of the following: • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)
PASSWORD_MIN_LOWERCASE_LETTERS	Minimum number of lowercase letters (a-z) that must be in a password, one of the following: • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)
PASSWORD_MIN_DIGITS	Minimum number of digits (0-9) that must be in a password, one of the following: • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)
PASSWORD_MIN_SYMBOLS	Minimum number of symbols—printable non-letter and non-digit characters such as $, #, @—that must be in a password, one of the following: • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)

PASSWORD_MIN_CHAR_CHANGE	Minimum number of characters that must be different from the previous password: • Default: 0 • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)

Privileges

Superuser

Profile settings and client authentication

The following profile settings affect client authentication methods , such as LDAP or GSS:

- FAILED_LOGIN_ATTEMPTS
- PASSWORD_LOCK_TIME

All other profile settings are used only by Vertica to manage its passwords.

Examples

```
ALTER PROFILE sample_profile LIMIT FAILED_LOGIN_ATTEMPTS 3;
```

See also

- CREATE PROFILE
- DROP PROFILE
- Creating a database name and password

ALTER PROFILE RENAME

Rename an existing profile.

Syntax

```
ALTER PROFILE name RENAME TO new-name;
```

Parameters

name

The current name of the profile.

new-name

The new name for the profile.

Privileges

Superuser

Examples

This example shows how to rename an existing profile.

```
ALTER PROFILE sample_profile RENAME TO new_sample_profile;
```

See also

- ALTER PROFILE
- CREATE PROFILE
- DROP PROFILE

ALTER PROJECTION

Changes the DDL of the specified projection.

Syntax

```
ALTER PROJECTION [[database.]schema.]projection
  { RENAME TO new-name | ON PARTITION RANGE BETWEEN min-val AND max-val | { ENABLE | DISABLE } }
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection

The projection to change, where *projection* can be one of the following:

- Projection base name: Rename all projections that share this base name.
- Projection name: Rename the specified projection and its base name. If the projection is segmented, its buddies are unaffected by this change.

See Projection naming for projection name conventions.

RENAME TO *new-name*

The new projection name.

ON PARTITION RANGE

> **Note**
> Valid only for projections that were created with a partition range.

Specifies to limit data of this projection to a range of partition keys, specified as follows:

```
ON PARTITION RANGE BETWEEN min-range-value AND max-range-value
```

where the following requirements apply to *min-range-value* and ≤ *max-range-value* :

- *min-range-value* must be ≤ *max-range-value*
- They must resolve to a data type that is compatible with the table partition expression.
- They can be:
  - String literals—for example, 2021-07-31
  - Expressions with stable or immutable functions, for example:

```
date_trunc('month', now()::timestamp - interval'1 month')
```

max-range-value can be set to NULL, to specify that the partition range has no upper bound.

min-range-value can be set to NULL, to specify that the partition range has no lower bound.

If both partition range projection *min-range-value* and *max-range-value* are set to NULL, it will drop the projection endpoints, becoming a regular projection.

If the new range of keys is outside the previous range, Vertica throws a warning that the projection is out of date and must be refreshed before it can be used.

For other requirements and usage details, see Partition range projections .

ENABLE | DISABLE

Specifies whether to mark this projection as unavailable for queries on its anchor table. If a projection is the queried table's only superprojection, attempts to disable it return with a rollback message. ENABLE restores the projection's availability to query planning. You can also mark a projection as unavailable for individual queries using the hint SKIP_PROJS .

Default: ENABLE

Privileges

Non-superuser, CREATE and USAGE on the schema and one of the following anchor table privileges:

- Table owner
- ALTER privilege
- SELECT privilege only if defining a partition range projection

Syntactic sugar

The statement

```
=> ALTER PROJECTION foo REMOVE PARTITION RANGE;
```

has the same effect as

```
=> ALTER PROJECTION foo ON PARTITION RANGE BETWEEN NULL AND NULL;
```

Examples

```
=> SELECT export_tables('','public.store_orders');

               export_tables
--------------------------------------------

CREATE TABLE public.store_orders
(
    order_no int,
    order_date timestamp NOT NULL,
    shipper varchar(20),
    ship_date date NOT NULL
);
(1 row)

=> CREATE PROJECTION store_orders_p AS SELECT * from store_orders;
CREATE PROJECTION
=> ALTER PROJECTION store_orders_p RENAME to store_orders_new;
ALTER PROJECTION
=> ALTER PROJECTION store_orders_new DISABLE;
=> SELECT * FROM store_orders_new;
ERROR 3586:  Insufficient projections to answer query
DETAIL:  No projections eligible to answer query
HINT:  Projection store_orders_new not used in the plan because the projection is disabled.
=> ALTER PROJECTION store_orders_new ENABLE;
```

See also

CREATE PROJECTION

ALTER RESOURCE POOL

Modifies an existing resource pool by setting one or more parameters.

You can use ALTER RESOURCE POOL to modify some parameters in Vertica built-in resource pools. For details on default settings and restrictions, see Built-in resource pools configuration .

Important

Changes to parameters of the built-in GENERAL resource pool take effect only when the database restarts.

Syntax

```
ALTER RESOURCE POOL pool-name [ FOR subcluster ] parameter-name setting[...]
```

Arguments

pool-name

Name of the resource pool to modify.

FOR *subcluster*

Eon Mode only, the subcluster to associate with this resource pool, where *subcluster* is one of the following:

- SUBCLUSTER *subcluster-name* : Resource pool for an existing subcluster. You cannot be connected to this subcluster, otherwise Vertica returns an error.
- CURRENT SUBCLUSTER : Resource pool for the subcluster that you are connected to.

Important

You cannot use ALTER RESOURCE POOL to convert a global resource pool to a subcluster-level resource pool.

parameter-name setting

A resource pool parameter and its new setting. To reset this parameter to its default value, specify DEFAULT .

If you specify a subcluster, you can alter only the MAXMEMORYSIZE , MAXQUERYMEMORYSIZE , and MEMORYSIZE parameters for built-in pools .

Parameters

Note
Default values specified here pertain only to user-defined resource pools. For built-in pool default values, see Built-in resource pools configuration , or query system table RESOURCE_POOL_DEFAULTS .

CASCADE TO

Secondary resource pool for executing queries that exceed the RUNTIMECAP setting of their assigned resource pool:

```
CASCADE TO secondary-pool
```

CPUAFFINITYMODE

Specifies whether the resource pool has exclusive or shared use of the CPUs specified in CPUAFFINITYSET :

```
CPUAFFINITYMODE { SHARED | EXCLUSIVE | ANY }
```

- SHARED : Queries that run in this resource pool share its CPUAFFINITYSET CPUs with other Vertica resource pools.
- EXCLUSIVE : Dedicates CPUAFFINITYSET CPUs to this resource pool only, and excludes other Vertica resource pools. If CPUAFFINITYSET is set as a percentage, then that percentage of CPU resources available to Vertica is assigned solely for this resource pool.
- ANY : Queries in this resource pool can run on any CPU, invalid if CPUAFFINITYSET designates CPU resources.

Default : ANY

CPUAFFINITYSET

CPUs available to this resource pool. All cluster nodes must have the same number of CPUs. The CPU resources assigned to this set are unavailable to general resource pools.

```
CPUAFFINITYSET {
 'cpu-index[,...]'
| 'cpu-indexi-cpu-indexn'
| 'integer%'
| NONE
}
```

- *cpu-index* [,...] : Dedicates one or more comma-delimited CPUs to this resource pool.
- *cpu-indexi-cpu-indexn* : Dedicates a range of contiguous CPU indexes *i* through *n* to this resource pool.
- *integer* % : Percentage of all available CPUs to use for this resource pool. Vertica rounds this percentage down to include whole CPU units.
- NONE (empty string): No affinity set is assigned to this resource pool. Queries associated with this pool are executed on any CPU.

Default : NONE

Important
CPUAFFINITYSET and CPUAFFINITYMODE must be set together in the same statement.

EXECUTIONPARALLELISM

Number of threads used to process any single query issued in this resource pool.

```
EXECUTIONPARALLELISM { limit | AUTO }
```

- *limit* : An integer value between 1 and the number of cores. Setting this parameter to a reduced value increases throughput of short queries issued in the resource pool, especially if queries are executed concurrently.
- AUTO or 0 : Vertica calculates the setting from the number of cores, available memory, and amount of data in the system. Unless memory is limited, or the amount of data is very small, Vertica sets this parameter to the number of cores on the node.

Default : AUTO

MAXCONCURRENCY

Maximum number of concurrent execution slots available to the resource pool across the cluster:

```
MAXCONCURRENCY { integer | NONE }
```

NONE (empty string): Unlimited number of concurrent execution slots.

Default : NONE

MAXMEMORYSIZE

Maximum size per node the resource pool can grow by borrowing memory from the GENERAL pool:

```
MAXMEMORYSIZE {
 'integer%'
 |'integer{K|M|G|T}'
 NONE
}
```

- *integer* % : Percentage of total memory
- *integer* {K|M|G|T} : Amount of memory in kilobytes, megabytes, gigabytes, or terabytes
- NONE (empty string): Unlimited, resource pool can borrow any amount of available memory from the GENERAL pool.

Default : NONE

MAXQUERYMEMORYSIZE

Maximum amount of memory this resource pool can allocate at runtime to process a query. If the query requires more memory than this setting, Vertica stops execution and returns an error.
Set this parameter as follows:

```
MAXQUERYMEMORYSIZE {
 'integer%'
| 'integer{K|M|G|T}'
| NONE
}
```

- *integer* % : Percentage of MAXMEMORYSIZE for this resource pool.
- *integer* {K|M|G|T} : Amount of memory in kilobytes, megabytes, gigabytes, or terabytes, up to the value of MAXMEMORYSIZE .
- NONE (empty string): Unlimited; resource pool can borrow any amount of available memory from the GENERAL pool, within the limits set by MAXMEMORYSIZE .

Default : NONE

Important
Changes to MAXQUERYMEMORYSIZE are applied retroactively to queries that are currently executing. If you reduce this setting, queries that were budgeted with the previous memory size are liable to fail if they try to allocate more memory than the new setting allows.

MEMORYSIZE

Total per-node memory available to the Vertica resource manager that is allocated to this resource pool:

```
MEMORYSIZE {
 'integer%'
| 'integer{K|M|G|T}'
}
```

- *integer* % : Percentage of total memory
- *integer* {K|M|G|T} : Amount of memory in kilobytes, megabytes, gigabytes, or terabytes

Default: 0%. No memory allocated, the resource pool borrows memory from the GENERAL pool.

PLANNEDCONCURRENCY

Preferred number of queries to execute concurrently in the resource pool. This setting applies to the entire cluster:

```
PLANNEDCONCURRENCY { num-queries | AUTO }
```

- *num-queries* : Integer value ≥ 1, the preferred number of queries to execute concurrently in the resource pool. When possible, query resource budgets are limited to allow this level of concurrent execution.
- AUTO : Value is calculated automatically at query runtime. Vertica sets this parameter to the lower of these two calculations, but never less than 4:
  - Number of logical cores
  - Memory divided by 2GB

If the number of logical cores on each node is different, AUTO is calculated differently for each node. Distributed queries run like the minimal effective planned concurrency. Single node queries run with the planned concurrency of the initiator.

Default : AUTO

> **Tip**
> Change this parameter only after evaluating performance over a period of time.

PRIORITY

Priority of queries in this resource pool when they compete for resources in the GENERAL pool:

```
PRIORITY { integer | HOLD }
```

- *integer* : Negative or positive integer value, where higher numbers denote higher priority:
  - User-defined resource pool: -100 to 100
  - Built-in resource pools SYSQUERY , RECOVERY , and TM : -110 to 110
- HOLD : Sets priority to -999 . Queries in this resource pool are queued until QUEUETIMEOUT is reached.

Default : 0

QUEUETIMEOUT

Maximum time a request can wait for pool resources before it is rejected, not more than one year:

```
QUEUETIMEOUT { integer | 'interval' | 'NONE' }
```

- *integer* : Maximum wait time in seconds
- *[interval](/sql-reference/language-elements/literals/datetime-literals/interval-literal.html)* : Maximum wait time expressed in the following format:

  ```
  num year num months num [days] HH:MM:SS.ms
  ```

- NONE (empty string): No maximum wait time, request can be queued indefinitely, up to one year.

If the value that you specify resolves to more than one year, Vertica returns with a warning and sets the parameter to 365 days:

```
=> ALTER RESOURCE POOL user_0 QUEUETIMEOUT '11 months 50 days 08:32';
WARNING 5693:  Using 1 year for QUEUETIMEOUT
ALTER RESOURCE POOL
=> SELECT QUEUETIMEOUT FROM resource_pools WHERE name = 'user_0';
 QUEUETIMEOUT
--------------
 365
(1 row)
```

Default : **00:05 (5 minutes)**

RUNTIMECAP

Maximum execution time allowed to queries in this resource pool, not more than one year, otherwise Vertica returns with an error. If a query exceeds this setting, it tries to cascade to a secondary pool:

```
RUNTIMECAP { 'interval' | NONE }
```

- *interval* : Maximum wait time expressed in the following format:

  ```
  num year num month num [day] HH:MM:SS.ms
  ```

- NONE (empty string): No maximum wait time, request can be queued indefinitely, up to one year.

If the user or session also has a RUNTIMECAP , the shorter limit applies.

RUNTIMEPRIORITY

Determines how the resource manager should prioritize dedication of run-time resources (CPU, I/O bandwidth) to queries already running in this resource pool:

```
RUNTIMEPRIORITY { HIGH | MEDIUM | LOW }
```

Default : MEDIUM

RUNTIMEPRIORITYTHRESHOLD

Maximum time (in seconds) in which query processing must complete before the resource manager assigns to it the resource pool's RUNTIMEPRIORITY . All queries begin execution with a priority of HIGH.

```
RUNTIMEPRIORITYTHRESHOLD seconds
```

Default : 2

SINGLEINITIATOR

Set to false for backward compatibility. Do not change this setting.

Privileges

Superuser

Examples

Set resource pool PRIORITY to 5:

```
=> ALTER RESOURCE POOL ceo_pool PRIORITY 5;
```

Designate a secondary resource pool:

```
=> CREATE RESOURCE POOL second_pool;
=> ALTER RESOURCE POOL ceo_pool CASCADE TO second_pool;
```

Decrease to 0% the MAXMEMORYSIZE and MEMORYSIZE settings on the dashboard subcluster's built-in TM resource pool . Changing these settings to 0 prevents the subcluster from running mergeout operations:

```
=> ALTER RESOURCE POOL TM FOR SUBCLUSTER dashboard MEMORYSIZE '0%'
  MAXMEMORYSIZE '0%';
```

See Tuning tuple mover pool settings for more information.

See also

- CREATE RESOURCE POOL
- DROP RESOURCE POOL
- CREATE USER
- RESOURCE_POOL_STATUS
- SET SESSION RESOURCE_POOL
- SET SESSION MEMORYCAP
- Managing workloads

ALTER ROLE

Renames an existing role .

> **Note**
> You cannot use ALTER ROLE to rename a role that was added to the Vertica database with the LDAPLink service.

Syntax

```
ALTER ROLE name RENAME TO new-name
```

Parameters

name

The role to rename.

new-name

The role's new name.

Privileges

Superuser

Examples

```
=> ALTER ROLE applicationadministrator RENAME TO appadmin;
ALTER ROLE
```

See also

- CREATE ROLE
- DROP ROLE

ALTER ROUTING RULE

Changes an existing load balancing policy routing rule.

Syntax

```
ALTER ROUTING RULE rule_name {
   RENAME TO new_name |
   SET ROUTE TO 'cidr_range'|
   SET GROUP TO group_name
   }
```

Parameters

rule_name

The name of the existing routing rule to change.

RENAME TO new_name

Changes the name of the routing rule to *new_name* .

SET ROUTE TO ' *cidr_range* '

An IPv4 or IPv6 address range in CIDR format. Changes the address range of client connections this rule applies to.

SET GROUP TO *group_name*

Changes the load balancing group that handles the connections that match this rule .

Examples

This example changes the routing rule named etl_rule so it uses the load balancing group named etl_rule to handle incoming connections in the IP address range of 10.20.100.0 to 10.20.100.255.

```
=> ALTER ROUTING RULE etl_rule SET GROUP TO etl_group;
ALTER ROUTING RULE
=> ALTER ROUTING RULE etl_rule SET ROUTE TO '10.20.100.0/24';
ALTER ROUTING RULE
=> \x
Expanded display is on.
=> SELECT * FROM routing_rules WHERE NAME = 'etl_rule';
-[ RECORD 1 ]----+---------------
name             | etl_rule
source_address   | 10.20.100.0/24
destination_name | etl_group
```

See also

- ALTER LOAD BALANCE GROUP
- ALTER NETWORK ADDRESS
- CREATE LOAD BALANCE GROUP
- LOAD_BALANCE_GROUPS
- NETWORK_ADDRESSES

ALTER SCHEDULE

Modifies a schedule .

Syntax

```
ALTER SCHEDULE [[database.]schema.]schedule {
        OWNER TO new_owner
      | SET SCHEMA new_schema
      | RENAME TO new_schedule
      | USING CRON new_cron_expression
      | USING DATETIMES new_timestamp_list
   }
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

schedule

The schedule to modify.

new_owner

The new owner of the schedule.

new_schema

The new schema of the schedule.

new_schedule

The new name for the schedule.

new_cron_expression

A cron expression. For example, to execute every day at 1:00PM:

```
=> ALTER SCHEDULE sched1 USING CRON '0 13 * * *';
```

new_timestamp_list

A comma-separated list of timestamps. For example, to execute at noon on October 2nd and November 2nd 2022:

```
=> ALTER SCHEDULE sched2 USING DATETIMES('2022-10-02 12:00:00', '2022-11-02 12:00:00');
```

Privileges

Superuser

Examples

To change the cron expression for a schedule:

```
=> ALTER SCHEDULE daily_schedule USING CRON '0 8 * * *';
```

To change a schedule that uses a cron expression to use a timestamp list instead:

```
=> ALTER SCHEDULE my_schedule USING DATETIMES('2023-10-01 12:30:00', '2022-11-01 12:30:00');
```

To rename a schedule:

```
=> ALTER SCHEDULE daily_schedule RENAME TO daily_8am_gmt;
```

ALTER SCHEMA

Changes one or more schemas in one of the following ways:

- Enable or disable inheritance of schema privileges by tables created in the schemas.
- Reassign schema ownership to another user.
- Change schema disk quota.
- Rename one or more schemas.

Syntax

```
ALTER SCHEMA [database.]schema
   DEFAULT {INCLUDE | EXCLUDE} SCHEMA PRIVILEGES
   | OWNER TO user-name [CASCADE]
   | DISK_QUOTA { value | SET NULL }
```

You can rename more than one schema in a single operation:

```
ALTER SCHEMA [database.]schema[,...] RENAME TO new-schema-name[,...]
```

Parameters

database

Name of the database containing the schema. If specified, it must be the current database.

schema

Name of the schema to modify.

DEFAULT {INCLUDE | EXCLUDE} SCHEMA PRIVILEGES

Specifies whether to enable or disable default inheritance of privileges for new tables in the specified schema:

- EXCLUDE SCHEMA PRIVILEGES (default): Disables inheritance of schema privileges.
- INCLUDE SCHEMA PRIVILEGES : Specifies to grant tables in the specified schema the same privileges granted to that schema. This option has no effect on existing tables in the schema.

See also Enabling schema inheritance .

OWNER TO

Reassigns schema ownership to the specified user:

```
OWNER TO user-name [CASCADE]
```

By default, ownership of objects in the reassigned schema remain unchanged. To reassign ownership of schema objects to the new schema owner, qualify the OWNER TO clause with CASCADE . For details, see Cascading Schema Ownership below.

DISK_QUOTA

One of the following:

- A string, an integer followed by a supported unit: K, M, G, or T. If the new value is smaller than the current usage, the operation succeeds but no further disk space can be used until usage is reduced below the new quota.
- SET NULL to remove a quota.

For more information, see Disk quotas .

RENAME TO

Renames one or more schemas:

```
RENAME TO new-schema-name[,...]
```

The following requirements apply:

- The new schema name conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, models, and schemas in the database.
- If you specify multiple schemas to rename, the source and target lists must have the same number of names.

> **Caution**
> Renaming a schema referenced by a view causes the view to fail unless another schema is created to replace it.

Privileges

One of the following:

- Superuser
- Schema owner

Cascading schema ownership

By default, ALTER SCHEMA...OWNER TO does not affect ownership of objects in the target schema or the privileges granted on them. If you qualify the OWNER TO clause with CASCADE , Vertica acts as follows on objects in the target schema:

- Transfers ownership of objects owned by the previous schema owner to the new owner.
- Revokes all object privileges granted by the previous schema owner.

If issued by non-superusers, ALTER SCHEMA...OWNER TO CASCADE ignores all objects that belong to other users, and returns with notices on the objects that it cannot change. For example:

1. Schema ms is owned by user mayday , and contains two tables: ms.t1 owned by mayday, and ms.t2 owned by user joe :

```
=> \dt
                    List of tables
   Schema    |        Name        | Kind  |  Owner  | Comment
----------------+-----------------------+-------+---------+---------
 ms          | t1                 | table | mayday  |
 ms          | t2                 | table | joe     |
```

2. User mayday transfers ownership of schema ms to user dbadmin , using CASCADE . On return, ALTER SCHEMA reports that it cannot transfer ownership of table ms.t2 and its projections, which are owned by user joe :

```
=> \c - mayday
You are now connected as user "mayday".
=> ALTER SCHEMA ms OWNER TO dbadmin CASCADE;
NOTICE 3583:  Insufficient privileges on ms.t2
NOTICE 3583:  Insufficient privileges on ms.t2_b0
NOTICE 3583:  Insufficient privileges on ms.t2_b1
ALTER SCHEMA
=> \c
You are now connected as user "dbadmin".
=> \dt
                    List of tables
   Schema    |        Name        | Kind  |  Owner  | Comment
----------------+-----------------------+-------+---------+---------
 ms          | t1                 | table | dbadmin |
 ms          | t2                 | table | joe     |
```

3. User dbadmin transfers ownership of schema ms to user pat , again using CASCADE . This time, because dbadmin is a superuser, ALTER SCHEMA transfers ownership of all ms tables to user pat

```
=> ALTER SCHEMA ms OWNER TO pat CASCADE;
ALTER SCHEMA
=> \dt
                    List of tables
   Schema    |        Name        | Kind  |  Owner  | Comment
----------------+-----------------------+-------+---------+---------
 ms          | t1                 | table | pat     |
 ms          | t2                 | table | pat     |
```

Swapping schemas

Renaming schemas is useful for swapping schemas without actually moving data. To facilitate the swap, enter a non-existent, temporary placeholder schema. For example, the following ALTER SCHEMA statement uses the temporary schema *temps* to facilitate swapping schema S1 with schema S2. In this example, *S1* is renamed to *temps* . Then *S2* is renamed to *S1* . Finally, *temps* is renamed to *S2* .

```
=> ALTER SCHEMA S1, S2, temps RENAME TO temps, S1, S2;
```

Examples

The following example renames schemas S1 and S2 to S3 and S4 , respectively:

```
=> ALTER SCHEMA S1, S2 RENAME TO S3, S4;
```

This example sets the default behavior for new table t2 to automatically inherit the schema's privileges:

```
=> ALTER SCHEMA s1 DEFAULT INCLUDE SCHEMA PRIVILEGES;

=> CREATE TABLE s1.t2 (i, int);
```

This example sets the default for new tables to not automatically inherit privileges from the schema:

```
=> ALTER SCHEMA s1 DEFAULT EXCLUDE SCHEMA PRIVILEGES;
```

See also

- CREATE SCHEMA
- DROP SCHEMA

ALTER SEQUENCE

Changes a sequence in two ways:

- Resets parameters that control sequence behavior—for example, its start value, and range of minimum and maximum values. These changes take effect only when you start a new database session.
- Resets sequence name, schema, or ownership. These changes take effect immediately.

Syntax

Change sequence behavior:

```
ALTER SEQUENCE [[database.]schema.]sequence
   [ INCREMENT [ BY ] integer ]
   [ MINVALUE integer | NO MINVALUE ]
   [ MAXVALUE integer | NO MAXVALUE ]
   [ RESTART [ WITH ] integer ]
   [ CACHE integer | NO CACHE ]
   [ CYCLE | NO CYCLE ]
```

Change sequence name, schema, or ownership:

```
ALTER SEQUENCE [schema.]sequence-name {
   RENAME TO seq-name
   | SET SCHEMA schema-name]
   | OWNER TO owner-name
}
```

Parameters

schema

Database and schema . The default schema is public . If you specify a database, it must be the current database.
If you do not specify a schema, the table is created in the default schema.

sequence

Name of the sequence to alter.

In the case of IDENTITY table columns, Vertica generates the sequence name using the following convention:

```
table-name_col-name_seq
```

To obtain this name, query the SEQUENCES system table.

INCREMENT

Positive or negative integer that specifies how much to increment or decrement the sequence on each call to NEXTVAL , by default set to 1.

> **Note**
> Setting this parameter to *integer* guarantees that column values always increment by at least *integer* . However, column values can sometimes increment by more than *integer* unless you also set the NO CACHE parameter.

MINVALUE|NO MINVALUE

Maximum integer value of the sequence. Vertica automatically changes the sequence value in two cases:

- Ascending sequence: If *currentSequenceValue* < *newMinValue* , sequence value resets to *newMinValue* .
- Descending sequence: If *currentSequenceValue* < *newMinValue* , sequence value cycles back to MAXVALUE .

MAXVALUE|NO MAXVALUE

Maximum integer value of the sequence. Vertica automatically changes the sequence value in two cases:

- Ascending sequence: If *currentSequenceValue* > *newMaxValue* , sequence value cycles back to MINVALUE .
- Descending sequence: If *currentSequenceValue* > *newMaxValue* , sequence value resets to to *newMaxValue* .

RESTART

New integer start value of the sequence. The next call to NEXTVAL returns the new start value.

> **Caution**
> Using ALTER SEQUENCE to set a sequence start value below its current value can result in duplicate keys.

CACHE|NO CACHE

Whether to cache unique sequence numbers on each node for faster access. CACHE takes an integer argument as follows:

- >1 specifies how many unique sequence numbers are pre-allocated and stored in memory for faster access. Vertica sets up caching for each session, and distributes it across all nodes.

  > **Caution**
  > If sequence caching is set to a low number, nodes are liable to request a new set of cache values more frequently. While it supplies a new cache, Vertica must lock the catalog. Until Vertica releases the lock, other database activities such as table inserts are blocked, which can adversely affect overall performance.

- 0 or 1 specifies to disable caching (equivalent to NO CACHE).

By default, the sequence cache is set to 250,000.

For details, see Distributing sequences .

CYCLE|NO CYCLE

Specifies whether the sequence can wrap when its minimum or maximum values are reached:

- CYCLE : The sequence wraps as follows:
  - When an incrementing sequence reaches its upper limit, it is reset to its minimum value.
  - When an decrementing sequence reaches its lower limit, it is reset to its maximum value.
- NO CYCLE (default): Calls to NEXTVAL return an error after the sequence reaches its maximum or minimum value.

RENAME TO

Supported only for named sequences , renames a sequence within the current schema, where *seq-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

SET SCHEMA

Supported only for named sequences, moves the sequence to schema *schema-name* .

OWNER TO

Reassigns sequence ownership to another user.

Privileges

For named sequences , USAGE on the schema and one of the following:

- Sequence owner
- ALTER privilege on the sequence

For certain operations, non-superusers must have the following schema privileges:

Schema privileges required...	For these operations...
CREATE, USAGE	Rename sequence
CREATE: destination schema USAGE: current schema	Move sequence to another schema

For IDENTITY column sequences, USAGE on the table schema and one of the following:

- Table owner
- ALTER privileges

Non-superusers must also have SELECT privileges to enable or disable constraint enforcement , or remove partitioning.

Examples

See Altering sequences .

See also

CREATE SEQUENCE

ALTER SESSION

ALTER SESSION sets and clears session-level configuration parameter values for the current session. To identify session-level parameters, query system table CONFIGURATION_PARAMETERS .

Syntax

```
ALTER SESSION {
   SET [PARAMETER] parameter-name=value[,...]
   | CLEAR { [PARAMETER] parameter-name[,...] | PARAMETER ALL }
   | SET UDPARAMETER [ FOR libname ] key=value[,...]
   | CLEAR UDPARAMETER { [ FOR libname ] key[,...] | ALL }
}
```

Parameters

SET [PARAMETER]

Sets one or more configuration parameters to the specified value.

CLEAR [PARAMETER]

Clears the specified configuration parameters of changes that were set in the current session.

CLEAR PARAMETER ALL

Clears all session-level configuration parameters of changes that were set in the current session.

SET UDPARAMETER

Sets one or more user-defined session parameters (*key* = *value*) to be used with a UDx. Key value sizes are restricted as follows:

- Set from client side: 128 characters
- Set from UDx side: unlimited

You can limit the SET operation's scope to a single library by including the clause FOR *libname* . For example:

```
=> ALTER SESSION SET UDPARAMETER FOR securelib username='alice';
```

If you specify a library, then only that library can access the parameter's value. Use this restriction to protect parameters that hold sensitive data, such as credentials.

CLEAR UDPARAMETER

Clears user-defined parameters, specified by one of the following options:

- [FOR *libname*] *key* [,...] : Clears the *key* -specified parameters, optionally scoped to library *libname* .
- ALL : Clears all user-defined parameters in the current session.

Privileges

None

Examples

Set and clear a parameter

- Force all UDxes that support fenced mode to run in fenced mode, even if their definition specifies NOT FENCED :

```
=> ALTER SESSION SET ForceUDxFencedMode = 1;
ALTER SESSION
```

- Clear ForceUDxFencedMode at the session level. Its value is reset to its default value 0 :

```
=> ALTER SESSION CLEAR ForceUDxFencedMode;
ALTER SESSION
=> SELECT parameter_name, current_value, default_value FROM configuration_parameters WHERE  parameter_name = 'ForceUDxFencedMode';
   parameter_name   | current_value | default_value
--------------------+---------------+---------------
 ForceUDxFencedMode | 0             | 0
(1 row)
```

- Clear all session-level configuration parameters of changes that were set in this session:

```
=> ALTER SESSION CLEAR PARAMETER ALL;
ALTER SESSION
```

Set and clear a user-defined parameter

- Set the value of user-defined parameter RowCount in library MyLibrary to 25.

```
=> ALTER SESSION SET UDPARAMETER FOR MyLibrary RowCount = 25;
ALTER SESSION
```

- Clear RowCount at the session level:

```
=> ALTER SESSION CLEAR UDPARAMETER FOR MyLibrary RowCount;
ALTER SESSION
```

ALTER SUBCLUSTER

Changes the configuration of a subcluster. You can use this statement to rename a subcluster or make it the default subcluster .

Syntax

```
ALTER SUBCLUSTER subcluster-name {
   RENAME TO new-name |
   SET DEFAULT
}
```

Parameters

subcluster-name

The name of the subcluster to alter.

RENAME TO *new-name*

Changes the name of the subcluster to *new-name* .

SET DEFAULT

Makes the subcluster the default subcluster. When you add new nodes to the database and do not specify a subcluster to contain them, Vertica adds them to the default subcluster. There can be only one default subcluster at a time. The subcluster that was previously the default subcluster becomes a non-default subcluster.

Privileges

Superuser

Examples

This example makes the analytics_cluster the default subcluster:

```
=> SELECT DISTINCT subcluster_name FROM SUBCLUSTERS WHERE is_default = true;
  subcluster_name
--------------------
 default_subcluster
(1 row)

=> ALTER SUBCLUSTER analytics_cluster SET DEFAULT;
ALTER SUBCLUSTER
=> SELECT DISTINCT subcluster_name FROM SUBCLUSTERS WHERE is_default = true;
  subcluster_name
--------------------
 analytics_cluster
(1 row)
```

This example renames default_subcluster to load_subcluster:

```
=> ALTER SUBCLUSTER default_subcluster RENAME TO load_subcluster;
ALTER SUBCLUSTER

=> SELECT DISTINCT subcluster_name FROM subclusters;
  subcluster_name
--------------------
 load_subcluster
 analytics_cluster
(2 rows)
```

See also

- CRITICAL_SUBCLUSTERS
- DEMOTE_SUBCLUSTER_TO_SECONDARY

- PROMOTE_SUBCLUSTER_TO_PRIMARY
- SHUTDOWN_SUBCLUSTER
- SUBCLUSTERS

ALTER SUBNET

Renames an existing subnet.

Syntax

```
ALTER SUBNET subnet-name RENAME TO new-subnet-name
```

Parameters

subnet-name

The name of the existing subnet.

new-subnet-name

The new name for the subnet.

Privileges

Superuser

Examples

```
=> ALTER SUBNET mysubnet RENAME TO myNewSubnet;
```

ALTER TABLE

Modifies the metadata of an existing table. All changes are auto-committed.

Syntax

```
ALTER TABLE [[database.]schema.]table {
   ADD COLUMN [ IF NOT EXISTS ] column datatype
     [ column-constraint ]
     [ ENCODING encoding-type ]
     [ PROJECTIONS (projections-list) | ALL PROJECTIONS ]
   | ADD table-constraint
   | ALTER COLUMN column {
      ENCODING encoding-type PROJECTIONS (projection-list)
      | { SET | DROP } expression }
   | ALTER CONSTRAINT constraint-name { ENABLED | DISABLED }
   | DISK_QUOTA { value | SET NULL }
   | DROP CONSTRAINT constraint-name [ CASCADE | RESTRICT ]
   | DROP [ COLUMN ] [ IF EXISTS ] column [ CASCADE | RESTRICT ]
   | FORCE OUTER integer
   | { INCLUDE | EXCLUDE | MATERIALIZE } [ SCHEMA ] PRIVILEGES
   | OWNER TO owner
   | partition-clause [ REORGANIZE ]
   | REMOVE PARTITIONING
   | RENAME [ COLUMN ] name TO new-name
   | RENAME TO new-table-name[,...]
   | REORGANIZE
   | SET {
      ActivePartitionCount { count | DEFAULT }
      | IMMUTABLE ROWS
      | MERGEOUT { 1 | 0 }
      | SCHEMA schema }
}
```

Note

Several ALTER TABLE clauses cannot be specified with other clauses in the same statement (see Exclusive ALTER TABLE Clauses below). Otherwise, ALTER TABLE supports multiple comma-delimited clauses. For example, the following ALTER TABLE statement changes the my_table table in two ways: reassigns ownership to Joe , and sets a UNIQUE constraint on the b column:

```
=> ALTER TABLE my_table OWNER TO Joe,
  ADD CONSTRAINT unique_b UNIQUE (b) ENABLED;
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

The table to alter.

ADD COLUMN

Adds a column to the table and, by default, to all its superprojections:

```
ADD COLUMN [IF NOT EXISTS]
   column datatype
   [ NULL | NOT NULL ]
   [ ENCODING encoding-type]
   [ PROJECTIONS (projections-list) | ALL PROJECTIONS]
```

Restrictions on columns of complex types also apply to columns that you add using ADD COLUMN .
The optional IF NOT EXISTS clause generates an informational message if *column* already exists under the specified name. If you omit this option and *column* exists, Vertica generates a ROLLBACK error message.

You can qualify the new column definition with one of these options:

- *column-constraint* specifies a column constraint as follows:

  ```
  {NULL | NOT NULL}
  | [ DEFAULT default-expr ] [ SET USING using-expr ] } | DEFAULT USING exp}
  ```

- ENCODING specifies the column's encoding type , by default set to AUTO .
- PROJECTIONS adds the new column to one or more existing projections of this table, specified as a comma-delimted list of projection base names . Vertica adds the column to all buddies of each projection. The projection list cannot include projections with pre-aggregated data such as live aggregate projections; otherwise, Vertica rolls back the ALTER TABLE statement.
- ALL PROJECTIONS adds the column to all projections of this table, excluding projections with pre-aggregated data.

ADD *table-constraint*

Adds a constraint to a table that does not have any associated projections.

ALTER COLUMN

You can alter an existing column in one of two ways:

- Set encoding on a column for one or more projections of this table:

  ```
  ENCODING encoding-type PROJECTIONS (projections-list)
  ```

  where *projections-list* is a comma-delimited list of projections to update with the new encoding. You can specify each projection in two ways:
  - Projection base name: Update all projections that share this base name.
  - Projection name: Update the specified projection. If the projection is segmented, the change is propagated to all buddies.

  If one of the projections does not contain the target column, Vertica returns with a rollback error.
  For details, see Projection Column Encoding .
- Set or drop a setting for a column of scalar data, including primitive arrays:

  ```
  SET { DEFAULT expression
        | USING expression
    | DEFAULT USING expression
    | NOT NULL
    | DATA TYPE datatype
  }
  DROP { DEFAULT
     | SET USING
     | DEFAULT USING
     | NOT NULL
  }
  ```

You cannot change the data type of a column of any complex type that is neither a scalar type nor an array of scalar types. One exception applies: in external tables, you can change a primitive column type to a complex type.
Setting a DEFAULT or SET USING expression has no effect on existing column values. To refresh the column with its DEFAULT or SET USING expression, update it as follows

- SET USING column: Call REFRESH_COLUMNS on the table.
- DEFAULT column: update the column as follows:

```
UPDATE table-name SET column-name=DEFAULT;
```

Altering a column with DEFAULT or SET USING can increase disk usage, which can cause the operation to fail if it would violate the table or schema disk quota.

ALTER CONSTRAINT

Specifies whether to enforce primary key, unique key, and check constraints:

```
ALTER CONSTRAINT constraint-name {ENABLED | DISABLED}
```

DISK_QUOTA

One of the following:

- A string, an integer followed by a supported unit: K, M, G, or T. If the new value is smaller than the current usage, the operation succeeds but no further disk space can be used until usage is reduced below the new quota.
- SET NULL to remove a quota.

For more information, see Disk quotas .

DROP CONSTRAINT

Drops the specified table constraint from the table:

```
DROP CONSTRAINT constraint-name [CASCADE | RESTRICT]
```

You can qualify DROP CONSTRAINT with one of these options:

- CASCADE : Drops a constraint and all dependencies in other tables.
- RESTRICT : Does not drop a constraint if there are dependent objects. Same as the default behavior.

Dropping a table constraint has no effect on views that reference the table.

DROP [COLUMN]

Drops the specified column from the table and that column's ROS containers:

```
DROP [COLUMN] [IF EXISTS] column [CASCADE | RESTRICT]
```

You can qualify DROP COLUMN with one of these options:

- IF EXISTS generates an informational message if the column does not exist. If you omit this option and the column does not exist, Vertica generates a ROLLBACK error message.
- CASCADE is required if the column has dependencies.
- RESTRICT drops the column only from the given table.

The column's table cannot be immutable .

See Dropping table columns .

FORCE OUTER *integer*

Specifies whether a table is joined to another as an inner or outer input. For details, see Controlling join inputs .

{INCLUDE | EXCLUDE | MATERIALIZE} [SCHEMA] PRIVILEGES

Specifies default inheritance of schema privileges for this table:

- EXCLUDE PRIVILEGES (default) disables inheritance of privileges from the schema.
- INCLUDE PRIVILEGES grants the table the same privileges granted to its schema.
- MATERIALIZE PRIVILEGES copies grants to the table and creates a GRANT object on the table. This disables the inherited privileges flag on the table, so you can:
  - Grant more specific privileges at the table level.
  - Use schema-level privileges as a template.
  - Move the table to a different schema.
  - Change schema privileges without affecting the table.

> **Note**
> If inherited privileges are disabled at the database level , schema privileges can still be materialized.

See also Setting privilege inheritance on tables and views .

OWNER TO *owner*

Changes the table owner .

***partition-clause* [REORGANIZE]**

Invalid for external tables, logically divides table data storage through a PARTITION BY clause:

```
PARTITION BY partition-expression
 [ GROUP BY group-expression ]
 [ SET ACTIVEPARTITIONCOUNT integer ]
```

For details, see Partition clause .
If you qualify the partition clause with REORGANIZE and the table previously specified no partitioning, the Vertica Tuple Mover immediately implements the partition clause. If the table previously specified partitioning, the Tuple Mover evaluates ROS storage containers and reorganizes them as needed to conform with the new partition clause.

REMOVE PARTITIONING

Specifies to remove partitioning from a table definition. The Tuple Mover subsequently removes existing partitions from ROS containers.

RENAME [COLUMN]

Renames the specified column within the table. The column's table cannot be immutable .

RENAME TO

Renames one or more tables:

```
RENAME TO new-table-name[,...]
```

The following requirements apply:

- The renamed table must be in the same schema as the original table.
- The new table name conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.
- If you specify multiple tables to rename, the source and target lists must have the same number of names.

REORGANIZE

Valid only for partitioned tables, invokes the Tuple Mover to reorganize ROS storage containers as needed to conform with the table's current partition clause. ALTER TABLE...REORGANIZE and Vertica meta-function PARTITION_TABLE operate identically.

REORGANIZE can also qualify a new partition clause .

SET

Changes a table setting, one of the following:

- ActivePartitionCount { *count* | DEFAULT } , valid only for partitioned tables, specifies how many partitions are active for this table, one of the following:
  - *count* : Unsigned integer, supersedes configuration parameter ActivePartitionCount .
  - DEFAULT : Removes the table-level active partition count. The table obtains its active partition count from the configuration parameter ActivePartitionCount.

  For details on usage, see Active and inactive partitions .
- IMMUTABLE ROWS prevents changes to table row values by blocking DML operations such as UPDATE and DELETE. Once set, table immutability cannot be reverted.
  You cannot set a flattened table to be immutable. For details on all immutable table restrictions, see Immutable tables .
- MERGEOUT { 1 | 0 } specifies whether to enable or disable mergeout to ROS containers that consolidate projection data of this table. By default, mergeout is enabled (1) on all tables.
- SCHEMA *schema-name* moves the table from its current schema to *schema-name* . Vertica automatically moves all projections that are anchored to the source table to the destination schema. It also moves all IDENTITY columns to the destination schema. For details, see Moving tables to another schema

Privileges

Non-superuser: USAGE on the schema and one of the following:

- Table owner
- ALTER privileges

Non-superusers must also have SELECT privileges to enable or disable constraint enforcement , or remove partitioning.

For certain operations, non-superusers must have the following schema privileges:

- To rename a table: CREATE, USAGE

- To move a table to another schema: USAGE on the source schema, CREATE on the destination schema

Restrictions for complex types

Complex types used in native tables have some restrictions, in addition to the restrictions for individual types listed on their reference pages:

- A native table must have at least one column that is a primitive type or a native array (one-dimensional array of a primitive type). If a flex table has real columns, it must also have at least one column satisfying this restriction.
- Complex type columns cannot be used in ORDER BY or PARTITION BY clauses nor as FILLER columns.
- Complex type columns cannot have constraints .
- Expressions returning complex types cannot be used as projection columns, and projections cannot be segmented or ordered by columns of complex types.
- Tables with columns of complex types cannot use DEFAULT and SET USING.

Exclusive ALTER TABLE clauses

The following ALTER TABLE clauses cannot be combined with another ALTER TABLE clause:

- ADD COLUMN
- DROP COLUMN
- RENAME COLUMN
- SET SCHEMA
- RENAME [TO]

Node down limitations

Enterprise Mode only

The following ALTER TABLE operations are not supported when one or more database cluster nodes are down:

- ALTER COLUMN ... ADD *table-constraint*
- ALTER COLUMN ... SET DATA TYPE
- ALTER COLUMN ... { SET DEFAULT | DROP DEFAULT }
- ALTER COLUMN ... { SET USING | DROP SET USING }
- ALTER CONSTRAINT
- DROP COLUMN
- DROP CONSTRAINT

Pre-aggregated projection restrictions

You cannot modify the metadata of anchor table columns that are included in live aggregate or Top-K projections. You also cannot drop these columns. To make these changes, you must first drop all live aggregate and Top-K projections that are associated with it.

External table restrictions

Not all ALTER TABLE options pertain to external tables. For instance, you cannot add a column to an external table, but you can rename the table:

```
=> ALTER TABLE mytable RENAME TO mytable2;
ALTER TABLE
```

Locked tables

If the operation cannot obtain an O lock on the target table, Vertica tries to close any internal Tuple Mover sessions that are running on that table. If successful, the operation can proceed. Explicit Tuple Mover operations that are running in user sessions do not close. If an explicit Tuple Mover operation is running on the table, the operation proceeds only when the operation is complete.

See also

- Working with native tables
- Altering table definitions
- Adding table columns

In this section

- Projection column encoding
- Table-constraint

Projection column encoding

After you create a table and its projections, you can call ALTER TABLE...ALTER COLUMN to set or change the encoding type of an existing column in one or more projections. For example:

```
ALTER TABLE store.store_dimension ALTER COLUMN store_region
  ENCODING rle PROJECTIONS (store.store_dimension_p1_b0, store.store_dimension_p2);
```

In this example, the ALTER TABLE statement specifies to set RLE encoding on column store_region for two projections: store_dimension_p1_b0 and store_dimension_p2 . The PROJECTIONS list references the two projections by their projection name and base name, respectively. You can reference a projection either way; in both cases, the change is propagated to all buddies of the projection and stored in its DDL accordingly:

```
=> select export_objects('','store.store_dimension');

                        export_objects
------------------------------------------------------------------------
CREATE TABLE store.store_dimension
(
    store_key int NOT NULL,
    store_name varchar(64),
    store_number int,
    store_address varchar(256),
    store_city varchar(64),
    store_state char(2),
    store_region varchar(64)
);

CREATE PROJECTION store.store_dimension_p1
(
 store_key,
 store_name,
 store_number,
 store_address,
 store_city,
 store_state,
 store_region ENCODING RLE
)
AS
 SELECT store_dimension.store_key,
        store_dimension.store_name,
        store_dimension.store_number,
        store_dimension.store_address,
        store_dimension.store_city,
        store_dimension.store_state,
        store_dimension.store_region
 FROM store.store_dimension
 ORDER BY store_dimension.store_key
SEGMENTED BY hash(store_dimension.store_key) ALL NODES KSAFE 1;

CREATE PROJECTION store.store_dimension_p2
(
 store_key,
 store_name,
 store_number,
 store_address,
 store_city,
 store_state,
 store_region ENCODING RLE
)
AS
 SELECT ...
```

Important
When you add or change a column's encoding type, it has no immediate effect on existing projection data. Vertica applies the encoding only to newly loaded data, and to existing data on mergeout .

Table-constraint

Table-constraint

Adds a constraint to table metadata. You can specify table constraints with CREATE TABLE, or add a constraint to an existing table with ALTER TABLE. For details, see Setting constraints.

> **Note**
> Adding a constraint to a table that is referenced in a view does not affect the view.

Syntax

```
[ CONSTRAINT constraint-name ]
{
... PRIMARY KEY (column[,... ]) [ ENABLED | DISABLED ]
... | FOREIGN KEY (column[,... ] ) REFERENCES table [ (column[,...]) ]
... | UNIQUE (column[,...]) [ ENABLED | DISABLED ]
... | CHECK (expression) [ ENABLED | DISABLED ]
}
```

Parameters

CONSTRAINT *constraint-name*

Assigns a name to the constraint. Vertica recommends that you name all constraints.

PRIMARY KEY

Defines one or more NOT NULL columns as the primary key as follows:

```
PRIMARY KEY (column[,...]) [ ENABLED | DISABLED]
```

You can qualify this constraint with the keyword ENABLED or DISABLED. See Enforcing Constraints below.

If you do not name a primary key constraint, Vertica assigns the name C_PRIMARY.

FOREIGN KEY

Adds a referential integrity constraint defining one or more columns as foreign keys as follows:

```
FOREIGN KEY (column[,... ]) REFERENCES table [(column[,... ])]
```

If you omit *column*, Vertica references the primary key in *table*.

If you do not name a foreign key constraint, Vertica assigns the name C_FOREIGN.

Important

Adding a foreign key constraint requires the following privileges (in addition to privileges also required by ALTER TABLE):

- REFERENCES on the referenced table
- USAGE on the schema of the referenced table

UNIQUE

Specifies that the data in a column or group of columns is unique with respect to all table rows, as follows:

```
UNIQUE (column[,...]) [ENABLED | DISABLED]
```

You can qualify this constraint with the keyword ENABLED or DISABLED. See Enforcing Constraints below.

If you do not name a unique constraint, Vertica assigns the name C_UNIQUE.

CHECK

Specifies a check condition as an expression that returns a Boolean value, as follows:

```
CHECK (expression) [ENABLED | DISABLED]
```

You can qualify this constraint with the keyword ENABLED or DISABLED. See Enforcing Constraints below.

If you do not name a check constraint, Vertica assigns the name C_CHECK.

Privileges

Non-superusers: table owner, or the following privileges:

- USAGE on schema
- ALTER on table
- SELECT on table to enable or disable constraint enforcement

Enforcing constraints

A table can specify whether Vertica automatically enforces a primary key, unique key or check constraint with the keyword ENABLED or DISABLED . If you omit ENABLED or DISABLED , Vertica determines whether to enable the constraint automatically by checking the appropriate configuration parameter:

- EnableNewPrimaryKeysByDefault
- EnableNewUniqueKeysByDefault
- EnableNewCheckConstraintsByDefault

For details, see Constraint enforcement .

Examples

The following example creates a table (t01) with a primary key constraint.

```
CREATE TABLE t01 (id int CONSTRAINT sampleconstraint PRIMARY KEY);
CREATE TABLE
```

This example creates the same table without the constraint, and then adds the constraint with ALTER TABLE ADD CONSTRAINT

```
CREATE TABLE t01 (id int);
CREATE TABLE

ALTER TABLE t01 ADD CONSTRAINT sampleconstraint PRIMARY KEY(id);
WARNING 2623:  Column "id" definition changed to NOT NULL
ALTER TABLE
```

The following example creates a table (addapk) with two columns, adds a third column to the table, and then adds a primary key constraint on the third column.

```
=> CREATE TABLE addapk (col1 INT, col2 INT);
CREATE TABLE

=> ALTER TABLE addapk ADD COLUMN col3 INT;
ALTER TABLE

=> ALTER TABLE addapk ADD CONSTRAINT col3constraint PRIMARY KEY (col3) ENABLED;
WARNING 2623:  Column "col3" definition changed to NOT NULL
ALTER TABLE
```

Using the sample table addapk , check that the primary key constraint is enabled (is_enabled is t).

```
=> SELECT constraint_name, column_name, constraint_type, is_enabled FROM PRIMARY_KEYS WHERE table_name IN ('addapk');

 constraint_name | column_name | constraint_type | is_enabled
-----------------+-------------+-----------------+------------
 col3constraint  | col3        | p               | t
(1 row)
```

This example disables the constraint using ALTER TABLE ALTER CONSTRAINT .

```
=> ALTER TABLE addapk ALTER CONSTRAINT col3constraint DISABLED;
```

Check that the primary key is now disabled (is_enabled is f).

```
=> SELECT constraint_name, column_name, constraint_type, is_enabled FROM PRIMARY_KEYS WHERE table_name IN ('addapk');

 constraint_name | column_name | constraint_type | is_enabled
-----------------+-------------+-----------------+------------
 col3constraint  | col3        | p               | f
(1 row)
```

For a general discussion of constraints, see Constraints . For additional examples of creating and naming constraints, see Naming constraints .

ALTER TLS CONFIGURATION

Alters a specified TLS Configuration object. For information on existing TLS Configuration objects, query TLS_CONFIGURATIONS .

Syntax

```
ALTER TLS CONFIGURATION tls_config_name {
   [ CERTIFICATE { NULL | cert_name } ]
   [ ADD CA CERTIFICATES ca_cert_name [,...] ]
   [ REMOVE CA CERTIFICATES ca_cert_name [,...] ]
   [ CIPHER SUITES { '' | 'openssl_cipher [,...]' } ]
   [ TLSMODE 'tlsmode' ]
   [ OWNER TO user_name ]
}
```

Parameters

tls_config_name

The TLS Configuration object to alter.

NULL

Removes the non-CA certificate from the TLS Configuration.

cert_name

A certificate created with CREATE CERTIFICATE .

You must have USAGE privileges on the certificate (either from ownership of the certificate or USAGE on its key , if any) to add it to a TLS Configuration.

ca_cert_name

A CA certificate created with CREATE CERTIFICATE .

You must have USAGE privileges on the certificate (either from ownership of the certificate or USAGE on its key , if any) to add it to a TLS Configuration.

openssl_cipher

A comma-separated list of cipher suites to use instead of the default set of cipher suites. Providing an empty string for this parameter clears the alternate cipher suite list and instructs the specified TLS Configuration to use the default set of cipher suites.

To view enabled cipher suites, use LIST_ENABLED_CIPHERS .

tlsmode

How Vertica establishes TLS connections and handles certificates, one of the following, in order of ascending security:

- DISABLE : Disables TLS. All other options for this parameter enable TLS.
- ENABLE : Enables TLS. Vertica does not check client certificates.
- TRY_VERIFY : Establishes a TLS connection if one of the following is true:
  - the other host presents a valid certificate
  - the other host doesn't present a certificate

  If the other host presents an invalid certificate, the connection will use plaintext.
- VERIFY_CA : Connection succeeds if Vertica verifies that the other host's certificate is from a trusted CA. If the other host does not present a certificate, the connection uses plaintext.
- VERIFY_FULL : Connection succeeds if Vertica verifies that the other host's certificate is from a trusted CA and the certificate's cn (Common Name) or subjectAltName attribute matches the hostname or IP address of the other host.
  Note that for client certificates, cn is used for the username, so subjectAltName must match the hostname or IP address of the other host.

VERIFY_FULL is unsupported for client-server TLS (the connection type handled by ServerTLSConfig) and behaves like VERIFY_CA .

> **Note**
> Whether Vertica or the other party acts as the client or server depends on the type of connection. For connections between the Vertica database and an LDAP server for LDAP Link or LDAP authentication, the Vertica database is the client and the LDAP server is the server:
>
> - LDAPLinkTLSConfig
> - LDAPAuthTLSConfig
>
> For all other connection types, Vertica is the server and the other party is the client:

- ServerTLSConfig
- InternodeTLSConfig (the other Vertica nodes are both the client and server)

Privileges

Non-superuser: ALTER privileges on the TLS Configuration.

Examples

To remove all certificates and CA certificates from the LDAPLink TLS Configuration:

```
=>  SELECT * FROM tls_configurations WHERE name='LDAPLink';
   name   |  owner  | certificate | ca_certificate | cipher_suites |  mode
----------+---------+-------------+----------------+---------------+---------
 LDAPLink | dbadmin | server_cert | ca             |               | DISABLE
 LDAPLink | dbadmin | server_cert | ica            |               | DISABLE
(2 rows)

=> ALTER TLS CONFIGURATION LDAPLink CERTIFICATE NULL REMOVE CA CERTIFICATES ca, ica;
ALTER TLS CONFIGURATION

=> SELECT * FROM tls_configurations WHERE name='LDAPLink';
   name   |  owner  | certificate | ca_certificate | cipher_suites |  mode
----------+---------+-------------+----------------+---------------+---------
 LDAPLink | dbadmin |             |                |               | DISABLE
(3 rows)
```

To use an alternate set of cipher suites for client-server TLS:

```
 => ALTER TLS CONFIGURATION server CIPHER SUITES
    'DHE-PSK-AES256-CBC-SHA384,
     DHE-PSK-AES128-GCM-SHA256,
     PSK-AES128-CBC-SHA256';
ALTER TLS CONFIGURATION

 => SELECT name, cipher_suites FROM tls_configurations WHERE name='server';
   name   |                          cipher_suites
 server   | DHE-PSK-AES256-CBC-SHA384,DHE-PSK-AES128-GCM-SHA256,PSK-AES128-CBC-SHA256
(1 row)
```

For other examples, see:

- Configuring client-server TLS
- TLS for LDAP link
- TLS for LDAP authentication
- Internode TLS

ALTER TRIGGER

Modifies a trigger .

Syntax

```
ALTER TRIGGER [[database.]schema.]trigger {
        OWNER TO new_owner
      | SET SCHEMA new_schema
      | RENAME TO new_trigger
      | PROCEDURE TO new_procedure
   }
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

trigger

The trigger to modify.

new_owner

The new owner and definer of the trigger. This affects the behavior of AS DEFINER .

new_schema

The new schema of the trigger.

new_trigger

The new name for the trigger.

new_procedure

The function signature of the stored procedure .

Privileges

Superuser

Examples

To attach a different procedure to the trigger:

```
=> ALTER TRIGGER daily_1am PROCEDURE TO log_user_actions(10, 20);
```

To rename a trigger:

```
=> ALTER TRIGGER daily_1am RENAME TO daily_1am_gmt;
```

To move the trigger to a different schema:

```
=> ALTER TRIGGER daily_1am_gmt
```

ALTER USER

Changes user account parameters and user-level configuration parameters.

Syntax

```
ALTER USER user-name {
   account-parameter value[,...]
   | SET [PARAMETER] cfg-parameter=value[,...]
   | CLEAR [PARAMETER] cfg-parameter[,...]
}
```

Parameters

user-name

Name of the user. Names that contain special characters must be double-quoted. To enforce case-sensitivity, use double-quotes.

For details on name requirements, see Creating a database name and password .

account-parameter value

Specifies user account settings (see below).

> **Note**
> Changes to a user account apply only to the current session and to all later sessions launched by this user.

SET [PARAMETER]

Sets the specified configuration parameters. The new setting applies only to the current session, and to all later sessions launched by this user. Concurrent user sessions are unaffected by new settings unless they call meta-function RESET_SESSION .

CLEAR [PARAMETER]

Resets the specified configuration parameters to their default values.

Important

SET | CLEAR PARAMETER can specify only user-level configuration parameters, otherwise Vertica returns an error. For details, see Setting User-Level Configuration Parameters below.

User account parameters

Specify one or more user-account parameters and their settings as a comma-delimited list:

```
account-parameter value[,...]
```

Important

The following user-account parameters are invalid for a user who is added to the Vertica database with the LDAPLink service:

- IDENTIFIED BY
- PROFILE
- SECURITY ALGORITHM

Parameter	Setting
ACCOUNT	Locks or unlocks user access to the database, one of the following: • UNLOCK (default) • LOCK prevents a new user from logging in. This can be useful when creating an account for a user who does not need immediate access. **Tip** To automate account locking, set a maximum number of failed login attempts with CREATE PROFILE .
DEFAULT ROLE	Specifies what roles are the default roles for this user, set to one of the following: • NONE (default): Removes all default roles. • *role* [,...] : Comma-delimited list of roles. • ALL : Sets as default all user roles. • ALL EXCEPT *role* [,...] : Comma-delimited list of roles to exclude as default roles. Default roles are automatically activated when a user logs in. The roles specified by this parameter supersede any roles assigned earlier. **Note** DEFAULT ROLE cannot be specified in combination with other ALTER USER parameters.
GRACEPERIOD	Specifies how long a user query can block on any session socket, one of the following: • NONE (default): Removes any grace period previously set on session queries. • ' *interval* ' : Specifies as an interval the maximum grace period for current session queries, up to 20 days. For details, see Handling session socket blocking .

IDENTIFIED BY	Changes the user's password:
	`IDENTIFIED BY '[new-password]'`
	`  | ['hashed-password' SALT 'hash-salt']`
	`  [REPLACE 'current-password']`
	• *new-password* : ASCII password that Vertica then hashes for internal storage. An empty string enables this user to access the database with no password.
	• *hashed-password* : A pre-hashed password and its associated hex string *hash-salt* . Setting a password this way bypasses all password complexity requirements .
	• REPLACE: Required for non-superusers, who must supply their current password. Non-superusers can only change their own passwords.
	For details, see Password guidelines and Creating a database name and password .

IDLESESSIONTIMEOUT	The length of time the system waits before disconnecting an idle session, one of the following: • NONE (default): No limit set for this user. If you omit this parameter, no limit is set for this user. • ' *interval* ' : An interval value, up to one year. For details, see Managing client connections .
MAXCONNECTIONS	Sets the maximum number of connections the user can have to the server, one of the following: • NONE (default): No limit set. If you omit this parameter, the user can have an unlimited number of connections across the database cluster. • *integer* ON DATABASE : Sets to *integer* the maximum number of connections across the database cluster. • *integer* ON NODE : Sets to *integer* the maximum number of connections to each node. For details, see Managing client connections .

MEMORYCAP	Sets how much memory can be allocated to user requests, one of the following:
	• NONE (default): No limit
	• A string value that specifies the memory limit, one of the following:
	  ◦ ' *int* %' expresses the maximum as a percentage of total memory available to the Resource Manager, where *int* is an integer value between 0 and 100.For example:
	  MEMORYCAP '40%'
	  ◦ ' *int* {K|M|G|T}' expresses memory allocation in kilobytes, megabytes, gigabytes, or terabytes. For example:
	  MEMORYCAP '10G'

PASSWORD EXPIRE	Forces immediate expiration of the user's password. The user must change the password on the next login. **Note** PASSWORD EXPIRE has no effect when using external password authentication methods such as LDAP or Kerberos.
PROFILE	Assigns a profile that controls password requirements for this user, one of the following: • DEFAULT (default): Assigns the default database profile to this user. • *profile-name* : A profile that is defined by CREATE PROFILE .

RENAME TO	Assigns the user a new user name. All privileges assigned to the user remain unchanged. **Note** RENAME TO cannot be specified in combination with other ALTER USER parameters.
RESOURCE POOL *pool-name* [FOR SUBCLUSTER *sc-name*]	Assigns a resource pool to this user. The user must also be granted privileges to this pool , unless privileges to the pool are set to PUBLIC . The FOR SUBCLUSTER clause assigns a subcluster-specific resource pool to the user. You can assign only one subcluster-specific resource pool to each user.
RUNTIMECAP	Sets how long this user's queries can execute, one of the following: • NONE (default): No limit set for this user. If you omit this parameter, no limit is set for this user. • ' *interval* ' : An interval value, up to one year. A query's runtime limit can be set at three levels: the user's runtime limit, the user's resource pool, and the session setting. For more information, see Setting a runtime limit for queries .
SEARCH_PATH	Specifies the user's default search path, that tells Vertica which schemas to search for unqualified references to tables and UDFs, one of the following: • DEFAULT (default): Sets the search path as follows: "$user", public, v_catalog, v_monitor, v_internal • Comma-delimited list of schemas. For details, see Setting Search Paths .
SECURITY_ALGORITHM ' *algorithm* '	Sets the user-level security algorithm for hash authentication , where *algorithm* is one of the following: • NONE (default): Uses the system-level parameter, SecurityAlgorithm • SHA512 • MD5 The user's password expires when you change the SECURITY_ALGORITHM value and must be reset.

TEMPSPACECAP	Sets how much temporary file storage is available for user requests, one of the following:
	
	• NONE (default): No limit
	• String value that specifies the storage limit, one of the following:
	  ◦ *int* % expresses the maximum as a percentage of total temporary storage available to the Resource Manager, where *int* is an integer value between 0 and 100. For example:
	  TEMPSPACECAP '40%'
	  ◦ *int* {K|M|G|T} expresses storage allocation in kilobytes, megabytes, gigabytes, or terabytes. For example:
	  TEMPSPACECAP '10G'

Privileges

Non-superusers can change the following options on their own user accounts:

- IDENTIFIED BY
- RESOURCE POOL
- SEARCH_PATH
- SECURITY_ALGORITHM

When changing a another user's resource pool to one outside of the PUBLIC schema, the user must have USAGE privileges on the resource pool from at least one of the following:

- Object ownership
- Explicit grant to the user
- Default role

Setting user-level configuration parameters

SET | CLEAR PARAMETER can specify only user-level configuration parameters, otherwise Vertica returns an error. Only superusers can set and clear user-level parameters, unless they are also supported at the session level.

To get the names of user-level parameters, query system table CONFIGURATION_PARAMETERS . For example:

```
=> SELECT parameter_name, allowed_levels FROM configuration_parameters
    WHERE allowed_levels ilike '%USER%' AND parameter_name ilike '%depot%' ORDER BY parameter_name;
     parameter_name       |     allowed_levels
----------------------------+------------------------
 BackgroundDepotWarming     | SESSION, USER, DATABASE
 DepotOperationsForQuery    | SESSION, USER, DATABASE
 EnableDepotWarmingFromPeers | SESSION, USER, DATABASE
 UseDepotForReads           | SESSION, USER, DATABASE
 UseDepotForWrites          | SESSION, USER, DATABASE
(5 rows)
```

The following example sets the user-level configuration parameter UseDepotForWrites for two users, Yvonne and Ahmed:

```
=> SHOW USER Yvonne PARAMETER ALL;
 user |       parameter        | setting
--------+------------------------+---------
 Yvonne | DepotOperationsForQuery | Fetches
(1 row)

=> ALTER USER Yvonne SET PARAMETER UseDepotForWrites = 0;
ALTER USER
=> SHOW USER Yvonne PARAMETER ALL;
 user |       parameter        | setting
--------+------------------------+---------
 Yvonne | DepotOperationsForQuery | Fetches
 Yvonne | UseDepotForWrites       | 0
(2 rows)

=> ALTER USER Ahmed SET PARAMETER DepotOperationsForQuery = 'Fetches';
ALTER USER
=> SHOW USER ALL PARAMETER ALL;
 user |       parameter        | setting
--------+------------------------+---------
 Ahmed  | DepotOperationsForQuery | Fetches
 Yvonne | DepotOperationsForQuery | Fetches
 Yvonne | UseDepotForWrites       | 0
(3 rows)
```

Examples

Set a user's password

```
=> CREATE USER user1;
=> ALTER USER user1 IDENTIFIED BY 'newpassword';
```

Set user's security algorithm and password

This example sets a user's security algorithm and password to SHA-512 and newpassword , respectively. When you execute the ALTER USER statement, Vertica hashes the password with the SHA-512 algorithm and saves the hash:

```
=> CREATE USER user1;
     => ALTER USER user1 SECURITY_ALGORITHM 'SHA512' IDENTIFIED BY 'newpassword'
```

Assign default roles to a user

This example make a user's assigned roles their default roles . Default roles are automatically set (enabled) when a user logs in:

```
=> CREATE USER user1;
CREATE USER
=> GRANT role1, role2, role3 to user1;
=> ALTER USER user1 DEFAULT ROLE ALL;
```

You can pair ALL with EXCEPT to exclude certain roles:

```
=> CREATE USER user2;
CREATE USER
=> GRANT role1, role2, role3 to user2;
=> ALTER USER user2 DEFAULT ROLE ALL EXCEPT role1;
```

See also

- CREATE USER
- DROP USER
- SHOW USER

ALTER VIEW

Modifies the metadata of an existing view . The changes are auto-committed.

Syntax

General usage:

```
ALTER VIEW [[database.]schema.]view {
   | OWNER TO owner
   | SET SCHEMA schema
   | { INCLUDE | EXCLUDE | MATERIALIZE } [ SCHEMA ] PRIVILEGES
}
```

Rename view:

```
ALTER VIEW [[database.]schema.]view[,...] RENAME TO new-view-name[,...]
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

view

The view to alter.

SET SCHEMA *schema* ``

Moves the view from one schema to another.

OWNER TO *owner*

Changes the view owner.

Important

The new view owner should also have SELECT privileges on the objects that the view references; otherwise the view is inaccessible to that user.

{ INCLUDE | EXCLUDE | MATERIALIZE } [SCHEMA] PRIVILEGES

Specifies default inheritance of schema privileges for this view:

- EXCLUDE [SCHEMA] PRIVILEGES (default) disables inheritance of privileges from the schema.
- INCLUDE [SCHEMA] PRIVILEGES grants the view the same privileges granted to its schema.
- MATERIALIZE : Copies grants to the view and creates a GRANT object on the view. This disables the inherited privileges flag on the view, so you can:
  - Grant more specific privileges at the view level
  - Use schema-level privileges as a template
  - Move the view to a different schema
  - Change schema privileges without affecting the view

> **Note**
> If inherited privileges are disabled at the database level , schema privileges can still be materialized.

See also Setting privilege inheritance on tables and views .

RENAME TO

Renames one or more views:

```
RENAME TO new-view-name[,...]
```

The following requirements apply:

- The new view name conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.
- If you specify multiple views to rename, the source and target lists must have the same number of names.
- Renaming a view requires USAGE and CREATE privileges on the schema that contains the view.

Privileges

Non-superuser: USAGE on the schema and one of the following:

- View owner
- ALTER privilege on the view

For certain operations, non-superusers must have the following schema privileges:

Schema privileges required...	For these operations...
CREATE, USAGE	Rename view
CREATE: destination schema USAGE: current schema	Move view to another schema

Examples

Rename view view1 to view2 :

```
=> CREATE VIEW view1 AS SELECT * FROM t;
CREATE VIEW
=> ALTER VIEW view1 RENAME TO view2;
ALTER VIEW
```

BEGIN

Starts a transaction block.

> **Note**
> BEGIN is a synonym for START TRANSACTION .

Syntax

```
BEGIN [ WORK | TRANSACTION ] [ isolation-level ] [ READ [ONLY] | WRITE ]
```

Parameters

WORK | TRANSACTION

Optional keywords for readability only.

isolation-level

Specifies the transaction's isolation level, which determines what data the transaction can access when other transactions are running concurrently, one of the following:

- READ COMMITTED (default)
- SERIALIZABLE
- REPEATABLE READ (automatically converted to SERIALIZABLE)
- READ UNCOMMITTED (automatically converted to READ COMMITTED)

For details, see Transactions .

READ [ONLY] | WRITE

Specifies the transaction mode, one of the following:

- READ WRITE (default): Transaction is read/write.
- READ ONLY: Transaction is read-only.

Setting the transaction session mode to read-only disallows the following SQL statements, but does not prevent all disk write operations:

- INSERT, UPDATE, DELETE, and COPY if the target table is not a temporary table
- All CREATE, ALTER, and DROP commands
- GRANT, REVOKE, and EXPLAIN if the SQL to run is one of the statements cited above.

Privileges

None

Examples

Create a transaction with the isolation level set to READ COMMITTED and the transaction mode to READ WRITE:

```
=> BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
BEGIN
=> CREATE TABLE sample_table (a INT);
CREATE TABLE
=> INSERT INTO sample_table (a) VALUES (1);
OUTPUT
--------
1
(1 row)

=> END;
COMMIT
```

See also

- Transactions
- Creating and rolling back transactions
- COMMIT
- END
- ROLLBACK

CALL

Invokes a stored procedure created with CREATE PROCEDURE (stored) .

Syntax

```
CALL [[database.]schema.]procedure( [ argument-list] );
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

procedure

The name of the stored procedure, where *procedure* conforms to conventions described in Identifiers .

argument-list

A comma-delimited list of arguments to pass to the stored procedure, whose types correspond to the types of the argument's IN parameters .

Privileges

Non-superuser: EXECUTE on the procedure

Examples

See Executing stored procedures and Stored procedures: use cases and examples .

See also

- DO
- CREATE PROCEDURE (stored)

- DROP PROCEDURE (stored)

COMMENT ON statements

COMMENT ON statements let you create comments on database objects, such as schemas, tables, and libraries. Each object can have one comment. Comments are stored in the system table COMMENTS .

In this section

- COMMENT ON AGGREGATE FUNCTION
- COMMENT ON ANALYTIC FUNCTION
- COMMENT ON CONSTRAINT
- COMMENT ON FUNCTION
- COMMENT ON LIBRARY
- COMMENT ON NODE
- COMMENT ON PROJECTION
- COMMENT ON PROJECTION COLUMN
- COMMENT ON SCHEMA
- COMMENT ON SEQUENCE
- COMMENT ON TABLE
- COMMENT ON TABLE COLUMN
- COMMENT ON TRANSFORM FUNCTION
- COMMENT ON VIEW

COMMENT ON AGGREGATE FUNCTION

Adds, revises, or removes a comment on an aggregate function. Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON AGGREGATE FUNCTION [[database.]schema.]function (function-args) IS { 'comment' | NULL };
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

The name of the aggregate function with which to associate the comment.

function-args

The function arguments.

comment

Specifies the comment text to add. If a comment already exists for this function, this overwrites the previous one.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the APPROXIMATE_MEDIAN(x FLOAT) function:

```
=> COMMENT ON AGGREGATE FUNCTION APPROXIMATE_MEDIAN(x FLOAT) IS 'alias of APPROXIMATE_PERCENTILE with 0.5 as its parameter';
```

The following example removes a comment from the APPROXIMATE_MEDIAN(x FLOAT) function:

```
=> COMMENT ON AGGREGATE FUNCTION APPROXIMATE_MEDIAN(x FLOAT) IS NULL;
```

COMMENT ON ANALYTIC FUNCTION

Adds, revises, or removes a comment on an analytic function. Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON ANALYTIC FUNCTION [[database.]schema.]function (function-args) IS { 'comment' | NULL };
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

The name of the analytic function with which to associate the comment.

function-args

The function arguments.

comment

Specifies the comment text to add. If a comment already exists for this function, this overwrites the previous one.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the user-defined an_rank() function:

```
=> COMMENT ON ANALYTIC FUNCTION an_rank() IS 'built from the AnalyticFunctions library';
```

The following example removes a comment from the user-defined an_rank() function:

```
=> COMMENT ON ANALYTIC FUNCTION an_rank() IS NULL;
```

COMMENT ON CONSTRAINT

Adds, revises, or removes a comment on a constraint. Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON CONSTRAINT constraint ON [[database.]schema.]table IS ... {'comment' | NULL };
```

Parameters

constraint

The name of the constraint associated with the comment.

[*database.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

The name of the table constraint with which to associate a comment.

comment

Specifies the comment text to add. If a comment already exists for this constraint, this comment overwrites the previous one.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the constraint_x constraint on the promotion_dimension table:

```
=> COMMENT ON CONSTRAINT constraint_x ON promotion_dimension IS 'Primary key';
```

The following example removes a comment from the constraint_x constraint on the promotion_dimension table:

```
=> COMMENT ON CONSTRAINT constraint_x ON promotion_dimension IS NULL;
```

COMMENT ON FUNCTION

Adds, revises, or removes a comment on a function. Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON FUNCTION [[database.]schema.]function (function-args) IS { 'comment' | NULL };
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

The name of the function with which to associate the comment.

function-args

The function arguments.

comment

Specifies the comment text to add. If a comment already exists for this function, this overwrites the previous one.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the macros.zerowhennull (x INT) function:

```
=> COMMENT ON FUNCTION macros.zerowhennull(x INT) IS 'Returns a 0 if not NULL';
```

The following example removes a comment from the macros.zerowhennull (x INT) function:

```
=> COMMENT ON FUNCTION macros.zerowhennull(x INT) IS NULL;
```

COMMENT ON LIBRARY

Adds, revises, or removes a comment on a library . Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON LIBRARY [[database.]schema.]library IS {'comment' | NULL}
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

library

The name of the library associated with the comment.

comment

Specifies the comment text to add. If a comment already exists for this library, this comment overwrites the previous one.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the library MyFunctions:

```
=> COMMENT ON LIBRARY MyFunctions IS 'In development';
```

The following example removes a comment from the library MyFunctions:

```
=> COMMENT ON LIBRARY MyFunctions IS NULL;
```

See also

- COMMENTS

COMMENT ON NODE

Adds, revises, or removes a comment on a node. Each object can have one comment. Comments are stored in the system table COMMENTS .

Dropping an object drops all comments associated with the object.

Syntax

```
COMMENT ON NODE node-name IS  { 'comment' | NULL }
```

Parameters

node-name

The name of the node associated with the comment.

comment

Specifies the comment text to add. If a comment already exists for this node, this comment overwrites the previous one.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment for the initiator node :

```
=> COMMENT ON NODE initiator IS 'Initiator node';
```

The following example removes a comment from the initiator node :

```
=> COMMENT ON NODE initiator IS NULL;
```

See also

COMMENTS

COMMENT ON PROJECTION

Adds, revises, or removes a comment on a projection. Each object can have one comment. Comments are stored in the system table COMMENTS .

Dropping an object drops all comments associated with the object.

Syntax

```
COMMENT ON PROJECTION [[database.]schema.]projection IS { 'comment' | NULL }
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection

The name of the projection associated with the comment.

comment

Specifies the text of the comment to add. If a comment already exists for this projection, the comment you enter here overwrites the previous comment.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the customer_dimension_vmart_node01 projection:

```
=> COMMENT ON PROJECTION customer_dimension_vmart_node01 IS 'Test data';
```

The following example removes a comment from the customer_dimension_vmart_node01 projection:

```
=> COMMENT ON PROJECTION customer_dimension_vmart_node01 IS NULL;
```

See also

COMMENTS

COMMENT ON PROJECTION COLUMN

Adds, revises, or removes a projection column comment. Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON COLUMN [[database.]schema.]projection.column IS {'comment' | NULL}
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection* . *column

The name of the projection and column with which to associate the comment.

comment

Specifies the comment text to add. If a comment already exists for this column, this comment overwrites the previous comment.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the customer_name column in the customer_dimension projection:

```
=> COMMENT ON COLUMN customer_dimension_vmart_node01.customer_name IS 'Last name only';
```

The following example removes a comment from the customer_name column in the customer_dimension projection:

```
=> COMMENT ON COLUMN customer_dimension_vmart_node01.customer_name IS NULL;
```

COMMENT ON SCHEMA

Adds, revises, or removes a comment on a schema. Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON SCHEMA schema-name IS {'comment' | NULL}
```

Parameters

schema-name

The schema associated with the comment.

comment

Text of the comment to add. If a comment already exists for this schema, the comment you enter here overwrites the previous comment.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the public schema:

```
=> COMMENT ON SCHEMA public  IS 'All users can access this schema';
```

The following example removes a comment from the public schema.

```
=> COMMENT ON SCHEMA public IS NULL;
```

COMMENT ON SEQUENCE

Adds, revises, or removes a comment on a sequence. Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON SEQUENCE [[database.]schema.]sequence IS { 'comment' | NULL }
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

sequence

The name of the sequence associated with the comment.

comment

Specifies the text of the comment to add. If a comment already exists for this sequence, this comment overwrites the previous one.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the sequence called prom_seq.

```
=> COMMENT ON SEQUENCE prom_seq IS 'Promotion codes';
```

The following example removes a comment from the prom_seq sequence.

```
=> COMMENT ON SEQUENCE prom_seq IS NULL;
```

COMMENT ON TABLE

Adds, revises, or removes a comment on a table. Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON TABLE [[database.]schema.]table IS { 'comment' | NULL }
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

The name of the table with which to associate the comment.

comment

Specifies the text of the comment to add. Enclose the text of the comment within single-quotes. If a comment already exists for this table, the comment you enter here overwrites the previous comment.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes a previously added comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the promotion_dimension table:

```
=> COMMENT ON TABLE promotion_dimension IS '2011 Promotions';
```

The following example removes a comment from the promotion_dimension table:

```
=> COMMENT ON TABLE promotion_dimension IS NULL;
```

COMMENT ON TABLE COLUMN

Adds, revises, or removes a table column comment. Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON COLUMN [[database.]schema.]table.column IS {'comment' | NULL}
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table* . *column

The name of the table and column with which to associate the comment.

comment

Specifies the comment text to add. If a comment already exists for this column, this comment overwrites the previous comment.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the transaction_time column in the store_sales_fact table in the store schema:

```
=> COMMENT ON COLUMN store.store_sales_fact.transaction_time IS 'GMT';
```

The following example removes a comment from the transaction_time column in the store_sales_fact table in the store schema:

```
=> COMMENT ON COLUMN store.store_sales_fact.transaction_time IS NULL;
```

COMMENT ON TRANSFORM FUNCTION

Adds, revises, or removes a comment on a user-defined transform function. Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON TRANSFORM FUNCTION [[database.]schema.]tfunction
...( [ tfunction-arg-name tfunction-arg-type ][,...] ) IS {'comment' | NULL}
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

tfunction

The name of the transform function with which to associate the comment.

tfunction-arg-name tfunction-arg-type

The names and data types of one or more transform function arguments. If you supply argument names and types, each type must match the type specified in the library used to create the original transform function.

comment

Specifies the comment text to add. If a comment already exists for this transform function, this comment overwrites the previous one.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment to the macros.zerowhennull (x INT) UTF function:

```
=> COMMENT ON TRANSFORM FUNCTION macros.zerowhennull(x INT) IS 'Returns a 0 if not NULL';
```

The following example removes a comment from the acros.zerowhennull (x INT) function by using the NULL option:

```
=> COMMENT ON TRANSFORM FUNCTION macros.zerowhennull(x INT) IS NULL;
```

COMMENT ON VIEW

Adds, revises, or removes a comment on a view. Each object can have one comment. Comments are stored in the system table COMMENTS .

Syntax

```
COMMENT ON VIEW [[database.]schema.]view IS { 'comment' | NULL }
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

view

The name of the view with which to associate the comment.

comment

Specifies the text of the comment to add. If a comment already exists for this view, this comment overwrites the previous one.

Comments can be up to 8192 characters in length. If a comment exceeds that limitation, Vertica truncates the comment and alerts the user with a message.

NULL

Removes an existing comment.

Privileges

Non-superuser: object owner

Examples

The following example adds a comment from the curr_month_ship view:

```
=> COMMENT ON VIEW curr_month_ship IS 'Shipping data for the current month';
```

The following example removes a comment from the curr_month_ship view:

```
=> COMMENT ON VIEW curr_month_ship IS NULL;
```

COMMIT

Ends the current transaction and makes all changes that occurred during the transaction permanent and visible to other users.

COMMIT is a synonym for END

Syntax

```
COMMIT [ WORK | TRANSACTION ]
```

Parameters

WORK TRANSACTION

Optional keywords for readability only.

Privileges

None

Examples

This example shows how to commit an insert.

```
=> CREATE TABLE sample_table (a INT);
=> INSERT INTO sample_table (a) VALUES (1);
OUTPUT
--------
1
=> COMMIT;
```

See also

- Transactions
- Creating and rolling back transactions
- BEGIN
- ROLLBACK
- START TRANSACTION

CONNECT TO VERTICA

Connects to another Vertica database to enable importing and exporting data across Vertica databases, with COPY FROM VERTICA and EXPORT TO VERTICA, respectively.

After you establish a connection to another database, the connection remains open in the current session until you explicitly close it with DISCONNECT. You can have only one connection to another database at a time. However, you can establish successive connections to different databases in the same session.

By default, invoking CONNECT TO VERTICA occurs over the Vertica private network. For information about creating a connection over a public network, see Using public and private IP networks.

Important

Copy and export operations can fail if either side of the connection is a single-node cluster installed on localhost.

Syntax

```
CONNECT TO VERTICA db-spec USER username PASSWORD 'password' ON 'host', port
   [ TLS CONFIGURATION tls_configuration ]
   [ TLSMODE PREFER ]
```

Parameters

db-spec

The target database, either the database name or DEFAULT .

username

The username to use when connecting to the other database.

password

A string containing the password to use to connect to the target database.

If the target database has no password, and you supply one, the connection succeeds; however, Vertica returns no indication that you supplied an incorrect password.

host

A string containing the host name of one of the nodes in the other database.

port

The port number of the other database as an integer.

TLS CONFIGURATION *tls_configuration*

The TLS Configuration to use for TLS. The TLS Configuration is ignored if ImportExportTLSMode is set to any of the following:

- REQUIRE_FORCE
- VERIFY_CA_FORCE
- VERIFY_FULL_FORCE

The effective TLS mode of CONNECT TO VERTICA changes depending on the TLSMODE of the TLS Configuration and the value of ImportExportTLSMode (for non-FORCE values). For details, see Effective TLSMode .

TLSMODE PREFER

> **Deprecated**
> This parameter has been superseded by the TLS CONFIGURATION parameter. TLSMODE PREFER only takes effect if TLS CONFIGURATION is not set.

Overrides the value of configuration parameter ImportExportTLSMode for this connection to PREFER . If TLS CONFIGURATION is set or ImportExportTLSMode is set to REQUIRE_FORCE, VERIFY_CA_FORCE, or VERIFY_FULL_FORCE, then TLSMODE PREFER has no effect.
If TLSMODE PREFER and ImportExportTLSMode are both not set, CONNECT TO VERTICA uses ENABLE.

Effective TLS mode

The effective TLS mode of CONNECT TO VERTICA is determined by the TLSMODE of the TLS Configuration and the value of ImportExportTLSMode. The following table summarizes this interaction for non-FORCE values of ImportExportTLSMode:

TLS Configuration	ImportExportTLSMode	Effective TLS mode
ENABLE	PREFER	PREFER
ENABLE	Anything except PREFER	REQUIRE
TRY_VERIFY, VERIFY_CA	Anything	VERIFY_CA
VERIFY_FULL	Anything	VERIFY_FULL

Privileges

None

Security requirements

When importing from or exporting to a Vertica database, you can connect only to a database that uses trusted (username only) or password-based authentication, as described in Security and authentication . SSL, Kerberos, and OAuth authentication methods are not supported.

If configured with a certificate, Vertica encrypts data during transmission using TLS and attempts to encrypt plan metadata. You can set configuration parameter ImportExportTLSMode to require encryption for plan metadata.

Examples

```
=> CONNECT TO VERTICA ExampleDB USER dbadmin PASSWORD 'Password123' ON 'VerticaHost01',5433;
CONNECT
```

COPY

COPY bulk-loads data into a Vertica database. By default, COPY automatically commits itself and any current transaction except when loading temporary tables. If COPY is terminated or interrupted Vertica rolls it back.

COPY reads data as UTF-8 encoding.

For information on loading one or more files or pipes on a cluster host or on a client system, see COPY LOCAL .

Syntax

```
COPY [ /*+ LABEL (label-string)*/ ] [[database.]schema-name.]target-table
  [ ( { column-as-expression | column }
     [ DELIMITER [ AS ] 'char' ]
     [ ENCLOSED [ BY ] 'char' ]
     [ ENFORCELENGTH ]
     [ ESCAPE [ AS ] 'char' | NO ESCAPE ]
     [ FILLER datatype]
     [ FORMAT 'format' ]
     [ NULL [ AS ] 'string' ]
     [ TRIM 'byte' ]
     [,...] ) ]
  [ COLUMN OPTION (column
     [ DELIMITER [ AS ] 'char' ]
     [ ENCLOSED [ BY ] 'char' ]
     [ ENFORCELENGTH ]
     [ ESCAPE [ AS ] 'char' | NO ESCAPE ]
     [ FORMAT 'format' ]
     [ NULL [ AS ] 'string' ]
     [ TRIM 'byte' ]
    [,...] ) ]
FROM {
  [ LOCAL ] STDIN [ compression ]
  | { 'path-to-data'
     [ ON { nodename | (nodeset) | ANY NODE | EACH NODE } ] [ compression ] }[,...]
    [ PARTITION COLUMNS column[,...] ]
  | LOCAL 'path-to-data' [ compression ] [,...]
  | VERTICA source-database.[source-schema.]source-table[( source-column[,...] ) ]
 }
 [ NATIVE
   | FIXEDWIDTH COLSIZES {( integer )[,...]}
   | NATIVE VARCHAR
   | ORC
   | PARQUET
 ]
 | [ WITH ] UDL-clause[...]
}
  [ ABORT ON ERROR ]
  [ DELIMITER [ AS ] 'char' ]
  [ ENCLOSED [ BY ] 'char'
  [ ENFORCELENGTH ]
  [ ERROR TOLERANCE ]
  [ ESCAPE [ AS ] 'char' | NO ESCAPE ]
  [ EXCEPTIONS 'path' [ ON nodename] [,...]
  [ NULL [ AS ] 'string' ]
  [ RECORD TERMINATOR 'string' ]
  [ REJECTED DATA {'path' [ ON nodename] [,...] | AS TABLE reject-table} ]
  [ REJECTMAX integer ]
  [ SKIP integer ]
  [ SKIP BYTES integer ]
  [ STREAM NAME  'streamName']
  [ TRAILING NULLCOLS ]
  [ TRIM 'byte' ]
  [ [ WITH ] PARSER parser ([ arg=value[,...] ]) ] ]
  [ NO COMMIT ]
```

Parameters

See Parameters .

Restrictions

See Restrictions .

Privileges

Superusers have full COPY privileges. The following requirements apply to non-superusers:

- INSERT privilege on table
- USAGE privilege on schema
- USER-accessible storage location
- Applicable READ or WRITE privileges granted to the storage location where files are read or written

COPY can specify a path to store rejected data and exceptions. If the path resolves to a storage location, the following privileges apply to non-superusers:

- The storage location was created with the USER option (see CREATE LOCATION).
- The user must has READ access to the storage location, as described in GRANT (storage location)

In this section

- Examples
- Parameters
- Parsers
- Restrictions

Examples

For additional COPY examples, see the reference pages for specific parsers, including: DELIMITED , ORC , PARQUET , FJSONPARSER , and FAVROPARSER .

Specifying string options

Use COPY with FORMAT , DELIMITER , NULL , and ENCLOSED BY options:

```
=> COPY public.customer_dimension (customer_since FORMAT 'YYYY')
  FROM STDIN
  DELIMITER ','
  NULL AS 'null'
  ENCLOSED BY '"';
```

Use COPY with DELIMITER and NULL options. This example sets and references a vsql variable for the input file:

```
=> \set input_file ../myCopyFromLocal/large_table.gzip
=> COPY store.store_dimension
  FROM :input_file
  DELIMITER '|'
  NULL ''
  RECORD TERMINATOR E'\f';
```

Including multiple source files

Create a table and then copy multiple source files to it:

```
=> CREATE TABLE sampletab (a int);
CREATE TABLE

=> COPY sampletab FROM '/home/dbadmin/one.dat', 'home/dbadmin/two.dat';
 Rows Loaded
-------------
           2
(1 row)
```

Use wildcards to indicate a group of files:

```
=> COPY myTable FROM 'webhdfs:///mydirectory/ofmanyfiles/*.dat';
```

Wildcards can include regular expressions:

```
=> COPY myTable FROM 'webhdfs:///mydirectory/*_[0-9]';
```

Specify multiple paths in a single COPY statement:

```
=> COPY myTable FROM 'webhdfs:///data/sales/01/*.dat', 'webhdfs:///data/sales/02/*.dat',
   'webhdfs:///data/sales/historical.dat';
```

Distributing a load

Load data that is shared across all nodes. Vertica distributes the load across all nodes, if possible:

```
=> COPY sampletab FROM '/data/file.dat' ON ANY NODE;
```

Load data from two files. Because the first load file does not specify nodes (or ON ANY NODE), the initiator performs the load. Loading the second file is distributed across all nodes:

```
=> COPY sampletab FROM '/data/file1.dat', '/data/file2.dat' ON ANY NODE;
```

Specify different nodes for each load file:

```
=> COPY sampletab FROM '/data/file1.dat' ON (v_vmart_node0001, v_vmart_node0002),
   '/data/file2.dat' ON (v_vmart_node0003, v_vmart_node0004);
```

Loading data from shared storage

To load data from shared storage, use URLs in the corresponding schemes:

- HDFS : [[s]web]hdfs://[*nameservice*]/ *path*
- S3 : s3:// *bucket* / *path*
- Google Cloud : gs:// *bucket* / *path*
- Azure : azb:// *account* / *container* / *path*

> **Note**
> Loads from HDFS, S3, GCS, and Azure default to ON ANY NODE ; you do not need to specify it.

Load a file stored in HDFS using the default name node or name service:

```
=> COPY t FROM 'webhdfs:///opt/data/file1.dat';
```

Load data from a particular HDFS name service (testNS). You specify a name service if your database is configured to read from more than one HDFS cluster:

```
=> COPY t FROM 'webhdfs://testNS/opt/data/file2.csv';
```

Load data from an S3 bucket:

```
=> COPY t FROM 's3://AWS_DataLake/*' ORC;
```

Partitioned data

Data files can be partitioned using the directory structure, such as:

```
path/created=2016-11-01/region=northeast/*
path/created=2016-11-01/region=central/*
path/created=2016-11-01/region=southeast/*
path/created=2016-11-01/...
path/created=2016-11-02/region=northeast/*
path/created=2016-11-02/region=central/*
path/created=2016-11-02/region=southeast/*
path/created=2016-11-02/...
path/created=2016-11-03/...
path/...
```

Load partition columns using the PARTITION COLUMNS option:

```
=> CREATE EXTERNAL TABLE records (id int, name varchar(50), created date, region varchar(50))
  AS COPY FROM 'webhdfs:///path/*/*/*'
  PARTITION COLUMNS created, region;
```

Using filler columns

In the following example, the table has columns for first name, last name, and full name, but the data being loaded contains columns for first, middle,

and last names. The COPY statement reads all of the source data but only loads the source columns for first and last names. It constructs the data for the full name by concatenating each of the source data columns, including the middle name. The middle name is read as a FILLER column so it can be used in the concatenation, but is ignored otherwise. (There is no table column for middle name.)

```
=> CREATE TABLE names(first VARCHAR(20), last VARCHAR(20), full VARCHAR(60));
CREATE TABLE
=> COPY names(first,
              middle FILLER VARCHAR(20),
              last,
              full AS first||' '||middle||' '||last)
      FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> Marc|Gregory|Smith
>> Sue|Lucia|Temp
>> Jon|Pete|Hamilton
>> \.
=> SELECT * from names;
 first |   last   |        full
-------+----------+--------------------
 Jon   | Hamilton | Jon Pete Hamilton
 Marc  | Smith    | Marc Gregory Smith
 Sue   | Temp     | Sue Lucia Temp
(3 rows)
```

Loading data into a flex table

Create a Flex table and copy JSON data into it using FJSONPARSER :

```
=> CREATE FLEX TABLE darkdata();
CREATE TABLE
=> COPY tweets FROM '/myTest/Flexible/DATA/tweets_12.json' PARSER FJSONPARSER();
 Rows Loaded
-------------
          12
(1 row)
```

Using named pipes

COPY supports named pipes that follow the same naming conventions as file names on the given file system. Permissions are open , write , and close .

Create named pipe, pipe1 , and set two vsql variables:

```
=> \! mkfifo  pipe1
=> \set dir `pwd`/
=> \set file '':dir'pipe1'
```

Copy an uncompressed file from the named pipe:

```
=> \! cat pf1.dat > pipe1 &
=> COPY large_tbl FROM :file delimiter '|';
=> SELECT * FROM large_tbl;
=> COMMIT;
```

Loading compressed data

Copy a GZIP file from a named pipe and uncompress it:

```
=> \! gzip pf1.dat
=> \! cat pf1.dat.gz > pipe1 &
=> COPY large_tbl FROM :file ON site01 GZIP delimiter '|';
=> SELECT * FROM large_tbl;
=> COMMIT;
=> \!gunzip pf1.dat.gz
```

Parameters

COPY parameters and their descriptions are divided into the following sections:

- Target Options
- Column Options
- Input Options
- Handling Options
- Parser-Specific Options

Target options

The following options apply to the target tables and their columns:

LABEL

Assigns a label to a statement to identify it for profiling and debugging.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.
COPY ignores *schema-name* when used in CREATE EXTERNAL TABLE or CREATE FLEX EXTERNAL TABLE statements.

target-table

The target columnar or flexible table for loading new data. Vertica loads the data into all projections that include columns from the schema table.

column-as-expression

An expression used to compute values for the target column, which must not be of a complex type. For example:

```
=> COPY t(year AS TO_CHAR(k, 'YYYY')) FROM 'myfile.dat'
```

Use this option to transform data when it is loaded into the target database.
For details, see Transforming data during loads .

column

Restricts the load to one or more specified columns in the table. If you omit specifying columns, COPY loads all columns by default.

Table columns that you omit from the column list are assigned their DEFAULT or SET USING values, if any; otherwise, COPY inserts NULL .
If you leave the *column* parameter blank to load all columns in the table, you can use the optional parameter COLUMN OPTION to specify parsing options for specific columns.

The data file must contain the same number of columns as the COPY command's column list.

COLUMN OPTION

Specifies load metadata for one or more columns declared in the table column list. For example, you can specify that a column has its own DELIMITER , ENCLOSED BY , or NULL AS expression, and so on. You do not have to specify every column name explicitly in the COLUMN OPTION list, but each column you specify must correspond to a column in the table column list.

Column options

Depending on how they are specified, the following COPY options can qualify specific columns or all columns. Some parser-specific options can also apply to either specific columns or all columns. See Global and column-specific options For details about these two modes.

ENFORCELENGTH

If specified, COPY rejects data rows of type CHAR , VARCHAR , BINARY , and VARBINARY , or elements of those types in collections, if they are larger than the declared size.

By default, COPY truncates offending rows of these data types and elements of these types in collections, but does not reject the rows. For more details, see Handling Messy Data .
If a collection does not fit with all of its elements, COPY rejects the row without truncating. It does not reduce the number of elements. This can happen if each element is individually within limits but the number of elements causes the collection to exceed the maximum size for the column.

FILLER *datatype*

Reads but does not copy the data of an input column. Use filler columns to ignore input columns that do not have columns in the table. You can also use filler columns to transform data (see Examples and Transforming data during loads). Filler columns cannot be of complex types.

FORMAT ' *format* '

Input format, one of the following:

- octal

- hex
- bitstream

See Binary (native) data to learn more about these formats.
When loading date/time columns, using FORMAT significantly improves load performance. COPY supports the same formats as the TO_DATE function. See Template patterns for date/time formatting.

If you specify invalid format strings, the COPY operation returns an error.

NULL [AS]

The string representing a null value. The default is an empty string (''). You can specify a null value as any ASCII value in the range E'\000' to E'\177' inclusive. You cannot use the same character for both the DELIMITER and NULL options. For details, see Delimited data.

Input options

The following options are available for specifying source data:

LOCAL

Loads data files (up to 65,535 files) on a client system, rather than on a cluster host. LOCAL can qualify the STDIN and *[path-to-data] (#pathToData)* parameters. For details, see COPY LOCAL.

Restrictions: Invalid for CREATE EXTERNAL TABLE AS COPY

STDIN

Reads from the client a standard input instead of a file. STDIN takes one input source only. To load multiple input sources, use *[path-to-data] (#pathToData)* .

User must have INSERT privileges on the table and USAGE privileges on its schema.
Restrictions: Invalid for CREATE EXTERNAL TABLE AS COPY

path-to-data

Specifies the absolute path of the file (or files) containing the data, which can be from multiple input sources.

- If the file is stored in HDFS, *path-to-data* is a URI in the webhdfs scheme, typically [[s]web]hdfs://[*nameservice*]/ *path* . See HDFS file system.
- If the file is stored in an S3 bucket, *path-to-data* is a URI in the format s3:// *bucket* / *path* . See S3 object store.
- If the file is stored in Google Cloud Storage, *path-to-data* is a URI in the format gs:// *bucket* / *path* . See Google Cloud Storage (GCS) object store.
- If the file is stored in Azure Blob Storage, *path-to-data* is a URI in the format azb:// *account* / *container* / *path* . See Azure Blob Storage object store.
- If the file is on the local Linux file system or an NFS mount, *path-to-data* is a local absolute file path.

path-to-data can optionally contain wildcards to match more than one file. The file or files must be accessible to the local client or the host on which the COPY statement runs. COPY skips empty files in the file list. A file list that includes directories causes the query to fail. See Specifying where to load data from. The supported patterns for wildcards are specified in the Linux Manual Page for Glob (7), and for ADO.net platforms, through the .NET Directory.getFiles method.
You can use variables to construct the pathname as described in Using load scripts.

If *path-to-data* resolves to a storage location on a local file system, and the user invoking COPY is not a superuser, the following requirements apply:

- The storage location was created with CREATE LOCATION...USAGE USER.
- The user must already have READ access to the file storage location.

Further, if a user has privileges but is not a superuser, and invokes COPY from that storage location, Vertica ensures that symbolic links do not result in unauthorized access.

PARTITION COLUMNS *column* [,...]

Columns whose values are specified in the directory structure and not in the data itself. This option is valid if the path contains directory names of the form *colname* = *value* , such as:

```
/data/created=2016-01-01/*.dat
/data/created=2016-01-02/*.dat
/data/created=2016-01-03/*.dat
```

The value is parsed from the *value* part of the directory name. If it cannot be coerced to the column data type, it is rejected. If the value is missing, COPY loads it as null.
The following example loads these dates into a table column, along with other columns in the data files:

```
=> CREATE EXTERNAL TABLE records
   (id int, name varchar(50), created date)
  AS COPY FROM 'webhdfs:///path/*/*'
  PARTITION COLUMNS created;
```

For more information, see Partitioned file paths .

ON *nodename*

Specifies the node on which the data to copy resides and the node that should parse the load file. If you omit *nodename* , the location of the input file defaults to the initiator node. Use *nodename* to copy and parse a load file from a node other than the COPY initiator node.

> **Note**
> *nodename* is invalid with STDIN and LOCAL .

ON (*nodeset*)

Specifies a set of nodes on which to perform the load. The same data must be available for load on all named nodes. *nodeset* is a comma-separated list of node names in parentheses. For example:

```
=> COPY t FROM 'file1.txt' ON (v_vmart_node0001, v_vmart_node0002);
```

Vertica apportions the load among all of the specified nodes. If you also specify ERROR TOLERANCE or REJECTMAX , Vertica instead chooses a single node on which to perform the load.
If the data is available on all nodes, you usually use ON ANY NODE , which is the default for loads from HDFS and cloud object stores. However, you can use ON *nodeset* to do manual load-balancing among concurrent loads.

ON ANY NODE

Specifies that the data to load is available on all nodes, so COPY opens the path and parses it from any node in the cluster. For an Eon Mode database, COPY uses nodes within the same subcluster as the initiator.

> **Caution**
> The data must be the same on all nodes. If the data differs on two nodes, an incorrect or incomplete result is returned, with no error or warning.

Vertica attempts to apportion the load among several nodes if a file is large enough to benefit from apportioning. It chooses a single node if ERROR TOLERANCE or REJECTMAX is specified.
You can use a wildcard or glob (such as *.dat) to load multiple input files, combined with the ON ANY NODE clause. If you use a glob, COPY distributes the list of files to all cluster nodes and spreads the workload.

ON ANY NODE is invalid with STDIN and LOCAL . STDIN can only use the client host, and LOCAL indicates a client node.

ON ANY NODE is the default for loads from all paths other than Linux (HDFS and cloud object stores).

ON EACH NODE

Loads data from the specified path on each node. Use this option when the path exists on all nodes but the data files it contains are different on each node. If the path is not valid on all nodes, COPY loads the valid paths and produces a warning. If the path is a shared location, COPY loads it only once as for ON ANY NODE .

compression

The input compression type, one of the following:

- UNCOMPRESSED (default)
- BZIP
- GZIP
- LZO
- ZSTD

Input files can be of any format. If you use wildcards, all qualifying input files must be in the same format. To load different file formats, specify the format types specifically.
The following requirements and restrictions apply:

- When using concatenated BZIP or GZIP files, verify that all source files terminate with a record terminator before concatenating them.
- Concatenated BZIP and GZIP files are not supported for NATIVE (binary) and NATIVE VARCHAR formats.

- LZO files are assumed to be compressed with lzop . Vertica supports the following lzop arguments :
  - --no-checksum / -F
  - --crc32
  - --adler32
  - --no-name / -n
  - --name / -N
  - --no-mode
  - --no-time
  - --fast
  - --best
  - Numbered compression levels
- BZIP , GZIP , ZSTD , and LZO compression cannot be used with ORC format.

VERTICA

See COPY FROM VERTICA .

[WITH] *UDL-clause* [...]

Specifies one or more user-defined load functions —one source, and optionally one or more filters and one parser, as follows:

```
SOURCE source( [arg=value[,...] ]
[ FILTER filter( [arg=value[,...] ] ) ]...
[ PARSER parser( [arg=value[,...] ] ) ]
```

To use a flex table parser for column tables, use the PARSER parameter followed by a flex table parser argument. For supported flex table parsers, see Bulk loading data into flex tables .

Handling options

The following options control how COPY handles different contingencies:

ABORT ON ERROR

Specifies that COPY stops if any row is rejected. The statement is rolled back and no data is loaded.

COLSIZES (*integer* [,...])

Specifies column widths when loading fixed-width data. COPY requires that you specify COLSIZES when using the FIXEDWIDTH parser. COLSIZES and the list of integers must correspond to the columns listed in the table column list. For details, see Fixed-width format data .

ERROR TOLERANCE

Specifies that COPY treats each source during execution independently when loading data. The statement is not rolled back if a single source is invalid. The invalid source is skipped and the load continues.

Using this parameter disables apportioned load.
Restrictions: Invalid for ORC or Parquet data

EXCEPTIONS

Specifies the file name or absolute path of the file in which to write exceptions, as follows:

```
EXCEPTIONS 'path' [ ON nodename[,...]]
```

Exceptions describe why each rejected row was rejected. Each exception describes the corresponding record in the file specified by the REJECTED DATA option.
Files are written on the node or nodes executing the load. If the file already exists, it is overwritten.

To collect all exceptions in one place, use the REJECTED DATA AS TABLE clause and exceptions are automatically listed in the table's rejected_reason column.

> **Note**
> EXCEPTIONS is incompatible with REJECTED DATA AS TABLE .

The ON *nodename* clause moves existing exceptions files on *nodename* to the indicated *path* on the same node. For details, see Saving load exceptions (EXCEPTIONS) .

If you use this parameter with COPY...ON ANY NODE , you must still specify the individual nodes for the exception files, as in the following example:

```
EXCEPTIONS '/home/ex01.txt' on v_db_node0001,'/home/ex02.txt'
on v_db_node0002,'/home/ex03.txt' on v_db_node0003
```

If *path* resolves to a storage location, the following privileges apply to non-superusers:

- The storage location must be created with the USER option (see CREATE LOCATION).
- The user must have READ access to the storage location where the files exist, as described in GRANT (storage location) .

REJECTED DATA

Specifies where to write each row that failed to load. If this parameter is specified, records that failed due to parsing errors are always written. Records that failed due to an error during a transformation are written only if configuration parameter CopyFaultTolerantExpression s is set.

The syntax for this parameter is:

```
REJECTED DATA
{ 'path' [ ON nodename ] [,...] | AS TABLE reject-table }
```

Vertica can write rejected data to the specified path or to a table:

- ' *path* ' [ON *nodename*] : Copies the rejected row data to the specified path on the node executing the load. If qualified by ON *nodename* , Vertica moves existing rejected data files on *nodename* to *path* on the same node.
  The value of *path* can be a directory or a file prefix. If there are multiple load sources, *path* is always treated as a directory. If there are not multiple load sources but *path* ends with ' / ', or if a directory of that name already exists, it is also treated as a directory. Otherwise, *path* is treated as a file prefix.
  Files are written on the node or nodes executing the load. If the file already exists, it is overwritten.
  When this parameter is used with LOCAL , the output is written to the client.

  > **Note**
  > Do not qualify *path* with ON ANY NODE . To collect all rejected data in one place regardless of how the load is distributed, use a table.

- AS TABLE *reject-table* : Saves rejected rows to *reject-table* .

  > **Note**
  > REJECTED DATA AS TABLE is incompatible with EXCEPTIONS .

For details about both options, see Handling messy data .

REJECTMAX *integer*

The maximum number of logical records that can be rejected before a load fails. For details, see Handling messy data .

REJECTMAX disables apportioned load.

SKIP *integer*

The number of records to skip in a load file. For example, you can use the SKIP option to omit table header information.

Restrictions: Invalid for ORC or Parquet data

STREAM NAME

Supplies a COPY load stream identifier. Using a stream name helps to quickly identify a particular load. The STREAM NAME value that you supply in the load statement appears in the STREAM_NAME column of system tables LOAD_STREAMS and LOAD_SOURCES .

A valid stream name can contain any combination of alphanumeric or special characters up to 128 bytes in length.
For example:

```
=> COPY mytable FROM myfile
  DELIMITER '|' STREAM NAME 'My stream name';
```

WITH *parser*

Specifies the parser to use when bulk loading columnar tables, one of the following:

- NATIVE
- NATIVE VARCHAR
- FIXEDWIDTH

- ORC
- PARQUET

By default, COPY uses the DELIMITER parser for UTF-8 format, delimited text input data. You do not specify the DELIMITER parser directly; absence of a specific parser indicates the default.
To use a flex table parser for column tables, use the PARSER parameter followed by a flex table parser argument. For supported flex table parsers, see Bulk loading data into flex tables .

When loading into flex tables, you must use a compatible parser. For supported flex table parsers, see Bulk loading data into flex tables .

COPY LOCAL does not support the NATIVE , NATIVE VARCHAR , ORC , and PARQUET parsers.

For parser support for complex data types, see the documentation of the specific parser.

For parser details, see Data formats in Data load .

NO COMMIT

Prevents the COPY statement from committing its transaction automatically when it finishes copying data. This option must be the last COPY statement parameter.

For details, see Using transactions to stage a load .
Restrictions: Invalid for ORC or Parquet data, ignored by CREATE EXTERNAL TABLE AS COPY

Parser-specific options

The following options apply only when using specific parsers.

DELIMITED parser

DELIMITER

Indicates the single ASCII character used to separate columns within each record of a file. You can use any ASCII value in the range E'\000' to E'\177', inclusive. You cannot use the same character for both the DELIMITER and NULL parameters. For more information, see Delimited data .
Default: Vertical bar ('|').

ENCLOSED [BY]

Sets the quote character within which to enclose data, allowing delimiter characters to be embedded in string values. You can choose any ASCII value in the range E'\001' to E'\177' inclusive (any ASCII character except NULL: E'\000'). By default, ENCLOSED BY has no value, meaning data is not enclosed by any sort of quote character.

ESCAPE [AS]

Sets the escape character. Once set, the character following the escape character is interpreted literally, rather than as a special character. You can define an escape character using any ASCII value in the range E'\001' to E'\177', inclusive (any ASCII character except NULL: E'\000').
The COPY statement does not interpret the data it reads in as String literals . It also does not follow the same escape rules as other SQL statements (including the COPY parameters). When reading data, COPY interprets only the characters defined by these options as special values:

- ESCAPE [AS]
- DELIMITER
- ENCLOSED [BY]
- RECORD TERMINATOR
- All COLLECTION options

Default: Backslash ('\').

NO ESCAPE

Eliminates escape-character handling. Use this option if you do not need any escape character and you want to prevent characters in your data from being interpreted as escape sequences.

RECORD TERMINATOR

Specifies the literal character string indicating the end of a data file record. For more information about using this parameter, see Delimited data .

TRAILING NULLCOLS

Specifies that if Vertica encounters a record with insufficient data to match the columns in the table column list, COPY inserts the missing columns with NULL values. For other information and examples, see Fixed-width format data .

COLLECTIONDELIMITER

For columns of collection types, indicates the single ASCII character used to separate elements within each collection. You can use any ASCII value in the range E'\000' to E'\177', inclusive. No COLLECTION option may have the same value as any other COLLECTION option. For more information, see Delimited data .
Default: Comma (',').

COLLECTIONOPEN , COLLECTIONCLOSE

For columns of collection types, these options indicate the characters that mark the beginning and end of the collection. It is an error to use these characters elsewhere within the list of elements without escaping them. No COLLECTION option may have the same value as any other COLLECTION option.
Default: Square brackets ('[' and ']').

COLLECTIONNULLELEMENT

The string representing a null element value in a collection. You can specify a null value as any ASCII value in the range E'\001' to E'\177' inclusive (any ASCII value except NULL: E'\000'). No COLLECTION option may have the same value as any other COLLECTION option. For more information, see Delimited data .
Default: 'null'

COLLECTIONENCLOSE

For columns of collection types, sets the quote character within which to enclose individual elements, allowing delimiter characters to be embedded in string values. You can choose any ASCII value in the range E'\001' to E'\177' inclusive (any ASCII character except NULL: E'\000'). No COLLECTION option may have the same value as any other COLLECTION option.

Default: double quote ('"')

FIXEDWIDTH parser

SKIP BYTES ***integer***

The total number of bytes in a record to skip.

TRIM

Trims the number of bytes you specify from a column. This option is only available when loading fixed-width data. You can set TRIM at the table level for a column, or as part of the COLUMN OPTION parameter.

Parsers

Vertica supports several parsers to load different types of data. Some parsers are for use only with flex tables, as noted.

In this section

- DELIMITED
- FAVROPARSER
- FCEFPARSER
- FCSVPARSER
- FDELIMITEDPAIRPARSER
- FDELIMITEDPARSER
- FJSONPARSER
- FREGEXPARSER
- ORC
- PARQUET

DELIMITED

Use the DELIMITED parser, which is the default, to load delimited text data using COPY . You can specify the delimiter, escape characters, how to handle null values, and other parameters.

The DELIMITED parser supports both apportioned load and cooperative parse .

COPY options

The following options are specific to this parser. See Parameters for other applicable options.

DELIMITER

Indicates the single ASCII character used to separate columns within each record of a file. You can use any ASCII value in the range E'\000' to E'\177', inclusive. You cannot use the same character for both the DELIMITER and NULL parameters. For more information, see Delimited data .
Default: Vertical bar ('|').

ENCLOSED [BY]

Sets the quote character within which to enclose data, allowing delimiter characters to be embedded in string values. You can choose any ASCII value in the range E'\001' to E'\177' inclusive (any ASCII character except NULL: E'\000'). By default, ENCLOSED BY has no value, meaning data is not enclosed by any sort of quote character.

ESCAPE [AS]

Sets the escape character. Once set, the character following the escape character is interpreted literally, rather than as a special character. You can define an escape character using any ASCII value in the range E'\001' to E'\177', inclusive (any ASCII character except NULL: E'\000').
The COPY statement does not interpret the data it reads in as String literals . It also does not follow the same escape rules as other SQL statements (including the COPY parameters). When reading data, COPY interprets only the characters defined by these options as special values:

- ESCAPE [AS]
- DELIMITER
- ENCLOSED [BY]
- RECORD TERMINATOR
- All COLLECTION options

Default: Backslash ('\').

NO ESCAPE

Eliminates escape-character handling. Use this option if you do not need any escape character and you want to prevent characters in your data from being interpreted as escape sequences.

RECORD TERMINATOR

Specifies the literal character string indicating the end of a data file record. For more information about using this parameter, see Delimited data .

TRAILING NULLCOLS

Specifies that if Vertica encounters a record with insufficient data to match the columns in the table column list, COPY inserts the missing columns with NULL values. For other information and examples, see Fixed-width format data .

COLLECTIONDELIMITER

For columns of collection types, indicates the single ASCII character used to separate elements within each collection. You can use any ASCII value in the range E'\000' to E'\177', inclusive. No COLLECTION option may have the same value as any other COLLECTION option. For more information, see Delimited data .
Default: Comma (',').

COLLECTIONOPEN , COLLECTIONCLOSE

For columns of collection types, these options indicate the characters that mark the beginning and end of the collection. It is an error to use these characters elsewhere within the list of elements without escaping them. No COLLECTION option may have the same value as any other COLLECTION option.
Default: Square brackets ('[' and ']').

COLLECTIONNULLELEMENT

The string representing a null element value in a collection. You can specify a null value as any ASCII value in the range E'\001' to E'\177' inclusive (any ASCII value except NULL: E'\000'). No COLLECTION option may have the same value as any other COLLECTION option. For more information, see Delimited data .
Default: 'null'

COLLECTIONENCLOSE

For columns of collection types, sets the quote character within which to enclose individual elements, allowing delimiter characters to be embedded in string values. You can choose any ASCII value in the range E'\001' to E'\177' inclusive (any ASCII character except NULL: E'\000'). No COLLECTION option may have the same value as any other COLLECTION option.

Default: double quote ('"')

Data types

The DELIMITED parser supports reading one-dimensional collections (arrays or sets) of scalar types.

If the total size of an array exceeds the size defined by the target table, the parser rejects the row.

Examples

The following example shows the default behavior, in which the delimiter character is '|'

```
=> CREATE TABLE employees (id INT, name VARCHAR(50), department VARCHAR(50));
CREATE TABLE

=> COPY employees FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 42|Sheldon Cooper|Physics
>> 17|Howard Wolowitz|Astronomy
>> \.

=> SELECT * FROM employees;
 id |      name       |  department
----+-----------------+--------------
 17 | Howard Wolowitz | Astrophysics
 42 | Sheldon Cooper  | Physics
(2 rows)
```

The following example shows loading array values with the default options.

```
=> CREATE TABLE researchers (id INT, name VARCHAR, grants ARRAY[VARCHAR], values ARRAY[INT]);
CREATE TABLE

=> COPY researchers FROM STDIN;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 42|Sheldon Cooper|[US-7376,DARPA-1567]|[65000,135000]
>> 17|Howard Wolowitz|[NASA-1683,NASA-7867,SPX-76]|[16700,85000,45000]
>> \.

=> SELECT * FROM researchers;
 id |      name       |              grants               |       values
----+-----------------+-----------------------------------+---------------------
 17 | Howard Wolowitz | ["NASA-1683","NASA-7867","SPX-76"] | [16700,85000,45000]
 42 | Sheldon Cooper  | ["US-7376","DARPA-1567"]           | [65000,135000]
(2 rows)
```

In the following example, collections are enclosed in braces and delimited by periods, and the arrays contain null values.

```
=> COPY researchers FROM STDIN COLLECTIONOPEN '{' COLLECTIONCLOSE '}' COLLECTIONDELIMITER '.';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 19|Leonard|{"us-1672".null."darpa-1963"}|{16200.null.16700}
>> \.

=> SELECT * FROM researchers;
 id |      name       |              grants               |       values
----+-----------------+-----------------------------------+---------------------
 17 | Howard Wolowitz | ["NASA-1683","NASA-7867","SPX-76"] | [16700,85000,45000]
 42 | Sheldon Cooper  | ["US-7376","DARPA-1567"]           | [65000,135000]
 19 | Leonard         | ["us-1672",null,"darpa-1963"]      | [16200,null,16700]
(3 rows)
```

FAVROPARSER

Parses data from an Avro file. The following requirements apply:

- Avro files must be encoded in the Avro binary serialization encoding format, described in the Apache Avro standard . The parser also supports Snappy and deflate compression.
- FAVROPARSER does not support Avro files with separate schema files. The Avro file must include the schema.

You can load complex types in the Avro source (arrays, structs, or combinations) with strong typing or as flexible complex types . A flexible complex type is loaded into a VMap column, as in flex tables. To load complex types as VMap columns, specify a column type of LONG VARBINARY . To preserve the indexing in complex types, set flatten_maps to false.

This parser can notify you if it finds keys in the data that are not part of the table definition. See Unmatched Keys .

When loading into a flex table, Vertica loads all data into the __raw__ (VMap) column, including complex types found in the data.

This parser does not support apportioned load or cooperative parse .

Syntax

```
FAVROPARSER ( [parameter=value[,...]] )
```

Parameters

flatten_maps

Boolean, whether to flatten all Avro maps. Key names are concatenated with nested levels. This value is recursive and affects all data in the load.

This parameter applies only to flex tables or VMap columns and is ignored when loading strongly-typed complex types.
Default: true

flatten_arrays

Boolean, whether to flatten all Avro arrays. Key names are concatenated with nested levels. This value is recursive and affects all data in the load.

This parameter applies only to flex tables or VMap columns and is ignored when loading strongly-typed complex types.
Default: false

flatten_records

Boolean, whether to flatten all Avro records. Key names are concatenated with nested levels. This value is recursive and affects all data in the load.

This parameter applies only to flex tables or VMap columns and is ignored when loading strongly-typed complex types.
Default: true

reject_on_materialized_type_error

Boolean, whether to reject a data row that contains a materialized column value that cannot be coerced into a compatible data type. If the value is false and the type cannot be coerced, the parser sets the value in that column to NULL.
If the column is an array and the data to be loaded is too large, then false sets the column value to NULL and true rejects the row.

If the column is a strongly-typed complex type , as opposed to a flexible complex type , then a type mismatch anywhere in the complex type causes the entire column to be treated as a mismatch. The parser does not partially load complex types.

Default: false

suppress_warnings

String, which warnings to suppress:

- unmatched_key (see Unmatched Keys)
- true or t (suppress all warnings)
- false or f (do not suppress warnings)

Default: false

Primitive data types

FAVROPARSER supports the following primitive data types, including as element types and field values in complex types.

AVRO Data Type	Vertica Data Type	Value
NULL	NULL value	No value

boolean	Boolean data type	A binary value
int	INTEGER	32-bit signed integer
long	INTEGER	64-bit signed integer
float	DOUBLE PRECISION (FLOAT) Synonymous with 64-bit IEEE FLOAT	Single precision (32-bit) IEEE 754 floating-point number
double	DOUBLE PRECISION (FLOAT)	Double precision (64-bit) IEEE 754 floating-point number
bytes	VARBINARY	Sequence of 8-bit unsigned bytes
string	VARCHAR	Unicode character sequence

Note
Vertica does not have an explicit 4-byte (32-bit integer) or smaller types. Instead, Vertica encoding and compression automatically eliminate the storage overhead of values that require less than 64 bits.

Avro logical types

FAVROPARSER supports the following Avro logical types. The target column must use a Vertica data type that supports the logical type. When you attempt to load data using an invalid logical type, the logical type is ignored and the underlying Avro type is used.

AVRO Logical Type	Base Avro Type	Supported Vertica Data Types
decimal 0 < *precision* ≤ 1024 0 ≤ *scale* ≤ *precision*	bytes or fixed	NUMERIC , Character Vertica rejects the value if: • The Avro precision setting is greater than the precision setting for the target column. • For fixed types, the precision value is greater than what is allowed by the size attribute. If the data type for the target column uses the default precision setting, the precision setting in the Avro schema overrides the default.
date	integer	DATE , Character
time-micros	long	TIME/TIMETZ , Character The time logical type does not provide a time zone value. For target columns that use the TIMETZ data type, Vertica uses UTC as the default.
time-millis	int	
timestamp-micros	long	TIMESTAMP/TIMESTAMPTZ , TIME/TIMETZ For timestamp-millis only, the timezone is included and is represented as an offset to UTC. Additionally, the millisecond values are right-extended with padded zeros.
timestamp-millis	long	
duration	fixed	INTERVAL , Character

Avro complex data types

The Avro format supports several complex data types. When loading into strongly-typed columns, you can use the ROW and ARRAY types to represent them. For example, Avro Record and Enums are structs (ROWs); see the Avro specification .

You can use ARRAY[ROW] to match an Avro map. You must name the ROW fields key and value . These are the names that the Avro format uses for

those fields in the data, and the parser relies on field names to match data to table columns.

If the total size of an array exceeds the size defined by the target table, the parser sets the value to null.

When loading into flex tables or using flexible complex types, this parser handles Avro complex types as follows:

Record

The name of each field is used as a virtual column name. If flatten_records is true and several nesting levels are present, Vertica concatenates the record names to create the key name.

Map

The value of each map key is used as a virtual column name. If flatten_maps is true and several nesting levels are present, Vertica concatenates the key names to create the key name.

Enum

Vertica treats Avro Enums like records, with the name of the Enum as the key and the value as the value.

Array

Vertica treats Avro Arrays as key/value pairs. By default, the index of each element is the key. In the following example, product_detail is a Record with a field, product_category , that is an Array:

```
=> CREATE FLEX TABLE products;
CREATE TABLE

=> COPY products FROM :datafile WITH PARSER FAVROPARSER();
 Rows Loaded
-------------
           2
(1 row)

=> SELECT MAPTOSTRING(__raw__) FROM products ORDER BY __identity__;
                    maptostring
-------------------------------------------------------------------------------
 {
    "__name__": "Order",
    "customer_id": "111222",
    "order_details": {
        "0.__name__": "OrderDetail",
        "0.product_detail.__name__": "Product",
        "0.product_detail.price": "46.21",
        "0.product_detail.product_category": {
            "0": "electronics",
            "1": "printers",
            "2": "computers"
        },
        "0.product_detail.product_description": "hp printer X11ew description :\
P",
        "0.product_detail.product_hash": "\u0000\u0001\u0002\u0003\u0004",
        "0.product_detail.product_id": "999012",
        "0.product_detail.product_map.one": "1.1",
        "0.product_detail.product_map.two": "1.1",
        "0.product_detail.product_name": "hp printer X11ew",
        "0.product_detail.product_status": "ONLY_FEW_LEFT",
        "0.quantity": "3",
        "0.total": "354.34"
    },
    "order_id": "2389646",
    "total": "132.43"
}
...
```

If flatten_arrays is true and several nesting levels are present, Vertica concatenates the indices to create the key name.

```
=> COPY products FROM :datafile WITH PARSER FAVROPARSER(flatten_arrays=true);
 Rows Loaded
-------------
           2
(1 row)

=> SELECT MAPTOSTRING(__raw__) FROM products ORDER BY __identity__;
                    maptostring
-------------------------------------------------------------------------------


 {
    "__name__": "Order",
    "customer_id": "111222",
    "order_details.0.__name__": "OrderDetail",
    "order_details.0.product_detail.__name__": "Product",
    "order_details.0.product_detail.price": "46.21",
    "order_details.0.product_detail.product_category.0": "electronics",
    "order_details.0.product_detail.product_category.1": "printers",
    "order_details.0.product_detail.product_category.2": "computers",
    "order_details.0.product_detail.product_description": "hp printer X11ew des\
cription :P",
    "order_details.0.product_detail.product_hash": "\u0000\u0001\u0002\u0003\u0\
004",
    "order_details.0.product_detail.product_id": "999012",
    "order_details.0.product_detail.product_map.one": "1.1",
    "order_details.0.product_detail.product_map.two": "1.1",
    "order_details.0.product_detail.product_name": "hp printer X11ew",
    "order_details.0.product_detail.product_status": "ONLY_FEW_LEFT",
    "order_details.0.quantity": "3",
    "order_details.0.total": "354.34",
    "order_id": "2389646",
    "total": "132.43"
}
...
```

Union

Vertica treats Avro Unions as arrays.

Unmatched keys

Data being loaded can contain keys that are not part of the table definition. If you are loading into a flex table (or a flexible complex type column), no data is lost. For a table with strongly-defined columns, however, new keys cannot be loaded because the table does not have a place to put them.

This parser emits warnings if it finds new keys and if both of the following are true:

- The target table is not a flex table.
- The new key is not nested within a flexible complex type column.

New keys are logged in the UDX_EVENTS system table. If a new key is a complex type with nested keys, only the top-level key is logged. When you see a warning about unmatched keys, you can query this table and then use ALTER TABLE to modify your table definition for future loads.

Querying an external table loads data and thus can trigger these warnings. To prevent them, set the suppress_warnings parameter to 'unmatched_keys' or 'true':

```
=> CREATE EXTERNAL TABLE restaurants(
        name VARCHAR(50),
        menu ARRAY[ROW(item VARCHAR(50), price NUMERIC(8,2)),100])
  AS COPY FROM '/data/rest.json'
  PARSER FAVROPARSER(suppress_warnings='unmatched_key');
```

Examples

This example shows how to create and load a flex table with Avro data using favroparser . After loading the data, you can query virtual columns:

```
=> CREATE FLEX TABLE avro_basic();
CREATE TABLE

=> COPY avro_basic FROM '/home/dbadmin/data/weather.avro' PARSER FAVROPARSER();
Rows Loaded
-------------
5
(1 row)

=> SELECT station, temp, time FROM avro_basic;
station | temp |     time
---------+------+---------------
mohali  | 0    | -619524000000
lucknow | 22   | -619506000000
norwich | -11  | -619484400000
ams     | 111  | -655531200000
baddi   | 78   | -655509600000
(5 rows)
```

For more information, see Avro data .

FCEFPARSER

Parses ArcSight Common Event Format (CEF) log files. This parser loads values directly into any table column with a column name that matches a source data key. The parser stores the data loaded into a flex table in a single VMap.

This parser is for use in Flex tables only. All flex parsers store the data as a single VMap in the LONG VARBINAR_raw__ column. If a data row is too large to fit in the column, it is rejected. Vertica supports null values for loading data with NULL -specified columns.

Syntax

```
FCEFPARSER ( [parameter-name='value'[,...]] )
```

Parameters

delimiter

Single-character delimiter.

Default: ' '

record_terminator

Single-character record terminator.

**Default ****value: ** newline

trim

Boolean, specifies whether to trim white space from header names and key values.

Default: true

reject_on_unescaped_delimiter

Boolean, specifies whether to reject rows containing unescaped delimiters. The CEF standard does not permit them.

Default: false

Examples

The following example illustrates creating a sample flex table for CEF data, with two real columns, eventId and priority .

1. Create a flex table cefdata :

```
=> create flex table cefdata();
CREATE TABLE
```

2. Load some basic CEF data, using the flex parser fcefparser :

```
=> copy cefdata from stdin parser fcefparser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> CEF:0|ArcSight|ArcSight|2.4.1|machine:20|New alert|High|
>> \.
```

3. Use the maptostring() function to view the contents of your cefdata flex table:

```
=> select maptostring(__raw__) from cefdata;
                maptostring
-------------------------------------------------------------
 {
   "deviceproduct" : "ArcSight",
   "devicevendor" : "ArcSight",
   "deviceversion" : "2.4.1",
   "name" : "New alert",
   "severity" : "High",
   "signatureid" : "machine:20",
   "version" : "0"
}


(1 row)
```

4. Select some virtual columns from the cefdata flex table:

```
= select deviceproduct, severity, deviceversion from cefdata;
 deviceproduct | severity | deviceversion
---------------+----------+---------------
 ArcSight      | High     | 2.4.1
(1 row)
```

For more information, see Common event format (CEF) data

See also

- FAVROPARSER
- FCSVPARSER
- FDELIMITEDPARSER
- FDELIMITEDPAIRPARSER
- FJSONPARSER
- FREGEXPARSER

FCSVPARSER

Parses CSV format (comma-separated values) data. Use this parser to load CSV data into columnar, flex, and hybrid tables. All data must be encoded in Unicode UTF-8 format. The fcsvparser parser supports the RFC 4180 standard for CSV data, and other options, to accommodate variations in CSV file format definitions. Invalid records are rejected. For more information about data formats, see Handling Non-UTF-8 input .

This parser is for use in Flex tables only. All flex parsers store the data as a single VMap in the LONG VARBINAR_raw__ column. If a data row is too large to fit in the column, it is rejected. Vertica supports null values for loading data with NULL -specified columns.

Syntax

```
FCSVPARSER ( [parameter='value'[,...]] )
```

Parameters

type

The default parameter values for the parser, one of the following strings:

- rfc4180
- traditional

You do not have to use the type parameter when loading data that conforms to the RFC 4180 standard (such as MS Excel files). See Loading CSV data for the RFC4180 default parameters, and other options you can specify for traditional CSV files.

Default: RFC4180

delimiter

A single-character value used to separate fields in the CSV data.

Default: , (for rfc4180 and traditional)

escape

A single-character value used as an escape character to interpret the next character in the data literally.

Default:

- rfc4180 : "
- traditional : \

enclosed_by

A single-character value. Use enclosed_by to include a value that is identical to the delimiter, but should be interpreted literally. For example, if the data delimiter is a comma (,), and you want to use a comma within the data ("my name is jane, and his is jim").

Default: "

record_terminator

A single-character value used to specify the end of a record.

Default:

- rfc4180 : \n
- traditional : \r\n

header

Boolean, specifies whether to use the first row of data as a header column. When header=true (default), and no header exists, fcsvparser uses a default column heading. The default header consists of ucol *n* , where *n* is the column offset number, starting with 0 for the first column. You can specify custom column heading names using the header_names parameter, described next.

If you specify header=false , the fcsvparser parses the first row of input as data, rather than as column headers.
Default: true

header_names

A list of column header names, delimited by the character defined by the parser's delimiter parameter. Use this parameter to specify header names in a CSV file without a header row, or to override the column names present in the CSV source. To override one or more existing column names, specify the header names to use. This parameter overrides any header row in the data.

trim

Boolean, specifies whether to trim white space from header names and key values.

Default: true

omit_empty_keys

Boolean, specifies how the parser handles header keys without values. If true, keys with an empty value in the header row are not loaded.

Default: false

reject_on_duplicate

Boolean, specifies whether to ignore duplicate records (false), or to reject duplicates (true). In either case, the load continues.

Default: false

reject_on_empty_key

Boolean, specifies whether to reject any row containing a key without a value.

Default: false

reject_on_materialized_type_error

Boolean, specifies whether to reject any materialized column value that the parser cannot coerce into a compatible data type. See Loading CSV data .

Default: false

Examples

This example shows how you can use fcsvparser to load a flex table, build a view, and then query that view.

1. Create a flex table for CSV data:

```
=> CREATE FLEX TABLE rfc();
CREATE TABLE
```

2. Use fcsvparser to load the data from STDIN. Specify that no header exists, and enter some data as shown:

```
=> COPY rfc FROM stdin PARSER fcsvparser(header='false');
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 10,10,20
>> 10,"10",30
>> 10,"20""5",90
>> \.
```

3. Run the compute_flextable_keys_and_build_view function, and query the rfc_view . Notice that the default enclosed_by character permits an escape character (") within a field ("20""5"). Thus, the resulting value was parsed correctly. Since no header existed in the input data, the function added ucol *n* for each column:

```
=> SELECT compute_flextable_keys_and_build_view('rfc');
                        compute_flextable_keys_and_build_view
-------------------------------------------------------------------------------------------
 Please see public.rfc_keys for updated keys
The view public.rfc_view is ready for querying
(1 row)

=> SELECT * FROM rfc_view;
 ucol0 | ucol1 | ucol2
-------+-------+-------
 10    | 10    | 20
 10    | 10    | 30
 10    | 20"5  | 90
(3 rows)
```

For more information and examples using other parameters of this parser, see Loading CSV data .

See also

- FAVROPARSER
- FDELIMITEDPAIRPARSER
- FDELIMITEDPARSER
- FDELIMITEDPAIRPARSER
- FJSONPARSER
- FREGEXPARSER

FDELIMITEDPAIRPARSER

Parses delimited data files. This parser provides a subset of the functionality in the parser fdelimitedparser . Use the fdelimitedpairparser when the data you are loading specifies pairs of column names with data in each row.

This parser is for use in Flex tables only. All flex parsers store the data as a single VMap in the LONG VARBINAR_raw__ column. If a data row is too large to fit in the column, it is rejected. Vertica supports null values for loading data with NULL -specified columns.

Syntax

```
FDELIMITEDPAIRPARSER ( [parameter-name='value'[,...]] )
```

Parameters

delimiter

Specifies a single-character delimiter.

Default: ' '

record_terminator

Specifies a single-character record terminator.

Default: newline

trim

Boolean specifies whether to trim white space from header names and key values.

Default: true

Examples

The following example illustrates creating a sample flex table for simple delimited data, with two real columns, eventId and priority .

1. Create a table:

```
=> create flex table CEFData(eventId int default(eventId::int), priority int default(priority::int) );
CREATE TABLE
```

2. Load a sample delimited OpenText ArcSight log file into the CEFData table, using the fcefparser :

```
=> copy CEFData from '/home/release/kmm/flextables/sampleArcSight.txt' parser fdelimitedpairparser();
Rows Loaded | 200
```

3. After loading the sample data file, use maptostring() to display the virtual columns in the __raw__ column of CEFData :

```
=> select maptostring(__raw__) from CEFData limit 1;
maptostring
-------------------------------------------------------------
  "agentassetid" : "4-WwHuD0BABCCQDVAeX21vg==",
  "agentzone" : "3083",
  "agt" : "265723237",
  "ahost" : "svsvm0176",
  "aid" : "3tGoHuD0BABCCMDVAeX21vg==",
  "art" : "1099267576901",
  "assetcriticality" : "0",
  "at" : "snort_db",
  "atz" : "America/Los_Angeles",
  "av" : "5.3.0.19524.0",
  "cat" : "attempted-recon",
  "categorybehavior" : "/Communicate/Query",
  "categorydevicegroup" : "/IDS/Network",
  "categoryobject" : "/Host",
  "categoryoutcome" : "/Attempt",
  "categorysignificance" : "/Recon",
  "categorytechnique" : "/Scan",
  "categorytupledescription" : "An IDS observed a scan of a host.",
  "cnt" : "1",
  "cs2" : "3",
  "destinationgeocountrycode" : "US",
  "destinationgeolocationinfo" : "Richardson",
  "destinationgeopostalcode" : "75082",
  "destinationgeoregioncode" : "TX",
  "destinationzone" : "3133",
  "device product" : "Snort",
  "device vendor" : "Snort",
  "device version" : "1.8",
  "deviceseverity" : "2",
  "dhost" : "198.198.121.200",
  "dlat" : "329913940429",
  "dlong" : "-966644973754",
  "dst" : "3334896072",
  "dtz" : "America/Los_Angeles",
  "dvchost" : "unknown:eth1",
  "end" : "1364676323451",
  "eventid" : "1219383333",
  "fdevice product" : "Snort",
  "fdevice vendor" : "Snort",
  "fdevice version" : "1.8",
  "fdtz" : "America/Los_Angeles",
  "fdvchost" : "unknown:eth1",
  "lblstring2label" : "sig_rev",
  "locality" : "0",
  "modelconfidence" : "0",
```

```
  "mrt" : "1364675789222",
  "name" : "ICMP PING NMAP",
  "oagentassetid" : "4-WwHuD0BABCCQDVAeX21vg==",
  "oagentzone" : "3083",
  "oagt" : "265723237",
  "oahost" : "svsvm0176",
  "oaid" : "3tGoHuD0BABCCMDVAeX21vg==",
  "oat" : "snort_db",
  "oatz" : "America/Los_Angeles",
  "oav" : "5.3.0.19524.0",
  "originator" : "0",
  "priority" : "8",
  "proto" : "ICMP",
  "relevance" : "10",
  "rt" : "1099267573000",
  "severity" : "8",
  "shost" : "198.198.104.10",
  "signature id" : "[1:469]",
  "slat" : "329913940429",
  "slong" : "-966644973754",
  "sourcegeocountrycode" : "US",
  "sourcegeolocationinfo" : "Richardson",
  "sourcegeopostalcode" : "75082",
  "sourcegeoregioncode" : "TX",
  "sourcezone" : "3133",
  "src" : "3334891530",
  "start" : "1364676323451",
  "type" : "0"
}

(1 row)
```

4. Select the eventID and priority real columns, along with two virtual columns, atz and destinationgeoregioncode :

```
=>  select eventID, priority, atz, destinationgeoregioncode from CEFData limit 10;
 eventID   | priority |         atz         | destinationgeoregioncode
-----------+----------+---------------------+--------------------------
1218325417 |        5 | America/Los_Angeles |
1219383333 |        8 | America/Los_Angeles | TX
1219533691 |        9 | America/Los_Angeles | TX
1220034458 |        5 | America/Los_Angeles | TX
1220034578 |        9 | America/Los_Angeles |
1220067119 |        5 | America/Los_Angeles | TX
1220106960 |        5 | America/Los_Angeles | TX
1220142122 |        5 | America/Los_Angeles | TX
1220312009 |        5 | America/Los_Angeles | TX
1220321355 |        5 | America/Los_Angeles | CA
(10 rows)
```

See also

- FAVROPARSER
- FCEFPARSER
- FCSVPARSER
- FDELIMITEDPARSER
- FJSONPARSER
- FREGEXPARSER

FDELIMITEDPARSER

Parses data using a delimiter character to separate values. The fdelimitedparser loads delimited data, storing it in a single-value VMap.

This parser is for use in Flex tables only. All flex parsers store the data as a single VMap in the LONG VARBINAR_raw__ column. If a data row is too large to fit in the column, it is rejected. Vertica supports null values for loading data with NULL -specified columns.

> **Note**
> By default, fdelimitedparser treats empty fields as NULL , rather than as an empty string (''). This behavior makes casting easier. Casting a NULL to an integer (NULL::int) is valid, while casting an empty string to an integer (''::int) is not. If required, use the treat_empty_val_as_null parameter to change the default behavior of fdelimitedparser .

Syntax

```
FDLIMITEDPARSER ( [parameter-name='value'[,...]] )
```

Parameters

delimiter

Single character delimiter.

Default: |

record_terminator

Single-character record terminator.

Default: \n

trim

Boolean, specifies whether to trim white space from header names and key values.

Default: true

header

Boolean, specifies that a header column exists. The parser uses col### for the column names if you use this parameter but no header exists.

Default: true

omit_empty_keys

Boolean, specifies how the parser handles header keys without values. If omit_empty_keys=true , keys with an empty value in the header row are not loaded.

Default: false

reject_on_duplicate

Boolean, specifies whether to ignore duplicate records (false), or to reject duplicates (true). In either case, the load continues.

Default: false

reject_on_empty_key

Boolean, specifies whether to reject any row containing a key without a value.

Default: false

reject_on_materialized_type_error

Boolean, specifies whether to reject any row value for a materialized column that the parser cannot coerce into a compatible data type. See Using flex table parsers .

Default: false

treat_empty_val_as_null

Boolean, specifies that empty fields become NULLs , rather than empty strings ('').

Default: true

Examples

1. Create a flex table for delimited data:

```
t=> CREATE FLEX TABLE delim_flex ();
CREATE TABLE
```

2. Use the fdelimitedparser to load some delimited data from STDIN , specifying a comma (,) column delimiter:

```
=> COPY delim_flex FROM STDIN parser fdelimitedparser (delimiter=',');
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> deviceproduct, severity, deviceversion
>> ArcSight, High, 2.4.1
>> \.
```

You can now query virtual columns in the delim_flex flex table:

```
=> SELECT deviceproduct, severity, deviceversion from delim_flex;
 deviceproduct | severity | deviceversion
---------------+----------+---------------
 ArcSight      | High     | 2.4.1
(1 row)
```

See also

- FAVROPARSER
- FCEFPARSER
- FCSVPARSER
- FDELIMITEDPAIRPARSER
- FJSONPARSER
- FREGEXPARSER

FJSONPARSER

Parses and loads a JSON file. This file can contain either repeated JSON data objects (including nested maps), or an outer list of JSON elements.

When loading into a flex or hybrid table, the parser stores the JSON data in a single-value VMap. When loading into a hybrid or columnar table, the parser loads data directly into any table column with a column name that matches a key in the JSON source data.

You can load complex types in the JSON source (arrays, structs, or combinations) with strong typing or as flexible complex types . A flexible complex type is loaded into a VMap column, as in flex tables. To load complex types as VMap columns, specify a column type of LONG VARBINARY . To preserve the indexing in complex types, set flatten_maps to false.

This parser can notify you if it finds keys in the data that are not part of the table definition. See Unmatched Keys .

FJSONPARSER supports cooperative parse only if record_terminator is specified. It does not support apportioned load .

Syntax

```
FJSONPARSER ( [parameter=value[,...]] )
```

Parameters

flatten_maps

Boolean, whether to flatten sub-maps within the JSON data, separating map levels with a period (.). This value affects all data in the load, including nested maps.

This parameter applies only to flex tables or VMap columns and is ignored when loading strongly-typed complex types.
Default: true

flatten_arrays

Boolean, whether to convert lists to sub-maps with integer keys. When lists are flattened, key names are concatenated as for maps. Lists are not flattened by default. This value affects all data in the load, including nested lists.

This parameter applies only to flex tables or VMap columns and is ignored when loading strongly-typed complex types.
Default: false

reject_on_duplicate

Boolean, whether to ignore duplicate records (false), or to reject duplicates (true). In either case, the load continues.

Default: false

reject_on_empty_key

Boolean, whether to reject any row containing a field key without a value.

Default: false

omit_empty_keys

Boolean, whether to omit any field key from the data that does not have a value. Other fields in the same record are loaded.

Default: false

record_terminator

When set, any invalid JSON records are skipped and parsing continues with the next record. Records must be terminated uniformly. For example, if your input file has JSON records terminated by newline characters, set this parameter to E'\n') . If any invalid JSON records exist, parsing continues after the next record_terminator .

Even if the data does not contain invalid records, specifying an explicit record terminator can improve load performance by allowing cooperative parse and apportioned load to operate more efficiently.
When you omit this parameter, parsing ends at the first invalid JSON record.

reject_on_materialized_type_error

Boolean, whether to reject a data row that contains a materialized column value that cannot be coerced into a compatible data type. If the value is false and the type cannot be coerced, the parser sets the value in that column to NULL.
If the column is an array and the data to be loaded is too large, then false sets the column value to NULL and true rejects the row.

If the column is a strongly-typed complex type , as opposed to a flexible complex type , then a type mismatch anywhere in the complex type causes the entire column to be treated as a mismatch. The parser does not partially load complex types.

Default: false

start_point

String, the name of a key in the JSON load data at which to begin parsing. The parser ignores all data before the start_point value. The value is loaded for each object in the file. The parser processes data after the first instance, and up to the second, ignoring any remaining data.

start_point_occurrence

Integer, the *n* th occurrence of the value you specify with start_point . Use in conjunction with start_point when the data has multiple start values and you know the occurrence at which to begin parsing.

Default: 1

suppress_nonalphanumeric_key_chars

Boolean, whether to suppress non-alphanumeric characters in JSON key values. The parser replaces these characters with an underscore (_) when this parameter is true.

Default: false

key_separator

Character for the parser to use when concatenating key names.

Default: period (.)

suppress_warnings

String, which warnings to suppress:

- unmatched_key (see Unmatched Keys)
- true or t (suppress all warnings)
- false or f (do not suppress warnings)

Default: false

Data types

If the total size of an array exceeds the size defined by the target table, the parser sets the value to null.

Unmatched keys

Data being loaded can contain keys that are not part of the table definition. If you are loading into a flex table (or a flexible complex type column), no data is lost. For a table with strongly-defined columns, however, new keys cannot be loaded because the table does not have a place to put them.

This parser emits warnings if it finds new keys and if both of the following are true:

- The target table is not a flex table.
- The new key is not nested within a flexible complex type column.

New keys are logged in the UDX_EVENTS system table. If a new key is a complex type with nested keys, only the top-level key is logged. When you see a warning about unmatched keys, you can query this table and then use ALTER TABLE to modify your table definition for future loads.

Querying an external table loads data and thus can trigger these warnings. To prevent them, set the suppress_warnings parameter to 'unmatched_keys' or 'true':

```
=> CREATE EXTERNAL TABLE restaurants(
        name VARCHAR(50),
        menu ARRAY[ROW(item VARCHAR(50), price NUMERIC(8,2)),100])
  AS COPY FROM '/data/rest.json'
  PARSER FJSONPARSER(suppress_warnings='unmatched_key');
```

Examples

The following example loads JSON data from STDIN using the default parameters:

```
=> CREATE TABLE people(age INT, name VARCHAR);
CREATE TABLE

=> COPY people FROM STDIN PARSER FJSONPARSER();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"age": 5, "name": "Tim"}
>>  {"age": 3}
>>  {"name": "Fred"}
>>  {"name": "Bob", "age": 10}
>> \.
=> SELECT * FROM people;
 age | name
-----+------
     | Fred
  10 | Bob
   5 | Tim
   3 |
(4 rows)
```

The following example uses the reject_on_duplicate parameter to reject duplicate values:

```
=> CREATE FLEX TABLE json_dupes();
CREATE TABLE
=> COPY json_dupes FROM stdin PARSER fjsonparser(reject_on_duplicate=true)
exceptions '/home/dbadmin/load_errors/json_e.out'
rejected data '/home/dbadmin/load_errors/json_r.out';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> {"a":"1","a":"2","b":"3"}
>> \.
=>  \!cat /home/dbadmin/load_errors/json_e.out
COPY: Input record 1 has been rejected (Rejected by user-defined parser).
Please see /home/dbadmin/load_errors/json_r.out, record 1 for the rejected record.
COPY: Loaded 0 rows, rejected 1 rows.
```

The following example loads array data:

```
$ cat addrs.json
 {"number": 301, "street": "Grant", "attributes": [1, 2, 3, 4]}

=> CREATE EXTERNAL TABLE customers(number INT, street VARCHAR, attributes ARRAY[INT])
    AS COPY FROM 'addrs.json' PARSER fjsonparser();

=> SELECT number, street, attributes FROM customers;
 num | street| attributes
-----+-----------+---------------
301  | Grant | [1,2,3,4]
(1 row)
```

The following example loads a flexible complex type, rejecting rows that have empty keys within the nested records. Notice that while the data has two restaurants, one has a key name that is an empty string. This one is rejected:

```
$ cat rest1.json
{
   "name" : "Bob's pizzeria",
   "cuisine" : "Italian",
   "location_city" : ["Cambridge", "Pittsburgh"],
   "menu" : [{"item" : "cheese pizza", "" : "$8.25"},
           {"item" : "spinach pizza", "price" : "$10.50"}]
}
{
   "name" : "Bakersfield Tacos",
   "cuisine" : "Mexican",
   "location_city" : ["Pittsburgh"],
   "menu" : [{"item" : "veggie taco", "price" : "$9.95"},
           {"item" : "steak taco", "price" : "$10.95"}]
}

=> CREATE TABLE rest (name VARCHAR, cuisine VARCHAR, location_city LONG VARBINARY, menu LONG VARBINARY);

=> COPY rest FROM '/data/rest1.json'
   PARSER fjsonparser(flatten_maps=false, reject_on_empty_key=true);
Rows Loaded
------------
       1
(1 row)

=> SELECT maptostring(location_city), maptostring(menu) FROM rest;
       maptostring        |                        maptostring
--------------------------+-----------------------------------------------------
 {
   "0": "Pittsburgh"
} | {
   "0": {
      "item": "veggie taco",
      "price": "$9.95"
   },
   "1": {
      "item": "steak taco",
      "price": "$10.95"
   }
}
(1 row)
```

To instead load partial data, use omit_empty_keys to bypass the missing keys while loading everything else:

```
=> COPY rest FROM '/data/rest1.json'
    PARSER fjsonparser(flatten_maps=false, omit_empty_keys=true);
 Rows Loaded
-------------
           2
(1 row)

=> SELECT maptostring(location_city), maptostring(menu) from rest;
                    maptostring                     |                maptostring
----------------------------------------------------+---------------------------------
 {
    "0": "Pittsburgh"
}                                                   | {
    "0": {
        "item": "veggie taco",
        "price": "$9.95"
    },
    "1": {
        "item": "steak taco",
        "price": "$10.95"
    }
}
 {
    "0": "Cambridge",
    "1": "Pittsburgh"
} | {
    "0": {
        "item": "cheese pizza"
    },
    "1": {
        "item": "spinach pizza",
        "price": "$10.50"
    }
}
(2 rows)
```

To instead load this data with strong typing, define the complex types in the table:

```
=> CREATE EXTERNAL TABLE restaurants
  (name VARCHAR, cuisine VARCHAR,
   location_city ARRAY[VARCHAR(80)],
   menu ARRAY[ ROW(item VARCHAR(80), price FLOAT) ]
  )
 AS COPY FROM '/data/rest.json' PARSER fjsonparser();

=> SELECT * FROM restaurants;
       name        | cuisine |       location_city        |                    \
                menu
--------------------+---------+----------------------------+--------------------\
-----------------------------------------------------------
 Bob's pizzeria     | Italian | ["Cambridge","Pittsburgh"] | [{"item":"cheese pi\
zza","price":0.0},{"item":"spinach pizza","price":0.0}]
 Bakersfield Tacos | Mexican | ["Pittsburgh"]             | [{"item":"veggie ta\
co","price":0.0},{"item":"steak taco","price":0.0}]
(2 rows)
```

In the following example, the data contains two new fields. One is a top-level field (a new column), and the other is a new field on an existing struct. The new fields are recorded in the UDX_EVENTS system table:

```
=> COPY rest FROM '/data/rest2.json' PARSER FJSONPARSER();
WARNING 10596:  Warning in UDx call in user-defined object [FJSONParser], code: 0, message:
Data source contained keys which did not match table schema
HINT:  SELECT key, sum(num_instances) FROM v_monitor.udx_events WHERE event_type = 'UNMATCHED_KEY' GROUP BY key
 Rows Loaded
-------------
           2
(1 row)

=> SELECT key, SUM(num_instances) FROM v_monitor.UDX_EVENTS
   WHERE event_type = 'UNMATCHED_KEY' GROUP BY key;
         key           | SUM
-----------------------+-----
 chain                 |   1
 menu.elements.calories |   7
(2 rows)
```

For other examples, see JSON data .

FREGEXPARSER

Parses a regular expression, matching columns to the contents of the named regular expression groups.

This parser is for use in Flex tables only. All flex parsers store the data as a single VMap in the LONG VARBINAR_raw__ column. If a data row is too large to fit in the column, it is rejected. Vertica supports null values for loading data with NULL -specified columns.

Syntax

```
FREGEXPARSER ( pattern=[parameter-name='value'[,...]] )
```

Parameters

pattern

Specifies the regular expression of data to match.

Default: Empty string ("")

use_jit

Boolean, specifies whether to use just-in-time compiling when parsing the regular expression.

Default: false

record_terminator

Specifies the character used to separate input records.

Default: \n

logline_column

A string that captures the destination column containing the full string that the regular expression matched.

Default: Empty string ("")

Example

These examples use the following regular expression, which searches for information that includes the timestamp , date , thread_name , and thread_id strings.

> **Caution**
> For display purposes, this sample regular expression adds new line characters to split long lines of text. To use this expression in a query, first copy and edit the example to remove any new line characters.

This example expression loads any thread_id hex value, regardless of whether it has a 0x prefix, (<thread_id>(?:0x)?[0-9a-f]+) .

```
'^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
(?<thread_name>[A-Za-z ]+):(?<thread_id>(?:0x)?[0-9a-f]+)
-?(?<transaction_id>[0-9a-f])?(?:[(?<component>\w+)]
\<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @[?(?<enode>\w+)]?: )
?(?<text>.*)'
```

1. Create a flex table (vlog) to contain the results of a Vertica log file. For this example, we made a copy of a log file in the directory /home/dbadmin/data/vertica.log :

```
=> CREATE FLEX TABLE vlog1();
CREATE TABLE
```

2. Use the fregexparser with the sample regular expression to load data from the log file. Be sure to remove any line characters before using this expression shown here:

```
=>  COPY vlog1 FROM '/home/dbadmin/tempdat/KMvertica.log'
PARSER FREGEXPARSER(pattern=
'^(?<time>\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d\.\d+)
 (?<thread_name>[A-Za-z ]+):(?<thread_id>(?:0x)?[0-9a-f]+)
-?(?<transaction_id>[0-9a-f])?(?:[(?<component>\w+)]
\<(?<level>\w+)\> )?(?:<(?<elevel>\w+)> @[?(?<enode>\w+)]?: )
?(?<text>.*)'


);

 Rows Loaded
-------------
       31049
(1 row)
```

3. After successfully loading data, use the MAPTOSTRING () function with the table's __raw__ column. The four rows (limt 4) that the query returns are regular expression results of the KMvertica.log file, parsed with fregexparser . The output shows thread_id values with a preceding 0x or without:

```
=> SELECT maptostring(__raw__) FROM vlog1 LIMIT 4;
                maptostring
-------------------------------------------------------------------------------
{
  "text" : " [Init] <INFO> Log /home/dbadmin/VMart/v_vmart_node0001_catalog/vertica.log
opened; #2",
  "thread_id" : "0x7f2157e287c0",
  "thread_name" : "Main",
  "time" : "2017-03-21 23:30:01.704"
}

{
  "text" : " [Init] <INFO> Processing command line: /opt/vertica/bin/vertica -D
/home/dbadmin/VMart/v_vmart_node0001_catalog -C VMart -n v_vmart_node0001 -h
10.20.100.247 -p 5433 -P 4803 -Y ipv4",
  "thread_id" : "0x7f2157e287c0",
  "thread_name" : "Main",
  "time" : "2017-03-21 23:30:01.704"
}

{
  "text" : " [Init] <INFO> Starting up Vertica Analytic Database v8.1.1-20170321",
  "thread_id" : "7f2157e287c0",
  "thread_name" : "Main",
  "time" : "2017-03-21 23:30:01.704"
}

{
  "text" : " [Init] <INFO> Compiler Version: 4.8.2 20140120 (Red Hat 4.8.2-15)",
  "thread_id" : "7f2157e287c0",
  "thread_name" : "Main",
  "time" : "2017-03-21 23:30:01.704"
}
(4 rows)
```

See also

- FDELIMITEDPAIRPARSER
- FDELIMITEDPARSER
- FJSONPARSER

ORC

Use the ORC clause with the COPY statement to load data in the ORC format. When loading data into Vertica, you can read all primitive types, UUIDs, and complex types .

When loading ORC data, you must account for all columns in the data; you cannot select only some columns.

If the table definition includes columns of primitive types and those columns are not in the data, the parser fills those columns with NULL. If the table definition includes columns of complex types, those columns must be present in the data.

This parser does not support apportioned load or cooperative parse .

Syntax

```
ORC ( [ parameter=value[,...] ] )
```

Parameters

All parameters are optional.

hive_partition_cols

Comma-separated list of columns that are partition columns in the data.

> **Deprecated**

> Instead, use COPY PARTITION COLUMNS. See Partitioned file paths . If you use both this parameter and PARTITION COLUMNS, COPY ignores the parameter.

allow_no_match

Whether to accept a path containing a glob with no matching files and report zero rows in query results. If this parameter is not set, Vertica returns an error if the path in the FROM clause does not match at least one file.

Data types

This parser can read all primitive types, UUIDs, and complex types .

If the total size of an array exceeds the size defined by the target table, the parser rejects the row.

Examples

The ORC clause does not use the PARSER option:

```
=> CREATE EXTERNAL TABLE orders (...)
  AS COPY FROM 's3://DataLake/orders.orc' ORC;
```

You can read a map column as an array of rows, as in the following example:

```
=> CREATE EXTERNAL TABLE orders
(orderkey INT,
 custkey INT,
 prods ARRAY[ROW(key VARCHAR(10), value DECIMAL(12,2))],
 orderdate DATE
) AS COPY FROM '...' ORC;
```

PARQUET

Use the PARQUET parser with the COPY statement to load data in the Parquet format. When loading data into Vertica you can read all primitive types, UUIDs, and complex types .

By default, the Parquet parser uses strong schema matching, meaning that columns in the data must exactly match the columns in the table using the data. You can optionally use Loose Schema Matching .

When loading Parquet data, Vertica caches the Parquet metadata to improve efficiency. This cache uses local TEMP storage and is not used if TEMP is remote. See the ParquetMetadataCacheSizeMB configuration parameter to change the size of the cache.

This parser does not support apportioned load or cooperative parse .

Syntax

```
PARQUET ( [ parameter=value[,...] ] )
```

Parameters

All parameters are optional.

hive_partition_cols

Comma-separated list of columns that are partition columns in the data.

> **Deprecated**
> Instead, use COPY PARTITION COLUMNS. See Partitioned file paths . If you use both this parameter and PARTITION COLUMNS, COPY ignores the parameter.

allow_no_match

Boolean. Whether to accept a path containing a glob with no matching files and report zero rows in query results. If this parameter is not set, Vertica returns an error if the path in the FROM clause does not match at least one file.

allow_long_varbinary_match_complex_type

Boolean. Whether to enable flexible column types (see Flexible complex types). If true, the Parquet parser allows a complex type in the data to match a table column defined as LONG VARBINARY. If false, the Parquet parser requires strong typing of complex types. With the parameter set you can still use strong typing. Set this parameter to false if you want use of flexible columns to be treated as an error.

do_soft_schema_match_by_name

Boolean. Whether to enable loose schema matching (true) instead of the strict matching based on column order in the table definition and parquet file (false, default). See Loose Schema Matching for more information.

reject_on_materialized_type_error

Boolean, applies only if do_soft_schema_match_by_name is true. Specifies what to do when loose schema matching is being used and a value cannot be coerced from the data to the target column type. A value of true (default) means to reject the row; a value of false means to use NULL for the value or, for strings that are too long, truncate. See the table of type coercions for coercible type mappings.

Loose schema matching

By default, the Parquet parser uses strong schema matching. This means that all columns in the Parquet data must be loaded, in the same order as in the data. However, there are times when you only want to pull certain columns, or you want to be able to accommodate future changes in the Parquet schema.

Use the do_soft_schema_match_by_name parameter to enable soft schema matching. This setting has the following effects:

- Columns in the data are matched to columns in the table by their names. Names must exactly match but are case-insensitive.
- Columns that exist in the Parquet data but are not part of the table definition are ignored.
- Columns that exist in the table definition but not the Parquet data are filled with NULL. The parser logs an UNMATCHED_TABLE_COLUMNS_PARQUETPARSER event in QUERY_EVENTS .
- If the same case-insensitive column name occurs more than once in the Parquet data, the parser uses the last one. (This situation can arise when using data written by tools that are case-sensitive.)
- Column types do not need to exactly match, so long as the data type in the Parquet file can be coerced to the type used by the table. If a type cannot be coerced, the parser logs a TYPE_MISMATCH_COLUMNS_PARQUETPARSER event in QUERY_EVENTS. If reject_on_materialized_type_error is true then the parser rejects the row. If it is false, the parser uses NULL or, for string values that are too long, truncates the value.
- Columns using complex types (other than one-dimensional arrays of primitive types) can be defined but not queried.

Data types

The Parquet parser maps Parquet data types to Vertica data types as follows.

Parquet Logical Type	Vertica Data Type
StringLogicalType	VARCHAR
MapLogicalType	ARRAY[ROW]
ListLogicalType	ARRAY/SET
IntLogicalType	INT/NUMERIC
DecimalLogicalType	NUMERIC
DateLogicalType	DATE
TimeLogicalType	TIME
TimestampLogicalType	TIMESTAMP
UUIDLogicalType	UUID

If the total size of an array exceeds the size defined by the target table, the parser rejects the row.

The following logical types are not supported:

- EnumLogicalType
- IntervalLogicalType
- JSONLogicalType
- BSONLogicalType
- UnknownLogicalType

The Parquet parser supports the following mappings of physical types:

Parquet Physical Type	Vertica Data Type
BOOLEAN	BOOLEAN
INT32/INT64	INT
INT96	Supported only for TIMESTAMP
FLOAT	DOUBLE
DOUBLE	DOUBLE
BYTE_ARRAY	VARBINARY
FIXED_LEN_BYTE_ARRAY	BINARY

The following mappings are supported with type coercion and loose schema matching.

Parquet Physical Type	Coercible to Vertica Data Type
BOOLEAN	BOOLEAN
INT32, INT64, BOOLEAN	INT
FLOAT, DOUBLE	DOUBLE
INT32, INT96	DATE
INT64, INT96	TIMESTAMP, TIMESTAMPTZ
INT64 If precision > 0: INT32, BYTE_ARRAY, FIXED_LEN_BYTE_ARRAY	Numeric
BYTE_ARRAY	CHAR, VARCHAR, LONG VARCHAR, BINARY, VARBINARY, LONG VARBINARY
FIXED_LEN_BYTE_ARRAY	UUID

Vertica supports only 3-level-encoded arrays, not 2-level-encoded.

Examples

The PARQUET clause does not use the PARSER option:

```
=> COPY sales FROM 's3://DataLake/sales.parquet' PARQUET;
```

In the following example, the data directory contains no files:

```
=> CREATE EXTERNAL TABLE customers (...)
   AS COPY FROM 'webhdfs:///data/*.parquet' PARQUET;
=> SELECT COUNT(*) FROM customers;
ERROR 7869: No files match when expanding glob: [webhdfs:///data/*.parquet]
```

To read zero rows instead of producing an error, use the allow_no_match parameter:

```
=> CREATE EXTERNAL TABLE customers (...)
   AS COPY FROM 'webhdfs:///data/*.parquet'
      PARQUET(allow_no_match='true');
=> SELECT COUNT(*) FROM customers;
 count
-------
     0
(1 row)
```

To allow reading a complex type (menu, in this example) as a flexible column type, use the allow_long_varbinary_match_complex_type parameter:

```
=> CREATE EXTERNAL TABLE restaurants
   (name VARCHAR, cuisine VARCHAR, location_city ARRAY[VARCHAR], menu LONG VARBINARY)
   AS COPY FROM '/data/rest*.parquet'
   PARQUET(allow_long_varbinary_match_complex_type='True');
```

To read only some columns from the restaurant data, use loose schema matching:

```
=> CREATE EXTERNAL TABLE restaurants(name VARCHAR, cuisine VARCHAR)
   AS COPY FROM '/data/rest*.parquet'
   PARQUET(allow_long_varbinary_match_complex_type='True',
         do_soft_schema_match_by_name='True');

=> SELECT * from restaurant;
      name        | cuisine
------------------+----------
 Bob's pizzeria    | Italian
 Bakersfield Tacos | Mexican
(2 rows)
```

Restrictions

COPY has the following restrictions:

Invalid data

COPY considers the following data invalid:

- Missing columns (an input line has fewer columns than the recipient table).
- Extra columns (an input line has more columns than the recipient table).
- Empty columns for an INTEGER or DATE/TIME data type. If a column is empty for either of these types, COPY does not use the default value that was defined by CREATE TABLE . However, if you do not supply a column option as part of the COPY statement, the default value is used.
- Incorrect representation of a data type. For example, trying to load a non-numeric value into an INTEGER column is invalid.

Constraint violations

If primary key, unique key, or check constraints are enabled for automatic enforcement in the target table, Vertica enforces those constraints when you load new data. If a violation occurs, Vertica rolls back the operation and returns an error.

Empty line handling

When COPY encounters an empty line while loading data, the line is neither inserted nor rejected, but COPY increments the line record number. Consider this behavior when evaluating rejected records. If you return a list of rejected records and COPY encountered an empty row while loading data, the position of rejected records is not incremented by one, as demonstrated in the following example.

The example first loads values into a table that defines the first column as INT. Note the errors on rows 3, 4, and 8:

```
=> \! cat -n /home/dbadmin/test.txt
    1 1|A|2
    2 2|B|4
    3 A|D|7
    4 A|E|7
    5
    6
    7 6|A|3
    8 B|A|3
```

The empty rows (5 and 6) shift the reporting of the error on row 8:

```
=> SELECT row_number, rejected_data, rejected_reason FROM test_bad;
 row_number | rejected_data | rejected_reason
------------+---------------+-------------------------------------------------
          3 | A|D|7 | Invalid integer format 'A' for column 1 (c1)
          4 | A|E|7 | Invalid integer format 'A' for column 1 (c1)
          6 | B|A|3 | Invalid integer format 'B' for column 1 (c1)
(3 rows)
```

Compressed file errors

When loading compressed files, COPY might abort and report an error, if the file seems to be corrupted. For example, this behavior can occur if reading the header block fails.

Disk quota

Tables and schemas can have disk quotas. If a load would violate either quota, the operation fails. For more information, see Disk quotas .

COPY FROM VERTICA

Imports data from another Vertica database. COPY FROM VERTICA is similar to COPY , but supports only a subset of its parameters.

Important

The source database must be no more than one major release behind the target database.

Syntax

```
COPY [[database.]schema-name.]target-table
   [( target-columns )]
   FROM VERTICA source-database.[schema.]source-table
   [( source-columns )]
   [STREAM NAME 'stream name']
   [NO COMMIT]
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

target-table

The target table for the imported data. Vertica loads the data into all projections that include columns from the schema table.

target-columns

A comma-delimited list of columns in *target-table* to store the copied data.See Mapping Between Target and Source Columns below.

You cannot use FILLER columns or columns of complex types , except native arrays, as part of the column definition.

source-database

The source database of the data to import. A connection to this database must already exist in the current session before starting the copy operation; otherwise Vertica returns an error. For details, see CONNECT TO VERTICA .

[*schema* .] *source-table*

The table that is the source of the imported data. If *schema* is any schema other than public , you must supply the schema name.

source-columns

A comma-delimited list of the columns in the source table to import. If omitted, all columns are exported.Columns cannot be of complex types. See Mapping Between Target and Source Columns below.

STREAM NAME

A COPY load stream identifier. Using a stream name helps to quickly identify a particular load. The STREAM NAME value that you specify in the load statement appears in the stream column of the LOAD_STREAMS system table.

NO COMMIT

Prevents COPY from committing its transaction automatically when it finishes copying data. For details, see Using transactions to stage a load .

Privileges

- Source table: SELECT

- Source table schema: USAGE
- Target table: INSERT
- Target table schema: USAGE

Mapping between target and source columns

If you copy all table data from one database to another, COPY FROM VERTICA can omit specifying column lists if column definitions in both tables comply with the following conditions:

- Same number of columns
- Identical column names
- Same sequence of columns
- Matching or compatible column data types
- No complex data types (ARRAY, SET, or ROW), except for native arrays

If any of these conditions is not true, the COPY FROM VERTICA statement must include column lists that explicitly map target and source columns to each other, as follows:

- Contain the same number of columns.
- List source and target columns in the same order.
- Pair columns with the same (or compatible) data types.

Node failure during COPY

See Handling node failure during copy/export .

Examples

The following example copies the contents of an entire table from the vmart database to an identically-defined table in the current database:

```
=> CONNECT TO VERTICA vmart USER dbadmin PASSWORD 'myPassword' ON 'VertTest01',5433;
CONNECT
=> COPY customer_dimension FROM  VERTICA vmart.customer_dimension;
 Rows Loaded
-------------
      500000
(1 row)
=> DISCONNECT vmart;
DISCONNECT
```

For more examples, see Copying data from another Vertica database .

See also

EXPORT TO VERTICA

COPY LOCAL

Using the COPY statement with its LOCAL option lets you load data files (up to 65,535 files) on a client system, rather than on a cluster host. COPY LOCAL supports the STDIN and 'pathToData' parameters, but not the [ON nodename] clause. COPY LOCAL does not support multiple file batches in NATIVE or NATIVE VARCHAR formats. COPY LOCAL does not support reading ORC or Parquet files; use ON NODE instead. COPY LOCAL does not support CURRENT_LOAD_SOURCE().

The COPY LOCAL option is platform-independent. The statement works in the same way across all supported Vertica platforms and drivers. For more details about supported drivers, see Client drivers .

COPY LOCAL must be the first statement in any multi-statement query you make with the ODBC client library. Using it as the second or later statement results in an error. When using other client libraries, such as JDBC, COPY LOCAL should always be the first statement in a multi-statement query. Also, do not use it multiple times in the same query.

> **Note**
> On Windows clients, the path you supply for the COPY LOCAL file is limited to 216 characters due to limitations in the Windows API.

COPY LOCAL does not automatically create exceptions and rejections files, even if exceptions occur.

Privileges

User must have INSERT privilege on the table and USAGE privilege on the schema.

How COPY LOCAL works

COPY LOCAL loads data in a platform-neutral way. The COPY LOCAL statement loads all files from a local client system to the Vertica host, where the server processes the files. You can copy files in various formats: uncompressed, compressed, fixed-width format, in bzip or gzip format, or specified as a bash glob. Files of a single format (such as all bzip, or gzip) can be comma-separated in the list of input files. You can also use any of the applicable COPY statement options (as long as the data format supports the option). For instance, you can define a specific delimiter character, or how to handle NULLs, and so forth.

> **Note**
> The Linux glob command returns files that match the pattern you enter, as specified in the Linux Manual Page for Glob (7) . For ADO.net platforms, specify patterns and wildcards as described in the .NET Directory.getFiles Method .

For more information about using the COPY LOCAL option to load data, see COPY for syntactical descriptions, and Specifying where to load data from for detailed examples.

The Vertica host uncompresses and processes the files as necessary, regardless of file format or the client platform from which you load the files. Once the server has the copied files, Vertica maintains performance by distributing file parsing tasks, such as encoding, compressing, uncompressing, across nodes.

Viewing copy local operations in a query plan

When you use the COPY LOCAL option, the GraphViz query plan includes a label for Load-Client-File , rather than Load-File . Following is a section from a sample query plan:

```
----------------------------------------------------
 PLAN:  BASE BULKLOAD PLAN  (GraphViz Format)
----------------------------------------------------
digraph G {
graph [rankdir=BT, label = " BASE BULKLOAD PLAN \nAll Nodes Vector:
\n\n  node[0]=initiator (initiator) Up\n", labelloc=t, labeljust=l ordering=out]
.
.
.
10[label = "Load-Client-File(/tmp/diff) \nOutBlk=[UncTuple]",
color = "green", shape = "ellipse"];
```

Examples

The following example shows a load from a local file.

```
$ cat > t.dat
12
17
9
^C

=> CREATE TABLE numbers (value INT);
CREATE TABLE

=> COPY numbers FROM LOCAL 't.dat';
 Rows Loaded
-------------
           3
(1 row)

=> SELECT * FROM numbers;
 value
-------
    12
    17
     9
(3 rows)
```

CREATE statements

CREATE statements let you create new database objects such as tables and users.

In this section

- CREATE ACCESS POLICY
- CREATE AUTHENTICATION
- CREATE CA BUNDLE
- CREATE CERTIFICATE
- CREATE DIRECTED QUERY
- CREATE EXTERNAL TABLE AS COPY
- CREATE EXTERNAL TABLE ICEBERG
- CREATE FAULT GROUP
- CREATE FLEXIBLE EXTERNAL TABLE AS COPY
- CREATE FLEXIBLE TABLE
- CREATE FUNCTION statements
- CREATE HCATALOG SCHEMA
- CREATE KEY
- CREATE LIBRARY
- CREATE LOAD BALANCE GROUP
- CREATE LOCAL TEMPORARY VIEW
- CREATE LOCATION
- CREATE NETWORK ADDRESS
- CREATE NETWORK INTERFACE
- CREATE NOTIFIER
- CREATE PROCEDURE (external)
- CREATE PROCEDURE (stored)
- CREATE PROFILE
- CREATE PROJECTION
- CREATE RESOURCE POOL
- CREATE ROLE
- CREATE ROUTING RULE
- CREATE SCHEDULE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE SUBNET
- CREATE TABLE
- CREATE TEMPORARY TABLE

- CREATE TEXT INDEX
- CREATE TLS CONFIGURATION
- CREATE TRIGGER
- CREATE USER
- CREATE VIEW

CREATE ACCESS POLICY

Creates an access policy that filters access to table data to users and roles. You can create access policies for table rows and columns. Vertica applies the access policy filters with each query and returns only the data that is permissible for the current user or role.

You cannot set access policies on columns of complex data types other than native arrays. If the table contains complex-type columns, you can still set row access policies and column access policies on other columns.

Syntax

```
CREATE ACCESS POLICY ON [[database.]schema.]table
  { FOR COLUMN column | FOR ROWS WHERE } expression [GRANT TRUSTED] { ENABLE | DISABLE }
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

The table with the target column or rows.

FOR COLUMN *column*

The column on which to apply this access policy. The column can be a native array, but other complex types are not supported. (See Complex types .)

FOR ROWS WHERE

The rows on which to apply this access policy.

expression

A SQL expression that specifies conditions for accessing row or column data:

- Row access policies limit access to specific rows in a table, as specified by the policy's WHERE expression. Only rows that satisfy this expression are fetched from the table. For details and sample usage, see Creating row access policies .
- Column access policies limit access to specific table columns. The access policy expression can also specify how to render column data to specific users and roles. For details and sample usage, see Creating column access policies .

GRANT TRUSTED

Specifies that GRANT statements take precedence over the access policy in determining whether users can perform DML operations on the target table. If omitted, users can only modify table data if the access policy allows them to see the stored data in its original, unaltered state. For more information, see Access policies and DML operations .

Important

GRANT TRUSTED only affects DML operations and does not enable users to see data that the access policy would otherwise mask. Specifying this option may allow users with certain grants to update data that they cannot see.

ENABLE | DISABLE

Whether to enable the access policy. You can enable and disable existing access policies with ALTER ACCESS POLICY .

Privileges

Non-superuser: Ownership of the table

Restrictions

The following limitations apply to access policies:

- A column can have only one access policy.
- Column access policies cannot be set on columns of complex types other than native arrays.
- Column access policies cannot be set for materialized columns on flex tables. While it is possible to set an access policy for the __raw__ column, doing so restricts access to the whole table.
- Row access policies are invalid on temporary tables and tables with aggregate projections.
- Access policy expressions cannot contain:

    - Subqueries
    - Aggregate functions
    - Analytic functions
    - User-defined transform functions (UDTF)
- If the query optimizer cannot replace a deterministic expression that involves only constants with their computed values, it blocks all DML operations such as INSERT .

See also

- Access policies
- ALTER ACCESS POLICY
- DROP ACCESS POLICY

CREATE AUTHENTICATION

Creates and enables an authentication record associated with users or roles. Authentication records are automatically enabled after creation.

Syntax

```
CREATE AUTHENTICATION auth-record-name
    METHOD 'auth-method'
    access-method
    [ FALLTHROUGH ]
```

Parameters

Name	Description
auth-record-name	Name of the authentication record, where *auth-record-name* conforms to conventions described in Identifiers .
auth-method	The authentication method, one of the following: • trust : Users can authenticate with a valid username (that is, without a password). • reject : Rejects the connection attempt. • hash : Users must provide a valid username and password. For details, see Hash authentication . • gss : Authorizes clients that connect to Vertica with an MIT Kerberos implementation. The Key Distribution Center (KDC) must support Kerberos 5 using the GSS-API. Non-MIT Kerberos implementations must use the GSS-API. For details, see Kerberos authentication . • ident : Authenticates the client against a username on an Ident server. For details, see Ident authentication . • ldap : Authenticates a client and their username and password with an LDAP or Active Directory server. For details, see LDAP authentication . • tls : Authenticates clients that provide a certificate with a Common Name (CN) that specifies a valid database username. Vertica must be configured for mutual mode TLS to use this method. For details, see TLS authentication • oauth : Authenticates a client with an access token. For details, see OAuth 2.0 authentication . For details, see Supported Client Authentication Methods .

access-method	The access method the client uses to connect, specified in one of the following ways:
	• LOCAL : Matches connection attempts made using local domain sockets.
	• HOST [TLS | NO TLS] ' *host-ip-address* ' : Matches connection attempts made using TCP/IP, where *host-ip-address* can be an IPv4 or IPv6 address. You can qualify HOST with one of the following options:
	  ◦ TLS (default): Match an SSL/TLS-wrapped TCP socket.
	  ◦ NO TLS : Match a plain (non-SSL/TLS) socket only.

[FALLTHROUGH]	Whether to enable fallthrough authentication for this record. To disable fallthrough, see ALTER AUTHENTICATION . Fallthrough cannot be enabled for authentication records that use the following authentication methods: • gss • oauth • reject • trust

Privileges

DBADMIN

Examples

See Creating authentication records .

See also

- ALTER AUTHENTICATION
- DROP AUTHENTICATION
- GRANT (authentication)
- REVOKE (authentication)

CREATE CA BUNDLE

> **Deprecated**
> CA bundles are only usable with certain deprecated parameters in Kafka notifiers . You should prefer using TLS configurations and the TLS CONFIGURATION parameter for notifiers instead.

Creates a certificate authority (CA) bundle. These contain root CA certificates.

Syntax

```
CREATE CA BUNDLE name [CERTIFICATES ca_cert[, ca_cert[, ...]]
```

Parameters

name
The name of the CA bundle.

ca_cert
The name of the CA certificate. If no certificates are specified, the bundle will be empty.

Privileges

Ownership of the CA certificates in the CA bundle.

Examples

See Managing CA bundles .

See also

- ALTER CA BUNDLE
- DROP CA BUNDLE

CREATE CERTIFICATE

Creates or imports a certificate, Certificate Authority (CA), or intermediate CA. These certificates can be used with ALTER TLS CONFIGURATION to set up client-server TLS , LDAPLink TLS , LDAPAuth TLS , and internode TLS .

CREATE CERTIFICATE generates x509v3 certificates.

Syntax

```
CREATE [TEMP[ORARY]] [CA] CERTIFICATE certificate_name
  {AS cert [KEY key_name]
  | SUBJECT subject
    [ SIGNED BY ca_cert ]
    [ VALID FOR days ]
    [ EXTENSIONS ext = val[,...] ]
    [ KEY private_key ]}
```

Parameters

TEMPORARY

Create with session scope. The key is stored in memory and is valid only for the current session.

CA

Designates the certificate as a CA or intermediate certificate. If omitted, the operation creates a normal certificate.

certificate_name

The name of the certificate.

AS *cert*

The imported certificate (string).

This parameter should include the entire chain of certificates, excluding the CA certificate.

KEY *key_name*

The name of the key.

This parameter only needs to be set for client/server certificates and CA certificates that you intend to sign other certificates with in Vertica. If your imported CA certificate will only be used for validating other certificates, you do not need to specify a key.

SUBJECT *subject*

The entity to issue the certificate to (string).

SIGNED BY *ca_cert*

The name of the CA that signed the certificate.

When adding a CA certificate, this parameter is optional. Specifying it will create an intermediate CA that cannot be used to sign other CA certificates.
When creating a certificate, this parameter is required.

VALID FOR *days*

The number of days that the certificate is valid.

EXTENSIONS *ext* = *val*

Strings specifying certificate extensions. For a full list of extensions, see the OpenSSL documentation .

KEY *private_key*

The name of the certificate's private key.

When importing a certificate, this parameter is required.

Privileges

Superuser

Default extensions

CREATE CERTIFICATE generates x509v3 certificates and includes several extensions by default. These differ based on the type of certificate you create:

CA Certificate:

- 'basicConstraints' = 'critical, CA:true'
- 'keyUsage' = 'critical, digitalSignature, keyCertSign'
- 'nsComment' = Vertica generated [CA] certificate'
- 'subjectKeyIdentifier' = 'hash'

Certificate:

- 'basicConstraints' = 'CA:false'
- 'keyUsage' = 'critical, digitalSignature, keyEncipherment'

Examples

See Generating TLS certificates and keys .

See also

- CREATE KEY

CREATE DIRECTED QUERY

Saves an association between an input query and a query that is annotated with optimizer hints.

CREATE DIRECTED QUERY has two variants:

- CREATE DIRECTED QUERY OPTIMIZER directs the query optimizer to generate annotated SQL from the specified input query. The annotated query contains hints that the optimizer can use to recreate its current query plan for that input query.
- CREATE DIRECTED QUERY CUSTOM specifies an annotated query supplied by the user. Vertica associates the annotated query with the input query specified by the last SAVE QUERY statement.

In both cases, Vertica associates the annotated query and input query, and registers their association in the system table DIRECTED_QUERIES under query_name .

Syntax

Optimizer-generated

```
CREATE DIRECTED QUERY OPT[IMIZER] query-name [COMMENT 'comments'] input-query
```

User-defined (custom)

```
CREATE DIRECTED QUERY CUSTOM query-name [COMMENT 'comments'] annotated-query
```

Parameters

OPT[IMIZER]

Directs the query optimizer to generate an annotated query from *input-query* , and associate both in the new directed query.

CUSTOM

Specifies to associate *annotated-query* with the query previously specified by SAVE QUERY .

query* - *name

A unique identifier for the directed query, a string that conforms to conventions described in Identifiers .

COMMENT ' *comments* '

Comment about the directed query, up to 128 characters. Comments can be useful for future reference—for example, to explain why a given directed query was created.

If you omit this argument, Vertica inserts one of the following comments:

- Optimizer-generated directed query
- Custom directed query

input-query

The input query to associate with an optimizer-generated directed query. The input query supports only one optimizer hint, :v (alias IGNORECONST).

annotated-query

A query with embedded optimizer hints to associate with the input query most recently saved with SAVE QUERY .

Privileges

Superuser

See also

Creating directed queries

CREATE EXTERNAL TABLE AS COPY

CREATE EXTERNAL TABLE AS COPY creates a table definition for data external to your Vertica database. This statement is a combination of the CREATE TABLE and COPY statements, supporting a subset of each statement's parameters.

Canceling a CREATE EXTERNAL TABLE AS COPY statement can cause unpredictable results. If you need to make a change, allow the statement to complete, drop the table, and then retry.

You can use ALTER TABLE to change the data types of columns instead of dropping and recreating the table.

You can use CREATE EXTERNAL TABLE AS COPY with any types except types from the Place package.

Vertica also supports external tables backed by Iceberg data. See CREATE EXTERNAL TABLE ICEBERG .

> **Note**
> Vertica does not create superprojections for external tables, since external tables are not stored in the database.

Syntax

```
CREATE EXTERNAL TABLE [ IF NOT EXISTS ] [[database.]schema.]table-name
   ( column-definition[,...] )
[{INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES]
AS COPY
   [ ( { column-as-expression | column }
      [ DELIMITER [ AS ] 'char' ]
      [ ENCLOSED [ BY ] 'char' ]
      [ ENFORCELENGTH ]
      [ ESCAPE [ AS ] 'char' | NO ESCAPE ]
      [ FILLER datatype ]
      [ FORMAT 'format' ]
      [ NULL [ AS ] 'string' ]
      [ TRIM 'byte' ]
   [,...] ) ]
   [ COLUMN OPTION ( column
      [ DELIMITER [ AS ] 'char' ]
      [ ENCLOSED [ BY ] 'char' ]
      [ ENFORCELENGTH ]
      [ ESCAPE [ AS ] 'char' | NO ESCAPE ]
      [ FORMAT 'format' ]
      [ NULL [ AS ] 'string' ]
      [ TRIM 'byte' ]
   [,...] ) ]


FROM {
  { 'path-to-data'
      [ ON { nodename | (nodeset) | ANY NODE | EACH NODE } ] [ compression ] }[,...]
    [ PARTITION COLUMNS column[,...] ]
  | LOCAL 'path-to-data' [ compression ] [,...]
  | VERTICA source-database.[source-schema.]source-table[( source-column[,...] ) ]
 }
     [ NATIVE
       | FIXEDWIDTH COLSIZES {( integer )[,...]}
       | NATIVE VARCHAR
       | ORC
       | PARQUET
     ]
  [ ABORT ON ERROR ]
  [ DELIMITER [ AS ] 'char' ]
  [ ENCLOSED [ BY ] 'char' ]
  [ ENFORCELENGTH ]
  [ ERROR TOLERANCE ]
  [ ESCAPE AS 'char' | NO ESCAPE ]
  [ EXCEPTIONS 'path' [ ON nodename ] [,...] ]
  [ [ WITH ] FILTER filter( [ arg=value[,...] ] ) ]
  [ NULL [ AS ] 'string' ]
  [ [ WITH ] PARSER parser([arg=value [,...] ]) ]
  [ RECORD TERMINATOR 'string' ]
  [ REJECTED DATA 'path' [ ON nodename ] [,...] ]
  [ REJECTMAX integer ]
  [ SKIP integer ]
  [ SKIP BYTES integer ]
  [ TRAILING NULLCOLS ]
  [ TRIM 'byte' ]
```

Parameters

For all supported parameters, see the CREATE TABLE and COPY statements. For information on using this statement with UDLs, see User-defined load (UDL).

For additional guidance on using COPY parameters, see Specifying where to load data from .

Privileges

Superuser, or non-superuser with the following privileges:

- READ privileges on the USER-accessible storage location. See GRANT (storage location)
- Full access (including SELECT) to an external table that the user has privileges to create

Partitioned data

Data can be partitioned using its directory structure and Vertica can take advantage of that partitioning to improve query performance for external tables. For details, see Partitioned file paths .

If you see unexpected results when reading data, verify that globs in your file paths correctly align with the partition structure. See Troubleshooting external tables .

ORC and Parquet data

When using the ORC and Parquet formats, Vertica supports some additional options in the COPY statement and data structures for columns. See ORC and PARQUET .

Examples

The following example defines an external table for delimited data stored in HDFS:

```
=> CREATE EXTERNAL TABLE sales (itemID INT, date DATE, price FLOAT)
   AS COPY FROM 'hdfs:///data/ext1.csv' DELIMITER ',';
```

The following example uses data in the ORC format that is stored in S3. The data has two partition columns. For more information about partitions, see Partitioned file paths .

```
=> CREATE EXTERNAL TABLE records (id int, name varchar(50), created date, region varchar(50))
  AS COPY FROM 's3://datalake/sales/*/*/*'
  PARTITION COLUMNS created, region;
```

The following example shows how you can read from all Parquet files in a local directory, with no partitions and no globs:

```
=> CREATE EXTERNAL TABLE sales (itemID INT, date DATE, price FLOAT)
   AS COPY FROM '/data/sales/*.parquet' PARQUET;
```

The following example creates an external table for data containing arrays:

```
=> CREATE EXTERNAL TABLE cust (cust_custkey int,
      cust_custname varchar(50),
      cust_custstaddress ARRAY[varchar(100)],
      cust_custaddressln2 ARRAY[varchar(100)],
      cust_custcity ARRAY[varchar(50)],
      cust_custstate ARRAY[char(2)],
      cust_custzip ARRAY[int],
      cust_email varchar(50), cust_phone varchar(30))
  AS COPY FROM 'webhdfs://data/*.parquet' PARQUET;
```

To allow users without superuser access to use external tables with data on the local file system, S3, or GCS, create a location for 'user' usage and grant access to it. This example shows granting access to a user named Bob to any external table whose data is located under /tmp (including in subdirectories to any depth):

```
=> CREATE LOCATION '/tmp' ALL NODES USAGE 'user';
=> GRANT ALL ON LOCATION '/tmp' to Bob;
```

The following example shows CREATE EXTERNAL TABLE using a user-defined source:

```
=> CREATE SOURCE curl AS LANGUAGE 'C++' NAME 'CurlSourceFactory' LIBRARY curllib;
=> CREATE EXTERNAL TABLE curl_table1 as COPY SOURCE CurlSourceFactory;
```

See also

Creating external tables
CREATE EXTERNAL TABLE ICEBERG

Creates an external table for data stored by Apache Iceberg . An Iceberg table consists of data files and metadata describing the schema. Unlike other external tables, an Iceberg external table need not specify column definitions (DDL). The information is read from Iceberg metadata at query time. For certain data types you can adjust column definitions, for example to specify VARCHAR sizes.

A single Iceberg table can have more than one metadata file, each describing a different version of the table. You can create an external table using either the base location of the table or a specific metadata file.

If a metadata file specifies columns that are not present in the data, Vertica treats the missing values as NULL. The same is not true for missing fields of complex types (structs). If a metadata file specifies struct fields that are not present in the data, Vertica treats this as an error.

All Iceberg files, both data and metadata, must be accessible to all database nodes.

Iceberg can store data in several file formats. Vertica can read Iceberg data in the Parquet format only.

Syntax

```
CREATE EXTERNAL TABLE [[database.]schema.]table
  STORED BY ICEBERG LOCATION { path | metadata-file }
  [COLUMN TYPES (column-name type[,...])] ;
```

Arguments

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Name of the table to create, which must be unique among names of all sequences, tables, projections, views, and models within the schema.

STORED BY ICEBERG LOCATION { *path* | *metadata-file* }

Location of Iceberg data, one of:

- Base location of an Iceberg File System table . Vertica uses the latest metadata file.
- Path to a metadata file with a name ending in .metadata.json . Vertica uses this metadata file even if it is not the latest.

On S3, an Iceberg table is not a File System table but a metastore. This means you cannot specify a base location on S3. You must specify the full path to a metadata file.
If the paths embedded in the Iceberg data are not accessible to Vertica, use the IcebergPathMapping configuration parameter to provide mappings. See Path Prefixes .

COLUMN TYPES (*column-name type* [,...])

Column names and types for VARCHAR, VARBINARY, or ARRAY columns or ROW fields only. You can specify types only to set lengths or array bounds, not type coercion. See Data Types . If you do not specify a type, the table uses the Vertica defaults.

You cannot specify any other column properties, such as defaults or constraints.
Columns that are specified but not found in the Iceberg schema are ignored.

Privileges

Superuser, or non-superuser with the following privileges:

- READ privileges on the USER-accessible storage location. See GRANT (storage location)
- Full access (including SELECT) to an external table that the user has privileges to create

Path prefixes

Iceberg tables store file paths in the metadata as absolute URIs (host and port). Sometimes this URI differs from the URI that Vertica can use to access the data. This can particularly be an issue for files stored on HDFS, where the metadata can use a different URI scheme and port number than what Vertica expects.

To change the URIs, set the IcebergPathMapping configuration parameter. The value is a list of one or more pairs of Iceberg URI prefixes and corresponding Vertica prefixes:

```
=> ALTER SESSION SET IcebergPathMapping=
  '{"hdfs://node-196.example.com:9000":"webhdfs://node-196.example.com:9870"}';
```

Include only the URI prefix (up through the port), not complete paths. If IcebergPathMapping contains more than one mapping that could apply, Vertica uses the longest entry that matches.

You can set IcebergPathMapping at the database, session, or user level.

Data types

The following table shows the mappings of Iceberg data types to Vertica data types. For types that allow it, you can use the COLUMN TYPES clause to override these defaults.

Iceberg Type	Vertica Type	Allows Override?
boolean	BOOLEAN	No
int (32-bit)	INT	No
long (64-bit)	INT	No
float (32-bit)	FLOAT	No
double (64-bit)	FLOAT	No
decimal(precision, scale)	NUMERIC with same precision and scale	No
date	DATE	No
time	TIME	No
timestamp	TIMESTAMP	No
timestamptz	TIMESTAMP WITH TIMEZONE	No
string	VARCHAR(80)	VARCHAR or LONG VARCHAR with custom length
uuid	UUID	No
fixed(length)	BINARY(length) if length <= 65000 LONG VARBINARY(length) otherwise	No
binary (variable length)	VARBINARY(80)	VARBINARY or LONG VARBINARY with custom length
struct	ROW	No, but you can override individual fields if their types permit
list	ARRAY (default bound)	ARRAY with custom bound
map	Not supported	No

Restrictions

The following restrictions apply to external tables backed by Iceberg:

- Data files must be in Parquet format and have Iceberg field IDs.
- All fields in complex types (structs) that are specified in the metadata file must be present in the data.
- Iceberg column defaults are not supported. The values must be present in the data.
- Iceberg delete files are not supported.
- Malformed data is an error that aborts the load. You cannot reject bad data and continue.
- VARCHAR values are not truncated. If a string is too long, it is treated as an error.

The following restrictions apply to queries of Iceberg tables:

- You cannot use a column in an Iceberg table as a DEFAULT or SET USING option in another table. The following example is an error:

```
=> CREATE TABLE t(
   id INT DEFAULT (SELECT COUNT(*) FROM iceberg_table));
ERROR 0:  Default and set using expressions cannot refer to external iceberg tables
```

- Errors in Iceberg data or metadata, such as missing files or type mismatches, can manifest as query errors such as the following:

```
=> SELECT * FROM iceberg_table;
ERROR 0:  Problem reading metadata for table iceberg_table. Detail: Could not determine type of column a. User specified type: int. Iceberg type: boolean
```

Examples

The following example creates a table based on the Parquet data files with no overrides:

```
=> CREATE EXTERNAL TABLE sales
  STORED BY ICEBERG LOCATION 's3:/sales/*';
```

In the following example, the data uses a struct for the shipping address, with fields for street address (string), city (string), and zip code (integer). The following table definition overrides the default VARCHAR lengths. Note that the zip code is not included in COLUMN TYPES overrides. The ROW column contains only the fields being changed, but all fields including the zip code are part of the table definition and are included in query results:

```
=> CREATE EXTERNAL TABLE sales
  STORED BY ICEBERG LOCATION 's3:/sales/*'
  COLUMN TYPES (address ROW(street VARCHAR(50), city VARCHAR(50)));
```

CREATE FAULT GROUP

Enterprise Mode only

Creates a fault group, which can contain the following:

- One or more nodes
- One or more child fault groups
- One or more nodes and one or more child fault groups

CREATE FAULT GROUP creates an empty fault group. Use ALTER FAULT GROUP to add nodes or other fault groups to an existing fault group.

Syntax

```
CREATE FAULT GROUP name
```

Parameters

name

The name of the fault group to create, unique among all fault groups, where *name* conforms to conventions described in Identifiers .

Privileges

Superuser

Examples

The following command creates a fault group called parent0 :

```
=> CREATE FAULT GROUP parent0;
CREATE FAULT GROUP
```

See also

- FAULT_GROUPS
- CLUSTER_LAYOUT
- Fault groups
- High availability with fault groups

CREATE FLEXIBLE EXTERNAL TABLE AS COPY

CREATE FLEXIBLE EXTERNAL TABLE AS COPY creates a flexible external table. This statement combines statements CREATE FLEXIBLE TABLE and COPY statements, supporting a subset of each statement's parameters.

You can also use user-defined load functions (UDLs) to create external flex tables. For details about creating and using flex tables, see Using Flex Tables.

> **Note**
> Vertica does not create a superprojection for an external table when you create it.

For details about creating and using flex tables, see Creating flex tables in Using Flex Tables.

Caution
Canceling a CREATE FLEX EXTERNAL TABLE AS COPY statement can cause unpredictable results. Vertica recommends that you allow the statement to finish, then use DROP TABLE after the table exists.

Syntax

```
CREATE FLEX[IBLE] EXTERNAL TABLE [ IF NOT EXISTS ] [[database.]schema.]table-name
  ( [ column-definition[,...] ] )
  [ INCLUDE | EXCLUDE [SCHEMA] PRIVILEGES ]
AS COPY [ ( { column-as-expression | column } [ FILLER datatype ] ]
  FROM {
    'path-to-data' [ ON nodename | ON ANY NODE | ON (nodeset) ] input-format [,...]
    | [ WITH ] UDL-clause[...]
  }
  [ ABORT ON ERROR ]
  [ DELIMITER [ AS ] 'char' ]
  [ ENCLOSED [ BY ] 'char' ]
  [ ENFORCELENGTH ]
  [ ESCAPE [ AS ] 'char' | NO ESCAPE ]
  [ EXCEPTIONS 'path' [ ON nodename ] [,...] ]
  [ NULL [ AS ] 'string' ]
  [ RECORD TERMINATOR 'string' ]
  [ REJECTED DATA 'path' [ ON nodename ][,...] ]
  [ REJECTMAX integer ]
  [ SKIP integer ]
  [ SKIP BYTES integer ]
  [ TRAILING NULLCOLS ]
  [ TRIM 'byte' ]
```

Parameters

For parameter descriptions, see CREATE TABLE and Parameters.

Note
CREATE FLEXIBLE EXTERNAL TABLE AS COPY supports only a subset of CREATE TABLE and COPY parameters.

Privileges

Superuser, or non-superuser with the following privileges:

- READ privileges on the USER-accessible storage location. See GRANT (storage location)
- Full access (including SELECT) to an external table that the user has privileges to create

Examples

To create an external flex table:

```
=> CREATE flex external table mountains() AS COPY FROM 'home/release/KData/kmm_ountains.json' PARSER fjsonparser();
CREATE TABLE
```

As with other flex tables, creating an external flex table produces two regular tables: the named table and its associated _keys table. The keys table is not an external table:

```
=> \dt mountains
                List of tables
 Schema |   Name    | Kind  |  Owner  | Comment
--------+-----------+-------+---------+---------
 public | mountains | table | release |
(1 row)
```

You can use the helper function, COMPUTE_FLEXTABLE_KEYS_AND_BUILD_VIEW , to compute keys and create a view for the external table:

```
=> SELECT compute_flextable_keys_and_build_view ('appLog');

                compute_flextable_keys_and_build_view
-------------------------------------------------------------------------------------------
Please see public.appLog_keys for updated keys
The view public.appLog_view is ready for querying
(1 row)
```

Check the keys from the _keys table for the results of running the helper application:

```
=> SELECT * FROM appLog_keys;
                    key_name                    | frequency |   data_type_guess
------------------------------------------------+-----------+-------------------
contributors                                    |         8 | varchar(20)
coordinates                                     |         8 | varchar(20)
created_at                                      |         8 | varchar(60)
entities.hashtags                               |         8 | long varbinary(186)
.
.
.
retweeted_status.user.time_zone                 |         1 | varchar(20)
retweeted_status.user.url                       |         1 | varchar(68)
retweeted_status.user.utc_offset                |         1 | varchar(20)
retweeted_status.user.verified                  |         1 | varchar(20)
(125 rows)
```

You can query the view:

```
=> SELECT "user.lang" FROM appLog_view;
 user.lang
-----------
it
en
es
en
en
es
tr
en
(12 rows)
```

See also

- CREATE EXTERNAL TABLE AS COPY

CREATE FLEXIBLE TABLE

Creates a flexible (flex) table in the logical schema.

When you create a flex table, Vertica automatically creates two dependent objects:

- Keys table that is named *flex-table-name* _keys
- View that is named *flex-table-name* _view

The flex table requires the keys table and view. Neither of these objects can exist independently of the flex table.

Syntax

Create with column definitions :

```
CREATE [[ scope ] TEMP[ORARY]] FLEX[IBLE] TABLE [ IF NOT EXISTS ]
  [[database.]schema.]table-name
  ( [ column-definition[,...] [, table-constraint ][,...] ] )
  [ ORDER BY column[,...] ]
  [ segmentation-spec ]
  [ KSAFE [k-num] ]
  [ partition-clause]
  [ {INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES ]
  [ DISK_QUOTA quota ]
```

Create from another table:

```
CREATE FLEX[IBLE] TABLE [[database.]schema.] table-name
  [ ( column-name-list ) ]
  [ {INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES ]
  AS  query [ ENCODED BY column-ref-list ]
  [ DISK_QUOTA quota ]
```

Parameters

For general parameter descriptions, see CREATE TABLE ; for parameters specific to temporary flex tables, see CREATE TEMPORARY TABLE and Creating flex tables .

You cannot partition a flex table on any virtual column (key).

Privileges

Non-superuser: CREATE privilege on table schema

Default columns

The CREATE statement can omit specifying any column definitions. CREATE FLEXIBLE TABLE always creates two columns automatically:

__raw__
: LONG VARBINARY type column to store unstructured data that you load. By default, this column has a NOT NULL constraint.

__identity__
: IDENTITY column that is used for segmentation and sorting when no other column is defined.

Default projections

Vertica automatically creates superprojections for both the flex table and keys tables when you create them.

If you create a flex table with one or more of the ORDER BY, ENCODED BY, SEGMENTED BY, or KSAFE clauses, the clause information is used to create projections. If no clauses are in use, Vertica uses the following defaults:

Table	Sort order	Encoding	Segmentation	K-safety
Flexible table	ORDER BY *.__identity__	none	SEGMENTED BY hash *.__identity__ ALL NODES OFFSET 0	1
Keys table	ORDER BY *._keys_frequency	none	UNSEGMENTED ALL NODES	1

> **Note**
> When you build a view for a flex table (see BUILD_FLEXTABLE_VIEW), the view is ordered by frequency , desc , and key_name .

Examples

The following example creates a flex table named darkdata without specifying any column information. Vertica creates a default superprojection and buddy projection as part of creating the table:

```
=> CREATE FLEXIBLE TABLE darkdata();
CREATE TABLE
=> \dj darkdata1*
                    List of projections
 Schema |        Name         |  Owner  |       Node       | Comment
--------+---------------------+---------+------------------+---------
 public | darkdata1_b0        | dbadmin |                  |
 public | darkdata1_b1        | dbadmin |                  |
 public | darkdata1_keys_super | dbadmin | v_vmart_node0001 |
 public | darkdata1_keys_super | dbadmin | v_vmart_node0002 |
 public | darkdata1_keys_super | dbadmin | v_vmart_node0003 |
(5 rows)

=> SELECT export_objects('','darkdata1_b0');
CREATE PROJECTION public.darkdata1_b0 /*+basename(darkdata1),createtype(P)*/
(
 __identity__,
 __raw__
)
AS
 SELECT darkdata1.__identity__,
        darkdata1.__raw__
 FROM public.darkdata1
 ORDER BY darkdata1.__identity__
SEGMENTED BY hash(darkdata1.__identity__) ALL NODES OFFSET 0;

SELECT MARK_DESIGN_KSAFE(1);
(1 row)

=> select export_objects('','darkdata1_keys_super');
CREATE PROJECTION public.darkdata1_keys_super /*+basename(darkdata1_keys),createtype(P)*/
(
 key_name,
 frequency,
 data_type_guess
)
AS
 SELECT darkdata1_keys.key_name,
        darkdata1_keys.frequency,
        darkdata1_keys.data_type_guess
 FROM public.darkdata1_keys
 ORDER BY darkdata1_keys.frequency
UNSEGMENTED ALL NODES;

SELECT MARK_DESIGN_KSAFE(1);
(1 row)
```

The following example creates a table called darkdata1 with one column definition (date_col). The statement specifies the partition by clause to partition the data by year. Vertica creates a default superprojection and buddy projections as part of creating the table:

```
=> CREATE FLEX TABLE darkdata1 (date_col date NOT NULL) partition by
  extract('year' from date_col);
CREATE TABLE
```

See also

- Creating flex tables
- CREATE FLEXIBLE EXTERNAL TABLE AS COPY

CREATE FUNCTION statements

Vertica provides CREATE statements for each type of user-defined extension . Each CREATE statement adds a user-defined function to the Vertica catalog:

CREATE statement	Extension
CREATE FUNCTION (scalar)	User-defined scalar functions (UDSFs)
CREATE AGGREGATE FUNCTION	User-defined aggregate functions (UDAFs)
CREATE ANALYTIC FUNCTION	User-defined analytic functions (UDAnF)
CREATE TRANSFORM FUNCTION	User-defined transform functions (UDTFs)
CREATE statements for user-defined load :	
• CREATE SOURCE	Load source functions
• CREATE FILTER	Load filter functions
• CREATE PARSER	Load parser functions

Vertica also provides CREATE FUNCTION (SQL) , which stores SQL expressions as functions that you can invoke in a query.

In this section

- CREATE AGGREGATE FUNCTION
- CREATE ANALYTIC FUNCTION
- CREATE FILTER
- CREATE FUNCTION (scalar)
- CREATE FUNCTION (SQL)
- CREATE PARSER
- CREATE SOURCE
- CREATE TRANSFORM FUNCTION

CREATE AGGREGATE FUNCTION

Adds a user-defined aggregate function (UDAF) to the catalog. The library containing the function must have been previously added using CREATE LIBRARY .

CREATE AGGREGATE FUNCTION automatically determines the function parameters and return value from data supplied by the factory class. Vertica supports overloading aggregate functions. When you call the SQL function, Vertica passes the input table to the function to process.

User-defined aggregate functions run in unfenced mode only.

Syntax

```
CREATE [ OR REPLACE ] AGGREGATE FUNCTION [ IF NOT EXISTS ]
 [[database.]schema.]function AS
 [ LANGUAGE 'language' ]
 NAME 'factory'
 LIBRARY library
 [ NOT FENCED ];
```

Arguments

OR REPLACE

If a function with the same name and arguments exists, replace it. You can use this to change between fenced and unfenced modes, for example. If you do not use this directive and the function already exists, the CREATE statement returns with a rollback error.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

IF NOT EXISTS

If a function with the same name and arguments exists, return without creating the function.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

Name of the function to create. This is the name used in SQL invocations of the function. It does not need to match the name of the factory, but it is less confusing if they are the same or similar.
The function name must conform to the restrictions on Identifiers .

LANGUAGE ' *language* '

The language used to develop this function, currently C++ only (the default).

NAME ' *factory* '

Name of the factory class that generates the function instance.

LIBRARY *library*

Name of the shared library that contains the function. This library must have already been loaded by CREATE LIBRARY .

NOT FENCED

Indicates that the function runs in unfenced mode. Aggregate functions cannot be run in fenced mode.

Privileges

Non-superuser:

- CREATE privilege on the function's schema
- USAGE privilege on the function's library

Examples

The following example demonstrates loading a library named AggregateFunctions and then defining functions named ag_avg and ag_cat . The functions are mapped to the AverageFactory and ConcatenateFactory classes in the library:

```
=> CREATE LIBRARY AggregateFunctions AS '/opt/vertica/sdk/examples/build/AggregateFunctions.so';
CREATE LIBRARY
=> CREATE AGGREGATE FUNCTION ag_avg AS LANGUAGE 'C++' NAME 'AverageFactory'
   library AggregateFunctions;
CREATE AGGREGATE FUNCTION
=> CREATE AGGREGATE FUNCTION ag_cat AS LANGUAGE 'C++' NAME 'ConcatenateFactory'
   library AggregateFunctions;
CREATE AGGREGATE FUNCTION
=> \x
Expanded display is on.
select * from user_functions;
-[ RECORD 1 ]----------+-----------------------------------------------------------------
schema_name            | public
function_name          | ag_avg
procedure_type         | User Defined Aggregate
function_return_type   | Numeric
function_argument_type | Numeric
function_definition    | Class 'AverageFactory' in Library 'public.AggregateFunctions'
volatility             |
is_strict              | f
is_fenced              | f
comment                |
-[ RECORD 2 ]----------+-----------------------------------------------------------------
schema_name            | public
function_name          | ag_cat
procedure_type         | User Defined Aggregate
function_return_type   | Varchar
function_argument_type | Varchar
function_definition    | Class 'ConcatenateFactory' in Library 'public.AggregateFunctions'
volatility             |
is_strict              | f
is_fenced              | f
comment                |
```

See also

- DROP AGGREGATE FUNCTION
- USER_FUNCTIONS
- Developing user-defined extensions (UDxs)
- Aggregate functions (UDAFs)

CREATE ANALYTIC FUNCTION

Adds a user-defined analytic function (UDAnF) to the catalog. The library containing the function must have been previously added using CREATE LIBRARY .

CREATE ANALYTIC FUNCTION automatically determines the function parameters and return value from data supplied by the factory class. Vertica supports overloading analytic functions. When you call the SQL function, Vertica passes the input table to the function in the library to process.

Syntax

```
CREATE [ OR REPLACE ] ANALYTIC FUNCTION [ IF NOT EXISTS ]
   [[database.]schema.]function AS
   [ LANGUAGE 'language' ]
   NAME 'factory'
   LIBRARY library
   [ FENCED | NOT FENCED ]
```

Arguments

OR REPLACE

If a function with the same name and arguments exists, replace it. You can use this to change between fenced and unfenced modes, for example. If you do not use this directive and the function already exists, the CREATE statement returns with a rollback error.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

IF NOT EXISTS

If a function with the same name and arguments exists, return without creating the function.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

Name of the function to create. This is the name used in SQL invocations of the function. It does not need to match the name of the factory, but it is less confusing if they are the same or similar.
The function name must conform to the restrictions on Identifiers .

LANGUAGE ' *language* '

Language used to develop this function, one of the following:

- C++ (default)
- Java

NAME ' *factory* '

Name of the factory class that generates the function instance.

LIBRARY *library*

Name of the library that contains the function. This library must already be loaded by CREATE LIBRARY .

FENCED | NOT FENCED

Enables or disables fenced mode for this function.

Default: FENCED

Privileges

Non-superuser:

- CREATE privilege on the function's schema
- USAGE privilege on the function's library

Examples

This example creates an analytic function named an_rank based on the factory class named RankFactory in the AnalyticFunctions library:

```
=> CREATE ANALYTIC FUNCTION an_rank AS LANGUAGE 'C++'
  NAME 'RankFactory' LIBRARY AnalyticFunctions;
```

See also

Analytic functions (UDAnFs)
CREATE FILTER

Adds a user-defined load filter function to the catalog. The library containing the filter function must have been previously added using CREATE LIBRARY .

CREATE FILTER automatically determines the function parameters and return value from data supplied by the factory class. Vertica supports overloading load filter functions. When you call the SQL function, Vertica passes the input table to the function in the library to process.

Important
Installing an untrusted UDL function can compromise the security of the server. UDxs can contain arbitrary code. In particular, user-defined source functions can read data from any arbitrary location. It is up to the developer of the function to enforce proper security limitations. Superusers must not grant access to UDxs to untrusted users.

Syntax

```
CREATE [ OR REPLACE ] FILTER [ IF NOT EXISTS ]
  [[database.]schema.]function AS
  [ LANGUAGE 'language' ]
  NAME 'factory' LIBRARY library
  [ FENCED | NOT FENCED ]
```

Arguments

OR REPLACE

If a function with the same name and arguments exists, replace it. You can use this to change between fenced and unfenced modes, for example. If you do not use this directive and the function already exists, the CREATE statement returns with a rollback error.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

IF NOT EXISTS

If a function with the same name and arguments exists, return without creating the function.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

Name of the function to create. This is the name used in SQL invocations of the function. It does not need to match the name of the factory, but it is less confusing if they are the same or similar.
The function name must conform to the restrictions on Identifiers .

LANGUAGE ' *language* '

The language used to develop this function, one of the following:

- C++ (default)
- Java
- Python

NAME ' *factory* '

Name of the factory class that generates the function instance. This is the same name used by the RegisterFactory class.

LIBRARY *library*

Name of the C++ library shared object file, Python file, or Java Jar file. This library must already have been loaded by CREATE LIBRARY .

FENCED | NOT FENCED

Enables or disables fenced mode for this function.

Default: FENCED

Privileges

Superuser

Examples

The following example demonstrates loading a library named iConverterLib , then defining a filter function named Iconverter that is mapped to the iConverterFactory factory class in the library:

```
=> CREATE LIBRARY iConverterLib as '/opt/vertica/sdk/examples/build/IconverterLib.so';
CREATE LIBRARY
=> CREATE FILTER Iconverter AS LANGUAGE 'C++' NAME 'IconverterFactory' LIBRARY IconverterLib;
CREATE FILTER FUNCTION
=> \x
Expanded display is on.
=> SELECT * FROM user_functions;
-[ RECORD 1 ]----------+--------------------
schema_name            | public
function_name          | Iconverter
procedure_type         | User Defined Filter
function_return_type   |
function_argument_type |
function_definition    |
volatility             |
is_strict              | f
is_fenced              | f
comment                |
```

See also

- DROP FILTER
- USER_FUNCTIONS
- User-defined load (UDL)

CREATE FUNCTION (scalar)

Adds a user-defined scalar function (UDSF) to the catalog. The library containing the function must have been previously added using CREATE LIBRARY .

A UDSF takes in a single row of data and returns a single value. These functions can be used anywhere a native Vertica function or statement can be used, except CREATE TABLE with its PARTITION BY or any segmentation clause.

CREATE FUNCTION automatically determines the function parameters and return value from data supplied by the factory class. Vertica supports overloading UDxs. When you call the function, Vertica passes the parameters to the function in the library to process.

Syntax

```
CREATE [ OR REPLACE ] FUNCTION [ IF NOT EXISTS ]
  [[database.]schema.]function AS
  [ LANGUAGE 'language' ]
  NAME 'factory'
  LIBRARY library
  [ FENCED | NOT FENCED ]
```

Arguments

OR REPLACE

If a function with the same name and arguments exists, replace it. You can use this to change between fenced and unfenced modes, for example. If you do not use this directive and the function already exists, the CREATE statement returns with a rollback error.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

IF NOT EXISTS

If a function with the same name and arguments exists, return without creating the function.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

Name of the function to create. This is the name used in SQL invocations of the function. It does not need to match the name of the factory, but it is less confusing if they are the same or similar.
The function name must conform to the restrictions on Identifiers .

LANGUAGE ' *language* '

Language used to develop this function, one of the following:

- C++ (default)
- Python
- Java
- R

NAME ' *factory* '

Name of the factory class that generates the function instance.

LIBRARY *library*

Name of the C++ shared object file, Python file, Java Jar file, or R functions file. This library must already have been loaded by CREATE LIBRARY .

FENCED | NOT FENCED

Enables or disables fenced mode for this function. Functions written in Java and R always run in fenced mode.

Default: FENCED

Privileges

- CREATE privilege on the function's schema
- USAGE privilege on the function's library

Examples

The following example loads a library named ScalarFunctions and then defines a function named Add2ints that is mapped to the Add2intsInfo factory class in the library:

```
=> CREATE LIBRARY ScalarFunctions AS '/opt/vertica/sdk/examples/build/ScalarFunctions.so';
CREATE LIBRARY
=> CREATE FUNCTION Add2Ints AS LANGUAGE 'C++' NAME 'Add2IntsFactory' LIBRARY ScalarFunctions;
CREATE FUNCTION
=> \x
Expanded display is on.
=> SELECT * FROM USER_FUNCTIONS;

-[ RECORD 1 ]----------+-----------------------------------------------------
schema_name            | public
function_name          | Add2Ints
procedure_type         | User Defined Function
function_return_type   | Integer
function_argument_type | Integer, Integer
function_definition    | Class 'Add2IntsFactory' in Library 'public.ScalarFunctions'
volatility             | volatile
is_strict              | f
is_fenced              | t
comment                |

=> \x
Expanded display is off.
=> -- Try a simple call to the function
=> SELECT Add2Ints(23,19);
 Add2Ints
----------
       42
(1 row)
```

The following example uses a scalar function that returns a ROW:

```
=> CREATE FUNCTION div_with_rem AS LANGUAGE 'C++' NAME 'DivFactory' LIBRARY ScalarFunctions;

=> SELECT div_with_rem(18,5);
       div_with_rem
-------------------------------
 {"quotient":3,"remainder":3}
(1 row)
```

See also

Developing user-defined extensions (UDxs)

CREATE FUNCTION (SQL)

Stores SQL expressions as functions for use in queries. User-defined SQL functions are useful for executing complex queries and combining Vertica built-in functions. You simply call the function in a given query. If multiple SQL functions with the same name and argument types are in the search path, Vertica calls the first match that it finds.

SQL functions are flattened in all cases, including DDL.

Syntax

```
CREATE [ OR REPLACE ] FUNCTION [ IF NOT EXISTS ]
   [[database.]schema.]function( [ arg_list ] )
   RETURN return_type
   AS
   BEGIN
     RETURN expression;
   END;
```

Arguments

OR REPLACE

If a function of the same name and arguments exists, replace it. If you only change the function arguments, Vertica ignores this option and maintains both functions under the same name.

OR REPLACE and IF NOT EXISTS are mutually exclusive.

IF NOT EXISTS

If a function with the same name and arguments exists, return without creating the function.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

SQL function to create, where *function* conforms to conventions described in Identifiers .

arg_list

A comma-delimited list of argument names and their data types, specified in this format:

```
argname argtype[,...]
```

where:

- *argname* is the name of an argument passed to *function* .
- *argtype* is *argname* 's data type .

return_type

The data type that this function returns.

RETURN *expression*

The SQL function body, where *expression* can contain built-in functions, operators, and argument names specified in the CREATE FUNCTION statement.

A semicolon at the end of the expression is required.

> **Note**
> CREATE FUNCTION allows only one RETURN expression. Return expressions do not support the following:
>
> - FROM, WHERE, GROUP BY, ORDER BY, and LIMIT clauses
> - Aggregation, analytics, and meta-functions

Privileges

Non-superuser:

- CREATE privilege on the function's schema
- USAGE privilege on the function's library

Strictness and volatility

Vertica infers the strictness and volatility (stable , immutable , or volatile) of a SQL function from its definition. Vertica then determines the correctness of usage, such as where an immutable function is expected but a volatile function is provided.

SQL functions and views

You can create views on the queries that use SQL functions and then query the views. When you create a view, a SQL function replaces a call to the user-defined function with the function body in a view definition. Therefore, when the body of the user-defined function is replaced, the view should also be replaced.

Examples

See Creating user-defined SQL functions .

See also

- ALTER FUNCTION (scalar)
- DROP FUNCTION
- User-defined SQL functions

CREATE PARSER

Adds a user-defined load parser function to the catalog. The library containing the function must have been previously added using CREATE LIBRARY.

CREATE PARSER automatically determines the function parameters and return value from data supplied by the factory class. Vertica supports overloading load parser functions. When you call the SQL function, Vertica passes the input table to the function in the library to process.

Important
Installing an untrusted UDL function can compromise the security of the server. UDxs can contain arbitrary code. In particular, user-defined source functions can read data from any arbitrary location. It is up to the developer of the function to enforce proper security limitations. Superusers must not grant access to UDxs to untrusted users.

Syntax

```
CREATE [ OR REPLACE ] PARSER [ IF NOT EXISTS ]
  [[database.]schema.]function AS
  [ LANGUAGE 'language' ]
  NAME 'factory'
  LIBRARY library
  [ FENCED | NOT FENCED ]
```

Arguments

OR REPLACE

If a function with the same name and arguments exists, replace it. You can use this to change between fenced and unfenced modes, for example. If you do not use this directive and the function already exists, the CREATE statement returns with a rollback error.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

IF NOT EXISTS

If a function with the same name and arguments exists, return without creating the function.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

[*database* .] *schema*

Database and schema. The default schema is public. If you specify a database, it must be the current database.

function

Name of the function to create. This is the name used in SQL invocations of the function. It does not need to match the name of the factory, but it is less confusing if they are the same or similar.
The function name must conform to the restrictions on Identifiers.

LANGUAGE ' *language* '

The language used to develop this function, one of the following:

- C++ (default)
- Java
- Python

NAME ' *factory* '

Name of the factory class that generates the function instance. This is the same name used by the RegisterFactory class.

LIBRARY *library*

Name of the C++ library shared object file, Python file, or Java Jar file. This library must already have been loaded by CREATE LIBRARY.

FENCED | NOT FENCED

Enables or disables fenced mode for this function.

Default: FENCED

Privileges

Superuser

Examples

The following example demonstrates loading a library named BasicIntegrerParserLib, then defining a parser function named BasicIntegerParser that is mapped to the BasicIntegerParserFactory factory class in the library:

```
=> CREATE LIBRARY BasicIntegerParserLib as '/opt/vertica/sdk/examples/build/BasicIntegerParser.so';
CREATE LIBRARY
=> CREATE PARSER BasicIntegerParser AS LANGUAGE 'C++' NAME 'BasicIntegerParserFactory' LIBRARY BasicIntegerParserLib;
CREATE PARSER FUNCTION
=> \x
Expanded display is on.
=> SELECT * FROM user_functions;
-[ RECORD 1 ]----------+--------------------
schema_name            | public
function_name          | BasicIntegerParser
procedure_type         | User Defined Parser
function_return_type   |
function_argument_type |
function_definition    |
volatility             |
is_strict              | f
is_fenced              | f
comment                |
```

See also

- DROP PARSER
- USER_FUNCTIONS
- User-defined load (UDL)

CREATE SOURCE

Adds a user-defined load source function to the catalog. The library containing the function must have been previously added using CREATE LIBRARY .

CREATE SOURCE automatically determines the function parameters and return value from data supplied by the factory class. Vertica supports overloading load source functions. When you call the SQL function, Vertica passes the input table to the function in the library to process.

Important
Installing an untrusted UDL function can compromise the security of the server. UDxs can contain arbitrary code. In particular, user-defined source functions can read data from any arbitrary location. It is up to the developer of the function to enforce proper security limitations. Superusers must not grant access to UDxs to untrusted users.

Syntax

```
CREATE [ OR REPLACE ] SOURCE [ IF NOT EXISTS ]
   [[database.]schema.]function AS
   [ LANGUAGE 'language' ]
   NAME 'factory'
   LIBRARY library
   [ FENCED | NOT FENCED ]
```

Arguments

OR REPLACE

If a function with the same name and arguments exists, replace it. You can use this to change between fenced and unfenced modes, for example. If you do not use this directive and the function already exists, the CREATE statement returns with a rollback error.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

IF NOT EXISTS

If a function with the same name and arguments exists, return without creating the function.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

Name of the function to create. This is the name used in SQL invocations of the function. It does not need to match the name of the factory, but it is less confusing if they are the same or similar.
The function name must conform to the restrictions on Identifiers .

LANGUAGE ' *language* '

Language used to develop this function, one of the following:

- C++ (default)
- Java

NAME ' *factory* '

Name of the factory class that generates the function instance. This is the same name used by the RegisterFactory class.

LIBRARY *library*

Name of the C++ library shared object file or Java Jar file. This library must already have been loaded by CREATE LIBRARY .

FENCED | NOT FENCED

Enables or disables fenced mode for this function.

**Default: ** FENCED

Privileges

Superuser

Examples

The following example demonstrates loading a library named curllib , then defining a source function named curl that is mapped to the CurlSourceFactory factory class in the library:

```
=> CREATE LIBRARY curllib as '/opt/vertica/sdk/examples/build/cURLLib.so';
CREATE LIBRARY
=> CREATE SOURCE curl AS LANGUAGE 'C++' NAME 'CurlSourceFactory' LIBRARY curllib;
CREATE SOURCE
=> \x
Expanded display is on.
=> SELECT * FROM user_functions;
-[ RECORD 1 ]----------+--------------------
schema_name            | public
function_name          | curl
procedure_type         | User Defined Source
function_return_type   |
function_argument_type |
function_definition    |
volatility             |
is_strict              | f
is_fenced              | f
comment                |
```

See also

- DROP SOURCE
- USER_FUNCTIONS
- User-defined load (UDL)

CREATE TRANSFORM FUNCTION

Adds a user-defined transform function (UDTF) to the catalog. The library containing the function must have been previously added using CREATE LIBRARY .

CREATE TRANSFORM FUNCTION automatically determines the function parameters and return value from data supplied by the factory class. Vertica supports overloading transform functions. When you call the SQL function, Vertica passes the input table to the transform function in the library to process.

Syntax

```
CREATE [ OR REPLACE ] TRANSFORM FUNCTION [ IF NOT EXISTS ]
  [[database.]schema.]function AS
  [ LANGUAGE 'language' ]
  NAME 'factory'
  LIBRARY library
  [ FENCED | NOT FENCED ]
```

Arguments

OR REPLACE

If a function with the same name and arguments exists, replace it. You can use this to change between fenced and unfenced modes, for example. If you do not use this directive and the function already exists, the CREATE statement returns with a rollback error.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

IF NOT EXISTS

If a function with the same name and arguments exists, return without creating the function.
OR REPLACE and IF NOT EXISTS are mutually exclusive.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

Name of the function to create. This is the name used in SQL invocations of the function. It does not need to match the name of the factory, but it is less confusing if they are the same or similar.
The function name must conform to the restrictions on Identifiers .

LANGUAGE ' *language* '

The language used to develop this function, one of the following:

- C++ (default)
- Java
- R
- Python

NAME ' *factory* '

Name of the factory class that generates the function instance.

LIBRARY *library*

Name of the C++ shared object file, Python file, Java Jar file, or R functions file. This library must already have been loaded by CREATE LIBRARY .

FENCED | NOT FENCED

Enables or disables fenced mode for this function. Functions written in Java and R always run in fenced mode.

Default: FENCED

Privileges

Non-superuser:

- CREATE privilege on the function's schema
- USAGE privilege on the function's library

Restrictions

A query that includes a UDTF cannot:

- Include statements other than the SELECT statement that calls the UDTF and a PARTITION BY expression unless the UDTF is marked as a one-to-many UDTF
- Call an analytic function
- Call another UDTF
- Include one of the following clauses:
  - TIMESERIES
  - Pattern matching
  - Gap filling and interpolation

Examples

The following example loads a library named TransformFunctions and then defines a function named tokenize that is mapped to the TokenFactory factory class in the library:

```
=> CREATE LIBRARY TransformFunctions AS
  '/home/dbadmin/TransformFunctions.so';
CREATE LIBRARY
=> CREATE TRANSFORM FUNCTION tokenize
  AS LANGUAGE 'C++' NAME 'TokenFactory' LIBRARY TransformFunctions;
CREATE TRANSFORM FUNCTION
```

See also

- DROP FUNCTION
- Developing user-defined extensions (UDxs)

CREATE HCATALOG SCHEMA

Define a schema for data stored in a Hive data warehouse using the HCatalog Connector. For more information, see Using the HCatalog Connector .

Most of the optional parameters are read out of Hadoop configuration files if available. If you copied the Hadoop configuration files as described in Configuring Vertica for HCatalog , you can omit most parameters. By default this statement uses the values specified in those configuration files. If the configuration files are complete, the following is a valid statement:

```
=> CREATE HCATALOG SCHEMA hcat;
```

If a value is not specified in the configuration files and a default is shown in the parameter list, then that default value is used.

Some parameters apply only if you are using HiveServer2 (the default). Others apply only if you are using WebHCat, a legacy Hadoop service. When using HiveServer2, use HIVESERVER2_HOSTNAME to specify the server host. When using WebHCat, use WEBSERVICE_HOSTNAME to specify the server host.

If you need to use WebHCat you must also set the HCatalogConnectorUseHiveServer2 configuration parameter to 0. See Hadoop parameters .

After creating the schema, you can change many (but not all) parameters using ALTER HCATALOG SCHEMA .

Syntax

```
CREATE HCATALOG SCHEMA [IF NOT EXISTS] schemaName
  [AUTHORIZATION user-id]
  [WITH [param=value [,...] ] ]
```

Arguments

Argument	Description
[IF NOT EXISTS]	If given, the statement exits without an error when the schema named in *schemaName* already exists.
schemaName	The name of the schema to create in the Vertica catalog. The tables in the Hive database will be available through this schema.
AUTHORIZATION *user-id*	The name of a Vertica account to own the schema being created. This parameter is ignored if Kerberos authentication is being used; in that case the current vsql user is used.

Parameters

Parameter	Description

HOSTNAME	The hostname, IP address, or URI of the database server that stores the Hive data warehouse's metastore information. If you specify this parameter and do not also specify PORT , then this value must be in the URI format used for hive.metastore.uris in hive-site.xml. If the Hive metastore supports High Availability, you can specify a comma-separated list of URIs for this value. If this value is not specified, hive-site.xml must be available.
PORT	The port number on which the metastore database is running. If you specify this parameter, you must also specify HOSTNAME and it must be a name or IP address (not a URI).
HIVESERVER2_HOSTNAME	The hostname or IP address of the HiveServer2 service. This parameter is optional if in hive-site.xml you set one of the following properties: • hive.server2.thrift.bind.host to a valid host • hive.server2.support.dynamic.service.discovery to true This parameter is ignored if you are using WebHCat.
WEBSERVICE_HOSTNAME	The hostname or IP address of the WebHCat service, if using WebHCat instead of HiveServer2. If this value is not specified, webhcat-site.xml must be available.
WEBSERVICE_PORT	The port number on which the WebHCat service is running, if using WebHCat instead of HiveServer2. If this value is not specified, webhcat-site.xml must be available.
WEBHDFS_ADDRESS	The host and port ("host:port") for the WebHDFS service. This parameter is used only for reading ORC and Parquet files. If this value is not set, hdfs-site.xml must be available to read these file types through the HCatalog Connector.
HCATALOG_SCHEMA	The name of the Hive schema or database that the Vertica schema is being mapped to. The default is *schemaName* .
CUSTOM_PARTITIONS	Whether the Hive schema uses custom partition locations ('YES' or 'NO'). If the schema uses custom partition locations, then Vertica queries Hive to get those locations when executing queries. These additional Hive queries can be expensive, so use this parameter only if you need to. The default is 'NO' (disabled). For more information, see Using Partitioned Data .
HCATALOG_USER	The username of the HCatalog user to use when making calls to the HiveServer2 or WebHCat server. The default is the current database user.
HCATALOG_CONNECTION_TIMEOUT	The number of seconds the HCatalog Connector waits for a successful connection to the HiveServer or WebHCat server. A value of 0 means wait indefinitely.
HCATALOG_SLOW_TRANSFER_LIMIT	The lowest data transfer rate (in bytes per second) from the HiveServer2 or WebHCat server that the HCatalog Connector accepts. See HCATALOG_SLOW_TRANSFER_TIME for details.
HCATALOG_SLOW_TRANSFER_TIME	The number of seconds the HCatalog Connector waits before enforcing the data transfer rate lower limit. After this time has passed, the HCatalog Connector tests whether the data transfer rate is at least as fast as the value set in HCATALOG_SLOW_TRANSFER_LIMIT. If it is not, then the HCatalog Connector breaks the connection and terminates the query.
SSL_CONFIG	The path of the Hadoop ssl-client.xml configuration file. This parameter is required if you are using HiveServer2 and it uses SSL wire encryption. This parameter is ignored if you are using WebHCat.

The default values for HCATALOG_CONNECTOR_TIMEOUT, HCATALOG_SLOW_TRANSFER_LIMIT, and HCATALOG_SLOW_TRANSFER_TIME are set by the database configuration parameters HCatConnectionTimeout, HCatSlowTransferLimit, and HCatSlowTransferTime. See Hadoop parameters for more information.

Configuration files

The HCatalog Connector uses the following values from the Hadoop configuration files if you do not override them when creating the schema.

File	Properties
hive-site.xml	hive.server2.thrift.bind.host (used for HIVESERVER2_HOSTNAME)
	hive.server2.thrift.port
	hive.server2.transport.mode
	hive.server2.authentication
	hive.server2.authentication.kerberos.principal
	hive.server2.support.dynamic.service.discovery
	hive.zookeeper.quorum (used as HIVESERVER2_HOSTNAME if dynamic service discovery is enabled)
	hive.zookeeper.client.port
	hive.server2.zookeeper.namespace
	hive.metastore.uris (used for HOSTNAME and PORT)
ssl-client.xml	ssl.client.truststore.location
	ssl.client.truststore.password

Privileges

The user must be a superuser or be granted all permissions on the database to use this statement.

The user also requires access to Hive data in one of the following ways:

- Have USAGE permissions on *hcatalog_schema* , if Hive does not use an authorization service (Sentry or Ranger) to manage access.
- Have permission through an authorization service, if Hive uses it to manage access. In this case you must either set EnableHCatImpersonation to 0, to access data as the Vertica principal, or grant users access to the HDFS data. For Sentry, you can use ACL synchronization to manage HDFS access.
- Be the dbadmin user, with or without an authorization service.

Examples

The following example shows how to use CREATE HCATALOG SCHEMA to define a new schema for tables stored in a Hive database and then query the system tables that contain information about those tables:

```
=> CREATE HCATALOG SCHEMA hcat WITH HOSTNAME='hcathost' PORT=9083
   HCATALOG_SCHEMA='default' HIVESERVER2_HOSTNAME='hs.example.com'
   SSL_CONFIG='/etc/hadoop/conf/ssl-client.xml' HCATALOG_USER='admin';
CREATE SCHEMA
=> \x
Expanded display is on.

=> SELECT * FROM v_catalog.hcatalog_schemata;
-[ RECORD 1 ]----------------+-------------------------------------------
schema_id                    | 45035996273748224
schema_name                  | hcat
schema_owner_id              | 45035996273704962
schema_owner                 | admin
create_time                  | 2017-12-05 14:43:03.353404-05
hostname                     | hcathost
port                         | -1
hiveserver2_hostname         | hs.example.com
webservice_hostname          |
webservice_port              | 50111
webhdfs_address              | hs.example.com:50070
hcatalog_schema_name         | default
ssl_config                   | /etc/hadoop/conf/ssl-client.xml
hcatalog_user_name           | admin
hcatalog_connection_timeout  | -1
hcatalog_slow_transfer_limit | -1
hcatalog_slow_transfer_time  | -1
custom_partitions            | f

=> SELECT * FROM v_catalog.hcatalog_table_list;
-[ RECORD 1 ]------+------------------
table_schema_id    | 45035996273748224
table_schema       | hcat
hcatalog_schema    | default
table_name         | nation
hcatalog_user_name | admin
-[ RECORD 2 ]------+------------------
table_schema_id    | 45035996273748224
table_schema       | hcat
hcatalog_schema    | default
table_name         | raw
hcatalog_user_name | admin
-[ RECORD 3 ]------+------------------
table_schema_id    | 45035996273748224
table_schema       | hcat
hcatalog_schema    | default
table_name         | raw_rcfile
hcatalog_user_name | admin
-[ RECORD 4 ]------+------------------
table_schema_id    | 45035996273748224
table_schema       | hcat
hcatalog_schema    | default
table_name         | raw_sequence
hcatalog_user_name | admin
```

The following example shows how to specify more than one metastore host.

```
=> CREATE HCATALOG SCHEMA hcat
   WITH HOSTNAME='thrift://node1.example.com:9083,thrift://node2.example.com:9083';
```

The following example shows how to include custom partition locations:

```
=> CREATE HCATALOG SCHEMA hcat WITH HCATALOG_SCHEMA='default'
   HIVESERVER2_HOSTNAME='hs.example.com'
   CUSTOM_PARTITIONS='yes';
```

CREATE KEY

Creates a private key.

Syntax

```
CREATE [TEMP[ORARY]] KEY name
    { 'AES' [ PASSWORD 'password' ] | 'RSA' }
    {LENGTH length | AS key_text}
```

Parameters

TEMPORARY

Create with session scope. The key is stored in memory and is valid only for the current session.

name

The name of the key.

password

Password for the key.

length

Size of the key in bits.

Example: 2048

key_text

The contents of the key to import.

Example:

```
-----BEGIN RSA PRIVATE KEY-----...ABCD1234...-----END RSA PRIVATE KEY-----
```

Privileges

Superuser

Examples

See Generating TLS certificates and keys .

See also

- CREATE CERTIFICATE

CREATE LIBRARY

Loads a library containing user-defined extensions (UDxs) into the Vertica catalog. Vertica automatically distributes copies of the library file and supporting libraries to all cluster nodes.

Because libraries are added to the database catalog, they persist across database restarts.

After loading a library in the catalog, you can use statements such as CREATE FUNCTION to define the extensions contained in the library. See Developing user-defined extensions (UDxs) for details.

Syntax

```
CREATE [OR REPLACE] LIBRARY
   [[database.]schema.]name
   AS 'path'
   [ DEPENDS 'depends-path' ]
   [ LANGUAGE 'language' ]
```

Arguments

OR REPLACE

If a library with the same name exists, replace it. UDxs defined in the catalog that reference the updated library automatically start using the new library file.

If you do not use this directive and the library already exists, the CREATE statement returns with an error.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

name

Name of the library to create. This is the name used when creating functions in the library (see Creating UDx Functions). While not required, it is good practice to match the file name.

AS *path*

Path of the library to load, either an absolute path on the initiator node file system or a URI for another supported file system or object store .

DEPENDS ' *depends-path* '

Files or libraries on which this library depends, one or more files or directories on the initiator node file system or other supported file systems or object stores . For a directory, end the path entry with a slash (/), optionally followed by a wildcard (*). To specify more than one file, separate entries with colons (:).
If any path entry contain colons, such as a URI, place brackets around the entire DEPENDS path and use double quotes for the individual path elements, as in the following example:

```
DEPENDS '["s3://mybucket/gson-2.3.1.jar"]'
```

To specify libraries with multiple directory levels, see Multi-level Library Dependencies .

DEPENDS has no effect for libraries written in R. R packages must be installed locally on each node, including external dependencies.

Important
The performance of CREATE LIBRARY can degrade in Eon Mode, in proportion to the number and depth of dependencies specified by the DEPENDS clause.

If a Java library depends on native libraries (SO files), use DEPENDS to specify the path and call System.loadLibrary() in your UDx to load the native libraries from that path.

LANGUAGE ' *language* '

The programming language of the functions in the library, one of:

- C++ (default)
- Python
- Java
- R

Privileges

Superuser, or UDXDEVELOPER and CREATE on the schema. Non-superusers must explicitly enable the UDXDEVELOPER role, as in the following example:

```
=> SET ROLE UDXDEVELOPER;
SET

-- Not required, but you can confirm the role as follows:
=> SHOW ENABLED ROLES;
    name     |   setting
---------------+--------------
 enabled roles | udxdeveloper
(1 row)

=> CREATE LIBRARY MyLib AS '/home/dbadmin/my_lib.so';
CREATE LIBRARY

-- Create functions...

-- UDXDEVELOPER also grants DROP (replace):
=> CREATE OR REPLACE LIBRARY MyLib AS '/home/dbadmin/my_lib.so';
```

Requirements

- Vertica makes its own copies of the library files. Later modification or deletion of the original files specified in the statement does not affect the library defined in the catalog. To update the library, use ALTER LIBRARY .

- Loading a library does not guarantee that it functions correctly. CREATE LIBRARY performs some basic checks on the library file to verify it is compatible with Vertica. The statement fails if it detects that the library was not correctly compiled or it finds other basic incompatibilities. However, CREATE LIBRARY cannot detect many other issues in shared libraries.

Multi-level library dependencies

If a DEPENDS clause specifies a library with multiple directory levels, Vertica follows the library path to include all subdirectories of that library. For example, the following CREATE LIBRARY statement enables the UDx library mylib to import all Python packages and modules that it finds in subdirectories of site-packages :

```
=> CREATE LIBRARY mylib AS '/path/to/python_udx' DEPENDS '/path/to/python/site-packages' LANGUAGE 'Python';
```

Important

DEPENDS can specify Java library dependencies that are up to 100 levels deep.

Examples

Load a library in the home directory of the dbadmin account:

```
=> CREATE LIBRARY MyFunctions AS '/home/dbadmin/my_functions.so';
```

Load a library located in the directory where you started vsql :

```
=> \set libfile '\''`pwd`'/MyOtherFunctions.so\'';
=> CREATE LIBRARY MyOtherFunctions AS :libfile;
```

Load a library from the cloud:

```
=> CREATE LIBRARY SomeFunctions AS 'S3://mybucket/extensions.so';
```

Load a library that depends on multiple JAR files in the same directory:

```
=> CREATE LIBRARY DeleteVowelsLib AS '/home/dbadmin/JavaLib.jar'
  DEPENDS '/home/dbadmin/mylibs/*' LANGUAGE 'Java';
```

Load a library with multiple explicit dependencies:

```
=> CREATE LIBRARY mylib AS '/path/to/java_udx'
  DEPENDS '/path/to/jars/this.jar:/path/to/jars/that.jar' LANGUAGE 'Java';
```

Load a library with dependencies in the cloud:

```
=> CREATE LIBRARY s3lib AS 's3://mybucket/UdlLib.jar'
  DEPENDS '["s3://mybucket/gson-2.3.1.jar"]' LANGUAGE 'Java';
```

CREATE LOAD BALANCE GROUP

Creates a group of network addresses that can be targeted by a load balancing routing rule. You create a group either using a list of network addresses, or basing it on one or more fault groups or subclusters.

> **Note**
> You cannot add multiple network addresses for one node to the same load balancing group.

Syntax

```
CREATE LOAD BALANCE GROUP group_name WITH {
    ADDRESS address[,...]
  | FAULT GROUP  fault_group[,...] FILTER 'IP_range'
  | SUBCLUSTER subcluster[,...] FILTER 'IP_range'
  }
  [ POLICY 'policy_setting' ]
```

Parameters

group_name

Name of the group to create. You use this name later when defining load balancing rules.

***address* [,...]**

Comma-delimited list of network addresses you created earlier.

***fault_group* [,...]**

Comma-delimited list of fault groups to use as the basis of the load balance group.

> **Note**
> Before you create your load balance group from a fault group, you must create network addresses on the nodes you want in your load balance group. Load balance groups only work with the network addresses you define on nodes, rather than IP addresses. See CREATE NETWORK ADDRESS .

***subcluster* [,...]**

Comma-delimited list of subclusters to use as the basis of the load balance group.

> **Note**
> As with fault groups, you must create network addresses on the nodes in the subcluster you want to be part of the load balance group.

IP_range

Range of IP addresses in CIDR notation to include in the load balance group from the fault groups or subclusters. This range can be either IPv4 or IPv6. Only nodes that have a network address with an IP address that falls within this range are added to the load balancing group.

policy_setting

Determines how the initially-contacted node chooses a target from the group, one of the following:

- ROUNDROBIN (default) rotates among the available members of the load balancing group. The initially-contacted node keeps track of which node it chose last time, and chooses the next one in the cluster.

  > **Note**
  > Each node in the cluster maintains its own round-robin pointer that indicates which node it should pick next for each load-balancing group. Therefore, if clients connect to different initial nodes, they may be redirected to the same node.

- RANDOM chooses an available node from the group randomly.
- NONE disables load balancing.

Privileges

Superuser

Examples

The following statement demonstrates creating a load balance group that contains several network addresses:

```
=> CREATE NETWORK ADDRESS addr01 ON v_vmart_node0001 WITH '10.20.110.21';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS addr02 ON v_vmart_node0002 WITH '10.20.110.22';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS addr03 on v_vmart_node0003 WITH '10.20.110.23';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS addr04 on v_vmart_node0004 WITH '10.20.110.24';
CREATE NETWORK ADDRESS
=> CREATE LOAD BALANCE GROUP group_1 WITH ADDRESS addr01, addr02;
CREATE LOAD BALANCE GROUP
=> CREATE LOAD BALANCE GROUP group_2 WITH ADDRESS addr03, addr04;
CREATE LOAD BALANCE GROUP

=> SELECT * FROM LOAD_BALANCE_GROUPS;
   name   |  policy  |    filter     |        type         | object_name
----------+----------+---------------+---------------------+-------------
 group_1  | ROUNDROBIN |             | Network Address Group | addr01
 group_1  | ROUNDROBIN |             | Network Address Group | addr02
 group_2  | ROUNDROBIN |             | Network Address Group | addr03
 group_2  | ROUNDROBIN |             | Network Address Group | addr04
(4 rows)
```

This example demonstrates creating a load balancing group using a fault group:

```
=> CREATE FAULT GROUP fault_1;
CREATE FAULT GROUP
=> ALTER FAULT GROUP fault_1 ADD NODE  v_vmart_node0001;
ALTER FAULT GROUP
=> ALTER FAULT GROUP fault_1 ADD NODE  v_vmart_node0002;
ALTER FAULT GROUP
=> ALTER FAULT GROUP fault_1 ADD NODE  v_vmart_node0003;
ALTER FAULT GROUP
=> ALTER FAULT GROUP fault_1 ADD NODE  v_vmart_node0004;
ALTER FAULT GROUP
=> SELECT node_name,node_address,node_address_family,export_address
  FROM v_catalog.nodes;
   node_name    | node_address | node_address_family | export_address
----------------+--------------+---------------------+----------------
 v_vmart_node0001 | 10.20.110.21 | ipv4              | 10.20.110.21
 v_vmart_node0002 | 10.20.110.22 | ipv4              | 10.20.110.22
 v_vmart_node0003 | 10.20.110.23 | ipv4              | 10.20.110.23
 v_vmart_node0004 | 10.20.110.24 | ipv4              | 10.20.110.24
(4 rows)

=> CREATE LOAD BALANCE GROUP group_all WITH FAULT GROUP fault_1 FILTER
  '0.0.0.0/0';
CREATE LOAD BALANCE GROUP

=> CREATE LOAD BALANCE GROUP group_some WITH FAULT GROUP fault_1 FILTER
  '10.20.110.21/30';
CREATE LOAD BALANCE GROUP

=> SELECT * FROM LOAD_BALANCE_GROUPS;
     name     |  policy  |    filter     |        type         | object_name
--------------+----------+---------------+---------------------+-------------
 group_all    | ROUNDROBIN | 0.0.0.0/0     | Fault Group         | fault_1
 group_some   | ROUNDROBIN | 10.20.110.21/30 | Fault Group       | fault_1
(2 rows)
```

See also

- ALTER LOAD BALANCE GROUP

- ALTER NETWORK ADDRESS
- ALTER ROUTING RULE
- LOAD_BALANCE_GROUPS
- NETWORK_ADDRESSES

CREATE LOCAL TEMPORARY VIEW

Creates or replaces a local temporary view. Views are read only, so they do not support insert, update, delete, or copy operations. Local temporary views are session-scoped, so they are visible only to their creator in the current session. Vertica drops the view when the session ends.

> **Note**
> Vertica does not support global temporary views.

Syntax

```
CREATE [OR REPLACE] LOCAL TEMP[ORARY] VIEW view [ (column[,...] ) ] AS query
```

Parameters

OR REPLACE

Specifies to overwrite the existing view *view-name* . If you omit this option and *view-name* already exists, CREATE VIEW returns an error.

view

Identifies the view to create, where *view* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

***column* [,...]**

List of up to 9800 names to use as view column names. Vertica maps view column names to query columns according to the order of their respective lists. By default, the view uses column names as they are specified in the query.

AS *query*

A SELECT statement that the temporary view executes. The SELECT statement can reference tables, temporary tables, and other views.

Privileges

See Creating views .

Examples

The following CREATE LOCAL TEMPORARY VIEW statement creates the temporary view myview . This view sums all individual incomes of customers listed in the store.store_sales_fact table, and groups results by state:

```
=> CREATE LOCAL TEMP VIEW myview AS
  SELECT SUM(annual_income), customer_state FROM public.customer_dimension
    WHERE customer_key IN (SELECT customer_key FROM store.store_sales_fact)
    GROUP BY customer_state
    ORDER BY customer_state ASC;
```

The following example uses the temporary view myview with a WHERE clause that limits the results to combined salaries greater than $2 billion:

```
=> SELECT * FROM myview WHERE SUM > 2000000000;


    SUM     | customer_state
------------+----------------
 2723441590 | AZ
29253817091 | CA
 4907216137 | CO
 3769455689 | CT
 3330524215 | FL
 4581840709 | IL
 3310667307 | IN
 2793284639 | MA
 5225333668 | MI
 2128169759 | NV
 2806150503 | PA
 2832710696 | TN
14215397659 | TX
 2642551509 | UT
(14 rows)
```

See also

- ALTER VIEW
- CREATE VIEW
- Creating views

CREATE LOCATION

Creates a storage location where Vertica can store data. After you create the location, you create storage policies that assign the storage location to the database objects that will store data in the location.

> **Caution**
> While no technical issue prevents you from using CREATE LOCATION to add one or more Network File System (NFS) storage locations, Vertica does not support NFS data or catalog storage except for MapR mount points. You will be unable to run queries against any other NFS data. When creating locations on MapR file systems, you must specify ALL NODES SHARED .
>
> If you use HDFS storage locations, the HDFS data must be available when you start Vertica. Your HDFS cluster must be operational, and the ROS files must be present. If you moved data files, or they are corrupted, or your HDFS cluster is not responsive, Vertica cannot start.

Syntax

```
CREATE LOCATION 'path'
   [NODE 'node' | ALL NODES]
   [SHARED]
   [USAGE 'usage']
   [LABEL 'label']
   [LIMIT 'size']
```

Arguments

path

Where to store this location's data. The type of file system on which the location is based determines the *path* format:

- Linux: Absolute path to the directory where Vertica can write the storage location's data.
- Shared file systems: See the URL specifications in HDFS file system , S3 object store , Google Cloud Storage (GCS) object store , and Azure Blob Storage object store .

HDFS storage locations have additional requirements .

Important

Vertica performs no validation on storage location paths. Confirm that the path value points to a valid location.

ALL NODES | NODE ' *node* '

The node or nodes on which the storage location is defined, one of the following:

- ALL NODES (default): Create the storage location on each node. If SHARED is also specified, create the storage location once for use by all nodes.
- NODE ' *node* ' : Create the storage location on a single node, where *node* is the name of the node in the NODES system table. You cannot use this option with SHARED.

SHARED

Indicates the location set by *path* is shared (used by all nodes) rather than local to each node. You cannot specify individual nodes with SHARED; you must use ALL NODES.

Most remote file systems such as HDFS and S3 are shared. For these file systems, the *path* argument represents a single location in the remote file system where all nodes store data. If using a remote file system, you must specify SHARED, even for one-node clusters.

> **Note**
> If *path* is set to S3 communal storage, SHARED is always implied and can be omitted.

> **Deprecated**
> SHARED DATA and SHARED DATA,TEMP storage locations are deprecated.

USAGE ' *usage* '

The type of data the storage location can hold, where *usage* is one of the following:

- DATA,TEMP (default): The storage location can store persistent and temporary DML-generated data, and data for temporary tables.
- TEMP : A *path* -specified location to store DML-generated temporary data. If *path* is set to S3, then this location is used only when the RemoteStorageForTemp configuration parameter is set to 1, and TEMP must be qualified with ALL NODES SHARED. For details, see S3 Storage of Temporary Data .
- DATA : The storage location can only store persistent data.
- USER : Users with READ and WRITE privileges can access data and external tables of this storage location.
- DEPOT : The storage location is used in Eon Mode to store the depot. Only create DEPOT storage locations on local Linux file systems. Vertica allows a single DEPOT storage location per node. If you want to move your depot to different location (on a different file system, for example) you must first drop the old depot storage location, then create the new location.

LABEL ' *label* '

A label for the storage location, used when assigning the storage location to data objects. You use this name later when assigning the storage location to data objects.
Important
You must supply a label for depot storage locations.

LIMIT ' *size* '

Valid only if the storage location usage type is set to DEPOT , specifies the maximum amount of disk space that the depot can allocate from the storage location's file system.
You can specify *size* in two ways:

- *integer* % : Percentage of storage location disk size.
- *integer* {K|M|G|T} : Amount of storage location disk size in kilobytes, megabytes, gigabytes, or terabytes.

Important
The depot size cannot exceed 80 percent of the file system disk space where the depot is stored. If you specify a value that is too large, Vertica issues a warning and automatically changes the value to 80 percent of the file system size.

If you do not specify a limit, it is set to 60 percent.

Privileges

Superuser

File system access

The Vertica process must have read and write permissions to the location where data is to be stored. Each file system has its own requirements:

File system	Requirements
Linux	Database superuser account (usually named dbadmin) must have full read and write access to the directory in the *path* argument.
HDFS without Kerberos	A Hadoop user whose username matches the Vertica database administrator username (usually dbadmin) must have read and write access to the HDFS directory specified in the *path* argument. The UseServerIdentityOverUserIdentity configuration parameter must be set to true in the user session; otherwise Vertica tries to use the identity associated with the logged-in user.
HDFS with Kerberos	A Hadoop user whose username matches the principal in the keytab file on each Vertica node must have read and write access to the HDFS directory stored in the path argument. This is not the same as the database administrator username. The UseServerIdentityOverUserIdentity configuration parameter must be set to true in the user session; otherwise Vertica tries to use the Kerberos principal associated with the logged-in user.
Object stores (S3, GCS, Azure)	Database-level credentials must be specified and provide full read and write access to the location in the path argument. If session-level credentials are specified they are used, directly overriding the use of the storage location.

Examples

Create a storage location in the local Linux file system for temporary data storage:

```
=> CREATE LOCATION '/home/dbadmin/testloc' USAGE 'TEMP' LABEL 'tempfiles';
```

Create a storage location on HDFS. The HDFS cluster does not use Kerberos:

```
=> CREATE LOCATION 'hdfs://hadoopNS/vertica/colddata' ALL NODES SHARED
  USAGE 'data' LABEL 'coldstorage';
```

Create the same storage location, but on a Hadoop cluster that uses Kerberos. Note the output that reports the principal being used:

```
=> CREATE LOCATION 'hdfs://hadoopNS/vertica/colddata' ALL NODES SHARED
  USAGE 'data' LABEL 'coldstorage';
NOTICE 0: Performing HDFS operations using kerberos principal [vertica/hadoop.example.com]
CREATE LOCATION
```

Create a location for user data, grant access to it, and use it to create an external table:

```
=> CREATE LOCATION '/tmp' ALL NODES USAGE 'user';
CREATE LOCATION
=> GRANT ALL ON LOCATION '/tmp' to Bob;
GRANT PRIVILEGE
=> CREATE EXTERNAL TABLE ext1 (x integer) AS COPY FROM '/tmp/data/ext1.dat' DELIMITER ',';
CREATE TABLE
```

Create a user storage location on S3 and a role, so that users without their own S3 credentials can read data from S3 using the server credential:

```
  --- set database-level credential (once):
=> ALTER DATABASE DEFAULT SET AWSAuth = 'myaccesskeyid123456:mysecretaccesskey12345678901234567890123456';

=> CREATE LOCATION 's3://datalake' SHARED USAGE 'USER' LABEL 's3user';

=> CREATE ROLE ExtUsers;
  --- Assign users to this role using GRANT (Role).

=> GRANT READ ON LOCATION 's3://datalake' TO ExtUsers;
```

See also

- Managing storage locations
- Using HDFS storage locations
- ALTER_LOCATION_LABEL

- ALTER_LOCATION_USE
- RETIRE_LOCATION
- SET_OBJECT_STORAGE_POLICY

CREATE NETWORK ADDRESS

Creates a network address that can be used as part of a connection load balancing policy. A network address creates a name in the Vertica catalog for an IP address and port number associated with a node. Nodes can have multiple network addresses, up to one for each IP address they have on the network.

Syntax

```
CREATE NETWORK ADDRESS name ON node WITH 'ip-address' [PORT port-number] [ENABLED | DISABLED]
```

Parameters

name

The name of the new network address. Use this name when creating connection load balancing groups.

node

The name of the node on which to create the network address. This should be name of the node as it appears in the node_name column of system table NODES .

ip-address

The IPv4 or and IPv6 address on the node to associate with the network address.

> **Note**
> Vertica does not verify that the IP address you supply in this parameter is actually associated with the specified node. Be sure that the IP address actually belongs to the node. Otherwise, your load balancing policy is liable to send a client connection to the wrong node, or a non-Vertica host. Vertica rejects IP address that are invalid for a node. For example, it checks whether the IP address falls in the loopback address range of 127.0.0.0/8. If it finds that the IP address is invalid, CREATE NETWORK ADDRESS returns an error.

PORT ***port-number***

Sets the port number for the network address. You must supply a network address when altering the port number.

ENABLED | DISABLED

Enables or disables the network address.

Privileges

Superuser

Examples

Create three network addresses, one for each node in a three-node cluster:

```
=> SELECT node_name,export_address from v_catalog.nodes;
     node_name      | export_address
--------------------+----------------
 v_vmart_br_node0001 | 10.20.100.62
 v_vmart_br_node0002 | 10.20.100.63
 v_vmart_br_node0003 | 10.20.100.64
(3 rows)

=> CREATE NETWORK ADDRESS node01 ON v_vmart_br_node0001 WITH '10.20.100.62';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node02 ON v_vmart_br_node0002 WITH '10.20.100.63';
CREATE NETWORK ADDRESS
=> CREATE NETWORK ADDRESS node03 ON v_vmart_br_node0003 WITH '10.20.100.64';
```

See also

CREATE NETWORK INTERFACE

Identifies a network interface to which a node belongs.

Use this statement when you want to configure import/export operations from individual nodes to other Vertica clusters. By default, when you install Vertica, it creates interfaces for all connected networks. You would only need CREATE NETWORK INTERFACE in situations where the network topology has changed since you installed Vertica.

> **Note**
> Do not confuse this statement with CREATE NETWORK ADDRESS, which is used to identify network addresses for connection load balancing (see Connection load balancing policies).

Syntax

```
CREATE NETWORK INTERFACE network-interface-name ON node-name [WITH] 'node-IP-address' [PORT port-number] [ENABLED | DISABLED]
```

network-interface-name

The name you assign to the network interface, where *network-interface-name* conforms to conventions described in Identifiers.

node-name

The name of the node.

node-IP-address

The node's IP address, either a public or private IP address. For more information, see Using Public and Private IP Networks.

PORT *port-number*

Sets the port number for the network interface. You must supply a network interface when altering the port number.

[ENABLED | DISABLED]

Enables or disables the network interface.

Privileges

Superuser

Examples

Create a network interface:

```
=> CREATE NETWORK INTERFACE mynetwork ON v_vmart_node0001 WITH '123.4.5.6' PORT 456 ENABLED;
```

CREATE NOTIFIER

Creates a push-based notifier to send event notifications and messages out of Vertica.

Syntax

```
CREATE NOTIFIER [ IF NOT EXISTS ] notifier-name ACTION 'notifier-type'
  [ ENABLE | DISABLE ]
  [ MAXPAYLOAD 'integer{K|M}' ]
  MAXMEMORYSIZE 'integer{K|M|G|T}'
  [ TLS CONFIGURATION tls-configuration ]
  [ TLSMODE 'tls-mode' ]
  [ CA BUNDLE bundle-name [ CERTIFICATE certificate-name ] ]
  [ IDENTIFIED BY 'uuid' ]
  [ [NO] CHECK COMMITTED ]
  [ PARAMETERS 'adapter-params' ]
```

Parameters

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP.

notifier-name

This notifier's unique identifier.

ACTION ' *notifier-type* '

String, the type of notifier, one of the following:

- URL, with the following format, that identifies one or more target Kafka servers:
  ```
  kafka://kafka-server-ip-address:port-number
  ```
  To enable failover when a Kafka server is unavailable, specify additional hosts in a comma-delimited list. For example:
  ```
  kafka://192.0.2.0:9092,192.0.2.1:9092,192.0.2.2:9092
  ```
- syslog : Notifications are sent to syslog . To use notifiers of this type, you must set the SyslogEnabled parameter:
  ```
  => ALTER DATABASE DEFAULT SET SyslogEnabled = 1
  ```
  Events monitored by this notifier type are not logged to MONITORING_EVENTS nor vertica.log .
- sns : Notifications are sent to a Simple Notification Service (SNS) endpoint .

ENABLE | DISABLE

Specifies whether to enable or disable the notifier.

Default: ENABLE .

MAXPAYLOAD ' *integer* {K|M} '

The maximum size of the message, up to 10^9 bytes, specified in kilobytes or megabytes.

The following restrictions apply:

- MAXPAYLOAD cannot be greater than MAXMEMORYSIZE .
- If you configure syslog to send messages to a remote destination, ensure that MaxMessageSize (in /etc/rsyslog for rsyslog) is greater than or equal to MAXPAYLOAD .
- The MAXPAYLOAD for SNS notifiers cannot exceed 256KB.

Defaults:

- Kafka: 1M
- syslog: 1M
- SNS: 256K

MAXMEMORYSIZE ' *integer* {K|M|G|T} '

The maximum size of the internal notifier, up to 2 TB, specified in kilobytes, megabytes, gigabytes, or terabytes.

MAXMEMORYSIZE must be greater than MAXPAYLOAD .
If the size of the message queue exceeds MAXMEMORYSIZE , the notifier drops excess messages.

TLS CONFIGURATION *tls-configuration*

The TLS CONFIGURATION to use for TLS.
Notifiers support the following TLS modes:

- DISABLE
- TRY_VERIFY (behaves like VERIFY_CA)
- VERIFY_CA
- VERIFY_FULL

If the notifier ACTION is 'syslog' or 'sns' , this parameter has no effect.

To encrypt messages sent to syslog, you must configure syslog for TLS.

To encrypt messages sent to an SNS endpoint, you must set the following configuration parameters:

- SNSCAFile or AWSCAFile
- SNSCAPath or AWSCAPath
- SNSEnableHttps

TLSMODE ' *tls-mode* '

> **Deprecated**
> This parameter has been superseded by the TLS CONFIGURATION parameter. If you use this parameter while the TLS CONFIGURATION parameter is not set, Vertica automatically creates a new TLS Configuration for the notifier uses the same values as the deprecated parameter.

Specifies the type of connection between the notifier and an endpoint, one of the following:

- disable (default): Plaintext connection.
- verify-ca : Encrypted connection, and the server's certificate is verified as being signed by a trusted CA.

If you set this parameter to verify-ca , the generated TLS Configuration will be set to TRY_VERIFY, which has the same behavior as VERIFY_CA.

If the notifier ACTION is 'syslog' or 'sns' , this parameter has no effect.

To encrypt messages sent to syslog, you must configure syslog for TLS.

To encrypt messages sent to an SNS endpoint, you must set the following configuration parameters:

- SNSCAFile or AWSCAFile
- SNSCAPath or AWSCAPath
- SNSEnableHttps

CA BUNDLE ***bundle-name***

> **Deprecated**
> This parameter has been superseded by the TLS CONFIGURATION parameter. If you use this parameter while the TLS CONFIGURATION parameter is not set, Vertica automatically creates a new TLS Configuration for the notifier uses the same values as the deprecated parameter.

Specifies a CA bundle . The certificates inside the bundle are used to validate the Kafka server's certificate if the TLSMODE requires it.
If a CA bundle is specified for a notifier that currently uses disable , which doesn't validate the Kafka server's certificate, the bundle will go unused when connecting to the Kafka server. This behavior persists unless the TLSMODE is changed to one that validates server certificates.

Changes to contents of the CA bundle take effect either after the notifier is disabled and re-enabled or after the database restarts. However, changes to which CA bundle the notifier uses takes effect immediately.

If the notifier ACTION is 'syslog' or 'sns' , this parameter has no effect.

To encrypt messages sent to syslog, you must configure syslog for TLS.

To encrypt messages sent to an SNS endpoint, you must set the following configuration parameters:

- SNSCAFile or AWSCAFile
- SNSCAPath or AWSCAPath
- SNSEnableHttps

CERTIFICATE ***certificate-name***

> **Deprecated**
> This parameter has been superseded by the TLS CONFIGURATION parameter. If you use this parameter while the TLS CONFIGURATION parameter is not set, Vertica automatically creates a new TLS Configuration for the notifier uses the same values as the deprecated parameter.

Specifies a client certificate for validation by the endpoint.
If the notifier ACTION is 'syslog' or 'sns' , this parameter has no effect.

To encrypt messages sent to syslog, you must configure syslog for TLS.

To encrypt messages sent to an SNS endpoint, you must set the following configuration parameters:

- SNSCAFile or AWSCAFile
- SNSCAPath or AWSCAPath
- SNSEnableHttps

IDENTIFIED BY ***uuid***

Specifies the notifier's unique identifier. If set, all the messages published by this notifier have this attribute.

[NO] CHECK COMMITTED

Specifies to wait for delivery confirmation before sending the next message in the queue.

Some messaging systems, like syslog, do not support delivery confirmation.
For SNS notifiers, CHECK COMMITTED must be specified, and NO CHECK COMMITTED behaves like CHECK COMMITTED.

PARAMETERS ' *adapter-params* '

Specifies one or more optional adapter parameters that are passed as a string to the adapter. Adapter parameters apply only to the adapter associated with the notifier.

For Kafka notifiers, refer to Kafka and Vertica configuration settings .
For syslog notifiers, specify the severity of the event with eventSeverity= *severity* , where *severity* is one of the following:

- 0 : Emergency
- 1 : Alert
- 2 : Critical
- 3 : Error
- 4 : Warning
- 5 : Notice
- 6 : Informational
- 7 : Debug

Most syslog implementations, by default, do not log events with a severity level of 7. You must configure syslog to record these types of events.

Parameters cannot be set for SNS notifiers.

Privileges

Superuser

Encrypted notifiers for SASL_SSL Kafka configurations

Follow this procedure to create or alter notifiers for Kafka endpoints that use SASL_SSL. Note that you must repeat this procedure whenever you change the TLSMODE, certificates, or CA bundle for a given notifier.

1. Create a TLS Configuration with the desired TLS mode, certificate, and CA certificates.
2. Use CREATE or ALTER to disable the notifier and set the TLS Configuration:
```
=> ALTER NOTIFIER encrypted_notifier
   DISABLE
   TLS CONFIGURATION kafka_tls_config;
```
3. ALTER the notifier and set the proper rdkafka adapter parameters for SASL_SSL:
```
=> ALTER NOTIFIER encrypted_notifier PARAMETERS
   'sasl.username=user;sasl.password=password;sasl.mechanism=PLAIN;security.protocol=SASL_SSL';
```
4. Enable the notifier:
```
=> ALTER NOTIFIER encrypted_notifier ENABLE;
```

Examples

Kafka notifiers

Create a Kafka notifier:

```
=> CREATE NOTIFIER my_dc_notifier
   ACTION 'kafka://172.16.20.10:9092'
   MAXMEMORYSIZE '1G'
   IDENTIFIED BY 'f8b0278a-3282-4e1a-9c86-e0f3f042a971'
   NO CHECK COMMITTED;
```

Create a notifier with an adapter-specific parameter:

```
=> CREATE NOTIFIER my_notifier
   ACTION 'kafka://127.0.0.1:9092'
   MAXMEMORYSIZE '10M'
   PARAMETERS 'queue.buffering.max.ms=1000';
```

Create a notifier that uses an encrypted connection and verifies the Kafka server's certificate with the CA certificates in the notifier_tls_config object:

```
=> CREATE NOTIFIER encrypted_notifier
   ACTION 'kafka://127.0.0.1:9092'
   MAXMEMORYSIZE '10M'
   TLS CONFIGURATION 'notifier_tls_config'
```

Syslog notifiers

The following example creates a notifier that writes a message to syslog when the Data collector (DC) component LoginFailures updates:

1. Enable syslog notifiers for the current database:
   ```
   => ALTER DATABASE DEFAULT SET SyslogEnabled = 1;
   ```
2. Create and enable a syslog notifier v_syslog_notifier :
   ```
   => CREATE NOTIFIER v_syslog_notifier ACTION 'syslog'
      ENABLE
      MAXMEMORYSIZE '10M'
      IDENTIFIED BY 'f8b0278a-3282-4e1a-9c86-e0f3f042a971'
      PARAMETERS 'eventSeverity = 5';
   ```
3. Configure the syslog notifier v_syslog_notifier for updates to the LoginFailures DC component with SET_DATA_COLLECTOR_NOTIFY_POLICY :
   ```
   => SELECT SET_DATA_COLLECTOR_NOTIFY_POLICY('LoginFailures','v_syslog_notifier', 'Login failed!', true);
   ```

   This notifier writes the following message to syslog (default location: /var/log/messages) when a user fails to authenticate as the user Bob :
   ```
   Apr 25 16:04:58
   vertica_host_01
   vertica:
      Event Posted:
         Event Code:21
         Event Id:0
         Event Severity: Notice [5]
         PostedTimestamp: 2022-04-25 16:04:58.083063
         ExpirationTimestamp: 2022-04-25 16:04:58.083063
         EventCodeDescription: Notifier
         ProblemDescription: (Login failed!)
      {
        "_db":"VMart",
        "_schema":"v_internal",
        "_table":"dc_login_failures",
        "_uuid":"f8b0278a-3282-4e1a-9c86-e0f3f042a971",
        "authentication_method":"Reject",
        "client_authentication_name":"default: Reject",
        "client_hostname":"::1",
        "client_label":"",
        "client_os_user_name":"dbadmin",
        "client_pid":523418,
        "client_version":"",
        "database_name":"dbadmin",
        "effective_protocol":"3.8",
        "node_name":"v_vmart_node0001",
        "reason":"REJECT",
        "requested_protocol":"3.8",
        "ssl_client_fingerprint":"",
        "ssl_client_subject":"",
        "time":"2022-04-25 16:04:58.082568-05",
        "user_name":"Bob"
      }#012
      DatabaseName: VMart
      Hostname: vertica_host_01
   ```

For details on syslog notifiers, see Configuring reporting for syslog .

See also

- ALTER NOTIFIER

- DROP NOTIFIER
- Monitoring Vertica Using Notifiers

CREATE PROCEDURE (external)

Enterprise Mode only

Adds an external procedure to Vertica. See External procedures for more information.

Syntax

```
CREATE PROCEDURE [ IF NOT EXISTS ]
  [[database.]schema.]procedure( [ argument-list ] )
  AS executable
  LANGUAGE 'EXTERNAL'
  USER OS-user
```

Parameters

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

This option cannot be used with OR REPLACE .

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

procedure

Specifies a name for the external procedure, where * procedure-name *conforms to conventions described in Identifiers .

argument-list

A comma-delimited list of procedure arguments, where each argument is specified as follows:

```
[ argname ] argtype
```

- *argname* optionally provides a descriptive name for this argument.
- *argtype* must be one of the following data types supported byVertica:
  - BIGINT
  - BOOLEAN
  - DECIMAL
  - DOUBLE PRECISION
  - FLOAT
  - FLOAT8
  - INT
  - INT8
  - INTEGER
  - MONEY
  - NUMBER
  - NUMERIC
  - REAL
  - SMALLINT
  - TINYINT
  - VARCHAR

executable

The name of the executable program in the procedures directory, a string.

OS-user

The owner of the file, a string. The owner:

- Cannot be root
- Must have execute privileges on *executable*

Privileges

Superuser

System security

- The procedure file must be owned by the database administrator (OS account) or by a user in the same group as the administrator. The procedure file must also have the set UID attribute enabled, and allow read and execute permission for the group.
- External procedures that you create with CREATE PROCEDURE (external) are always run with Linux dbadmin privileges. If a dbadmin or pseudosuperuser grants a non-dbadmin permission to run a procedure using GRANT (procedure) , be aware that the non-dbadmin user runs the procedure with full Linux dbadmin privileges.

Examples

The following example shows how to create a procedure named helloplanet for the procedure file helloplanet.sh . This file accepts one VARCHAR argument.

Create the file:

```
#!/bin/bash
echo "hello planet argument: $1" >> /tmp/myprocedure.log
```

Create the procedure with the following SQL:

```
=> CREATE PROCEDURE helloplanet(arg1 varchar) AS 'helloplanet.sh' LANGUAGE 'external' USER 'dbadmin';
```

See also

- DROP PROCEDURE (external)
- Installing external procedure executable files

CREATE PROCEDURE (stored)

Creates a stored procedure .

Syntax

```
CREATE [ OR REPLACE ] PROCEDURE [ IF NOT EXISTS ]
  [[database.]schema.]procedure( [ parameter-list ] )
  [ LANGUAGE 'language-name' ]
  [ SECURITY { DEFINER | INVOKER } ]
  AS $$ source $$;
```

Parameters

OR REPLACE

If a procedure with the same name already exists, replace it. Users and roles with privileges on the original procedure retain these privileges on the new procedure.

This option cannot be used with IF NOT EXISTS .

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

This option cannot be used with OR REPLACE .

[database .] schema

Database and schema . The default schema is public . If you specify a database, it must be the current database.

procedure

The name of the stored procedure, where * procedure-name *conforms to conventions described in Identifiers .

parameter-list

A comma-delimited list of formal parameters, each specified as follows:

```
[ parameter-mode ] parameter-name parameter-type
```

- *parameter-name* : the name of the parameter.
- *parameter-type* : Any SQL data type , with the following exceptions:
  - DECIMAL
  - NUMERIC
  - NUMBER
  - MONEY
  - UUID
  - GEOGRAPHY
  - GEOMETRY
  - Complex types

language-name

Specifies the language of the procedure *source* , one of the following (both options refer to PLvSQL ; PLpgSQL is included to maintain compatibility with existing scripts):

- PLvSQL
- PLpgSQL

Default: PLvSQL

SECURITY { DEFINER | INVOKER }

Determines whose privileges to use when the procedure is called and executes it as if the user is one of the following:

- DEFINER: User who defined the procedure
- INVOKER: User who called the procedure

A procedure with SECURITY DEFINER effectively executes the procedure as that user, so changes to the database appear to be performed by the procedure's definer rather than its caller.

> **Caution**
> Improper use of SECURITY DEFINER can lead to the confused deputy problem and introduce vulnerabilities into your system like SQL injection.

For more information, see Executing stored procedures .

source

The procedure source code. For details, see Scope and structure .

Privileges

Non-superuser: CREATE on the procedure's schema

Examples

For more complex examples, see Stored procedures: use cases and examples

This procedure prints its arguments:

```
=> CREATE PROCEDURE raiseXY(IN x INT, y VARCHAR) LANGUAGE PLvSQL AS $$
BEGIN
   RAISE NOTICE 'x = %', x;
   RAISE NOTICE 'y = %', y;
   -- some processing statements
END
$$;

CALL raiseXY(3, 'some string');
NOTICE 2005:  x = 3
NOTICE 2005:  y = some string
```

For more information on RAISE NOTICE, see Errors and diagnostics .

See also

- PL/vSQL
- CALL
- DO

- DROP PROCEDURE (stored)

CREATE PROFILE

Creates a profile that controls password requirements for users.

Syntax

```
CREATE PROFILE profile-name LIMIT [
   PASSWORD_LIFE_TIME setting
   PASSWORD_MIN_LIFE_TIME setting
   PASSWORD_GRACE_TIME setting
   FAILED_LOGIN_ATTEMPTS setting
   PASSWORD_LOCK_TIME setting
   PASSWORD_REUSE_MAX setting
   PASSWORD_REUSE_TIME setting
   PASSWORD_MAX_LENGTH setting
   PASSWORD_MIN_LENGTH setting
   PASSWORD_MIN_LETTERS setting
   PASSWORD_MIN_UPPERCASE_LETTERS setting
   PASSWORD_MIN_LOWERCASE_LETTERS setting
   PASSWORD_MIN_DIGITS setting
   PASSWORD_MIN_SYMBOLS setting
   PASSWORD_MIN_CHAR_CHANGE setting ]
```

Parameters

> **Note**
> All parameters that are not explicitly set in a new profile are set to default , and inherit their settings from the default profile.

Name	Description
name	The name of the profile to create, where * name *conforms to conventions described in Identifiers . To modify the default profile, set *name* to default . For example: ALTER PROFILE DEFAULT LIMIT PASSWORD_MIN_SYMBOLS 1;
PASSWORD_LIFE_TIME	Set to an integer value, one of the following: • ≥ 1: The number of days a password remains valid. • UNLIMITED : Password remains valid indefinitely. After your password's lifetime and grace period expire, you must change your password on your next login, if you have not done so already.
PASSWORD_MIN_LIFE_TIME	Set to an integer value, one of the following: • Default: 0 • ≥ 1: The number of days a password must be set before it can be changed • UNLIMITED : Password can be reset at any time.
PASSWORD_GRACE_TIME	Set to an integer value, one of the following: • ≥ 1: The number of days a password can be used after it expires. • UNLIMITED : No grace period.

FAILED_LOGIN_ATTEMPTS	Set to an integer value, one of the following: • ≥ 1: The number of consecutive failed login attempts Vertica allows before locking your account. • UNLIMITED : Vertica allows an unlimited number of failed login attempts.
PASSWORD_LOCK_TIME	• ≥ 1: The number of days (units configurable with PasswordLockTimeUnit) a user's account is locked after FAILED_LOGIN_ATTEMPTS number of login attempts. The account is automatically unlocked when the lock time elapses. • UNLIMITED : Account remains indefinitely inaccessible until a superuser manually unlocks it.
PASSWORD_REUSE_MAX	Set to an integer value, one of the following: • ≥ 1: The number of times you must change your password before you can reuse an earlier password. • UNLIMITED : You can reuse an earlier password without any intervening changes.
PASSWORD_REUSE_TIME	Set to an integer value, one of the following: • ≥ 1: The number of days that must pass after a password is set before you can reuse it. • UNLIMITED : You can reuse an earlier password immediately.
PASSWORD_MAX_LENGTH	The maximum number of characters allowed in a password, one of the following: • Integer between 8 and 512, inclusive
PASSWORD_MIN_LENGTH	The minimum number of characters required in a password, one of the following: • 0 to PASSWORD_MAX_LENGTH • UNLIMITED : Minimum of PASSWORD_MAX_LENGTH
PASSWORD_MIN_LETTERS	Minimum number of letters (a-z and A-Z) that must be in a password, one of the following: • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)
PASSWORD_MIN_UPPERCASE_LETTERS	Minimum number of uppercase letters (A-Z) that must be in a password, one of the following: • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)
PASSWORD_MIN_LOWERCASE_LETTERS	Minimum number of lowercase letters (a-z) that must be in a password, one of the following: • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)
PASSWORD_MIN_DIGITS	Minimum number of digits (0-9) that must be in a password, one of the following: • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)

PASSWORD_MIN_SYMBOLS	Minimum number of symbols—printable non-letter and non-digit characters such as $, #, @—that must be in a password, one of the following: • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)
PASSWORD_MIN_CHAR_CHANGE	Minimum number of characters that must be different from the previous password: • Default: 0 • Integer between 0 and PASSWORD_MAX_LENGTH , inclusive • UNLIMITED : 0 (no minimum)

Privileges

Superuser

Profile settings and client authentication

The following profile settings affect client authentication methods , such as LDAP or GSS:

- FAILED_LOGIN_ATTEMPTS
- PASSWORD_LOCK_TIME

All other profile settings are used only by Vertica to manage its passwords.

Examples

```
=> CREATE PROFILE sample_profile LIMIT PASSWORD_MAX_LENGTH 20;
```

See also

- ALTER PROFILE
- DROP PROFILE
- Creating a database name and password
- Profiles

CREATE PROJECTION

Creates metadata for a projection in the Vertica catalog. Vertica supports four types of projections:

- Standard projection : Stores a collection of table data in a format that optimizes execution of certain queries on that table.
- Live aggregate projection : Stores the grouped results of queries that invoke aggregate functions (such as SUM) on table columns.
- Top-K projection : Stores the top *k* rows from partitions of selected rows.
- UDTF projection : Stores newly-loaded data after it is transformed and/or aggregated by user-defined transformation functions (UDTFs).

Complex data types have additional restrictions when used within a projection:

- Each projection must include at least one column that is a primitive type or native array.
- An AS SELECT clause can use a complex-type column, but any other expression must be of a scalar type or native array.
- The ORDER BY , PARTITION BY , and GROUP BY clauses cannot use complex types.
- If a projection does not include an ORDER BY or segmentation clause, Vertica uses only the primitive columns from the select list to order or segment data.
- Projection columns cannot be complex types returned from functions such as ARRAY_CAT .
- TopK and UDTF projections do not support complex types.

In this section

- Encoding types
- GROUPED clause
- Hash segmentation clause
- Live aggregate projection
- Standard projection
- Top-k projection
- UDTF projection
- Unsegmented clause

Encoding types

Vertica supports various encoding and compression types, specified by the following ENCODING parameter arguments:

- AUTO (default)
- BLOCK_DICT
- BLOCKDICT_COMP
- BZIP_COMP
- COMMONDELTA_COMP
- DELTARANGE_COMP
- DELTAVAL
- GCDDELTA
- GZIP_COMP
- RLE
- Zstandard Compression

> **Note**
> Vertica supports the following encoding for numeric data types :
>
> - Precision ≤ 18: AUTO , BLOCK_DICT , BLOCKDICT_COMP , COMMONDELTA_COMP , DELTAVAL , GCDDELTA , and RLE
> - Precision > 18: AUTO , BLOCK_DICT , BLOCKDICT_COMP , RLE

You can set encoding types on a projection column when you create the projection . You can also change the encoding of one or more projection columns for a given table with ALTER TABLE...ALTER COLUMN .

AUTO (default)

AUTO encoding is ideal for sorted, many-valued columns such as primary keys. It is also suitable for general purpose applications for which no other encoding or compression scheme is applicable. Therefore, it serves as the default if no encoding/compression is specified.

Column data type	Default encoding type
BINARY/VARBINARY BOOLEAN CHAR/VARCHAR FLOAT	Lempel-Ziv-Oberhumer-based (LZO) compression
DATE/TIME/TIMESTAMP INTEGER INTERVAL	Compression scheme based on the delta between consecutive column values.

The CPU requirements for this type are relatively small. In the worst case, data might expand by eight percent (8%) for LZO and twenty percent (20%) for integer data.

BLOCK_DICT

For each block of storage, Vertica compiles distinct column values into a dictionary and then stores the dictionary and a list of indexes to represent the data block.

BLOCK_DICT is ideal for few-valued, unsorted columns where saving space is more important than encoding speed. Certain kinds of data, such as stock prices, are typically few-valued within a localized area after the data is sorted, such as by stock symbol and timestamp, and are good candidates for BLOCK_DICT. By contrast, long CHAR/VARCHAR columns are not good candidates for BLOCK_DICT encoding.

CHAR and VARCHAR columns that contain 0x00 or 0xFF characters should not be encoded with BLOCK_DICT. Also, BINARY/VARBINARY columns do not support BLOCK_DICT encoding.

BLOCK_DICT encoding requires significantly higher CPU usage than default encoding schemes. The maximum data expansion is eight percent (8%).

BLOCKDICT_COMP

This encoding type is similar to BLOCK_DICT except dictionary indexes are entropy coded. This encoding type requires significantly more CPU time to encode and decode and has a poorer worst-case performance. However, if the distribution of values is extremely skewed, using BLOCK_DICT_COMP encoding can lead to space savings.

BZIP_COMP

BZIP_COMP encoding uses the bzip2 compression algorithm on the block contents. See bzip web site for more information. This algorithm results in higher compression than the automatic LZO and gzip encoding; however, it requires more CPU time to compress. This algorithm is best used on large string columns such as VARCHAR, VARBINARY, CHAR, and BINARY. Choose this encoding type when you are willing to trade slower load speeds for higher data compression.

COMMONDELTA_COMP

This compression scheme builds a dictionary of all deltas in the block and then stores indexes into the delta dictionary using entropy coding.

This scheme is ideal for sorted FLOAT and INTEGER-based (DATE/TIME/TIMESTAMP/INTERVAL) data columns with predictable sequences and only occasional sequence breaks, such as timestamps recorded at periodic intervals or primary keys. For example, the following sequence compresses well: 300, 600, 900, 1200, 1500, 600, 1200, 1800, 2400. The following sequence does not compress well: 1, 3, 6, 10, 15, 21, 28, 36, 45, 55.

If delta distribution is excellent, columns can be stored in less than one bit per row. However, this scheme is very CPU intensive. If you use this scheme on data with arbitrary deltas, it can cause significant data expansion.

DELTARANGE_COMP

This compression scheme is primarily used for floating-point data; it stores each value as a delta from the previous one.

This scheme is ideal for many-valued FLOAT columns that are sorted or confined to a range. Do not use this scheme for unsorted columns that contain NULL values, as the storage cost for representing a NULL value is high. This scheme has a high cost for both compression and decompression.

To determine if DELTARANGE_COMP is suitable for a particular set of data, compare it to other schemes. Be sure to use the same sort order as the projection, and select sample data that will be stored consecutively in the database.

DELTAVAL

For INTEGER and DATE/TIME/TIMESTAMP/INTERVAL columns, data is recorded as a difference from the smallest value in the data block. This encoding has no effect on other data types.

DELTAVAL is best used for many-valued, unsorted integer or integer-based columns. CPU requirements for this encoding type are minimal, and data never expands.

GCDDELTA

For INTEGER and DATE/TIME/TIMESTAMP/INTERVAL columns, and NUMERIC columns with 18 or fewer digits, data is recorded as the difference from the smallest value in the data block divided by the greatest common divisor (GCD) of all entries in the block. This encoding has no effect on other data types.

ENCODING GCDDELTA is best used for many-valued, unsorted, integer columns or integer-based columns, when the values are a multiple of a common factor. For example, timestamps are stored internally in microseconds, so data that is only precise to the millisecond are all multiples of 1000. The CPU requirements for decoding GCDDELTA encoding are minimal, and the data never expands, but GCDDELTA may take more encoding time than DELTAVAL.

GZIP_COMP

This encoding type uses the gzip compression algorithm. See gzip web site for more information. This algorithm results in better compression than the automatic LZO compression, but lower compression than BZIP_COMP. It requires more CPU time to compress than LZO but less CPU time than BZIP_COMP. This algorithm is best used on large string columns such as VARCHAR, VARBINARY, CHAR, and BINARY. Use this encoding when you want a better compression than LZO, but at less CPU time than bzip2.

RLE

RLE (run length encoding) replaces sequences (runs) of identical values with a single pair that contains the value and number of occurrences. Therefore, it is best used for low cardinality columns that are present in the ORDER BY clause of a projection.

The Vertica execution engine processes RLE encoding run-by-run and the Vertica optimizer gives it preference. Use it only when run length is large, such as when low-cardinality columns are sorted.

Zstandard compression

Vertica supports three ZSTD compression types:

- ZSTD_COMP provides high compression ratios. This encoding type has a higher compression than gzip. Use this when you want a better compression than gzip. For general use cases, use this or the ZSTD_FAST_COMP encoding type.
- ZSTD_FAST_COMP uses the fastest compression level that the zstd library provides. It is the fastest encoding type of the zstd library, but takes up more space than the other two encoding types. For general use cases, use this or the ZSTD_COMP encoding type.
- ZSTD_HIGH_COMP offers the best compression in the zstd library. It is slower than the other two encoding types. Use this type when you need the best compression, with slower CPU time.

GROUPED clause

Enterprise Mode only

Groups two or more columns into a single disk file. Doing so minimizes file I/O for the following tasks:

- Read a large percentage of the columns in a table.
- Perform single row look-ups.
- Query against many small columns.
- Frequently update data in these columns.

You can improve query performance by grouping columns that are always accessed together and are not used in predicates. Once columns are grouped, queries can no longer retrieve from disk records for one column independently of the others.

> **Note**
> RLE encoding is reduced when an RLE column is grouped with one or more non-RLE columns.

You can group columns in several ways:

- Group some of the columns:
  ```
  (a, GROUPED(b, c), d)
  ```
- Group all of the columns:
  ```
  (GROUPED(a, b, c, d))
  ```
- Create multiple groupings in the same projection:
  ```
  (GROUPED(a, b), GROUPED(c, d))
  ```

> **Note**
> Vertica performs dynamic column grouping. For example, to provide better read and write efficiency for small loads, Vertica ignores any projection-defined column grouping (or lack thereof) and groups all columns together by default.

Grouping columns

The following example shows how to group columns bid and ask . The stock column is stored separately.

```
=> CREATE TABLE trades (stock CHAR(5), bid INT, ask INT);
=> CREATE PROJECTION tradeproj (stock ENCODING RLE,
  GROUPED(bid ENCODING DELTAVAL, ask))
  AS (SELECT * FROM trades) KSAFE 1;
```

The following example show how to create a projection that uses expressions in the column definition. The projection contains two integer columns a and b , and a third column product_value that stores the product of a and b :

```
=> CREATE TABLE values (a INT, b INT);
=> CREATE PROJECTION product (a, b, product_value) AS
  SELECT a, b, a*b FROM values ORDER BY a KSAFE;
```

Hash segmentation clause

Specifies how to segment projection data for distribution across all cluster nodes. You can specify segmentation for a table and a projection. If a table definition specifies segmentation, Vertica uses it for that table's auto-projections .

It is strongly recommended that you use Vertica's built-in HASH function, which distributes data evenly across the cluster, and facilitates optimal query execution.

Syntax

```
SEGMENTED BY expression ALL NODES [ OFFSET offset ]
```

Parameters

SEGMENTED BY *expression*

A general SQL expression. Hash segmentation is the preferred method of segmentation. Vertica recommends using its built-in HASH function, whose arguments resolve to table columns. If you use an expression other than HASH , Vertica issues a warning.

The segmentation expression should specify columns with a large number of unique data values and acceptable skew in their data distribution. In general, primary key columns that meet these criteria are good candidates for hash segmentation.
For details, see Expression Requirements below.

ALL NODES

Automatically distributes data evenly across all nodes when the projection is created. Node ordering is fixed.

OFFSET *offset*

A zero-based offset that indicates on which node to start segmentation distribution.

This option is not valid for CREATE TABLE and CREATE TEMPORARY TABLE .

Important
If you create a projection for a table with the OFFSET option, be sure to create enough copies of each projection segment to satisfy system K-safety; otherwise, Vertica regards the projection as unsafe and cannot use it to query the table.

You can ensure K-safety compliance when you create projections by combining OFFSET and KSAFE options in the CREATE PROJECTION statement. On executing this statement, Vertica automatically creates the necessary number of projection copies.

Expression requirements

A segmentation expression must specify table columns as they are defined in the source table. Projection column names are not supported.

The following restrictions apply to segmentation expressions:

- All leaf expressions must be constants or column references to a column in the CREATE PROJECTION 's SELECT list.
- The expression must return the same value over the life of the database.
- Aggregate functions are not allowed.
- The expression must return non-negative INTEGER values in the range $0 <= x < 2^{63}$, and values are generally distributed uniformly over that range.

> **Note**
> If the expression produces a value outside the expected range—for example, a negative value—no error occurs, and the row is added to the projection's first segment.

Examples

The following CREATE PROJECTION statement creates projection public.employee_dimension_super . It specifies to include all columns in table public.employee_dimension . The hash segmentation clause invokes the Vertica HASH function to segment projection data on the column employee_key ; it also includes the ALL NODES clause, which specifies to distribute projection data evenly across all nodes in the cluster:

```
=> CREATE PROJECTION public.employee_dimension_super
   AS SELECT * FROM public.employee_dimension
   ORDER BY employee_key
   SEGMENTED BY hash(employee_key) ALL NODES;
```

Live aggregate projection

Stores the grouped results of queries that invoke aggregate functions (such as SUM) on table columns. For details, see Live aggregate projections .

Syntax

```
CREATE PROJECTION [ IF NOT EXISTS ] [[database.]schema.]projection
[ (
  { projection-column | grouped-clause
  [ ENCODING encoding-type ]
  [ ACCESSRANK integer ] }[,...]
) ]
AS SELECT { table-column | expr-with-table-columns }[,...] FROM [[database.]schema.]table [ [AS] alias]
  GROUP BY column-expr
  [ KSAFE [ k-num ] ]
```

Parameters

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

[*database* .] *schema*

Specifies the schema for this projection and its anchor table, where *schema* must be the same for both. If you specify a database, it must be the current database.

projection

Identifies the projection to create, where *projection* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

projection-column

The name of a projection column. The list of projection columns must match the SELECT list columns and expressions in number, type, and sequence.
If projection column names are omitted, Vertica uses the anchor table column names specified in the SELECT list.

grouped-clause

See GROUPED clause .

ENCODING *encoding-type*

The column encoding type , by default set to AUTO.

ACCESSRANK *integer*

Overrides the default access rank for a column. Use this parameter to increase or decrease the speed at which Vertica accesses a column. For more information, see Overriding Default Column Ranking .

AS SELECT

Specifies the table data to query:

```
{table-column | expr-with-table-columns } [ [AS] alias] }[,...]
```

You can optionally assign an alias to each column expression and reference that alias elsewhere in the SELECT statement.

> **Note**
> If you specify projection column names, the two lists of projection columns and table columns/expressions must exactly match in number and order.

GROUP BY *column-expr* [,...]

One or more column expressions from the SELECT list. The first *column-expr* must be the first column expression in the SELECT list, the second *column-expr* must be the second column expression in the SELECT list, and so on.

Privileges

Non-superusers:

- Anchor table owner
- CREATE privilege on the schema

Requirements and restrictions

Vertica does not regard live aggregate projections as superprojections , even those that include all table columns. For other requirements and restrictions, see Creating live aggregate projections .

Examples

See Live aggregate projection example .

Standard projection

Stores a collection of table data in a format that optimizes execution of certain queries on that table. For details, see Projections .

Syntax

```
CREATE PROJECTION [ IF NOT EXISTS ] [[database.]schema.]projection
[ (
  { projection-column | grouped-clause
  [ ENCODING encoding-type ]
  [ ACCESSRANK integer ] }[,...]
) ]
AS SELECT { * | { MATCH_COLUMNS('pattern') | expression [ [AS] alias ] }[,...] }
  FROM [[database.]schema.]table [ [AS] alias]
  [ ORDER BY column-expr[,...] ]
  [ segmentation-spec ]
  [ KSAFE [ k-num ]
  [ ON PARTITION RANGE BETWEEN min-val AND max-val ] ]
```

Parameters

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

[*database* .] *schema*

Specifies the schema for this projection and its anchor table, where *schema* must be the same for both. If you specify a database, it must be the current database.

projection

Identifies the projection to create, where *projection* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

projection-column

The name of a projection column. The list of projection columns must match the SELECT list columns and expressions in number, type, and sequence.
If projection column names are omitted, Vertica uses the anchor table column names specified in the SELECT list.

grouped-clause

See GROUPED clause .

ENCODING *encoding-type*

The column encoding type , by default set to AUTO.

ACCESSRANK *integer*

Overrides the default access rank for a column. Use this parameter to increase or decrease the speed at which Vertica accesses a column. For more information, see Overriding Default Column Ranking .

AS SELECT

Specifies the columns or column expressions to select from the specified table:

- * (asterisk)
  Lists all columns in the queried tables.
- MATCH_COLUMNS (' *pattern* ')
  Returns the names of all columns in the queried anchor table that match *pattern* .
- *expression* [[AS] *alias*]
  Resolves to column data from the queried anchor table.
  You can optionally assign an alias to each column expression and reference that alias elsewhere in the SELECT statement—for example, in the ORDER BY or segmentation clause.

> **Note**
> If you specify projection column names, the two lists of projection columns and table columns/expressions must exactly match in number and order.

ORDER BY

Specifies columns from the SELECT list on which to sort the projection. The ORDER BY clause can only be set to ASC (the default). Vertica always stores projection data in ascending sort order.

If you order by a column with a collection data type (ARRAY or SET), queries that use that column in an ORDER BY clause perform the sort again. This is because projections and queries perform the ordering differently.
If you omit the ORDER BY clause, Vertica uses the SELECT list to sort the projection.

segmentation-spec

Specifies how to distribute projection data with one of the following clauses:

- *hash-segmentation-clause* : Specifies to segment projection data evenly and distribute across cluster nodes:
  ```
  SEGMENTED BY expression ALL NODES [ OFFSET offset ]
  ```
- *unsegmented-clause* : Specifies to create an unsegmented projection:
  ```
  UNSEGMENTED ALL NODES
  ```

If the anchor table and projection both omit specifying segmentation, the projection is defined with a hash segmentation clause that includes all columns in the SELECT list , as follows:

```
SEGMENTED BY HASH(column-expr[,...]) ALL NODES OFFSET 0;
```

> **Tip**
> Vertica recommends segmenting large tables.

KSAFE [*k-num*]

Specifies K-safety for the projection, where *k-num* must be equal to or greater than database K-safety. Vertica ignores this parameter if set for unsegmented projections. If you omit *k-num* , Vertica uses database K-safety.
For general information, see K-safety in an Enterprise Mode database .

ON PARTITION RANGE

Specifies to limit data of this projection to a range of partition keys, specified as follows:

```
ON PARTITION RANGE BETWEEN min-range-value AND max-range-value
```

where the following requirements apply to *min-range-value* and ≤ *max-range-value* :

- *min-range-value* must be ≤ *max-range-value*
- They must resolve to a data type that is compatible with the table partition expression.
- They can be:
  - String literals—for example, 2021-07-31
  - Expressions with stable or immutable functions, for example:
    ```
    date_trunc('month', now()::timestamp - interval'1 month')
    ```

max-range-value can be set to NULL, to specify that the partition range has no upper bound.

min-range-value can be set to NULL, to specify that the partition range has no lower bound.

If both partition range projection *min-range-value* and *max-range-value* are set to NULL, it will drop the projection endpoints, becoming a regular projection.

For other requirements and usage details, see Partition range projections .

Privileges

Non-superusers:

- Anchor table owner
- CREATE privilege on the schema

Examples

See:

- Segmented projections
- Unsegmented projections
- Partition range projections

Top-k projection

Stores the top *k* rows from partitions of selected rows. For details, see Top-k projections .

Syntax

```
CREATE PROJECTION [ IF NOT EXISTS ] [[database.]schema.]projection
[ (
  { projection-column | grouped-clause
  [ ENCODING encoding-type ]
  [ ACCESSRANK integer ] }[,...]
) ]
AS SELECT { table-column | expr-with-table-columns }[,...] FROM [[database.]schema.]table [ [AS] alias]
  LIMIT num-rows OVER ( window-partition-clause [window-order-clause] )
  [ KSAFE [ k-num ] ]
```

Parameters

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

[*database* .] *schema*

Specifies the schema for this projection and its anchor table, where *schema* must be the same for both. If you specify a database, it must be the current database.

projection

Identifies the projection to create, where *projection* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

projection-column

The name of a projection column. The list of projection columns must match the SELECT list columns and expressions in number, type, and sequence.
If projection column names are omitted, Vertica uses the anchor table column names specified in the SELECT list.

grouped-clause

See GROUPED clause .

ENCODING *encoding-type*

The column encoding type , by default set to AUTO.

ACCESSRANK *integer*

Overrides the default access rank for a column. Use this parameter to increase or decrease the speed at which Vertica accesses a column. For more information, see Overriding Default Column Ranking .

AS SELECT

Specifies the table data to query:

```
{table-column | expr-with-table-columns } [ [AS] alias] }[,...]
```

You can optionally assign an alias to each column expression and reference that alias elsewhere in the SELECT statement.

> **Note**
> If you specify projection column names, the two lists of projection columns and table columns/expressions must exactly match in number and order.

AS SELECT

Specifies the table data to query:

```
{table-column | expr-with-table-columns } [ [AS] alias] }[,...]
```

You can optionally assign an alias to each column expression and reference that alias elsewhere in the SELECT statement.

> **Note**
> If you specify projection column names, the two lists of projection columns and table columns/expressions must exactly match in number and order.

LIMIT ***num-rows***

The number of rows to return from the specified partition.

window-partition-clause

Specifies window partitioning by one or more comma-delimited column expressions from the SELECT list. The first partition expression must be the first SELECT list item, the second partition expression the second SELECT list item, and so on.

window-order-clause

Specifies the order in which the top *k* rows are returned, by default in ascending (ASC) order. All column expressions must be from the SELECT list, where the first window order expression must be the first SELECT list item not specified in the window partition clause.

Top-K projections support ORDER BY NULLS FIRST/LAST .

Privileges

Non-superusers:

- Anchor table owner
- CREATE privilege on the schema

Requirements and restrictions

Vertica does not regard Top-K projections as superprojections , even those that include all table columns. For other requirements and restrictions, see Creating top-k projections .

Examples

See Top-k projection examples .

UDTF projection

Stores newly-loaded data after it is transformed and/or aggregated by user-defined transformation functions (UDTFs). For details and examples, see Pre-aggregating UDTF results .

Important

Currently, projections can only reference UDTFs developed in C++ .

Syntax

```
CREATE PROJECTION [ IF NOT EXISTS ] [[database.]schema.]projection
[ (
  { projection-column | grouped-clause
  [ ENCODING encoding-type ]
  [ ACCESSRANK integer ]  }[,...]
) ]
AS { [batch-query](#UDTFBatchQuery) FROM { prepass-query sq-ref | table [[AS] alias] }
    | prepass-query }
```

```
batch-query
SELECT { table-column | expr-with-table-columns }[,...], batch-udtf(batch-args)
  OVER (PARTITION BATCH BY partition-column-expr[,...])
  [ AS (batch-output-columns) ]
```

```
prepass-query
SELECT { table-column | expr-with-table-columns }[,...], prepass-udtf(prepass-args)
  OVER (PARTITION PREPASS BY partition-column-expr[,...])
  [ AS (prepass-output-columns) ] FROM table
```

Parameters

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

[*database* .] *schema*

Specifies the schema for this projection and its anchor table, where *schema* must be the same for both. If you specify a database, it must be the current database.

projection

Identifies the projection to create, where *projection* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

projection-column

The name of a projection column. The list of projection columns must match the SELECT list columns and expressions in number, type, and sequence.
If projection column names are omitted, Vertica uses the anchor table column names specified in the SELECT list.

grouped-clause

See GROUPED clause .

ENCODING *encoding-type*

The column encoding type , by default set to AUTO.

ACCESSRANK *integer*

Overrides the default access rank for a column. Use this parameter to increase or decrease the speed at which Vertica accesses a column. For more information, see Overriding Default Column Ranking .

AS SELECT

Specifies the table data to query:

```
{table-column | expr-with-table-columns } [ [AS] alias] }[,...]
```

You can optionally assign an alias to each column expression and reference that alias elsewhere in the SELECT statement.

> **Note**
> If you specify projection column names, the two lists of projection columns and table columns/expressions must exactly match in number and order.

***batch-udtf* (*batch-args*)**

The batch UDTF to invoke each time the following events occur:

- Tuple mover mergeout
- Queries on the projection
- If invoked singly, on data load operations

Important

If the projection definition includes a pre-pass subquery, *batch-args* must exactly match the pre-pass UDTF output columns, in name and order.

***prepass-udtf* (*prepass-args*)**

The pre-pass UDTF to invoke on each load operation such as COPY or INSERT.

If specified in a subquery, the pre-pass UDTF returns transformed data to the batch query for further processing. Otherwise, the pre-pass query results are added to projection data storage.

OVER (PARTITION { BATCH | PREPASS } BY *partition-column-expr* [,...]

Specifies the UDTF type and how to partition the data it returns:

- BATCH identifies the UDTF as a batch UDTF.
- PREPASS identifies the UDTF as a pre-pass UDTF.

In both cases, the OVER clause specifies partitioning with one or more column expressions from the SELECT list. The first *partition-column-expr* is the first column expression in the SELECT list, the second *partition-column-expr* is the second column expression in the SELECT list, and so on.

> **Note**
> The projection is implicitly segmented and ordered on PARTITION BY columns.

AS (*batch-output-columns*) AS (*prepass-output-columns*)

Optionally names columns that are returned by the UDTF.

If a pre-pass subquery omits this clause, the outer batch query UDTF arguments (*batch-args*) must reference the column names as they are defined in the pre-pass UDTF.

***table* [[AS] *alias*]**

Specifies the projection's anchor table, optionally qualified by an alias.

sq-results

Subquery result set that is returned to the outer batch UDTF.

Privileges

Non-superusers:

- Anchor table owner
- CREATE privilege on the schema
- EXECUTE privileges on all UDTFs that are referenced by the projection

Examples

See Pre-aggregating UDTF results.

Unsegmented clause

Specifies to distribute identical copies of table or projection data on all nodes across the cluster. Use this clause to facilitate distributed query execution on tables and projections that are too small to benefit from segmentation.

Vertica uses the same name to identify all instances of an unsegmented projection. For more information about projection name conventions, see Projection naming.

Syntax

```
UNSEGMENTED ALL NODES
```

Examples

This example creates an unsegmented projection for table store.store_dimension :

```
=> CREATE PROJECTION store.store_dimension_proj (storekey, name, city, state)
          AS SELECT store_key, store_name, store_city, store_state
          FROM store.store_dimension
          UNSEGMENTED ALL NODES;
CREATE PROJECTION

=>  SELECT anchor_table_name anchor_table, projection_name, node_name
    FROM PROJECTIONS WHERE projection_basename='store_dimension_proj';
 anchor_table  |  projection_name   |    node_name
---------------+-------------------+-----------------
 store_dimension | store_dimension_proj | v_vmart_node0001
 store_dimension | store_dimension_proj | v_vmart_node0002
 store_dimension | store_dimension_proj | v_vmart_node0003
(3 rows)
```

CREATE RESOURCE POOL

Creates a user-defined resource pool.

Syntax

```
CREATE RESOURCE POOL pool-name [ FOR subcluster ] [ parameter-name setting ]...
```

Arguments

pool-name

Name of the resource pool. If you specify a resource pool name with uppercase letters, Vertica converts them to lowercase letters. If you specify a resource pool name with uppercase letters, Vertica converts them to lowercase letters.

FOR *subcluster*

Eon Mode only, the subcluster to associate with this resource pool, where *subcluster* is one of the following:

- SUBCLUSTER *subcluster-name* : Resource pool for an existing subcluster. You cannot be connected to this subcluster, otherwise Vertica returns an error.
- CURRENT SUBCLUSTER : Resource pool for the subcluster that you are connected to.

If omitted, the resource pool is created globally. Attempts to create a global resource pool with the same name as a subcluster-specific resource pool return an error.

parameter-name setting

A resource pool parameter and its initial value. If you omit this argument, Vertica sets this resource pool's parameters to their default values (see Parameters).

Parameters

> **Note**
> Default values specified here pertain only to user-defined resource pools. For built-in pool default values, see Built-in resource pools configuration , or query system table RESOURCE_POOL_DEFAULTS .

CASCADE TO

Secondary resource pool for executing queries that exceed the RUNTIMECAP setting of their assigned resource pool:

```
CASCADE TO secondary-pool
```

CPUAFFINITYMODE

Specifies whether the resource pool has exclusive or shared use of the CPUs specified in CPUAFFINITYSET :

```
CPUAFFINITYMODE { SHARED | EXCLUSIVE | ANY }
```

- SHARED : Queries that run in this resource pool share its CPUAFFINITYSET CPUs with other Vertica resource pools.
- EXCLUSIVE : Dedicates CPUAFFINITYSET CPUs to this resource pool only, and excludes other Vertica resource pools. If CPUAFFINITYSET is set as a percentage, then that percentage of CPU resources available to Vertica is assigned solely for this resource pool.
- ANY : Queries in this resource pool can run on any CPU, invalid if CPUAFFINITYSET designates CPU resources.

Default : ANY

CPUAFFINITYSET

CPUs available to this resource pool. All cluster nodes must have the same number of CPUs. The CPU resources assigned to this set are unavailable to general resource pools.

```
CPUAFFINITYSET {
 'cpu-index[,...]'
| 'cpu-indexi-cpu-indexn'
| 'integer%'
| NONE
}
```

- *cpu-index* [,...] : Dedicates one or more comma-delimited CPUs to this resource pool.
- *cpu-indexi-cpu-indexn* : Dedicates a range of contiguous CPU indexes *i* through *n* to this resource pool.
- *integer* % : Percentage of all available CPUs to use for this resource pool. Vertica rounds this percentage down to include whole CPU units.
- NONE (empty string): No affinity set is assigned to this resource pool. Queries associated with this pool are executed on any CPU.

Default : NONE

Important
CPUAFFINITYSET and CPUAFFINITYMODE must be set together in the same statement.

EXECUTIONPARALLELISM

Number of threads used to process any single query issued in this resource pool.

```
EXECUTIONPARALLELISM { limit | AUTO }
```

- *limit* : An integer value between 1 and the number of cores. Setting this parameter to a reduced value increases throughput of short queries issued in the resource pool, especially if queries are executed concurrently.
- AUTO or 0 : Vertica calculates the setting from the number of cores, available memory, and amount of data in the system. Unless memory is limited, or the amount of data is very small, Vertica sets this parameter to the number of cores on the node.

Default : AUTO

MAXCONCURRENCY

Maximum number of concurrent execution slots available to the resource pool across the cluster:

```
MAXCONCURRENCY { integer | NONE }
```

NONE (empty string): Unlimited number of concurrent execution slots.

Default : NONE

MAXMEMORYSIZE

Maximum size per node the resource pool can grow by borrowing memory from the GENERAL pool:

```
MAXMEMORYSIZE {
 'integer%'
 |'integer{K|M|G|T}'
 NONE
}
```

- *integer* % : Percentage of total memory
- *integer* {K|M|G|T} : Amount of memory in kilobytes, megabytes, gigabytes, or terabytes
- NONE (empty string): Unlimited, resource pool can borrow any amount of available memory from the GENERAL pool.

Default : NONE

MAXQUERYMEMORYSIZE

Maximum amount of memory this resource pool can allocate at runtime to process a query. If the query requires more memory than this setting, Vertica stops execution and returns an error.
Set this parameter as follows:

```
MAXQUERYMEMORYSIZE {
 'integer%'
| 'integer{K|M|G|T}'
| NONE
}
```

- *integer* % : Percentage of MAXMEMORYSIZE for this resource pool.
- *integer* {K|M|G|T} : Amount of memory in kilobytes, megabytes, gigabytes, or terabytes, up to the value of MAXMEMORYSIZE .
- NONE (empty string): Unlimited; resource pool can borrow any amount of available memory from the GENERAL pool, within the limits set by MAXMEMORYSIZE .

Default : NONE

MEMORYSIZE

Total per-node memory available to the Vertica resource manager that is allocated to this resource pool:

```
MEMORYSIZE {
 'integer%'
| 'integer{K|M|G|T}'
}
```

- *integer* % : Percentage of total memory
- *integer* {K|M|G|T} : Amount of memory in kilobytes, megabytes, gigabytes, or terabytes

Default: 0%. No memory allocated, the resource pool borrows memory from the GENERAL pool.

PLANNEDCONCURRENCY

Preferred number of queries to execute concurrently in the resource pool. This setting applies to the entire cluster:

```
PLANNEDCONCURRENCY { num-queries | AUTO }
```

- *num-queries* : Integer value ≥ 1, the preferred number of queries to execute concurrently in the resource pool. When possible, query resource budgets are limited to allow this level of concurrent execution.
- AUTO : Value is calculated automatically at query runtime. Vertica sets this parameter to the lower of these two calculations, but never less than 4:
  - Number of logical cores
  - Memory divided by 2GB

  If the number of logical cores on each node is different, AUTO is calculated differently for each node. Distributed queries run like the minimal effective planned concurrency. Single node queries run with the planned concurrency of the initiator.

Default : AUTO

> **Tip**
> Change this parameter only after evaluating performance over a period of time.

PRIORITY

Priority of queries in this resource pool when they compete for resources in the GENERAL pool:

```
PRIORITY { integer | HOLD }
```

- *integer* : Negative or positive integer value, where higher numbers denote higher priority:
  - User-defined resource pool: -100 to 100
  - Built-in resource pools SYSQUERY , RECOVERY , and TM : -110 to 110
- HOLD : Sets priority to -999 . Queries in this resource pool are queued until QUEUETIMEOUT is reached.

Default : 0

QUEUETIMEOUT

Maximum time a request can wait for pool resources before it is rejected, not more than one year:

```
QUEUETIMEOUT { integer | 'interval' | 'NONE' }
```

- *integer* : Maximum wait time in seconds
- *[interval](/sql-reference/language-elements/literals/datetime-literals/interval-literal.html)* : Maximum wait time expressed in the following format:
  ```
  num year num months num [days] HH:MM:SS.ms
  ```
- NONE (empty string): No maximum wait time, request can be queued indefinitely, up to one year.

If the value that you specify resolves to more than one year, Vertica returns with a warning and sets the parameter to 365 days:

```
=> ALTER RESOURCE POOL user_0 QUEUETIMEOUT '11 months 50 days 08:32';
WARNING 5693:  Using 1 year for QUEUETIMEOUT
ALTER RESOURCE POOL
=> SELECT QUEUETIMEOUT FROM resource_pools WHERE name = 'user_0';
 QUEUETIMEOUT
--------------
 365
(1 row)
```

Default : **00:05 (5 minutes)**

RUNTIMECAP

Maximum execution time allowed to queries in this resource pool, not more than one year, otherwise Vertica returns with an error. If a query exceeds this setting, it tries to cascade to a secondary pool:

```
RUNTIMECAP { 'interval' | NONE }
```

- *interval* : Maximum wait time expressed in the following format:
  ```
  num year num month num [day] HH:MM:SS.ms
  ```
- NONE (empty string): No maximum wait time, request can be queued indefinitely, up to one year.

If the user or session also has a RUNTIMECAP , the shorter limit applies.

RUNTIMEPRIORITY

Determines how the resource manager should prioritize dedication of run-time resources (CPU, I/O bandwidth) to queries already running in this resource pool:

```
RUNTIMEPRIORITY { HIGH | MEDIUM | LOW }
```

Default : MEDIUM

RUNTIMEPRIORITYTHRESHOLD

Maximum time (in seconds) in which query processing must complete before the resource manager assigns to it the resource pool's RUNTIMEPRIORITY . All queries begin execution with a priority of HIGH.

```
RUNTIMEPRIORITYTHRESHOLD seconds
```

Default : 2

SINGLEINITIATOR

Set to false for backward compatibility. Do not change this setting.

Privileges

Superuser

Examples

This example shows how to create a resource pool with MEMORYSIZE of 1800 MB.

```
=> CREATE RESOURCE POOL ceo_pool MEMORYSIZE '1800M' PRIORITY 10;
CREATE RESOURCE POOL
```

Assuming the CEO report user already exists, associate this user with the preceding resource pool using ALTER USER statement.

```
=> GRANT USAGE ON RESOURCE POOL ceo_pool to ceo_user;
GRANT PRIVILEGE
=> ALTER USER ceo_user RESOURCE POOL ceo_pool;
ALTER USER
```

Issue the following command to confirm that the ceo_user is associated with the ceo_pool:

```
=> SELECT * FROM users WHERE user_name ='ceo_user';
-[ RECORD 1 ]-----+----------------------------------------------------
user_id           | 45035996273733402
user_name         | ceo_user
is_super_user     | f
profile_name      | default
is_locked         | f
lock_time         |
resource_pool     | ceo_pool
memory_cap_kb     | unlimited
temp_space_cap_kb | unlimited
run_time_cap      | unlimited
all_roles         |
default_roles     |
search_path       | "$user", public, v_catalog, v_monitor, v_internal
```

This exampleshows how to create and designate secondary resource pools.

```
=> CREATE RESOURCE POOL rp3 RUNTIMECAP '5 minutes';
=> CREATE RESOURCE POOL rp2 RUNTIMECAP '3 minutes' CASCADE TO rp3;
=> CREATE RESOURCE POOL rp1 RUNTIMECAP '1 minute' CASCADE TO rp2;
=> SET SESSION RESOURCE_POOL = rp1;
```

This Eon Mode example confirms the current subcluster name, then creates a resource pool for the current subcluster:

```
=> SELECT CURRENT_SUBCLUSTER_NAME();
 CURRENT_SUBCLUSTER_NAME
-------------------------
 analytics_1
(1 row)

=> CREATE RESOURCE POOL dashboard FOR SUBCLUSTER analytics_1;
CREATE RESOURCE POOL
```

See also

- ALTER RESOURCE POOL
- CREATE USER
- DROP RESOURCE POOL
- SET SESSION RESOURCE_POOL
- SET SESSION MEMORYCAP
- Managing workloads

In this section

- Built-in pools
- Built-in resource pools configuration

Built-in pools

Vertica is preconfigured with built-in pools for various system tasks:

- GENERAL
- BLOBDATA
- DBD
- JVM
- METADATA
- RECOVERY

- REFRESH
- SYSQUERY
- TM

For details on resource pool settings, see ALTER RESOURCE POOL .

GENERAL

Catch-all pool used to answer requests that have no specific resource pool associated with them. Any memory left over after memory has been allocated to all other pools is automatically allocated to the GENERAL pool. The MEMORYSIZE parameter of the GENERAL pool is undefined (variable), however, the GENERAL pool must be at least 1GB in size and cannot be smaller than 25% of the memory in the system.

The MAXMEMORYSIZE parameter of the GENERAL pool has special meaning; when set as a % value it represents the percent of total physical RAM on the machine that the Resource manager can use for queries. By default, it is set to 95%. MAXMEMORYSIZE governs the total amount of RAM that the Resource Manager can use for queries, regardless of whether it is set to a percent or to a specific value (for example, '10GB').

User-defined pools can borrow memory from the GENERAL pool to satisfy requests that need extra memory until the MAXMEMORYSIZE parameter of that pool is reached. If the pool is configured to have MEMORYSIZE equal to MAXMEMORYSIZE, it cannot borrow any memory from the GENERAL pool. When multiple pools request memory from the GENERAL pool, they are granted access to general pool memory according to their priority setting. In this manner, the GENERAL pool provides some elasticity to account for point-in-time deviations from normal usage of individual resource pools.

Vertica recommends reducing the GENERAL pool MAXMEMORYSIZE if your catalog uses over 5 percent of overall memory. You can calculate what percentage of GENERAL pool memory the catalog uses as follows:

```
=> WITH memory_use_metadata AS (SELECT node_name, memory_size_kb FROM resource_pool_status WHERE pool_name='metadata'),
      memory_use_general  AS (SELECT node_name, memory_size_kb FROM resource_pool_status WHERE pool_name='general')
  SELECT m.node_name, ((m.memory_size_kb/g.memory_size_kb) * 100)::NUMERIC(4,2) pct_catalog_usage
  FROM memory_use_metadata m JOIN memory_use_general g ON m.node_name = g.node_name;
   node_name     | pct_catalog_usage
-----------------+-------------------
 v_vmart_node0001 |              0.41
 v_vmart_node0002 |              0.37
 v_vmart_node0003 |              0.36
(3 rows)
```

BLOBDATA

Controls resource usage for in-memory blobs. *In-memory blobs* are objects used by a number of the machine learning SQL functions. You should adjust this pool if you plan on processing large machine learning workloads. For information about tuning the pool, see Tuning for machine learning .

If a query using the BLOBDATA pool exceeds its query planning budget, then it spills to disk. For more information about tuning your query budget, see Query budgeting .

DBD

Controls resource usage for Database Designer processing. Use of this pool is enabled by configuration parameter DBDUseOnlyDesignerResourcePool , by default set to false.

By default, QUEUETIMEOUT is set to 0 for this pool. When resources are under pressure, this setting causes the DBD to time out immediately, and not be queued to run later. Database Designer then requests the user to run the designer later, when resources are more available.

Important
Do not change QUEUETIMEOUT or any DBD resource pool parameters.

JVM

Controls Java Virtual Machine resources used by Java User Defined Extensions. When a Java UDx starts the JVM, it draws resources from the those specified in the JVM resource pool. Vertica does not reserve memory in advance for the JVM pool. When needed, the pool can expand to 10% of physical memory or 2 GB of memory, whichever is smaller. If you are buffering large amounts of data, you may need to increase the size of the JVM resource pool.

You can adjust the size of your JVM resource pool by changing its configuration settings. Unlike other resource pools, the JVM resource pool does not release resources until a session is closed.

METADATA

Tracks memory allocated for catalog data and storage data structures. This pool increases in size as Vertica metadata consumes additional resources. Memory assigned to the METADATA pool is subtracted from the GENERAL pool, enabling the Vertica resource manager to make more effective use of available resources. If the METADATA resource pool reaches 75% of the GENERAL pool, Vertica stops updating METADATA memory size and displays a

warning message in vertica.log . You can enable or disable the METADATA pool with configuration parameter EnableMetadataMemoryTracking .

If you created a "dummy" or "swap" resource pool to protect resources for use by your operating system, you can replace that pool with the METADATA pool.

Users cannot change the parameters of the METADATA resource pool.

RECOVERY

Used by queries issued when recovering another node of the database. The MAXCONCURRENCY parameter is used to determine how many concurrent recovery threads to use. You can use the PLANNEDCONCURRENCY parameter (by default, set to twice the MAXCONCURRENCY) to tune how to apportion memory to recovery queries.

See Tuning for recovery .

REFRESH

Used by queries issued by PROJECTION_REFRESHES operations. Refresh does not currently use multiple concurrent threads; thus, changes to the MAXCONCURRENCY values have no effect.

See Scenario: Tuning for Refresh .

SYSQUERY

Runs queries against all system monitoring and catalog tables . The SYSQUERY pool reserves resources for system table queries so that they are never blocked by contention for available resources.

TM

The Tuple Mover (TM) pool. You can set the MAXCONCURRENCY parameter for the TM pool to allow concurrent TM operations.

See Tuning tuple mover pool settings .

Built-in resource pools configuration

To view the current and default configuration for built-in resource pools, query the system tables RESOURCE_POOLS and RESOURCE_POOL_DEFAULTS, respectively. The sections below provide this information, and also indicate which built-in pool parameters can be modified with ALTER RESOURCE POOL :

- GENERAL
- BLOBDATA
- DBD
- JVM
- METADATA
- RECOVERY
- REFRESH
- SYSQUERY
- TM

GENERAL

Important
Changes to GENERAL resource pool parameters take effect only when the database restarts.

Parameter	Settings
MEMORYSIZE	Empty / cannot be set

MAXMEMORYSIZE	The maximum memory to use for all resource pools, one of the following: `MAXMEMORYSIZE { 'integer%' | 'integer{K|M|G|T}' }` • *integer %* : Percentage of total system RAM, must be ≥ 25% **Caution** Setting this parameter to 100% generates a warning of potential swapping. • *integer* {K|M|G|T} : Amount of memory in kilobytes, megabytes, gigabytes, or terabytes, must be ≥ 1GB For example, if your node has 64GB of memory, setting MAXMEMORYSIZE to 50% allocates half of available memory. Thus, the maximum amount of memory available to all resource pools is 32GB. **Default:** 95%
MAXQUERYMEMORYSIZE	The maximum amount of memory allocated by this pool to process any query: `MAXQUERYMEMORYSIZE { 'integer%' | 'integer{K|M|G|T}' }` • *integer %* : Percentage of MAXMEMORYSIZE for this pool. • *integer* {K|M|G|T} : Amount of memory in kilobytes, megabytes, gigabytes, or terabytes
EXECUTIONPARALLELISM	**Default:** AUTO
PRIORITY	**Default:** 0
RUNTIMEPRIORITY	**Default:** Medium
RUNTIMEPRIORITYTHRESHOLD	**Default:** 2
QUEUETIMEOUT	**Default:** 00:05 (minutes)
RUNTIMECAP	Prevents runaway queries by setting the maximum time a query in the pool can execute. If a query exceeds this setting, it tries to cascade to a secondary pool: RUNTIMECAP { ' *interval* ' | NONE } • *interval* : An interval of 1 minute or 100 seconds; should not exceed one year. • NONE (default): No time limit on queries running in this pool.

PLANNEDCONCURRENCY	The number of concurrent queries you expect to run against the resource pool, an integer ≥ 4. If set to AUTO (default), Vertica automatically sets PLANNEDCONCURRENCY at query runtime, choosing the lower of these two values: • Number of cores • Memory/2GB Important In systems with a large number of cores, the default AUTO setting of PLANNEDCONCURRENCY is liable to be too low. In this case, set the parameter to the actual number of cores: ALTER RESOURCE POOL general PLANNEDCONCURRENCY *#cores* ; **Default:** AUTO
MAXCONCURRENCY	**Caution** Must be set ≥ 1, otherwise Vertica generates a warning that system queries might be unable to execute. **Default:** Empty
SINGLEINITIATOR	Important Included for backwards compatibility. Do not change. **Default:** False
CPUAFFINITYSET	**Default:** Empty
CPUAFFINITYMODE	**Default:** ANY
CASCADETO	**Default:** Empty

BLOBDATA

Parameter	**Default Setting**
MEMORYSIZE	0%
MAXMEMORYSIZE	10
MAXQUERYMEMORYSIZE	Empty / cannot be set
EXECUTIONPARALLELISM	
PRIORITY	
RUNTIMEPRIORITY	
RUNTIMEPRIORITYTHRESHOLD	
QUEUETIMEOUT	
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	AUTO

MAXCONCURRENCY	Empty / cannot be set
SINGLEINITIATOR	
CPUAFFINITYSET	
CPUAFFINITYMODE	ANY / cannot be set
CASCADETO	Empty / cannot be set

DBD

Parameter	**Default Setting**
MEMORYSIZE	0%
MAXMEMORYSIZE	Unlimited
MAXQUERYMEMORYSIZE	Empty / cannot be set
EXECUTIONPARALLELISM	AUTO
PRIORITY	0
RUNTIMEPRIORITY	MEDIUM
RUNTIMEPRIORITYTHRESHOLD	0
QUEUETIMEOUT	0
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	AUTO
MAXCONCURRENCY	NONE
SINGLEINITIATOR	True Important Included for backwards compatibility. Do not change.
CPUAFFINITYSET	Empty / cannot be set
CPUAFFINITYMODE	ANY / cannot be set
CASCADETO	Empty / cannot be set

JVM

Parameter	**Default Setting**
MEMORYSIZE	0%
MAXMEMORYSIZE	10% of memory or 2 GB, whichever is smaller
MAXQUERYMEMORYSIZE	Empty / cannot be set
EXECUTIONPARALLELISM	AUTO

PRIORITY	0
RUNTIMEPRIORITY	MEDIUM
RUNTIMEPRIORITYTHRESHOLD	2
QUEUETIMEOUT	00:05 (minutes)
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	AUTO
MAXCONCURRENCY	Empty / cannot be set
SINGLEINITIATOR	FALSE Important Included for backwards compatibility. Do not change.
CPUAFFINITYSET	Empty / cannot be set
CPUAFFINITYMODE	ANY / cannot be set
CASCADETO	Empty / cannot be set

METADATA

Parameter	**Default Setting**
MEMORYSIZE	0%
MAXMEMORYSIZE	Unlimited
MAXQUERYMEMORYSIZE	Empty / cannot be set
EXECUTIONPARALLELISM	AUTO
PRIORITY	108
RUNTIMEPRIORITY	HIGH
RUNTIMEPRIORITYTHRESHOLD	0
QUEUETIMEOUT	0
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	AUTO
MAXCONCURRENCY	0
SINGLEINITIATOR	FALSE. Important Included for backwards compatibility. Do not change.
CPUAFFINITYSET	Empty / cannot be set

CPUAFFINITYMODE	ANY / cannot be set
CASCADETO	Empty / cannot be set

RECOVERY

Parameter	**Default Setting**
MEMORYSIZE	0%
MAXMEMORYSIZE	Maximum size per node the resource pool can grow by borrowing memory from the GENERAL pool: `MAXMEMORYSIZE { 'integer%' |'integer{K|M|G|T}' NONE }` • *integer* % : Percentage of total memory • *integer* {K|M|G|T} : Amount of memory in kilobytes, megabytes, gigabytes, or terabytes • NONE (empty string): Unlimited, resource pool can borrow any amount of available memory from the GENERAL pool. **Default** : NONE **Caution** Setting must resolve to ≥ 25%. Otherwise, Vertica generates a warning that system queries might be unable to execute.
MAXQUERYMEMORYSIZE	Empty / cannot be set
EXECUTIONPARALLELISM	AUTO
PRIORITY	One of the following: • Enterprise Mode: 107 • Eon Mode: 110 **Caution** Change these settings only under guidance from Vertica technical support.
RUNTIMEPRIORITY	MEDIUM
RUNTIMEPRIORITYTHRESHOLD	60
QUEUETIMEOUT	00:05 (minutes)
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	AUTO

MAXCONCURRENCY	By default, set as follows: (*numberCores* / 2) + 1 Thus, given a system with four cores, MAXCONCURRENCY has a default setting of 3. **Note** 0 or NONE (unlimited) are invalid settings.
SINGLEINITIATOR	True. Important Included for backwards compatibility. Do not change.
CPUAFFINITYSET	Empty / cannot be set
CPUAFFINITYMODE	ANY / cannot be set
CASCADETO	Empty / cannot be set

REFRESH

Parameter	**Default Setting**
MEMORYSIZE	0%
MAXMEMORYSIZE	NONE (unlimited)
MAXQUERYMEMORYSIZE	Empty / cannot be set
EXECUTIONPARALLELISM	AUTO
PRIORITY	-10
RUNTIMEPRIORITY	MEDIUM
RUNTIMEPRIORITYTHRESHOLD	60
QUEUETIMEOUT	00:05 (minutes)
RUNTIMECAP	NONE (unlimited)
PLANNEDCONCURRENCY	AUTO (4)
MAXCONCURRENCY	3 This parameter must be set ≥ 1.
SINGLEINITIATOR	True. Important Included for backwards compatibility. Do not change.
CPUAFFINITYSET	Empty / cannot be set

CPUAFFINITYMODE	ANY / cannot be set
CASCADETO	Empty / cannot be set

SYSQUERY

Parameter	Default Setting
MEMORYSIZE	1G **Caution** Setting must resolve to ≥ 20M, otherwise Vertica generates a warning that system queries might be unable to execute, and diagnosing problems might be difficult.
MAXMEMORYSIZE	Empty (unlimited)
MAXQUERYMEMORYSIZE	Empty / cannot be set
EXECUTIONPARALLELISM	AUTO
PRIORITY	110
RUNTIMEPRIORITY	HIGH
RUNTIMEPRIORITYTHRESHOLD	0
QUEUETIMEOUT	00:05 (minutes)
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	AUTO
MAXCONCURRENCY	Empty **Caution** Must be set ≥ 1, otherwise Vertica generates a warning that system queries might be unable to execute.
SINGLEINITIATOR	False. Important Included for backwards compatibility. Do not change.
CPUAFFINITYSET	Empty / cannot be set
CPUAFFINITYMODE	
CASCADETO	

TM

Parameter	Default Setting

MEMORYSIZE	5% (of the GENERAL pool's MAXMEMORYSIZE setting) + 2GB Important You can estimate the optimal amount of RAM for the TM resource pool as follows: *GbRAM* / (6 * *#table-cols*) > 10 where *#table-cols* is the number of columns in the largest database table. For example, given a 100-column table, MEMORYSIZE needs least 6GB of RAM: 6144MB / (6 * 100) = 10.24
MAXMEMORYSIZE	Unlimited
MAXQUERYMEMORYSIZE	Empty / cannot be set
EXECUTIONPARALLELISM	AUTO
PRIORITY	105
RUNTIMEPRIORITY	MEDIUM
RUNTIMEPRIORITYTHRESHOLD	60
QUEUETIMEOUT	00:05 (minutes)
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	7
MAXCONCURRENCY	Sets across all nodes the maximum number of concurrent execution slots available to TM pool. In databases created in Vertica releases ≥9.3, the default value is 7. In databases created in earlier versions, the default is 3.This setting specifies the maximum number of merges that can occur simultaneously on multiple threads. **Note** 0 or NONE (unlimited) are invalid settings.
SINGLEINITIATOR	True Important Included for backwards compatibility. Do not change.
CPUAFFINITYSET	Empty / cannot be set
CPUAFFINITYMODE	ANY / cannot be set
CASCADETO	Empty / cannot be set

CREATE ROLE

Creates a role . After creating a role, use GRANT statements to specify role permissions.

Syntax

```
CREATE ROLE role
```

Parameters

role

The name for the new role, where *role* conforms to conventions described in Identifiers .

Privileges

Superuser

Examples

This example shows to create an empty role called roleA.

```
=> CREATE ROLE roleA;
CREATE ROLE
```

See also

- ALTER ROLE
- DROP ROLE

CREATE ROUTING RULE

Creates a load balancing routing rule that directs incoming client connections from an IP address range to a group of Vertica nodes. This group of Vertica nodes is defined by a load balance group. Once you create a routing rule, any client connection originating from the rule's IP address range is redirected to one of the nodes in the load balance group if the client opts into load balancing.

Syntax

```
CREATE ROUTING RULE rule_name ROUTE 'address_range' TO group_name
```

Arguments

rule_name

A name for the routing rule.

address_range

An IPv4 or IPv6 address range in CIDR format. Sets the address range of client connections that this rule applies to.

group_name

The name of the load balance group to handle the client connections from the address range. You create this group using the CREATE LOAD BALANCE GROUP statement.

Privileges

Superuser

Examples

The following example creates a routing rule that routes all client connections from 192.168.1.0 to 192.168.1.255 to a load balance group named internal_clients:

```
=> CREATE ROUTING RULE internal_clients ROUTE '192.168.1.0/24' TO internal_clients;
CREATE ROUTING RULE
```

See also

CREATE SCHEDULE

Creates a schedule. For details, see Scheduled execution .

To view existing schedules, query USER_SCHEDULES .

Syntax

```
CREATE SCHEDULE [ IF NOT EXISTS ] [[database.]schema.]schedule
  { USING CRON 'cron_expression' | USING DATETIMES timestamp_list }
```

Parameters

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

schedule

The name of the schedule.

cron_expression

A cron expression. You should use this for recurring tasks. Steps and ranges are not yet supported.

timestamp_list

A comma-separated list of timestamps. You should use this to schedule non-recurring events at arbitrary times.

Privileges

Superuser

Examples

To create an schedule for January 1 at 12:00 PM:

```
=> CREATE SCHEDULE annual_schedule USING CRON '0 12 1 1 *';
```

To create a schedule for the first of every month:

```
=> CREATE SCHEDULE monthly_schedule USING CRON '0 0 1 * *';
```

To create a schedule for Sunday at 12:00 AM:

```
=> CREATE SCHEDULE weekly_schedule USING CRON '0 0 * * 0';
```

To create a schedule for every day at 1:00 PM:

```
=> CREATE SCHEDULE daily_schedule USING CRON '0 13 * * *';
```

To create a schedule for October 2, 2022 and November 2, 2022:

```
=> CREATE SCHEDULE oct_nov_2 USING DATETIMES('2022-10-02 12:00:00', '2022-11-02 12:00:00');
```

CREATE SCHEMA

Defines a schema.

Syntax

```
CREATE SCHEMA [ IF NOT EXISTS ] [database.]schema
  [ AUTHORIZATION username]
  [ DEFAULT { INCLUDE | EXCLUDE } [ SCHEMA ] PRIVILEGES ]
  [ DISK_QUOTA quota ]
```

Parameters

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

database

Name of the database in which to create the schema. If specified, it must be the current database.

schema

Name of the schema to create, with the following requirements:

- Must be unique among all other schema names in the database.
- Must comply with keyword restrictions and rules for Identifiers .
- Cannot begin with v_ ; this prefix is reserved for Vertica system tables .

AUTHORIZATION *username*

Valid only for superusers, assigns ownership of the schema to another user. By default, the user who creates a schema is also assigned ownership.

After you create a schema, you can reassign ownership to another user with ALTER SCHEMA .

DEFAULT {INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES

Specifies whether to enable or disable default inheritance of privileges for new tables in the specified schema:

- EXCLUDE SCHEMA PRIVILEGES (default): Disables inheritance of schema privileges.
- INCLUDE SCHEMA PRIVILEGES : Specifies to grant tables in the specified schema the same privileges granted to that schema. This option has no effect on existing tables in the schema.

If you omit INCLUDE PRIVILEGES , you must explicitly grant schema privileges on the desired tables.

For more information see Enabling schema inheritance .

DISK_QUOTA *quota*

String, an integer followed by a supported unit: K, M, G, or T. Data-load, DML, and ILM operations that increase the schema's usage beyond the set quota fail. For details, see Disk quotas .

If not specified, the schema has no quota.

Privileges

- Superuser (required to set disk quota)
- CREATE privilege for the database

Supported sub-statements

CREATE SCHEMA can include one or more sub-statements—for example, to create tables or projections within the new schema. Supported sub-statements include:

- CREATE TABLE / CREATE TEMPORARY TABLE
- GRANT statements
- CREATE PROJECTION
- CREATE SEQUENCE
- CREATE TEXT INDEX
- CREATE VIEW

CREATE SCHEMA statement and all sub-statements are treated as a single transaction. If any statement fails, Vertica rolls back the entire transaction. The owner of the new schema is assigned ownership of all objects that are created within this transaction.

For example, the following CREATE SCHEMA statement also grants privileges on the new schema, and creates a table and view of that table:

```
=> \c - Joan
You are now connected as user "Joan".
=> CREATE SCHEMA s1
    GRANT USAGE, CREATE ON SCHEMA s1 TO public
    CREATE TABLE s1.t1 (a varchar)
    CREATE VIEW s1.t1v AS SELECT * FROM s1.t1;
CREATE SCHEMA
=> \dtv s1.*
            List of tables
 Schema | Name | Kind  | Owner | Comment
--------+------+-------+-------+---------
 s1     | t1   | table | Joan  |
 s1     | t1v  | view  | Joan  |
(2 rows)
```

Examples

Create schema s1 :

```
=> CREATE SCHEMA s1;
```

Create schema s2 if it does not already exist:

```
=> CREATE SCHEMA IF NOT EXISTS s2;
```

If the schema already exists, Vertica returns a rollback message:

```
=> CREATE SCHEMA IF NOT EXISTS s2;
NOTICE 4214:  Object "s2" already exists; nothing was done
```

Create table t1 in schema s1 , then grant users Fred and Aniket access to all existing tables and all privileges on table t1 :

```
=> CREATE TABLE s1.t1 (c INT);
CREATE TABLE
=> GRANT USAGE ON SCHEMA s1 TO Fred, Aniket;
GRANT PRIVILEGE
=> GRANT ALL PRIVILEGES ON TABLE s1.t1 TO Fred, Aniket;
GRANT PRIVILEGE
```

Enable inheritance on new schema s3 so all tables created in it automatically inherit its privileges. In this case, new table s3.t2 inherits USAGE, CREATE, and SELECT privileges, which are automatically granted to all database users:

```
=> CREATE SCHEMA s3 DEFAULT INCLUDE SCHEMA PRIVILEGES;
CREATE SCHEMA

=> GRANT USAGE, CREATE, SELECT, INSERT ON SCHEMA S3 TO PUBLIC;
GRANT PRIVILEGE

=> CREATE TABLE s3.t2(i int);
WARNING 6978:  Table "t2" will include privileges from schema "s3"
CREATE TABLE
```

See also

- ALTER SCHEMA
- DROP SCHEMA

CREATE SEQUENCE

Defines a named sequence number generator object. Named sequences let you set the default values of primary key columns. Sequences guarantee uniqueness, and avoid constraint enforcement issues.

For more information about sequence types and usage, see Sequences .

Syntax

```
CREATE SEQUENCE [ IF NOT EXISTS ] [[database.]schema.]sequence
  [ INCREMENT [ BY ] integer ]
  [ MINVALUE integer | NO MINVALUE ]
  [ MAXVALUE maxvalue | NO MAXVALUE ]
  [ START [ WITH ] integer ]
  [ CACHE integer | NO CACHE ]
  [ CYCLE | NO CYCLE ]
```

Parameters

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

sequence

Name of the sequence to create, where *sequence* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

INCREMENT

Positive or negative integer that specifies how much to increment or decrement the sequence on each call to NEXTVAL , by default set to 1.

> **Note**
> Setting this parameter to *integer* guarantees that column values always increment by at least *integer* . However, column values can sometimes increment by more than *integer* unless you also set the NO CACHE parameter.

MINVALUE|NO MINVALUE

Minimum integer value of the sequence. If omitted, the following defaults apply:

- Ascending sequence: 1
- Descending sequence: -2^{63}

MAXVALUE|NO MAXVALUE

Maximum integer value of the sequence. If omitted, the following defaults apply:

- Ascending sequence: 2^{63}
- Descending sequence: -1

START

Integer start value of the sequence. The next call to NEXTVAL returns the start value. If omitted, the following defaults apply:

- Ascending sequence: MINVALUE
- Descending sequence: MAXVALUE

CACHE|NO CACHE

Whether to cache unique sequence numbers on each node for faster access. CACHE takes an integer argument as follows:

- >1 specifies how many unique sequence numbers are pre-allocated and stored in memory for faster access. Vertica sets up caching for each session, and distributes it across all nodes.

  > **Caution**
  > If sequence caching is set to a low number, nodes are liable to request a new set of cache values more frequently. While it supplies a new cache, Vertica must lock the catalog. Until Vertica releases the lock, other database activities such as table inserts are blocked, which can adversely affect overall performance.

- 0 or 1 specifies to disable caching (equivalent to NO CACHE).

By default, the sequence cache is set to 250,000.

For details, see Distributing sequences .

CYCLE|NO CYCLE

Whether the sequence wraps:

- CYCLE :
  - Incrementing sequence: On reaching MAXVALUE , wraps to MINVALUE .
  - Decrementing sequence: On reaching MINVALUE , wraps to MAXVALUE .
- NO CYCLE (default): Calls to NEXTVAL return an error after the sequence reaches its upper or lower limit.

Privileges

Non-superusers: CREATE privilege on the schema

Examples

See Creating and using named sequences .

See also

- ALTER SEQUENCE
- DROP SEQUENCE

CREATE SUBNET

Identifies the subnet to which the nodes of a Vertica database belong. Use this statement to configure import/export from a database to other Vertica clusters.

Syntax

```
CREATE SUBNET subnet-name WITH 'subnet-prefix'
```

Parameters

subnet-name

A name you assign to the subnet, where *subnet-name* conforms to conventions described in Identifiers .

subnet-prefix

The subnet prefix in either a dotted-quad number format for IPv4 addresses, or four colon-delimited four-digit hexadecimal numbers for IPv6 addresses. Refer to system table NETWORK_INTERFACES to get the prefix of all available IP networks.

You can then configure the database to use the subnet for import/export. For details, see Identify the database or nodes used for import/export .

Privileges

Superuser

Examples

```
=> CREATE SUBNET mySubnet WITH '123.4.5.6';
=> CREATE SUBNET mysubnet WITH 'fd9b:1fcc:1dc4:78d3::';
```

CREATE TABLE

Creates a table in the logical schema.

Syntax

Create with column definitions:

```
CREATE TABLE [ IF NOT EXISTS ] [[database.]schema.]table
  ( column-definition[,...] [, table-constraint [,...]] )
  [ ORDER BY column[,...] ]
  [ segmentation-spec ]
  [ KSAFE [safety] ]
  [ partition-clause]
  [ {INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES ]
  [ DISK_QUOTA quota ]
```

Create from another table:

```
CREATE TABLE [ IF NOT EXISTS ] [[database.]schema.]table
  { AS-clause | LIKE-clause }
  [ DISK_QUOTA quota ]
```

AS-clause:

```
[ ( column-name-list ) ]
[ {INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES ]
AS  [ /*+ LABEL */ ] [ AT epoch ] query [ ENCODED BY column-ref-list ] [ segmentation-spec ]
```

LIKE-clause:

```
LIKE [[database.]schema.]existing-table
  [ {INCLUDING | EXCLUDING} PROJECTIONS ]
  [ {INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES ]
```

Parameters

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Name of the table to create, which must be unique among names of all sequences, tables, projections, views, and models within the schema.

column-definition

Column name, data type , and optional constraints. A table can have up to 9800 columns. At least one column in the table must be of a scalar type or native array.

table-constraint

Table-level constraint, as opposed to column constraints.

ORDER BY *column* [,...]

Invalid for external tables, specifies columns from the SELECT list on which to sort the superprojection that is automatically created for this table. The ORDER BY clause cannot include qualifiers ASC or DESC . Vertica always stores projection data in ascending sort order.
If you omit the `ORDER BY` clause, Vertica uses the SELECT list order as the projection sort order.

segmentation-spec

Invalid for external tables, specifies how to distribute data for auto-projections of this table. Supply one of the following clauses:

- *hash-segmentation-clause* : Specifies to segment data evenly and distribute across cluster nodes. Vertica recommends segmenting large tables.
- *unsegmented-clause* : Specifies to create an unsegmented projection.

If this clause is omitted, Vertica generates auto-projections with default hash segmentation .

KSAFE [*safety*]

Invalid for external tables, specifies K-safety of auto-projections created for this table, where *k-num* must be equal to or greater than system K-safety. If you omit this option, the projection uses the system K-safety level.

partition-clause

Invalid for external tables, logically divides table data storage through a PARTITION BY clause:

```
PARTITION BY partition-expression
 [ GROUP BY group-expression ] [ ACTIVEPARTITIONCOUNT integer ]
```

column-name-list

Valid only when creating a table from a query (AS *query*), defines column names that map to the query output. If you omit this list, Vertica uses the query output column names.
This clause and the ENCODED BY clause are mutually exclusive. Column name lists are invalid for external tables.

The names in *column-name-list* and queried columns must be the same in number.

For example:

```
CREATE TABLE customer_occupations (name, profession)
  AS SELECT customer_name, occupation FROM customer_dimension;
```

{INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES

Default inheritance of schema privileges for this table:

- INCLUDE PRIVILEGES specifies that the table inherits privileges that are set on its schema. This is the default behavior if privileges inheritance is enabled for the schema.
- EXCLUDE PRIVILEGES disables inheritance of privileges from the schema.

For details, see Inherited privileges .

AS *query*

Creates and loads a table from the results of a query , specified as follows:

```
AS  [ /*+ LABEL */ ] [ AT ] epoch query
```

The query cannot include complex type columns.

ENCODED BY *column-ref-list*

A comma-delimited list of columns from the source table, where each column is qualified by one or both of the following encoding options:

- ACCESSRANK *integer* : Overrides the default access rank for a column, useful for prioritizing access to a column. See Prioritizing column access speed .
- ENCODING *encoding-type* : Specifies the type of encoding to use on the column. The default encoding type is AUTO .

This option and * column-name-list *are mutually exclusive. This option is invalid for external tables.

LIKE *existing-table*

Creates the table by replicating an existing table . You can qualify the LIKE clause with one of the following options:

- EXCLUDING PROJECTIONS (default): Do not copy projections from the source table.
- INCLUDING PROJECTIONS : Copy current projections from the source table for the new table.
- {INCLUDE|EXCLUDE} [SCHEMA] PRIVILEGES : See description above).

DISK_QUOTA *quota*

String, an integer followed by a supported unit: K, M, G, or T. Data-load, DML, and ILM operations that increase the table's usage beyond the set quota fail. For details, see Disk quotas .

If not specified, the table has no quota.

Privileges

Superuser to set disk quota.

Non-superuser:

- CREATE privileges on the table schema
- If creating a table that includes a named sequence:
  - SELECT privilege on sequence object
  - USAGE privilege on sequence schema
- If creating a table with the LIKE clause, source table owner

Restrictions for complex types

Complex types used in native tables have some restrictions, in addition to the restrictions for individual types listed on their reference pages:

- A native table must have at least one column that is a primitive type or a native array (one-dimensional array of a primitive type). If a flex table has real columns, it must also have at least one column satisfying this restriction.
- Complex type columns cannot be used in ORDER BY or PARTITION BY clauses nor as FILLER columns.
- Complex type columns cannot have constraints .
- Expressions returning complex types cannot be used as projection columns, and projections cannot be segmented or ordered by columns of complex types.
- Tables with columns of complex types cannot use DEFAULT and SET USING.

Examples

The following example creates a table in the public schema:

```
CREATE TABLE public.Premium_Customer
(
   ID IDENTITY ,
   lname varchar(25),
   fname varchar(25),
   store_membership_card int
);
```

The following example uses LIKE to create a new table from this one:

```
=> CREATE TABLE All_Customers LIKE Premium_Customer;
CREATE TABLE
```

The following example selects columns from one table to use in a new table, using an AS clause:

```
=> CREATE TABLE cust_basic_profile AS SELECT
    customer_key, customer_gender, customer_age, marital_status, annual_income, occupation
    FROM customer_dimension WHERE customer_age>18 AND customer_gender !='';
CREATE TABLE
=> SELECT customer_age, annual_income, occupation FROM cust_basic_profile
    WHERE customer_age > 23 ORDER BY customer_age;
 customer_age | annual_income |     occupation
--------------+---------------+--------------------
           24 |        469210 | Hairdresser
           24 |        140833 | Butler
           24 |        558867 | Lumberjack
           24 |        529117 | Mechanic
           24 |        322062 | Acrobat
           24 |        213734 | Writer
           ...
```

The following example creates a table using array columns:

```
=> CREATE TABLE orders(
   orderkey    INT,
   custkey     INT,
   prodkey     ARRAY[VARCHAR(10)],
   orderprices ARRAY[DECIMAL(12,2)],
   orderdate   DATE
);
```

The following example uses a ROW complex type:

```
=> CREATE TABLE inventory
   (store INT, products ROW(name VARCHAR, code VARCHAR));
```

The following example uses quotas:

```
=> CREATE SCHEMA internal DISK_QUOTA '10T';
CREATE SCHEMA

=> CREATE TABLE internal.sales (...) DISK_QUOTA '5T';
CREATE TABLE

=> CREATE TABLE internal.leads (...) DISK_QUOTA '12T';
WARNING 0: Table leads has disk quota greater than its schema internal
```

See also

- Creating tables
- Replicating a table
- Creating a table from a query
- CREATE TEMPORARY TABLE
- CREATE EXTERNAL TABLE AS COPY
- CREATE FLEXIBLE TABLE

In this section

- Column-constraint
- Column-definition
- Column-name-list
- Partition clause
- Table-constraint

Column-constraint

Adds a constraint to a column's metadata. For details, see Constraints .

Syntax

```
[ { AUTO_INCREMENT | IDENTITY } [ (args) ] ]
[ CONSTRAINT constraint-name ] {
  [ CHECK (expression) [ ENABLED | DISABLED ] ]
  [ [ DEFAULT expression ] [ SET USING expression } | DEFAULT USING expression ]
  [ NULL | NOT NULL ]
  [ { PRIMARY KEY [ ENABLED | DISABLED ] REFERENCES table [( column )] } ]
  [ UNIQUE [ ENABLED | DISABLED ] ]
}
```

Parameters

> **Note**
> You can specify enforcement of several constraints by qualifying them with the keywords ENABLED or DISABLED . See Enforcing Constraints below.

AUTO_INCREMENT | IDENTITY

Creates a table column whose values are automatically generated by and managed by the database. You cannot change or load values in this column. You can set this constraint on only one table column.

AUTO_INCREMENT and IDENTITY are synonyms. For details on this constraint and optional arguments, see IDENTITY sequences .
These options are invalid for temporary tables.

CONSTRAINT *constraint-name*

Assigns a name to the constraint, valid for the following constraints:

- PRIMARY KEY
- REFERENCES (foreign key)
- CHECK
- UNIQUE

If you omit assigning a name to these constraints, Vertica assigns its own name. For details, see Naming constraints .
Vertica recommends that you name all constraints.

CHECK (*expression*)

Adds check condition *expression* , which returns a Boolean value.

DEFAULT

Specifies this column's default value:

```
DEFAULT default-expr
```

Vertica evaluates the DEFAULT expression and sets the column on load operations, if the operation omits a value for the column. For details about valid expressions, see Defining column values .

SET USING

Specifies to set values in this column from the specified expression:

```
SET USING using-expr
```

Vertica evaluates the SET USING expression and refreshes column values only when the function REFRESH_COLUMNS is invoked. For details about valid expressions, see Defining column values .

DEFAULT USING

Defines the column with DEFAULT and SET USING constraints, specifying the same expression for both. DEFAULT USING columns support the same expressions as SET USING columns, and are subject to the same restrictions .

NULL | NOT NULL

Specifies whether the column can contain null values:

- NULL : Allows null values in the column. If you set this constraint on a primary key column, Vertica ignores it and sets it to NOT NULL .
- NOT NULL : Specifies that the column must be set to a value during insert and update operations. If the column has no default value and no value is provided, INSERT or UPDATE returns an error.

If you omit this constraint, the default is NULL for all columns except primary key columns, which Vertica always sets to NOT NULL .
External tables: If you specify NOT NULL and the column contains null values, queries are liable to return errors or generate unexpected behavior. Specify NOT NULL for an external table column only if you are sure that the column does not contain nulls.

PRIMARY KEY

Identifies this column as the table's primary key.

REFERENCES

Identifies this column as a foreign key:

```
REFERENCES table [column]
```

where *column* is the primary key in *table* . If you omit *column* , Vertica references the primary key in *table* .

UNIQUE

Requires column data to be unique with respect to all table rows.

Privileges

Table owner or user WITH GRANT OPTION is grantor.

- REFERENCES privilege on table to create foreign key constraints that reference this table
- USAGE privilege on schema that contains the table

Enforcing constraints

The following constraints can be qualified with the keyword ENABLED or DISABLED :

- PRIMARY KEY
- UNIQUE
- CHECK

If you omit ENABLED or DISABLED , Vertica determines whether to enable the constraint automatically by checking the appropriate configuration parameter:

- EnableNewPrimaryKeysByDefault
- EnableNewUniqueKeysByDefault
- EnableNewCheckConstraintsByDefault

For details, see Constraint enforcement .

Column-definition

Specifies the name, data type, and constraints to be applied to a column.

Syntax

```
column-name data-type
  [ column-constraint ][...]
  [ ENCODING encoding-type ]
  [ ACCESSRANK integer ]
```

Parameters

column-name

The name of a column to be created or added.

data-type

A Vertica-supported data type .

> **Tip**
> When specifying the maximum column width in a CREATE TABLE statement, use the width in bytes (octets) for any of the string types. Each UTF-8 character might require four bytes, but European languages generally require a little over one byte per character, while Oriental languages generally require a little under three bytes per character.

column-constraint

A constraint type that Vertica supports—for example, NOT NULL or UNIQUE . For general information, see Constraints .

ENCODING *encoding-type*

The column encoding type , by default set to AUTO.

ACCESSRANK *integer*

Overrides the default access rank for a column. Use this parameter to increase or decrease the speed at which Vertica accesses a column. For more information, see Overriding Default Column Ranking .

Examples

The following example creates a table named Employee_Dimension and its associated superprojection in the public schema. The Employee_key column is designated as a primary key, and RLE encoding is specified for the Employee_gender column definition:

```
=> CREATE TABLE public.Employee_Dimension (
    Employee_key                integer PRIMARY KEY NOT NULL,
    Employee_gender             varchar(8) ENCODING RLE,
    Courtesy_title              varchar(8),
    Employee_first_name         varchar(64),
    Employee_middle_initial     varchar(8),
    Employee_last_name          varchar(64)
);
```

Column-name-list

Used to rename columns when creating a table or temporary table from a query ; also used to specify the column's encoding type and access rank .

Syntax

```
column-name-list
   [ ENCODING encoding-type ]
   [ ACCESSRANK integer ]
   [ GROUPED ( column-reference[,...] ) ]
```

Parameters

column-name

Specifies the new name for the column.

ENCODING *[encoding-type](/sql-reference/statements/create-statements/create-projection/encoding-types.html)*

Specifies the type of encoding to use on the column. The default encoding type is AUTO .

ACCESSRANK *integer*

Overrides the default access rank for a column, useful for prioritizing access to a column. See Prioritizing column access speed .

GROUPED

Groups two or more columns . For detailed information, see GROUPED clause .

Requirements

- A column in the list can not specify the column's data type or any constraint. These are derived from the queried table.
- If the query output has expressions other than simple columns (for example, constants or functions) then an alias must be specified for that expression, or the column name list must include all queried columns.
- CREATE TABLE can specify encoding types and access ranks in the column name list or the query's ENCODED BY clause, but not in both. For example, the following CREATE TABLE statement sets encoding and access rank on two columns in the column name list:

  ```
  => CREATE TABLE promo1 (state ENCODING RLE ACCESSRANK 1, zip ENCODING RLE,...)
     AS SELECT * FROM customer_dimension ORDER BY customer_state;
  ```

  The next statement specifies the same encoding and access rank in the query's ENCODED BY clause.

  ```
  => CREATE TABLE promo2
     AS SELECT * FROM customer_dimension ORDER BY customer_state
     ENCODED BY customer_state ENCODING RLE ACCESSRANK 1, customer_zip ENCODING RLE;
  ```

Partition clause

Specifies partitioning of table data, through a PARTITION BY clause in the table definition:

```
PARTITION BY partition-expression [ GROUP BY group-expression ] [ active-partition-count-expr ]
```

PARTITION BY *partition-expression*

For each table row, resolves to a partition key that is derived from one or more table columns.

> **Caution**

Avoid partitioning tables on LONG VARBINARY and LONG VARCHAR columns. Doing so can adversely impact performance.

GROUP BY ***group-expression***

For each table row, resolves to a partition group key that is derived from the partition key. Vertica uses group keys to merge partitions into separate partition groups. GROUP BY must use the same expression as PARTITION BY. For example:

```
...PARTITION BY (i+j) GROUP BY (
   CASE WHEN (i+j) < 5 THEN 1
      WHEN (i+j) < 10 THEN 2
      ELSE 3);
```

For details on partitioning table data by groups, see Partition grouping and Hierarchical partitioning .

active-partition-count-expr

Specifies how many partitions are active for this table, specified as follows:

- In partition clause of CREATE TABLE:
  ```
  ACTIVEPARTITIONCOUNT integer
  ```
- In partition clause of ALTER TABLE:
  ```
  SET ACTIVEPARTITIONCOUNT integer
  ```

This setting supersedes configuration parameter ActivePartitionCount . For details on usage, see Active and inactive partitions .

Partitioning requirements and restrictions

PARTITION BY expressions can specify leaf expressions, functions, and operators. The following requirements and restrictions apply:

- All table projections must include all columns referenced in the expression; otherwise, Vertica cannot resolve the expression.
- The expression can reference multiple columns, but it must resolve to a single non-null value for each row.

> **Note**
> You can avoid null-related errors with the function ZEROIFNULL . This function can check a PARTITION BY expression for null values and evaluate them to 0. For example: CREATE TABLE t1 (a int, b int) PARTITION BY (ZEROIFNULL(a)); CREATE TABLE

- All leaf expressions must be constants or table columns.
- All other expressions must be functions and operators. The following restrictions apply to functions: * They must be immutable —that is, they return the same value regardless of time and locale and other session- or environment-specific conditions. * They cannot be aggregate functions . * They cannot be Vertica meta-functions.
- The expression cannot include queries.
- The expression cannot include user-defined data types such as Geometry .

GROUP BY expressions do not support modulo (%) operations.

Examples

The following statements create the store_orders table and load data into it. The CREATE TABLE statement includes a simple partition clause that specifies to partition data by year:

```
=> CREATE TABLE public.store_orders
(
   order_no int,
   order_date timestamp NOT NULL,
   shipper varchar(20),
   ship_date date
)
UNSEGMENTED ALL NODES
PARTITION BY YEAR(order_date);
CREATE TABLE
=> COPY store_orders FROM '/home/dbadmin/export_store_orders_data.txt';
41834
```

As COPY loads the new table data into ROS storage, the Tuple Mover executes the table's partition clause by dividing orders for each year into separate partitions, and consolidating these partitions in ROS containers.

In this case, the Tuple Mover creates four partition keys for the loaded data—2017, 2016, 2015, and 2014—and divides the data into separate ROS containers accordingly:

```
=> SELECT dump_table_partition_keys('store_orders');
... Partition keys on node v_vmart_node0001
 Projection 'store_orders_super'
  Storage [ROS container]
    No of partition keys: 1
    Partition keys: 2017
  Storage [ROS container]
    No of partition keys: 1
    Partition keys: 2016
  Storage [ROS container]
    No of partition keys: 1
    Partition keys: 2015
  Storage [ROS container]
    No of partition keys: 1
    Partition keys: 2014

 Partition keys on node v_vmart_node0002
  Projection 'store_orders_super'
  Storage [ROS container]
    No of partition keys: 1
    Partition keys: 2017
...

(1 row)
```

As new data is loaded into store_orders , the Tuple Mover merges it into the appropriate partitions, creating partition keys as needed for new years.

See also

Partitioning tables

Table-constraint

Table-constraint

Adds a constraint to table metadata. You can specify table constraints with CREATE TABLE , or add a constraint to an existing table with ALTER TABLE . For details, see Setting constraints .

> **Note**
> Adding a constraint to a table that is referenced in a view does not affect the view.

Syntax

```
[ CONSTRAINT constraint-name ]
{
... PRIMARY KEY (column[,... ]) [ ENABLED | DISABLED ]
... | FOREIGN KEY (column[,... ] ) REFERENCES table [ (column[,...]) ]
... | UNIQUE (column[,...]) [ ENABLED | DISABLED ]
... | CHECK (expression) [ ENABLED | DISABLED ]
}
```

Parameters

CONSTRAINT *constraint-name*

Assigns a name to the constraint. Vertica recommends that you name all constraints.

PRIMARY KEY

Defines one or more NOT NULL columns as the primary key as follows:

```
PRIMARY KEY (column[,...]) [ ENABLED | DISABLED]
```

You can qualify this constraint with the keyword ENABLED or DISABLED . See Enforcing Constraints below.

If you do not name a primary key constraint, Vertica assigns the name C_PRIMARY .

FOREIGN KEY

Adds a referential integrity constraint defining one or more columns as foreign keys as follows:

```
FOREIGN KEY (column[,... ]) REFERENCES table [(column[,... ])]
```

If you omit *column* , Vertica references the primary key in *table* .
If you do not name a foreign key constraint, Vertica assigns the name C_FOREIGN .

Important
Adding a foreign key constraint requires the following privileges (in addition to privileges also required by ALTER TABLE):

- REFERENCES on the referenced table
- USAGE on the schema of the referenced table

UNIQUE

Specifies that the data in a column or group of columns is unique with respect to all table rows, as follows:

```
UNIQUE (column[,...]) [ENABLED | DISABLED]
```

You can qualify this constraint with the keyword ENABLED or DISABLED . See Enforcing Constraints below.
If you do not name a unique constraint, Vertica assigns the name C_UNIQUE .

CHECK

Specifies a check condition as an expression that returns a Boolean value, as follows:

```
CHECK (expression) [ENABLED | DISABLED]
```

You can qualify this constraint with the keyword ENABLED or DISABLED . See Enforcing Constraints below.
If you do not name a check constraint, Vertica assigns the name C_CHECK .

Privileges

Non-superusers: table owner, or the following privileges:

- USAGE on schema
- ALTER on table
- SELECT on table to enable or disable constraint enforcement

Enforcing constraints

A table can specify whether Vertica automatically enforces a primary key, unique key or check constraint with the keyword ENABLED or DISABLED . If you omit ENABLED or DISABLED , Vertica determines whether to enable the constraint automatically by checking the appropriate configuration parameter:

- EnableNewPrimaryKeysByDefault
- EnableNewUniqueKeysByDefault
- EnableNewCheckConstraintsByDefault

For details, see Constraint enforcement .

Examples

The following example creates a table (t01) with a primary key constraint.

```
CREATE TABLE t01 (id int CONSTRAINT sampleconstraint PRIMARY KEY);
CREATE TABLE
```

This example creates the same table without the constraint, and then adds the constraint with ALTER TABLE ADD CONSTRAINT

```
CREATE TABLE t01 (id int);
CREATE TABLE

ALTER TABLE t01 ADD CONSTRAINT sampleconstraint PRIMARY KEY(id);
WARNING 2623:  Column "id" definition changed to NOT NULL
ALTER TABLE
```

The following example creates a table (addapk) with two columns, adds a third column to the table, and then adds a primary key constraint on the third column.

```
=> CREATE TABLE addapk (col1 INT, col2 INT);
CREATE TABLE

=> ALTER TABLE addapk ADD COLUMN col3 INT;
ALTER TABLE

=> ALTER TABLE addapk ADD CONSTRAINT col3constraint PRIMARY KEY (col3) ENABLED;
WARNING 2623:  Column "col3" definition changed to NOT NULL
ALTER TABLE
```

Using the sample table addapk , check that the primary key constraint is enabled (is_enabled is t).

```
=> SELECT constraint_name, column_name, constraint_type, is_enabled FROM PRIMARY_KEYS WHERE table_name IN ('addapk');

 constraint_name | column_name | constraint_type | is_enabled
-----------------+-------------+-----------------+------------
 col3constraint  | col3        | p               | t
(1 row)
```

This example disables the constraint using ALTER TABLE ALTER CONSTRAINT .

```
=> ALTER TABLE addapk ALTER CONSTRAINT col3constraint DISABLED;
```

Check that the primary key is now disabled (is_enabled is f).

```
=> SELECT constraint_name, column_name, constraint_type, is_enabled FROM PRIMARY_KEYS WHERE table_name IN ('addapk');

 constraint_name | column_name | constraint_type | is_enabled
-----------------+-------------+-----------------+------------
 col3constraint  | col3        | p               | f
(1 row)
```

For a general discussion of constraints, see Constraints . For additional examples of creating and naming constraints, see Naming constraints .

CREATE TEMPORARY TABLE

Creates a table whose data persists only during the current session. By default, temporary table data is not visible to other sessions.

Syntax

Create with column definitions:

```
CREATE [ scope ] TEMP[ORARY] TABLE [ IF NOT EXISTS ] [[database.]schema.]table-name
  ( column-definition[,...] )
  [ table-constraint ]
  [ ON COMMIT { DELETE | PRESERVE } ROWS ]
  [ NO PROJECTION ]
  [ ORDER BY table-column[,...] ]
  [ segmentation-spec ]
  [ KSAFE [safety-level] ]
  [ {INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES ]
  [ DISK_QUOTA quota ]
```

Create from another table:

```
CREATE TEMP[ORARY] TABLE [ IF NOT EXISTS ] [[database.]schema.]table-name
  [ ( column-name-list ) ]
  [ ON COMMIT { DELETE | PRESERVE } ROWS ]
  AS  [ /*+ LABEL */ ] [ AT epoch ] query [ ENCODED BY column-ref-list ]
  [ DISK_QUOTA quota ]
```

Parameters

scope

Visibility of the table definition:

- GLOBAL : The table definition is visible to all sessions, and persists until you explicitly drop the table.
- LOCAL : the table definition is visible only to the session in which it is created, and is dropped when the session ends.

If no scope is specified, Vertica uses the default that is set by the DefaultTempTableLocal configuration parameter.
Regardless of this setting, retention of temporary table data is set by the keywords ON COMMIT DELETE and ON COMMIT PRESERVE (see below).

For more information, see Creating temporary tables .

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.
If you do not specify a schema, the table is created in the default schema.

table-name

Name of the table to create, where *table-name* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

column-definition

Column names and types. A table can have up to 9800 columns.

table-constraint

Adds a constraint to table metadata.

ON COMMIT

Whether data is transaction- or session-scoped:

```
ON COMMIT {PRESERVE | DELETE} ROWS
```

- DELETE (default) marks the temporary table for transaction-scoped data. Vertica removes all table data after each commit.
- PRESERVE marks the temporary table for session-scoped data, which is preserved beyond the lifetime of a single transaction. Vertica removes all table data when the session ends.

NO PROJECTION

Prevents Vertica from creating auto-projections for this table. A superprojection is created only when data is explicitly loaded into this table.

NO PROJECTION is invalid with the following clauses:

- ORDER BY
- KSAFE
- Any segmentation clause (*hash-segmentation-clause* or *unsegmented-clause*).

{INCLUDE | EXCLUDE} [SCHEMA] PRIVILEGES

Default inheritance of schema privileges for this table:

- INCLUDE PRIVILEGES specifies that the table inherits privileges that are set on its schema. This is the default behavior if privileges inheritance is enabled for the schema.
- EXCLUDE PRIVILEGES disables inheritance of privileges from the schema.

For details, see Inherited privileges .

ORDER BY *table-column* [,...]

Invalid for external tables, specifies columns from the SELECT list on which to sort the superprojection that is automatically created for this table. The ORDER BY clause cannot include qualifiers ASC or DESC . Vertica always stores projection data in ascending sort order.
If you omit the `ORDER BY` clause, Vertica uses the SELECT list order as the projection sort order.

segmentation-spec

Invalid for external tables, specifies how to distribute data for auto-projections of this table. Supply one of the following clauses:

- *hash-segmentation-clause* : Specifies to segment data evenly and distribute across cluster nodes. Vertica recommends segmenting large tables.
- *unsegmented-clause* : Specifies to create an unsegmented projection.

If this clause is omitted, Vertica generates auto-projections with default hash segmentation .

KSAFE [*safety-level*]

Invalid for external tables, specifies K-safety of auto-projections created for this table, where *k-num* must be equal to or greater than system K-safety. If you omit this option, the projection uses the system K-safety level.
Eon Mode: K-safety of temporary tables is always set to 0, regardless of system K-safety. If a CREATE TEMPORARY TABLE statement sets *k-num* greater than 0, Vertica returns an warning.

column-name-list

Valid only when creating a table from a query (AS *query*), defines column names that map to the query output. If you omit this list, Vertica uses the query output column names.
This clause and the ENCODED BY clause are mutually exclusive. Column name lists are invalid for external tables.

The names in *column-name-list* and queried columns must be the same in number.

For example:

```
CREATE TEMP TABLE customer_occupations (name, profession)
  AS SELECT customer_name, occupation FROM customer_dimension;
```

AS *query*

Creates and loads a table from the results of a query , specified as follows:

```
AS  [ /*+ LABEL */ ] [ AT  ] epoch query
```

The query cannot include complex type columns.

ENCODED BY *column-ref-list*

A comma-delimited list of columns from the source table, where each column is qualified by one or both of the following encoding options:
- ACCESSRANK *integer* : Overrides the default access rank for a column, useful for prioritizing access to a column. See Prioritizing column access speed .
- ENCODING *encoding-type* : Specifies the type of encoding to use on the column. The default encoding type is AUTO .

This option and * column-name-list *are mutually exclusive. This option is invalid for external tables.

DISK_QUOTA *quota*

String, an integer followed by a supported unit: K, M, G, or T. If the schema has a quota, this value must be smaller than the schema quota. Data-load and ILM operations that increase the table's usage beyond the set quota fail. For details, see Disk quotas .

If not specified, the table has no quota.
Disk quota is valid for global temporary tables but not local ones.

Privileges

The following privileges are required:

- CREATE privileges on the table schema
- If creating a temporary table that includes a named sequence:
  - SELECT privilege on sequence object
  - USAGE privilege on sequence schema

Restrictions

- Queries on temporary tables are subject to the same restrictions on SQL support as persistent tables.
- You cannot add projections to non-empty, global temporary tables (ON COMMIT PRESERVE ROWS). Make sure that projections exist before you load data. See Auto-projections .
- While you can add projections for temporary tables that are defined with ON COMMIT DELETE ROWS specified, be aware that you might lose all data.
- Mergeout operations cannot be used on session-scoped temporary data.
- In general, session-scoped temporary table data is not visible using system (virtual) tables.

- Temporary tables do not recover. If a node fails, queries that use the temporary table also fail. Restart the session and populate the temporary table.
- Local temporary tables cannot have disk quotas.

Examples

See Creating temporary tables .

See also

- ALTER TABLE
- CREATE TABLE

CREATE TEXT INDEX

Creates a text index used to perform text searches. If data within a table is partitioned, then an extra column appears in the text index, showing the partition.

Syntax

```
CREATE TEXT INDEX [[database.]schema.]txtindex-name
 ON [schema.]source-table (unique-id, text-field [, column-name,...])
[STEMMER {stemmer-name(stemmer-input-data-type)| NONE}]
[TOKENIZER tokenizer-name(tokenizer-input-data-type)];
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

If you do not specify a schema, the table is created in the default schema.

txtindex-name

The text index name.

source-table

The source table to index.

unique-id

The name of the column in the source table that contains a unique identifier. Any data type is permissible. The column must be the primary key in the source table.

text-field

The name of the column in the source table that contains the text field. Valid data types are:

- CHAR
- VARCHAR
- LONG VARCHAR
- VARBINARY
- LONG VARBINARY

Nulls are allowed.

column-name

The name of a column or columns to be included as additional columns.

stemmer-name

The name of the stemmer.

stemmer-input-data-type

The input data type of the *stemmer-name* function.

tokenizer-name

Specifies the name of the tokenizer.

tokenizer-input-data-type

This value is the input data type of the *tokenizer-name* function. It can accept any number of arguments.

If a Vertica tokenizers is used, then this parameter can be omitted.

Privileges

The index automatically inherits the query permissions of its parent table. The table owner and dbadmin will be allowed to create and/or modify the indices.

Important

Do not alter the contents or definitions of the text index. If the contents or definitions of the text index are altered, then the results will not appropriately match the source table.

Requirements

- Requires there be a column with a unique identifier set as the primary key.
- The source table must have an associated projection, and must be both sorted and segmented by the primary key.

Examples

The following example shows how to create a text index with an additional unindexed column on the table t_log using the CREATE TEXT INDEX statement:

```
=> CREATE TEXT INDEX t_log_index ON t_log (id, text, day_of_week);
CREATE INDEX
=> SELECT * FROM t_log_index;
       token          | doc_id | day_of_week
----------------------+--------+-------------
'catalog               |      1 | Monday
'dbadmin'              |      2 | Monday
2014-06-04             |      1 | Monday
2014-06-04             |      2 | Monday
2014-06-04             |      3 | Monday
2014-06-04             |      4 | Monday
2014-06-04             |      5 | Monday
2014-06-04             |      6 | Monday
2014-06-04             |      7 | Monday
2014-06-04             |      8 | Monday
45035996273704966      |      3 | Tuesday
45035996273704968      |      4 | Tuesday
<INFO>                 |      1 | Tuesday
<INFO>                 |      6 | Tuesday
<INFO>                 |      7 | Tuesday
<INFO>                 |      8 | Tuesday
<WARNING>              |      2 | Tuesday
<WARNING>              |      3 | Tuesday
<WARNING>              |      4 | Tuesday
<WARNING>              |      5 | Tuesday

...

(97 rows)
```

The following example shows a text index, tpart_index, created from a partitioned source table:

```
=> SELECT * FROM tpart_index;
        token          | doc_id | partition
-----------------------+--------+-----------
 0                     |      4 |      2014
 0                     |      5 |      2014
 11:00:49.568          |      4 |      2014
 11:00:49.568          |      5 |      2014
 11:00:49.569          |      6 |      2014
 <INFO>                |      6 |      2014
 <WARNING>             |      4 |      2014
 <WARNING>             |      5 |      2014
 Database              |      6 |      2014
 Execute:              |      6 |      2014
 Object                |      4 |      2014
 Object                |      5 |      2014
 [Catalog]             |      4 |      2014
 [Catalog]             |      5 |      2014
 'catalog              |      1 |      2013
 'dbadmin'             |      2 |      2013
 0                     |      3 |      2013
 11:00:49.568          |      1 |      2013
 11:00:49.568          |      2 |      2013
 11:00:49.568          |      3 |      2013
 11:00:49.570          |      7 |      2013
 11:00:49.571          |      8 |      2013
 45035996273704966     |      3 |      2013

...

(89 rows)
```

See also

- Using text search
- DROP TEXT INDEX

CREATE TLS CONFIGURATION

Creates a TLS Configuration object. For information on existing TLS Configuration objects, query TLS_CONFIGURATIONS .

To modify an existing TLS Configuration object, see ALTER TLS CONFIGURATION .

Syntax

```
CREATE TLS CONFIGURATION tls_config_name {
   [ CERTIFICATE { NULL | cert_name } ]
   [ CA CERTIFICATES ca_cert_name [,...] ]
   [ CIPHER SUITES { '' | 'openssl_cipher [,...]' } ]
   [ TLSMODE 'tlsmode' ]
}
```

Parameters

tls_config_name

The name of the TLS Configuration object.

cert_name

A certificate created with CREATE CERTIFICATE .

ca_cert_name

A CA certificate created with CREATE CERTIFICATE .

openssl_cipher

A comma-separated list of cipher suites to use instead of the default set of cipher suites. Providing an empty string for this parameter clears the alternate cipher suite list and instructs the specified TLS Configuration to use the default set of cipher suites.

To view enabled cipher suites, use LIST_ENABLED_CIPHERS .

tlsmode

How Vertica establishes TLS connections and handles client certificates, one of the following, in order of ascending security:

- DISABLE : Disables TLS. All other options for this parameter enable TLS.
- ENABLE : Enables TLS. Vertica does not check client certificates.
- TRY_VERIFY : Establishes a TLS connection if one of the following is true:
  - the other host presents a valid certificate
  - the other host doesn't present a certificate

  If the other host presents an invalid certificate, the connection will use plaintext.
- VERIFY_CA : Connection succeeds if Vertica verifies that the other host's certificate is from a trusted CA. If the other host does not present a certificate, the connection uses plaintext.
- VERIFY_FULL : Connection succeeds if Vertica verifies that the other host's certificate is from a trusted CA and the certificate's cn (Common Name) or subjectAltName attribute matches the hostname or IP address of the other host.
  Note that for client certificates, cn is used for the username, so subjectAltName must match the hostname or IP address of the other host.

VERIFY_FULL is unsupported for client-server TLS (the server TLS Configuration context) and behaves as VERIFY_CA .

CREATE TRIGGER

Creates a trigger. For details, see Triggers .

Syntax

```
CREATE TRIGGER [ IF NOT EXISTS ] [[database.]schema.]trigger
  ON SCHEDULE [[database.]schema.]schedule
  EXECUTE PROCEDURE procedure AS DEFINER
```

Parameters

IF NOT EXISTS

If an object with the same name exists, do not create it and proceed. If you omit this option and the object exists, Vertica generates a ROLLBACK error message. In both cases, the object is not created if it already exists.
The IF NOT EXISTS clause is useful for SQL scripts where you want to create an object if it does not already exist.

For related information, see ON_ERROR_STOP .

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

trigger

The name of the trigger.

schedule

The schedule with which to associate the trigger.

procedure

The function signature of the stored procedure .

AS DEFINER

The user to execute the stored procedure as. Currently, the only option is DEFINER, which executes the stored procedure as the definer of the trigger.

Privileges

Superuser

Examples

CREATE TRIGGER requires a schedule and stored procedure :

```
=> CREATE PROCEDURE revoke_all_on_table(table_name VARCHAR, user_name VARCHAR)
LANGUAGE PLvSQL
AS $$
BEGIN
   EXECUTE 'REVOKE ALL ON ' || QUOTE_IDENT(table_name) || ' FROM ' || QUOTE_IDENT(user_name);
END;
$$;

=> CREATE SCHEDULE 24_hours_later USING DATETIMES('2022-12-16 12:00:00');
```

To create the trigger with schedule 24_hours_later and stored procedure revoke_all_on_table() with arguments customer_dimension and Bob :

```
=> CREATE TRIGGER revoke_trigger ON SCHEDULE 24_hours_later EXECUTE PROCEDURE revoke_all_on_table('customer_dimension', 'Bob') AS DEFINER;;
```

CREATE USER

Adds a name to the list of authorized database users.

> **Note**
> New users lack default access to schema PUBLIC. Be sure to grant new users USAGE privileges on the PUBLIC schema .

Syntax

```
CREATE USER user-name [ account-parameter value[,...] ]
```

Parameters

user-name

Name of the new user, where *user-name* conforms to conventions described in Identifiers .

account-parameter value

One or more user account parameter settings (see below).

User account parameters

Specify one or more user account parameters as a comma-delimited list:

```
account-parameter setting[,...]
```

Parameter	Settings
ACCOUNT	Locks or unlocks user access to the database, one of the following: • UNLOCK (default) • LOCK prevents a new user from logging in. This can be useful when creating an account for a user who does not need immediate access. **Tip** To automate account locking, set a maximum number of failed login attempts with CREATE PROFILE .
GRACEPERIOD	Specifies how long a user query can block on any session socket, one of the following: • NONE (default): Removes any grace period previously set on session queries. • ' *interval* ' : Specifies as an interval the maximum grace period for current session queries, up to 20 days. For details, see Handling session socket blocking .

IDENTIFIED BY	Sets the user's password as follows: IDENTIFIED BY { '[*password*]' | ' *hashed-password* ' SALT ' *hash-salt* ' } • *password* : ASCII password that Vertica then hashes for internal storage. An empty string enables this user to access the database with no password. • *hashed-password* : A pre-hashed password and its associated hex string *hash-salt* . Setting a password this way bypasses all password complexity requirements . Important If you omit this parameter, this user can access the database with no password. For details, see Password guidelines and Creating a database name and password .
IDLESESSIONTIMEOUT	The length of time the system waits before disconnecting an idle session, one of the following: • NONE (default): No limit set for this user. If you omit this parameter, no limit is set for this user. • ' *interval* ' : An interval value, up to one year. For details, see Managing client connections .
MAXCONNECTIONS	Sets the maximum number of connections the user can have to the server, one of the following: • NONE (default): No limit set. If you omit this parameter, the user can have an unlimited number of connections across the database cluster. • *integer* ON DATABASE : Sets to *integer* the maximum number of connections across the database cluster. • *integer* ON NODE : Sets to *integer* the maximum number of connections to each node. For details, see Managing client connections .
MEMORYCAP	Sets how much memory can be allocated to user requests, one of the following: • NONE (default): No limit • A string value that specifies the memory limit, one of the following: ◦ ' *int* %' expresses the maximum as a percentage of total memory available to the Resource Manager, where *int* is an integer value between 0 and 100.For example: MEMORYCAP '40%' ◦ ' *int* {K|M|G|T}' expresses memory allocation in kilobytes, megabytes, gigabytes, or terabytes. For example: MEMORYCAP '10G'
PASSWORD EXPIRE	Forces immediate expiration of the user's password. The user must change the password on the next login. **Note** PASSWORD EXPIRE has no effect when using external password authentication methods such as LDAP or Kerberos.
PROFILE	Assigns a profile that controls password requirements for this user, one of the following: • DEFAULT (default): Assigns the default database profile to this user. • *profile-name* : A profile that is defined by CREATE PROFILE . If you omit this parameter, the user is assigned the default profile.

RESOURCE POOL	Assigns a resource pool to this user. The user must also be granted privileges to this pool , unless privileges to the pool are set to PUBLIC .
RUNTIMECAP	Sets how long this user's queries can execute, one of the following: • NONE (default): No limit set for this user. If you omit this parameter, no limit is set for this user. • ' *interval* ' : An interval value, up to one year. A query's runtime limit can be set at three levels: the user's runtime limit, the user's resource pool, and the session setting. For more information, see Setting a runtime limit for queries .
SEARCH_PATH	Specifies the user's default search path, that tells Vertica which schemas to search for unqualified references to tables and UDFs, one of the following: • DEFAULT (default): Sets the search path as follows: `"$user", public, v_catalog, v_monitor, v_internal` • Comma-delimited list of schemas. For details, see Setting Search Paths .
TEMPSPACECAP	Sets how much temporary file storage is available for user requests, one of the following: • NONE (default): No limit • String value that specifies the storage limit, one of the following: ◦ *int* % expresses the maximum as a percentage of total temporary storage available to the Resource Manager, where *int* is an integer value between 0 and 100. For example: TEMPSPACECAP '40%' ◦ *int* {K|M|G|T} expresses storage allocation in kilobytes, megabytes, gigabytes, or terabytes. For example: TEMPSPACECAP '10G'

Privileges

Superuser

User name best practices

Vertica database user names are logically separate from user names of the operating system in which the server runs. If all the users of a particular server also have accounts on the server's machine, it makes sense to assign database user names that match their operating system user names. However, a server that accepts remote connections might many database users with no local operating system account. In this case, there is no need to connect database and system user names.

Examples

```
=> CREATE USER Fred IDENTIFIED BY 'Mxyzptlk';
=> GRANT USAGE ON SCHEMA PUBLIC to Fred;
```

See also

- ALTER USER
- DROP USER

CREATE VIEW

Defines a view . Views are read only, so they do not support insert, update, delete, or copy operations.

Syntax

```
CREATE [ OR REPLACE ] VIEW [[database.]schema.]view [ (column[,...]) ]
 [ {INCLUDE|EXCLUDE} [SCHEMA] PRIVILEGES ] AS query
```

Parameters

OR REPLACE

Specifies to overwrite the existing view *view-name* . If you omit this option and *view-name* already exists, CREATE VIEW returns an error.

Any grants assigned to the view before you execute a CREATE OR REPLACE remain on the updated view. See GRANT (view) .

[*database*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

view

Identifies the view to create, where *view* conforms to conventions described in Identifiers . It must also be unique among all names of sequences, tables, projections, views, and models within the same schema.

***column* [,...]**

List of up to 9800 names to use as view column names. Vertica maps view column names to query columns according to the order of their respective lists. By default, the view uses column names as they are specified in the query.

query

A SELECT statement that the temporary view executes. The SELECT statement can reference tables, temporary tables, and other views.

{INCLUDE|EXCLUDE}[SCHEMA] PRIVILEGES

Specifies whether this view inherits schema privileges:

- INCLUDE PRIVILEGES specifies that the view inherits privileges that are set on its schema. This is the default behavior if privileges inheritance is enabled for the schema.
- EXCLUDE PRIVILEGES disables inheritance of privileges from the schema.

For details, see Inherited privileges .

Privileges

See Creating views .

Examples

The following example shows how to create a view that contains data from multiple tables.

```
=> CREATE VIEW temp_t0 AS SELECT * from t0_p1 UNION ALL
   SELECT * from t0_p2 UNION ALL
    SELECT * from t0_p3 UNION ALL
     SELECT * from t0_p4 UNION ALL
      SELECT * from t0_p5;
```

See also

- ALTER VIEW
- CREATE LOCAL TEMPORARY VIEW
- Creating views
- DROP VIEW
- GRANT (view)
- REVOKE (view)

DEACTIVATE DIRECTED QUERY

Deactivates one or more directed queries previously activated by ACTIVATE DIRECTED QUERY .

Syntax

```
DEACTIVATE DIRECTED QUERY { query-name | input-query | where-clause }
```

Arguments

query-name

Name of the directed query to deactivate, as stored in the DIRECTED_QUERIES column query_name .

input-query

The input query of the directed queries to deactivate. Use this argument to deactivate multiple direct queries that map to the same input query.

where-clause

Resolves to one or more directed queries that are filtered from system table DIRECTED_QUERIES . For example, the following statement specifies to deactivate all directed queries with the same save_plans_version identifier:

```
=> DEACTIVATE DIRECTED QUERY WHERE save_plans_version = 21;
```

Privileges

Superuser

Examples

See Activating and deactivating directed queries .

DELETE

Removes the specified rows from a table and returns a count of the deleted rows. A count of 0 is not an error, but indicates that no rows matched the condition. An unqualified DELETE statement (one that omits a WHERE clause) removes all rows but leaves intact table columns, projections, and constraints.

DELETE supports subqueries and joins, so you can delete values in a table based on values in other tables.

Important

The Vertica implementation of DELETE differs from traditional databases: it does not delete data from disk storage, but instead marks rows as deleted so they are available for historical queries. Deleted data remains on disk, and counts against disk quota, until purged.

Syntax

```
DELETE [ /*+LABEL (label-string)*/ ]  FROM [[database.]schema.]table [ where-clause ]
```

Arguments

LABEL

Assigns a label to a statement to identify it for profiling and debugging.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

Any table, including temporary tables.

where-clause

Which rows to mark for deletion. If you omit this clause, DELETE behavior varies depending on whether the table is persistent or temporary. See below for details.

Privileges

Table owner or user with GRANT OPTION is grantor.

- DELETE privilege on table
- USAGE privilege on the schema of the target table
- SELECT privilege on a table when the DELETE statement includes a WHERE or SET clause that specifies columns from that table.

Restrictions

You cannot execute DELETE on an immutable table .

Committing successive table changes

Vertica follows the SQL-92 transaction model, so successive INSERT, UPDATE, and DELETE statements are included in the same transaction. You do not need to explicitly start this transaction; however, you must explicitly end it with COMMIT , or implicitly end it with COPY . Otherwise, Vertica discards all changes that were made within the transaction.

Persistent and temporary tables

When deleting from a persistent table, DELETE removes data directly from the ROS.

DELETE execution on temporary tables varies, depending on whether the table was created with ON COMMIT DELETE ROWS (default) or ON COMMIT PRESERVE ROWS :

- If DELETE contains a WHERE clause that specifies which rows to remove, behavior is identical: DELETE marks the rows for deletion. In both cases, you cannot roll back to an earlier savepoint.
- If DELETE omits a WHERE clause and the table was created with ON COMMIT PRESERVE ROWS , Vertica marks all table rows for deletion. If the table was created with ON COMMIT DELETE ROWS , DELETE behaves like TRUNCATE TABLE and removes all rows from storage.

> **Note**
> If you issue an unqualified DELETE statement on a temporary table created with ON COMMIT DELETE ROWS , Vertica removes all rows from storage but does not end the transaction.

Examples

The following statement removes all rows from a temporary table:

```
=> DELETE FROM temp1;
```

The following statement deletes all records from a schema-qualified table where a condition is satisfied:

```
=> DELETE FROM retail.customer WHERE state IN ('MA', 'NH');
```

For examples that show how to nest a subquery within a DELETE statement, see Subqueries in UPDATE and DELETE .

See also

- DROP TABLE
- TRUNCATE TABLE
- Removing table data
- Optimizing DELETE and UPDATE

DISCONNECT

Closes a connection to another Vertica database that was opened in the same session with CONNECT TO VERTICA .

> **Note**
> Closing your session also closes the database connection. However, it is a good practice to explicitly close the connection to the other database, both to free up resources and to prevent issues with other SQL scripts that might be running in your session. Always closing the connection prevents potential errors if you run a script in the same session that attempts to open a connection to the same database, since each session can only have one connection to a given database at a time.

Syntax

```
DISCONNECT db-spec
```

Parameters

db-spec

Specifies the target database, either the database name or DEFAULT .

Privileges

None

Examples

```
=> DISCONNECT DEFAULT;
DISCONNECT
```

DO

Executes an anonymous (unnamed) stored procedure without saving it.

Syntax

```
DO [ LANGUAGE 'language-name' ] $$
   source
$$;
```

Parameters

language-name

Specifies the language of the procedure *source* , one of the following (both options refer to PLvSQL ; PLpgSQL is included to maintain compatibility with existing scripts):

- PLvSQL
- PLpgSQL

Default: PLvSQL

source

The source code of the procedure.

Privileges

None

Examples

For more complex examples, see Stored procedures: use cases and examples

This procedure prints the variables in the DECLARE block:

```
DO LANGUAGE PLvSQL $$
DECLARE
   x int := 3;
   y varchar := 'some string';
BEGIN
   RAISE NOTICE 'x = %', x;
   RAISE NOTICE 'y = %', y;
END;
$$;

NOTICE 2005:  x = 3
NOTICE 2005:  y = some string
```

For more information on RAISE NOTICE, see Errors and diagnostics .

See also

- Stored procedures
- PL/vSQL
- CREATE PROCEDURE (stored)

DROP statements

DROP statements let you delete database objects such as schemas, tables, and users.

In this section

- DROP ACCESS POLICY
- DROP AGGREGATE FUNCTION
- DROP ANALYTIC FUNCTION
- DROP AUTHENTICATION
- DROP CA BUNDLE
- DROP CERTIFICATE
- DROP DIRECTED QUERY
- DROP FAULT GROUP
- DROP FILTER
- DROP FUNCTION
- DROP KEY
- DROP LIBRARY
- DROP LOAD BALANCE GROUP
- DROP MODEL
- DROP NETWORK ADDRESS
- DROP NETWORK INTERFACE
- DROP NOTIFIER
- DROP PARSER
- DROP PROCEDURE (external)
- DROP PROCEDURE (stored)
- DROP PROFILE
- DROP PROJECTION
- DROP RESOURCE POOL
- DROP ROLE
- DROP ROUTING RULE
- DROP SCHEDULE
- DROP SCHEMA
- DROP SEQUENCE
- DROP SOURCE

- DROP SUBNET
- DROP TABLE
- DROP TEXT INDEX
- DROP TLS CONFIGURATION
- DROP TRANSFORM FUNCTION
- DROP TRIGGER
- DROP USER
- DROP VIEW

DROP ACCESS POLICY

Removes an access policy from a column or row.

Syntax

```
DROP ACCESS POLICY ON table FOR { COLUMN column | ROWS}
```

Parameters

table

Name of the table that contains the column access policy to remove

column

Name of the column that contains the access policy to remove

Privileges

Non-superuser: Ownership of the table

Examples

These examples show various cases where you can drop an access policy.

Drop column access policy:

```
=> DROP ACCESS POLICY ON customer FOR COLUMN Customer_Number;
```

Drop row access policy on a table:

```
=> DROP ACCESS POLICY ON customer_info FOR ROWS;
```

DROP AGGREGATE FUNCTION

Drops a user-defined aggregate function (UDAnF) from the Vertica catalog.

Syntax

```
DROP AGGREGATE FUNCTION [ IF EXISTS ] [[database.]schema.]function( [ arglist ] )
```

Parameters

IF EXISTS

Specifies not to report an error if the function to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

Specifies a name of the SQL function to drop. If the function name is schema-qualified, the function is dropped from the specified schema (as noted above).

arglist

A comma delimited list of argument names and data types that are passed to the function, formatted as follows:

```
{ [argname] argtype }[,...]
```

- *argname* optionally specifies the argument name, typically a column name.
- *argtype* specifies the argument's data type, where *argtype* matches a Vertica data type .

Privileges

Non-superuser: Owner

Requirements

- To drop a function, you must specify the argument types because several functions might share the same name with different parameters.
- Vertica does not check for dependencies, so if you drop a SQL function where other objects reference it (such as views or other SQL functions), Vertica returns an error when those objects are used and not when the function is dropped.

Examples

The following command drops the ag_avg function:

```
=> DROP AGGREGATE FUNCTION ag_avg(numeric);
DROP AGGREGATE FUNCTION
```

See also

Aggregate functions (UDAFs)

DROP ANALYTIC FUNCTION

Drops a user-defined analytic function from the Vertica catalog.

Syntax

```
DROP ANALYTIC FUNCTION [ IF EXISTS ] [[database.]schema.]function( [ arglist ] )
```

Parameters

IF EXISTS

Specifies not to report an error if the function to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

Specifies a name of the SQL function to drop. If the function name is schema-qualified, the function is dropped from the specified schema (as noted above).

arglist

A comma delimited list of argument names and data types that are passed to the function, formatted as follows:

```
{ [argname] argtype }[,...]
```

- *argname* optionally specifies the argument name, typically a column name.
- *argtype* specifies the argument's data type, where *argtype* matches a Vertica data type .

Privileges

Non-superuser: Owner

Requirements

- To drop a function, you must specify the argument types because several functions might share the same name with different parameters.
- Vertica does not check for dependencies, so if you drop a SQL function where other objects reference it (such as views or other SQL functions), Vertica returns an error when those objects are used and not when the function is dropped.

Examples

The following command drops the analytic_avg function:

```
=> DROP ANALYTIC FUNCTION analytic_avg(numeric);
DROP ANALYTIC FUNCTION
```

See also

Analytic functions (UDAnFs)

DROP AUTHENTICATION

Drops an authentication method.

Syntax

```
DROP AUTHENTICATION [ IF EXISTS ] auth-method-name [ CASCADE ]
```

Parameters

IF EXISTS

Specifies not to report an error if the authentication method to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping

non-existent objects before attempting to create them.

auth-method-name

Name of the authentication method to drop.

CASCADE

Required if the authentication method to drop is granted to users. In this case, omission of this option causes the drop operation to fail.

Privileges

Superuser

Examples

Delete authentication method md5_auth :

```
=> DROP AUTHENTICATION md5_auth;
```

Use CASCADE to drop authentication method that was granted to a user:

```
=> CREATE AUTHENTICATION localpwd METHOD 'password' LOCAL;
=> GRANT AUTHENTICATION localpwd TO jsmith;
=> DROP AUTHENTICATION localpwd CASCADE;
```

See also

- ALTER AUTHENTICATION
- CREATE AUTHENTICATION
- GRANT (authentication)
- REVOKE (authentication)

DROP CA BUNDLE

> **Deprecated**
> CA bundles are only usable with certain deprecated parameters in Kafka notifiers . You should prefer using TLS configurations and the TLS CONFIGURATION parameter for notifiers instead.

Drops a certificate authority (CA) bundle.

Syntax

```
DROP CA BUNDLE [ IF EXISTS ] name [,...] [ CASCADE ]
```

Parameters

IF EXISTS

Vertica does not report an error if the CA bundle to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

name

The name of the CA bundle.

CASCADE

Drops dependent objects before dropping the CA bundle.

Privileges

Ownership of the CA bundle

Examples

See Managing CA bundles .

See also

- CREATE CA BUNDLE
- ALTER CA BUNDLE

DROP CERTIFICATE

Drops a TLS certificate from the database.

To view existing certificates, query CERTIFICATES .

Syntax

```
DROP CERTIFICATE [ IF EXISTS ] certificate-name [,...] [ CASCADE ]
```

Parameters

IF EXISTS

Vertica does not report an error if the certificate to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

certificate-name

The name of the certificate to drop.

CASCADE

Drops dependent objects before dropping the certificate.

Predefined TLS Configurations and TLS Configurations that manage a connection type cannot be dropped , nor can the keys and certificates referenced by such TLS Configurations. For details, see TLS configurations .

Privileges

Non-superuser, one of the following:

- Ownership of the certificate
- DROP privileges on the private key (when used with DROP KEY...CASCADE)

Examples

Drop server_cert , if it exists:

```
=> DROP CERTIFICATE server_cert;
DROP CERTIFICATE;
```

Drop a CA certificate and its dependencies (typically the certificates that it has signed):

```
=> DROP CERTIFICATE ca_cert CASCADE;
DROP CERTIFICATE;
```

See also

- CREATE CERTIFICATE
- DROP KEY

DROP DIRECTED QUERY

Removes a directed query from the database. If the directed query is active, Vertica deactivates it before removal.

Syntax

```
DROP DIRECTED QUERY { query-name | where-clause }
```

Arguments

query-name

Name of the directed query to remove from the database, as stored in the DIRECTED_QUERIES column query_name . You can also use GET DIRECTED QUERY to obtain names of all directed queries that map to an input query.

where-clause

Resolves to one or more directed queries that are filtered from system table DIRECTED_QUERIES . For example, the following statement specifies to drop all directed queries with the same save_plans_version identifier:

```
=> DROP DIRECTED QUERY WHERE save_plans_version = 21;
```

Privileges

Superuser

Examples

See Dropping directed queries .

DROP FAULT GROUP

Removes the specified fault group and its child fault groups, placing all nodes under the parent of the dropped fault group.

To drop all fault groups, use ALTER DATABASE..DROP ALL FAULT GROUP .

To add an orphaned node back to a fault group, you must manually reassign it to a new or existing fault group with CREATE FAULT GROUP and ALTER FAULT GROUP...ADD NODE .

> **Tip**
> For a list of all fault groups defined in the cluster, query system table FAULT_GROUPS .

Syntax

```
DROP FAULT GROUP [ IF EXISTS ] fault-group
```

Parameters

IF EXISTS

Specifies not to report an error if *fault-group* does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

fault-group

Specifies the name of the fault group to drop.

Privileges

Superuser

Examples

```
=> DROP FAULT GROUP group2;
DROP FAULT GROUP
```

See also

- Fault groups
- High availability with fault groups
- CLUSTER_LAYOUT

DROP FILTER

Drops a User Defined Load Filter function from the Vertica catalog.

Syntax

```
DROP FILTER [[database.]schema.]filter()
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

***filter* ()**

Specifies the filter function to drop. You must append empty parentheses to the function name.

Privileges

Non-superuser:

- Owner or DROP privilege
- USAGE privilege on schema

Examples

The following command drops the lconverter filter function::

```
=> drop filter lconverter();
DROP FILTER
```

See also

- ALTER FUNCTION (scalar)
- CREATE FILTER
- USER_FUNCTIONS
- User-defined load (UDL)

DROP FUNCTION

Drops an SQL function or user-defined functions (UDFs) from the Vertica catalog.

Syntax

```
DROP FUNCTION [ IF EXISTS ] [[database.]schema.]function[,...] ( [ arg-list ] )
```

Parameters

IF EXISTS

Specifies not to report an error if the function to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

The SQL or user-defined function (UDF) to drop, where UDFs can be one of the following types:

- Scalar function (UDSF)
- Analytic function (UDAnF)
- Load (UDL) functions: source , filter , and parser

> **Note**
> You drop aggregate and transformation functions with DROP AGGREGATE FUNCTION and DROP TRANSFORM FUNCTION , respectively.

arg-list

A comma-delimited list of arguments as defined for this function when it was created, specified as follows:
[*arg-name*] *arg-type* [,...]

where *arg-name* optionally qualifies *arg-type* :

- *arg-name* is typically a column name.
- *arg-type* is the name of an SQL data type supported by Vertica.

Privileges

Non-superuser, one of the following:

- Owner or DROP privilege
- USAGE privilege on schema

Requirements

- To drop a function, you must specify the argument types because several functions might share the same name with different parameters.
- Vertica does not check for dependencies when you drop a SQL function, so if other objects reference it (such as views or other SQL functions), Vertica returns an error only when those objects are used.

Examples

The following command drops the zerowhennull function in the macros schema:

```
=>  DROP FUNCTION macros.zerowhennull(x INT);
DROP FUNCTION
```

See also

DROP KEY

Drops a cryptographic key and its certificate, if any, from the database.

To view existing cryptographic keys, query CRYPTOGRAPHIC_KEYS .

Syntax

```
DROP KEY [ IF EXISTS ] key-name [,...] [ CASCADE ]
```

Parameters

IF EXISTS

Vertica does not report an error if the key to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

key-name

The name of the cryptographic key to drop.

CASCADE

Drops dependent objects before dropping the key.

Predefined TLS Configurations and TLS Configurations that manage a connection type cannot be dropped , nor can the keys and certificates referenced by such TLS Configurations. For details, see TLS configurations .

Privileges

Non-superuser, one of the following:

- Ownership of the key
- DROP privileges

Examples

Drop k_ca , if it exists:

```
=> DROP KEY k_ca IF EXISTS;
DROP KEY;
```

Drop k_client and its dependencies (the certificate it's associated with):

```
=> DROP KEY k_client CASCADE;
DROP KEY;
```

See also

- CREATE KEY
- DROP CERTIFICATE

DROP LIBRARY

Removes a UDx library from the database. The library file is deleted from managed directories on the Vertica nodes. The user-defined functions (UDFs) in the library are no longer available. See Developing user-defined extensions (UDxs) for details.

Syntax

```
DROP LIBRARY [ IF EXISTS ] [[database.]schema.]library [  CASCADE]
```

Arguments

IF EXISTS

Execute this command only if the library exists. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

library

The name of the library to drop, the same name used in CREATE LIBRARY to load the library.

CASCADE

Also drop any functions that were defined using the library. DROP LIBRARY fails if CASCADE is omitted and one or more UDxs use the target library.

Privileges

One of:

- Superuser
- UDXDEVELOPER and library owner (the user who created it with CREATE LIBRARY)
- UDXDEVELOPER and DROP grant on the library (see GRANT (library))

Examples

A superuser can drop any library:

```
=> DROP LIBRARY ml.MyLib CASCADE;
```

Users with the UDXDEVELOPER role can drop libraries that they created:

```
=> GRANT UDXDEVELOPER TO alice, bob;
GRANT ROLE

=> \c - alice;
You are now connected as user "alice".

-- Must enable the role before using:
=> SET ROLE UDXDEVELOPER;
SET

-- Create and use ml.mylib...

-- Drop library and dependencies:
DROP LIBRARY ml.mylib CASCADE;
DROP LIBRARY
```

A user can be granted explicit permission to drop a library:

```
=> \c - alice
You are now connected as user "alice".

=> GRANT DROP ON LIBRARY ml.mylib to bob;
GRANT PRIVILEGE

=> \c - bob
You are now connected as user "bob".

=> SET ROLE UDXDEVELOPER;
SET

=> DROP LIBRARY ml.mylib cascade;
DROP LIBRARY
```

DROP LOAD BALANCE GROUP

Deletes a load balancing group.

Syntax

```
DROP LOAD BALANCE GROUP [ IF EXISTS ] group_name [ CASCADE ]
```

Parameters

IF EXISTS

Specifies not to report an error if the load balance group to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

group_name

The name of the group to drop.

[CASCADE]

Also drops all load balancing routing rules that target this group. If you do not supply this keyword and one or more routing rules target *group_name* , this statement fails with an error message.

Privileges

Superuser

Examples

The following statement demonstrates the error you get if the load balancing group has a dependent routing rule, and the use of the CASCADE keyword:

```
=> DROP LOAD BALANCE GROUP group_all;
NOTICE 4927:  The RoutingRule catch_all depends on LoadBalanceGroup group_all
ROLLBACK 3128:  DROP failed due to dependencies
DETAIL:  Cannot drop LoadBalanceGroup group_all because other objects depend on it
HINT:  Use DROP ... CASCADE to drop the dependent objects too

=> DROP LOAD BALANCE GROUP group_all CASCADE;
DROP LOAD BALANCE GROUP
```

See also

DROP MODEL

Removes one or more models from the Vertica database.

Syntax

```
DROP MODEL [ IF EXISTS ] [[database.]schema.]model[,...]
```

Parameters

IF EXISTS

Specifies not to report an error if the models to drop do not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

model

The model to drop.

Privileges

One of the following:

- Superuser
- Non-superuser: model owner

Examples

See Dropping models .

DROP NETWORK ADDRESS

Deletes a network address from the catalog. A network address is a name for a IP address and port on a node for use in connection load balancing policies.

Syntax

```
DROP NETWORK ADDRESS [ IF EXISTS ] address-name [ CASCADE ]
```

Parameters

IF EXISTS

Specifies not to report an error if the network address to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

address-name

Name of the network address to drop.

CASCADE

Removes the network address from any load balancing groups that target it. If you do not supply this keyword and one or more load balance groups include this address, this statement fails with an error message.

Privileges

Superuser

Examples

The following statement demonstrates the error you get if the network address has a dependent load balance group, and the use of the CASCADE keyword:

```
=> DROP NETWORK ADDRESS node01;
NOTICE 4927:  The LoadBalanceGroup group_1 depends on NetworkInterface node01
NOTICE 4927:  The LoadBalanceGroup group_random depends on NetworkInterface node01
ROLLBACK 3128:  DROP failed due to dependencies
DETAIL:  Cannot drop NetworkInterface node01 because other objects depend on it
HINT:  Use DROP ... CASCADE to drop the dependent objects too
=> DROP NETWORK ADDRESS node01 CASCADE;
DROP NETWORK ADDRESS
```

DROP NETWORK INTERFACE

Removes a network interface from Vertica. You can use the CASCADE option to also remove the network interface from any node definition. (See Identify the database or nodes used for import/export for more information.)

Syntax

```
DROP NETWORK INTERFACE [ IF EXISTS ] network-interface-name [ CASCADE ]
```

Parameters

The parameters are defined as follows:

IF EXISTS

Specifies not to report an error if the network interface to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

network-interface-name

The network interface to remove.

CASCADE

Removes the network interface from all node definitions.

Privileges

Superuser

Examples

```
=> DROP NETWORK INTERFACE myNetwork;
```

DROP NOTIFIER

Drops a push-based notifier created by CREATE NOTIFIER .

Syntax

```
DROP NOTIFIER [ IF EXISTS ] notifier-name [ CASCADE ]
```

Parameters

IF EXISTS

Specifies not to report an error if notifier to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

notifier-name

The notifier's unique identifier.

CASCADE

Removes the notifier from any data collector (DC) table policies before dropping the notifier. If the notifier is set for a DC table and CASCADE is not specified, the DROP command fails.

To manually remove the notifier from DC table policies, use the SET_DATA_COLLECTOR_NOTIFY_POLICY function.

Examples

Drop the requests_issued notifier, specifying CASCADE to remove it from any DC table policies:

```
DROP NOTIFIER requests_issued CASCADE;
```

DROP PARSER

Drops a User Defined Load Parser function from the Vertica catalog.

Syntax

```
DROP PARSER[[database.]schema.]parser()
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

***parser* ()**

The name of the parser function to drop. You must append empty parentheses to the function name.

Privileges

Non-superuser:

- Owner or DROP privilege
- USAGE privilege on schema

Examples

```
=> DROP PARSER BasicIntegerParser();
DROP PARSER
```

See also

- ALTER FUNCTION (scalar)
- CREATE PARSER
- GRANT (User Defined Extension)
- REVOKE (User Defined Extension)
- USER_FUNCTIONS
- User-defined load (UDL)

DROP PROCEDURE (external)

Enterprise Mode only

Removes an external procedure from Vertica. Only the reference to the procedure inside Vertica is removed. The external file remains in the *database* / *procedures* directory of each database node.

Syntax

```
DROP PROCEDURE [ IF EXISTS ] [[database.]schema.]procedure( [ parameter-list ] )
```

Parameters

IF EXISTS

Specifies not to report an error if the procedure to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

procedure

Specifies the procedure to drop.

parameter-list

A comma-delimited list of formal parameters defined for this procedure, specified as follows:

[*parameter-name*] *parameter-type* [,...]

where *parameter-name* optionally qualifies *parameter-type* .

Privileges

Non-superuser:

- Owner or DROP privilege
- USAGE privilege on schema

Examples

```
=> DROP PROCEDURE helloplanet(arg1 varchar);
```

See also

CREATE PROCEDURE (external)
DROP PROCEDURE (stored)

Drops a stored procedure .

Syntax

```
DROP PROCEDURE [ IF EXISTS ] [[database.]schema.]procedure( [ parameter-type-list] ) [ CASCADE ];
```

Parameters

IF EXISTS

Specifies not to report an error if the procedure to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

procedure

The name of the stored procedure, where *procedure* conforms to conventions described in Identifiers .

parameter-type-list

A comma-delimited list of the IN parameters' types.

CASCADE

Drops the trigger that references the stored procedure, if any.

Privileges

Non-superuser:

- Owner or DROP privilege
- USAGE privilege on schema

Examples

Given the following procedure:

```
=> CREATE PROCEDURE raiseXY(IN x INT, y VARCHAR) LANGUAGE PLvSQL AS $$
BEGIN
   RAISE NOTICE 'x = %', x;
   RAISE NOTICE 'y = %', y;
   -- some processing statements
END;
$$;

CALL raiseXY(3, 'some string');
NOTICE 2005:  x = 3
NOTICE 2005:  y = some string
```

You can drop it with:

```
=> DROP PROCEDURE raiseXY(INT, VARCHAR);
DROP PROCEDURE
```

For more information on RAISE NOTICE, see Errors and diagnostics .

See also

- CREATE PROCEDURE (stored)
- DROP PROCEDURE (external)

DROP PROFILE

Removes a user-defined profile (created by CREATE PROFILE) from the database. You cannot drop the DEFAULT profile.

Syntax

```
DROP PROFILE [ IF EXISTS ] profile-name[,...] [ CASCADE ]
```

Parameters

IF EXISTS

Specifies not to report an error if the profile to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

profile-name

The profile to drop.

CASCADE

Moves all users assigned to the dropped profiles to the DEFAULT profile. If you omit this option and a targeted profile has users assigned to it, Vertica returns an error.

Privileges

Superuser

Examples

```
=> DROP PROFILE sample_profile;
```

DROP PROJECTION

Marks a projection to drop from the catalog so it is unavailable to user queries.

Syntax

```
DROP PROJECTION [ IF EXISTS ] { [[database.]schema.]projection[,...] } [ RESTRICT | CASCADE ]
```

Parameters

IF EXISTS

Specifies not to report an error if the projection to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

projection

Specifies a projection to drop:

- If the projection is unsegmented, all projection replicas in the database cluster are dropped.
- If the projection is segmented, drop all buddy projections by specifying the projection base name. You can also specify the name of a specific buddy projection as long as dropping it so does not violate system K-safety .

See Projection naming for projection name conventions.

RESTRICT | CASCADE

Specifies whether to drop the projection when it contains objects:

- RESTRICT (default): Drop the projection only if it contains no objects.
- CASCADE : Drop the projection even if it contains objects.

Privileges

Non-superuser: owner of the anchor table

Restrictions

The following restrictions apply to dropping a projection:

- The projection cannot be the anchor table's superprojection .
- You cannot drop a buddy projection if doing so violates system K-safety .
- Another projection must be available to enforce the same primary or unique key constraint.

See also

- CREATE PROJECTION
- GET_PROJECTIONS

DROP RESOURCE POOL

Drops a user-created resource pool. All memory allocated to the pool is returned to the GENERAL pool .

Syntax

```
DROP RESOURCE POOL resource-pool [ FOR { SUBCLUSTER subcluster | CURRENT SUBCLUSTER } ]
```

Parameters

resource-pool

Name of the resource pool to drop.

FOR {SUBCLUSTER *subcluster* | CURRENT SUBCLUSTER}

Eon Mode only, drops *resource-pool* from the specified subcluster, one of the following:

- SUBCLUSTER *subcluster* : Drops *resource-pool* from the named subcluster . You cannot be connected to this subcluster, otherwise Vertica returns an error.
- CURRENT SUBCLUSTER : Drops *resource-pool* from the subcluster you are connected to.

If you omit this parameter, the resource pool is dropped from all subclusters. If a resource pool was created for an individual subcluster, you must explicitly drop it from that subcluster with this parameter; otherwise, Vertica returns an error.

Privileges

Superuser

Resource pool transfers

Requests that are queued against the dropped pool are transferred to the GENERAL pool according to the priority of the pool compared to the GENERAL pool. If the pool's priority is higher than the GENERAL pool, the requests are placed at the head of the queue; otherwise, transferred requests are placed at the end of the queue.

Users who are assigned to the dropped pool are reassigned to the default user resource pool as set by DefaultResourcePoolForUser. The DROP request returns with a notice like this:

```
NOTICE:  Switched the following users to the <name> pool: <username>
```

If any user lacks permission to use the default user resource pool, Vertica rolls back the drop operation.

Existing sessions are transferred to the GENERAL pool regardless of whether the session user has privileges to use that pool. This can result in additional user privileges if access to the dropped pool is more restrictive than the GENERAL pool. In this case, you can prevent giving users additional privileges as follows:

1. Revoke permissions on the target resource pool from all users.
2. Close any sessions that had permissions on the resource pool.
3. Drop the resource pool.

Restrictions

- If you try to drop a resource pool that is the secondary pool for another resource pool, Vertica returns an error. The error lists the resource pools that depend on the secondary pool you tried to drop. To drop a secondary resource pool, first set the CASCADE TO parameter to DEFAULT on the primary resource pool, and then drop the secondary pool.
  For example, you can drop resource pool rp2 , which is a secondary pool for rp1 , as follows:
  ```
  => ALTER RESOURCE POOL rp1 CASCADE TO DEFAULT;
  => DROP RESOURCE POOL rp2;
  ```
- You cannot drop a resource pool that is configured as the default user resource pool by the DefaultResourcePoolForUsers parameter.

Examples

Drop a user-defined resource pool:

```
=> DROP RESOURCE POOL ceo_pool;
```

Get the name of the current subcluster for an Eon Mode database, then drop its resource pool:

```
=> SELECT CURRENT_SUBCLUSTER_NAME();
 CURRENT_SUBCLUSTER_NAME
-------------------------
 analytics_1
(1 row)

=> DROP RESOURCE POOL dashboard FOR CURRENT SUBCLUSTER;
DROP RESOURCE POOL
```

See also

- ALTER RESOURCE POOL
- CREATE RESOURCE POOL

- Managing workloads

DROP ROLE

Removes a role from the database.

> **Note**
> You cannot use DROP ROLE on a role added to the Vertica database with the LDAPLink service.

Syntax

```
DROP ROLE [ IF EXISTS ] role-name[,...] [ CASCADE ]
```

Parameters

IF EXISTS

Specifies not to report an error if the roles to drop do not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

role-name

The name of the role to drop

CASCADE

Revoke the role from users and other roles before dropping the role

Privileges

Superuser

Examples

```
=> DROP ROLE appadmin;
NOTICE:  User bob depends on Role appadmin
ROLLBACK:  DROP ROLE failed due to dependencies
DETAIL:  Cannot drop Role appadmin because other objects depend on it
HINT:  Use DROP ROLE ... CASCADE to remove granted roles from the dependent users/roles
=> DROP ROLE appadmin CASCADE;
DROP ROLE
```

See also

- ALTER ROLE
- CREATE ROLE

DROP ROUTING RULE

Deletes a routing rule from the catalog.

Syntax

```
DROP ROUTING RULE [ IF EXISTS ] rule-name
```

Parameters

`IF EXISTS`

Specifies not to report an error if the routing rule to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

rule-name

Name of the rule to drop.

Privileges

Superuser

Examples

```
=> DROP ROUTING RULE internal_clients;
DROP ROUTING RULE
```

DROP SCHEDULE

Drops schedules .

Syntax

```
DROP SCHEDULE [[database.]schema.]schedule[,...] [ CASCADE ]
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

schedule

The schedule to drop.

CASCADE

Drops triggers that use this schedule, if any.

Privileges

Superuser

Examples

```
=> DROP SCHEDULE monthly_schedule;
```

DROP SCHEMA

Permanently removes a schema from the database. Be sure that you want to remove the schema before you drop it, because DROP SCHEMA is an irreversible process. Use the CASCADE parameter to drop a schema containing one or more objects.

Syntax

```
DROP SCHEMA [ IF EXISTS ] [database.]schema[,...] [ CASCADE | RESTRICT ]
```

Parameters

IF EXISTS

Specifies not to report an error if the schemas to drop do not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Name of the schema to drop. If you specify a database, it must be the current database.

CASCADE

Specifies to drop the schema and all objects in it, regardless of who owns those objects.

> **Caution**
> Objects in other schemas that depend on objects in the dropped schema—for example, user-defined functions—also are silently dropped.

RESTRICT

Drops the schema only if it is empty (default).

Privileges

Non-superuser: schema owner

Restrictions

- You cannot drop the PUBLIC schema.
- If a user is accessing an object within a schema that is in the process of being dropped, the schema is not deleted until the transaction completes.
- Canceling a DROP SCHEMA statement can cause unpredictable results.

Examples

The following example drops schema S1 only if it doesn't contain any objects:

```
=> DROP SCHEMA S1;
```

The following example drops schema S1 whether or not it contains objects:

```
=> DROP SCHEMA S1 CASCADE;
```

DROP SEQUENCE

Removes the specified named sequence number generator.

Syntax

```
DROP SEQUENCE [ IF EXISTS ] [[database.]schema.]sequence[,...]
```

Parameters

IF EXISTS

Specifies not to report an error if the sequences to drop do not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

sequence

Name of the sequence to drop.

Privileges

Non-superusers: sequence or schema owner

Restrictions

- For sequences specified in a table's default expression, the default expression fails the next time you try to load data. Vertica does not check for these instances.
- DROP SEQUENCE does not support the CASCADE keyword. Sequences used in a default expression of a column cannot be dropped until all references to the sequence are removed from the default expression.

Examples

The following command drops the sequence named sequential .

```
=> DROP SEQUENCE sequential;
```

See also

- ALTER SEQUENCE
- CREATE SEQUENCE
- CURRVAL
- GRANT (sequence)
- NEXTVAL
- Sequences

DROP SOURCE

Drops a User Defined Load Source function from the Vertica catalog.

Syntax

```
DROP SOURCE [[database.]schema.]source()
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

***source* ()**

Specifies the source function to drop. You must append empty parentheses to the function name.

Privileges

Non-superuser:

- Owner or DROP privilege
- USAGE privilege on schema

Examples

The following command drops the curl source function:

```
=> DROP SOURCE curl();
DROP SOURCE
```

See also

- ALTER FUNCTION (scalar)
- CREATE SOURCE
- GRANT (User Defined Extension)
- REVOKE (User Defined Extension)
- USER_FUNCTIONS
- User-defined load (UDL)

DROP SUBNET

Removes a subnet from Vertica.

> **Caution**
> Before removing a subnet, be sure your database is not configured to allow export on the public subnet.

Syntax

```
DROP SUBNET [ IF EXISTS ] subnet-name[,...] [ CASCADE ]
```

Parameters

The parameters are defined as follows:

IF EXISTS

Specifies not to report an error if the subnets to drop do not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

subnet-name

A subnet to remove.

CASCADE

Removes the specified subnets from all database definitions.

Privileges

Superuser

Examples

```
=> DROP SUBNET mySubnet;
```

See also

Identify the database or nodes used for import/export

DROP TABLE

Removes one or more tables and their projections . When you run DROP TABLE , the change is auto-committed.

Syntax

```
DROP TABLE [ IF EXISTS ] [ [database.]schema.]table[,...] [ CASCADE ]
```

Parameters

IF EXISTS

Specifies not to report an error if one or more of the tables to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

The table to drop.

CASCADE

Specifies to drop all projections of the target tables. CASCADE is optional if the target tables have only auto-projections . If you omit this option and any of the tables has non-superprojections, Vertica returns an error and rolls back the entire drop operation.

This option is not valid for external tables.

Privileges

Non-superuser:

- Table: owner, or DROP privilege
- Table schema: owner, or USAGE privilege

Requirements

- Do not cancel an executing DROP TABLE . Doing so can leave the database in an inconsistent state.
- Check that the target table is not in use, either directly or indirectly—for example, in a view.
- If you drop and restore a table that is referenced by a view, the new table must have the same name and column definitions.

Examples

See Dropping tables

See also

- DROP PROJECTION
- TRUNCATE TABLE

DROP TEXT INDEX

Drops a text index used to perform text searches.

> **Note**
> When a source table is dropped that has a text index associated with it, the text index is also dropped.

Syntax

```
DROP TEXT INDEX [ IF EXISTS ] [[database.]schema.]idx-table
```

Parameters

IF EXISTS

Specifies not to report an error if the text index to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

idx-table

Specifies the text index name. When using more than one schema, specify the schema that contains the index in the DROP TEXT INDEX statement.

Privileges

- dbadmin
- Table owner
- DROP privileges on the source table

Examples

```
=> DROP TEXT INDEX t_text_index;
DROP INDEX
```

See also

- Using text search
- CREATE TEXT INDEX

DROP TLS CONFIGURATION

Drops an existing TLS Configuration.

You cannot drop a TLS Configuration if it set as a configuration parameter . For details, see TLS configurations .

Syntax

```
DROP TLS CONFIGURATION tls_config_name
```

Parameters

tls_config_name

The name of the TLS Configuration object to drop.

Privileges

Non-superuser, one of the following:

- Ownership of the TLS Configuration
- DROP privileges

DROP TRANSFORM FUNCTION

Drops a user-defined transform function (UDTF) from the Vertica catalog.

Syntax

```
DROP TRANSFORM FUNCTION [ IF EXISTS ] [[database.]schema.]function( [ arg-list ] )
```

Parameters

IF EXISTS

Specifies not to report an error if the function to drop does not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

Specifies the transform function to drop.

arg-list

A comma-delimited list of arguments as defined for this function when it was created, specified as follows:
[*arg-name*] *arg-type* [,...]

where *arg-name* optionally qualifies *arg-type* :

- *arg-name* is typically a column name.
- *arg-type* is the name of an SQL data type supported by Vertica.

> **Note**
> You can omit *arg-list* when dropping a polymorphic function.

Privileges

One of the following:

- Superuser
- Schema or function owner

Examples

The following command drops the tokenize UDTF in the macros schema:

```
=> DROP TRANSFORM FUNCTION macros.tokenize(varchar);
DROP TRANSFORM FUNCTION
```

The following command drops the Pagerank polymorphic function in the online schema:

```
=> DROP TRANSFORM FUNCTION online.Pagerank();
DROP TRANSFORM FUNCTION
```

See also

CREATE TRANSFORM FUNCTION
DROP TRIGGER

Drops triggers . Dropping a trigger disables its associated schedule , if any.

Syntax

```
DROP TRIGGER [[database.]schema.]trigger[,...]
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

trigger

The trigger to drop.

Privileges

Superuser

Examples

To drop a trigger:

```
=> DROP TRIGGER revoke_trigger;
```

DROP USER

Removes a name from the list of authorized database users.

> **Note**
> DROP USER can not remove a user that was added to the Vertica database with the LDAPLink service.

Syntax

```
DROP USER [ IF EXISTS ] user-name[,...] [ CASCADE ]
```

Parameters

IF EXISTS

Do not report an error if the users to drop do not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

user-name

Name of a user to drop.

CASCADE

Drop all user-defined objects created by *user-name* , including schemas, tables and all views that reference the table, and projections of that table.

> **Caution**
> Tables owned by the dropped user cannot be recovered after you issue DROP USER CASCADE.

Privileges

Superuser

Examples

DROP USER succeeds if no user-defined objects exist:

```
=> CREATE USER user2;
CREATE USER
=> DROP USER IF EXISTS user2;
DROP USER
```

DROP USER fails if objects that the user created still exist:

```
=> DROP USER IF EXISTS user1;
NOTICE:  Table T_tbd1 depends on User user1
ROLLBACK:  DROP failed due to dependencies
DETAIL:  Cannot drop User user1 because other objects depend on it
HINT:  Use DROP ... CASCADE to drop the dependent objects too
```

DROP USER CASCADE succeeds regardless of any existing user-defined objects. The statement forcibly drops all user-defined objects, such as schemas, tables and their associated projections:

```
=> DROP USER IF EXISTS user1 CASCADE;
DROP USER
```

See also

- ALTER USER
- CREATE USER

DROP VIEW

Removes the specified view. Vertica does not check for dependencies on the dropped view. After dropping a view, other views that reference it fail.

If you drop a view and replace it with another view or table with the same name and column names, other views that reference that name use the new view. If you change the column data type in the new view, the server coerces the old data type to the new one if possible; otherwise, it returns an error.

Syntax

```
DROP VIEW [ IF EXISTS ] [[database.]schema.]view[,...]
```

Parameters

IF EXISTS

Specifies not to report an error if the views to drop do not exist. Use this clause in SQL scripts to avoid errors on dropping non-existent objects before attempting to create them.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

view

Name of a view to drop.

Privileges

One of the following

- View owner and USAGE privileges
- Schema owner

Examples

```
=> DROP VIEW myview;
```

END

Ends the current transaction and makes all changes that occurred during the transaction permanent and visible to other users.

> **Note**
> COMMIT is a synonym for END.

Syntax

```
END [ WORK | TRANSACTION ]
```

Parameters

WORK | TRANSACTION

Optional keywords that have no effect, for readability only.

Privileges

None

Examples

```
=> BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
BEGIN
=> CREATE TABLE sample_table (a INT);
CREATE TABLE
=> INSERT INTO sample_table (a) VALUES (1);
OUTPUT
--------
1
(1 row)

=> END;
COMMIT
```

See also

- Transactions
- Creating and rolling back transactions
- BEGIN
- ROLLBACK
- START TRANSACTION

EXPLAIN

Returns a formatted description of the Vertica optimizer's plan for executing the specified statement.

Syntax

```
EXPLAIN [/*+ ALLNODES */] [explain-options] sql-statement
```

Parameters

/*+ ALLNODES */

Specifies to create a query plan that assumes all nodes are active, not valid with LOCAL option.

explain-options

One or more EXPLAIN options, specified in the order shown:

```
[ LOCAL ] [ VERBOSE ] [ JSON ] [ ANNOTATED ]
```

- LOCAL : On a multi-node database, shows the local query plans assigned to each node, which together comprise the total (global) query plan. If you omit this option, Vertica shows only the global query plan. Local query plans are shown only in DOT language source, which can be rendered in Graphviz .
  This option is incompatible with the hint /*+ALLNODES*/ . If you specify both, EXPLAIN returns with an error.
- VERBOSE : Increases the level of detail in the rendered query plan.
- JSON : Renders the query plan in JSON format. This option is compatible only with VERBOSE .
- ANNOTATED : Embeds optimizer hints that encapsulate the query plan for this query. This option is compatible with LOCAL and VERBOSE .

sql-statement

A query or DML statement—for example, SELECT , INSERT , UPDATE , COPY , and MERGE .

Privileges

The same privileges required by the specified statement.

Requirements

The following requirements apply to EXPLAIN's ability to produce useful information:

- Reasonably representative statistics of your data must be available. See Collecting Statistics for details.
- EXPLAIN produces useful output only if projections are available for the queried tables.
- Qualifier options must be specified in the order shown earlier , otherwise EXPLAIN returns with an error. If an option is incompatible with any preceding options, EXPLAIN ignores them.

Examples

See Viewing query plans .

EXPORT TO DELIMITED

Exports a table, columns from a table, or query results to delimited files. The files can be read back in using DELIMITED . Several exporter parameters have corresponding parser parameters, allowing you to change delimiters, null indicators, and other formatting.

There are some limitations on the queries you can use in an export statement. See Query Restrictions .

You can export data stored in Vertica in ROS format and data from external tables.

This statement returns the number of rows written and logs information about exported files in a system table. See Monitoring exports .

During an export to HDFS or an NFS mount point, Vertica writes files to a temporary directory in the same location as the destination and renames the directory when the export is complete. Do not attempt to use the files in the temporary directory. During an export to S3, GCS, or Azure, Vertica writes files directly to the destination path, so you must wait for the export to finish before reading the files. For more information, see Exporting to object stores .

Syntax

```
EXPORT [ /*+LABEL (label-string)*/ ] TO DELIMITED
  ( directory='path'[, param=value[,...] ] )
  [ OVER (over-clause ) ] AS SELECT query-expression
```

Arguments

LABEL

Assigns a label to a statement to identify it for profiling and debugging.

over-clause

Specifies how to partition table data using PARTITION BY. Within partitions you can sort using ORDER BY. See SQL analytics . This clause may contain column references but not expressions.

If you partition data, Vertica creates a partition directory structure, transforming column names to lowercase. See Partitioned file paths for a description of the directory structure. If you use the fileName parameter, you cannot use partitioning.
If you omit this clause, Vertica optimizes for maximum parallelism.

query-expression

Specifies the data to export. See Query Restrictions for important limitations.

Parameters

directory

The destination directory for the output files. The directory must not exist, and the current user must have permission to write it. The destination can be on any of the following file systems:

- HDFS file system
- S3 object store
- Google Cloud Storage (GCS) object store
- Azure Blob Storage object store
- Linux file system, either an NFS mount or local storage on each node

filename

If specified, all output is written to a single file of this name in the location specified by directory . While the query can be processed by multiple nodes, only a single node generates the output data. The fileSizeMB parameter is ignored, and the query cannot use partitioning in the OVER() clause.

addHeader

If true, add a header row to the beginning of each file.

Default: false

delimiter

Column delimiter character. To produce CSV in accordance with RFC 4180, set the delimiter character to , (comma).

Default: | (vertical bar)

escapeDelimitersInsideEnclosures

If true, delimiter characters are escaped within values that are enclosed by enclosedBy . This parameter is most relevant when enclosedBy and escapeAs have the same value. A value of false only applies when enclosedBy has a value (is not an empty string).

Default: true

recordTerminator

Character that marks the record end.

Default: \n

enclosedBy

Character to use to enclose string and date/time data. If you omit this parameter, no character encloses these data types.

Default: '' (empty string, no enclosing character)

escapeAs

Character to use to escape values in exported data that must be escaped, including the enclosedBy value.

Default: \ (backslash)

nullAs

String to represent null values in the data. If this parameter is included, the exporter exports all null values as this value. Otherwise, the exporter exports null values as zero-length strings.

binaryTypesFormat

Format for exported binary data type (BINARY, VARBINARY, and LONG VARBINARY) values, one of the following:

- Default : Printable ASCII characters where possible and escaped octal representations of the non-printable bytes. The DELIMITED parser reads this format.
- Hex : Base 16 (hexadecimal) representation; value is preceded by '0x' and bytes are not escaped.
- Octal : Base 8 (octal) representation, without escaping.
- Bitstring : Binary representation, without escaping.

For example, the value a\000b\001c can be exported as follows:

- Default (assuming an escape character of \ ): a\\000b\\001c
- Hex: 0x6100620163
- Octal: 141000142001143
- Bitstring: 01100001000000000011000100000000101100011

compression

Compression type, one of:

- Uncompressed
- BZip
- GZip

Default: Uncompressed

fileExtension

Output file extension. If using compression, a compression-specific extension such as .bz2 is appended.

Default: csv

fileSizeMB

The maximum file size of a single output file. This value is a hint, not a hard limit. A value of 0 specifies no limit. If filename is also specified, fileSizeMB is ignored.
This value affects the size of individual output files, not the total output size. For smaller values, Vertica divides the output into more files; all data is still exported.

Default: 10GB

fileMode

For writes to HDFS only, permission to apply to all exported files. You can specify the value in Unix octal format (such as 665) or *user - group - other* format—for example, rwxr-xr-x . The value must be formatted as a string even if using the octal format.
Valid octal values range between 0 and 1777 , inclusive. See HDFS Permissions in the Apache Hadoop documentation.

When writing files to any destination other than HDFS, this parameter has no effect.

Default: 660 , regardless of the value of fs.permissions.umask-mode in hdfs-site.xml .

dirMode

For writes to HDFS only, permission to apply to all exported directories. Values follow the same rules as those for *fileMode* . Further, you must give the Vertica HDFS user full permission, at least rwx------ or 700 .
When writing files to any destination other than HDFS, this parameter has no effect.

Default: 755 , regardless of the value of fs.permissions.umask-mode in hdfs-site.xml .

Privileges

Non-superusers:

- Source table: SELECT
- Source table schema: USAGE
- Destination directory: Write

Query restrictions

You must provide an alias column label for selected column targets that are expressions.

If you partition the output, you cannot specify schema and table names in the SELECT statement. Specify only the column name.

The query can contain only a single outer SELECT statement. For example, you cannot use UNION:

```
=> EXPORT TO DELIMITED(directory = '/mnt/shared_nfs/accounts/rm')
  OVER(PARTITION BY hash)
  AS
  SELECT 1 as account_id, '{}' as json, 0 hash
  UNION ALL
  SELECT 2 as account_id, '{}' as json, 1 hash;
ERROR 8975:  Only a single outer SELECT statement is supported
HINT:  Please use a subquery for multiple outer SELECT statements
```

Instead, rewrite the query to use a subquery:

```
=> EXPORT TO DELIMITED(directory = '/mnt/shared_nfs/accounts/rm')
  OVER(PARTITION BY hash)
  AS
  SELECT
   account_id,
   json
  FROM
  (
    SELECT 1 as account_id, '{}' as json, 0 hash
    UNION ALL
    SELECT 2 as account_id, '{}' as json, 1 hash
  ) a;
 Rows Exported
---------------
             2
(1 row)
```

To use composite statements such as UNION, INTERSECT, and EXCEPT, rewrite them as subqueries.

Data types

EXPORT TO DELIMITED does not support ARRAY, ROW, and SET types.

This operation exports raw Flex columns as binary data.

Output

The export operation always creates an output directory, even if all output is written to a single file or the query produces zero rows.

Output file names follow the pattern: [*8-character-hash*]-[*nodename*]-[*thread-id*]. *fileExtension* .

Column names in partition directories are lowercase.

Files exported to a local file system by any Vertica user are owned by the Vertica superuser. Files exported to HDFS or object stores are owned by the Vertica user who exported the data.

Making concurrent exports to the same output destination is an error and can produce incorrect results.

Exports to the local file system can be to an NFS mount (shared) or to the Linux file system on each node (non-shared). For details, see Exporting to the Linux file system . Exports to non-shared local file systems have the following restrictions:

- The output directory must not exist on any node.
- You must have a USER storage location or superuser privileges.
- You cannot override the permissions mode of 700 for directories and 600 for files.

Exports to object-store file systems are not atomic. Be careful to wait for the export to finish before using the data. For details, see Exporting to object stores .

Examples

The following example exports uncompressed comma-separated values (CSV) with a header row in each file:

```
=> EXPORT TO DELIMITED(directory='webhdfs:///user1/data', delimiter=',', addHeader='true')
 AS SELECT * FROM public.sales;
```

EXPORT TO JSON

Exports a table, columns from a table, or query results to JSON files. The files can be read back into Vertica using FJSONPARSER .

There are some limitations on the queries you can use in an export statement. See Query Restrictions .

You can export data stored in Vertica in ROS format and data from external tables.

This statement returns the number of rows written and logs information about exported files in a system table. See Monitoring exports .

During an export to HDFS or an NFS mount point, Vertica writes files to a temporary directory in the same location as the destination and renames the directory when the export is complete. Do not attempt to use the files in the temporary directory. During an export to S3, GCS, or Azure, Vertica writes files directly to the destination path, so you must wait for the export to finish before reading the files. For more information, see Exporting to object stores .

Syntax

```
EXPORT [ /*+LABEL (label)*/ ] TO JSON
  ( directory='path'[, param=value[,...] ] )
  [ OVER (over-clause ) ] AS SELECT query-expression
```

Arguments

LABEL

Assigns a label to a statement to identify it for profiling and debugging.

over-clause

Specifies how to partition table data using PARTITION BY. Within partitions you can sort using ORDER BY. See SQL analytics . This clause may contain column references but not expressions.

If you partition data, Vertica creates a partition directory structure, transforming column names to lowercase. See Partitioned file paths for a description of the directory structure. If you use the fileName parameter, you cannot use partitioning.

If you omit this clause, Vertica optimizes for maximum parallelism.

query-expression

Specifies the data to export. See Query Restrictions for important limitations.

Parameters

directory

The destination directory for the output files. The directory must not exist, and the current user must have permission to write it. The destination can be on any of the following file systems:

- HDFS file system
- S3 object store
- Google Cloud Storage (GCS) object store
- Azure Blob Storage object store

- Linux file system, either an NFS mount or local storage on each node

filename

If specified, all output is written to a single file of this name in the location specified by directory . While the query can be processed by multiple nodes, only a single node generates the output data. The fileSizeMB parameter is ignored, and the query cannot use partitioning in the OVER() clause.

omitNullFields

Boolean, whether to omit ROW fields with null values.

Default: false

compression

Compression type, one of:

- Uncompressed
- BZip
- GZip

Default: Uncompressed

fileSizeMB

The maximum file size of a single output file. This value is a hint, not a hard limit. A value of 0 specifies no limit. If filename is also specified, fileSizeMB is ignored.
This value affects the size of individual output files, not the total output size. For smaller values, Vertica divides the output into more files; all data is still exported.

Default: 10GB

fileMode

For writes to HDFS only, permission to apply to all exported files. You can specify the value in Unix octal format (such as 665) or *user - group - other* format—for example, rwxr-xr-x . The value must be formatted as a string even if using the octal format.
Valid octal values range between 0 and 1777 , inclusive. See HDFS Permissions in the Apache Hadoop documentation.

When writing files to any destination other than HDFS, this parameter has no effect.

Default: 660 , regardless of the value of fs.permissions.umask-mode in hdfs-site.xml .

dirMode

For writes to HDFS only, permission to apply to all exported directories. Values follow the same rules as those for *fileMode* . Further, you must give the Vertica HDFS user full permission, at least rwx------ or 700 .
When writing files to any destination other than HDFS, this parameter has no effect.

Default: 755 , regardless of the value of fs.permissions.umask-mode in hdfs-site.xml .

Privileges

Non-superusers:

- Source table: SELECT
- Source table schema: USAGE
- Destination directory: Write

Query restrictions

You must provide an alias column label for selected column targets that are expressions.

If you partition the output, you cannot specify schema and table names in the SELECT statement. Specify only the column name.

The query can contain only a single outer SELECT statement. For example, you cannot use UNION:

```
=> EXPORT TO JSON(directory = '/mnt/shared_nfs/accounts/rm')
  OVER(PARTITION BY hash)
  AS
  SELECT 1 as account_id, '{}' as json, 0 hash
  UNION ALL
  SELECT 2 as account_id, '{}' as json, 1 hash;
ERROR 8975:  Only a single outer SELECT statement is supported
HINT:  Please use a subquery for multiple outer SELECT statements
```

Instead, rewrite the query to use a subquery:

```
=> EXPORT TO JSON(directory = '/mnt/shared_nfs/accounts/rm')
  OVER(PARTITION BY hash)
  AS
  SELECT
   account_id,
   json
  FROM
  (
    SELECT 1 as account_id, '{}' as json, 0 hash
    UNION ALL
    SELECT 2 as account_id, '{}' as json, 1 hash
  ) a;
 Rows Exported
---------------
             2
(1 row)
```

To use composite statements such as UNION, INTERSECT, and EXCEPT, rewrite them as subqueries.

Data types

EXPORT TO JSON can export ARRAY and ROW types in any combination.

EXPORT TO JSON does not support binary output (VARBINARY).

Output

The export operation always creates an output directory, even if all output is written to a single file or the query produces zero rows.

Output file names follow the pattern: [*8-character-hash*]-[*nodename*]-[*thread-id*].json .

Column names in partition directories are lowercase.

Files exported to a local file system by any Vertica user are owned by the Vertica superuser. Files exported to HDFS or object stores are owned by the Vertica user who exported the data.

Making concurrent exports to the same output destination is an error and can produce incorrect results.

Exports to the local file system can be to an NFS mount (shared) or to the Linux file system on each node (non-shared). For details, see Exporting to the Linux file system . Exports to non-shared local file systems have the following restrictions:

- The output directory must not exist on any node.
- You must have a USER storage location or superuser privileges.
- You cannot override the permissions mode of 700 for directories and 600 for files.

Exports to object-store file systems are not atomic. Be careful to wait for the export to finish before using the data. For details, see Exporting to object stores .

Examples

In the following example, one of the ROW elements has a null value, which is omitted in the output. EXPORT TO JSON writes each JSON record on one line; line breaks have been inserted into the following output for readability:

```
=> SELECT name, menu FROM restaurants;
      name       |                                menu

-----------------+-------------------------------------------------------------
-----------------
 Bob's pizzeria    | [{"item":"cheese pizza","price":null},{"item":"spinach pizza","price":10.5}]
 Bakersfield Tacos | [{"item":"veggie taco","price":9.95},{"item":"steak taco","price":10.95}]
(2 rows)

=> EXPORT TO JSON (directory='/output/json', omitNullFields=true)
   AS SELECT * FROM restaurants;
 Rows Exported
---------------
             2
(1 row)

=> \! cat /output/json/*.json
{"name":"Bob's pizzeria","cuisine":"Italian","location_city":["Cambridge","Pittsburgh"],
 "menu":[{"item":"cheese pizza"},{"item":"spinach pizza","price":10.5}]}
{"name":"Bakersfield Tacos","cuisine":"Mexican","location_city":["Pittsburgh"],
 "menu":[{"item":"veggie taco","price":9.95},{"item":"steak taco","price":10.95}]}
```

EXPORT TO ORC

Exports a table, columns from a table, or query results to files in the ORC format.

You can use an OVER() clause to partition the data before export. You can partition data instead of or in addition to exporting the column data. Partitioning data can improve query performance by enabling partition pruning. See Partitioned file paths .

There are some limitations on the queries you can use in an export statement. See Query Restrictions .

You can export data stored in Vertica in ROS format and data from external tables.

This statement returns the number of rows written and logs information about exported files in a system table. See Monitoring exports .

During an export to HDFS or an NFS mount point, Vertica writes files to a temporary directory in the same location as the destination and renames the directory when the export is complete. Do not attempt to use the files in the temporary directory. During an export to S3, GCS, or Azure, Vertica writes files directly to the destination path, so you must wait for the export to finish before reading the files. For more information, see Exporting to object stores .

Syntax

```
EXPORT [ /*+LABEL (label-string)*/ ] TO ORC
  ( directory='path'[, param=value[,...] ] )
  [ OVER (over-clause ) ] AS SELECT query-expression
```

Arguments

LABEL

Assigns a label to a statement to identify it for profiling and debugging.

over-clause

Specifies how to partition table data using PARTITION BY. Within partitions you can sort using ORDER BY. See SQL analytics . This clause may contain column references but not expressions.

If you partition data, Vertica creates a partition directory structure, transforming column names to lowercase. See Partitioned file paths for a description of the directory structure. If you use the fileName parameter, you cannot use partitioning.

If you omit this clause, Vertica optimizes for maximum parallelism.

query-expression

Specifies the data to export. See Query Restrictions for important limitations.

Parameters

directory

The destination directory for the output files. The directory must not exist, and the current user must have permission to write it. The destination can be on any of the following file systems:

- HDFS file system
- S3 object store
- Google Cloud Storage (GCS) object store
- Azure Blob Storage object store
- Linux file system, either an NFS mount or local storage on each node

filename

If specified, all output is written to a single file of this name in the location specified by directory . While the query can be processed by multiple nodes, only a single node generates the output data. The fileSizeMB parameter is ignored, and the query cannot use partitioning in the OVER() clause.

compression

Column compression type, one of:

- Zlib
- Uncompressed

Default: Zlib

stripeSizeMB

The uncompressed size of exported stripes in MB, an integer value between 1 and 1024, inclusive.

Default: 250

rowIndexStride

Integer that specifies how frequently the exporter builds indexing statistics in the output, between 1 and 1000000 (1 million), inclusive. A value of 0 disables indexing. The exporter builds statistics after every rowIndexStride rows in each stripe, or once for stripes < rowIndexStride .

Default: 1000

fileSizeMB

The maximum file size of a single output file. This value is a hint, not a hard limit. A value of 0 specifies no limit. If filename is also specified, fileSizeMB is ignored.
This value affects the size of individual output files, not the total output size. For smaller values, Vertica divides the output into more files; all data is still exported.

Default: 10GB

fileMode

For writes to HDFS only, permission to apply to all exported files. You can specify the value in Unix octal format (such as 665) or *user - group - other* format—for example, rwxr-xr-x . The value must be formatted as a string even if using the octal format.
Valid octal values range between 0 and 1777 , inclusive. See HDFS Permissions in the Apache Hadoop documentation.

When writing files to any destination other than HDFS, this parameter has no effect.

Default: 660 , regardless of the value of fs.permissions.umask-mode in hdfs-site.xml .

dirMode

For writes to HDFS only, permission to apply to all exported directories. Values follow the same rules as those for *fileMode* . Further, you must give the Vertica HDFS user full permission, at least rwx------ or 700 .
When writing files to any destination other than HDFS, this parameter has no effect.

Default: 755 , regardless of the value of fs.permissions.umask-mode in hdfs-site.xml .

Privileges

Non-superusers:

- Source table: SELECT
- Source table schema: USAGE
- Destination directory: Write

Query restrictions

You must provide an alias column label for selected column targets that are expressions.

If you partition the output, you cannot specify schema and table names in the SELECT statement. Specify only the column name.

The query can contain only a single outer SELECT statement. For example, you cannot use UNION:

```
=> EXPORT TO ORC(directory = '/mnt/shared_nfs/accounts/rm')
  OVER(PARTITION BY hash)
  AS
  SELECT 1 as account_id, '{}' as json, 0 hash
  UNION ALL
  SELECT 2 as account_id, '{}' as json, 1 hash;
ERROR 8975:  Only a single outer SELECT statement is supported
HINT:  Please use a subquery for multiple outer SELECT statements
```

Instead, rewrite the query to use a subquery:

```
=> EXPORT TO ORC(directory = '/mnt/shared_nfs/accounts/rm')
  OVER(PARTITION BY hash)
  AS
  SELECT
   account_id,
   json
  FROM
  (
    SELECT 1 as account_id, '{}' as json, 0 hash
    UNION ALL
    SELECT 2 as account_id, '{}' as json, 1 hash
  ) a;
 Rows Exported
---------------
             2
(1 row)
```

To use composite statements such as UNION, INTERSECT, and EXCEPT, rewrite them as subqueries.

Data types

EXPORT TO ORC converts Vertica data types to Hive data types as shown in the following table.

Vertica Data Type	**Hive Data Type**
INTEGER, BIGINT	BIGINT
FLOAT, DECIMAL, SMALLINT, TINYINT, CHAR, BOOLEAN	Corresponding Hive type
VARCHAR, LONG VARCHAR	VARCHAR (max 64KB) or STRING (can be read as either)
BINARY, VARBINARY, LONG VARBINARY	BINARY
DATE	DATE if supported by your version of Hive, otherwise INT96 (can be read as TIMESTAMP)
TIMESTAMP, TIMESTAMPTZ	TIMESTAMP. Vertica does not convert TIMESTAMP values to UTC. To avoid problems arising from time zones, use TIMESTAMPTZ instead of TIMESTAMP.
TIME, TIMEZ, INTERVAL, UUID	Not supported
ARRAY, SET	Not supported
ROW	Not supported

Decimal precision must be <= 38.

The exported Hive types might not be identical to the Vertica types. For example, a Vertica INT is exported as a Hive BIGINT. When defining Hive external tables to read exported data, you might have to adjust column definitions.

This operation exports raw Flex columns as binary data.

Output

The export operation always creates an output directory, even if all output is written to a single file or the query produces zero rows.

Output file names follow the pattern: [*8-character-hash*]-[*nodename*]-[*thread-id*].orc .

Column names in partition directories are lowercase.

Files exported to a local file system by any Vertica user are owned by the Vertica superuser. Files exported to HDFS or object stores are owned by the Vertica user who exported the data.

Making concurrent exports to the same output destination is an error and can produce incorrect results.

Exports to the local file system can be to an NFS mount (shared) or to the Linux file system on each node (non-shared). For details, see Exporting to the Linux file system . Exports to non-shared local file systems have the following restrictions:

- The output directory must not exist on any node.
- You must have a USER storage location or superuser privileges.
- You cannot override the permissions mode of 700 for directories and 600 for files.

Exports to object-store file systems are not atomic. Be careful to wait for the export to finish before using the data. For details, see Exporting to object stores .

Examples

The following example demonstrates partitioning and exporting data. EXPORT TO ORC first partitions the data on region and then, within each partition, sorts by store.

```
=> EXPORT TO ORC(directory='gs://DataLake/user2/data')
  OVER(PARTITION BY store.region ORDER BY store.ID)
  AS SELECT sale.price, sale.date, store.ID
  FROM public.sales sale
  JOIN public.vendor store ON sale.distribID = store.ID;
```

For more examples, see EXPORT TO PARQUET , which (aside from a few parameters) behaves the same as EXPORT TO ORC.

EXPORT TO PARQUET

Exports a table, columns from a table, or query results to files in the Parquet format.

You can use an OVER() clause to partition the data before export. You can partition data instead of or in addition to exporting the column data. Partitioning data can improve query performance by enabling partition pruning. See Partitioned file paths .

There are some limitations on the queries you can use in an export statement. See Query Restrictions .

You can export data stored in Vertica in ROS format and data from external tables.

This statement returns the number of rows written and logs information about exported files in a system table. See Monitoring exports .

During an export to HDFS or an NFS mount point, Vertica writes files to a temporary directory in the same location as the destination and renames the directory when the export is complete. Do not attempt to use the files in the temporary directory. During an export to S3, GCS, or Azure, Vertica writes files directly to the destination path, so you must wait for the export to finish before reading the files. For more information, see Exporting to object stores .

After you export data, you can use the GET_METADATA function to inspect the results.

Syntax

```
EXPORT [ /*+LABEL (label-string)*/ ] TO PARQUET
  ( directory='path'[, param=value[,...] ] )
  [ OVER (over-clause ) ] AS SELECT query-expression
```

Arguments

LABEL

Assigns a label to a statement to identify it for profiling and debugging.

over-clause

Specifies how to partition table data using PARTITION BY. Within partitions you can sort using ORDER BY. See SQL analytics . This clause may contain column references but not expressions.

If you partition data, Vertica creates a partition directory structure, transforming column names to lowercase. See Partitioned file paths for a description of the directory structure. If you use the fileName parameter, you cannot use partitioning.
If you omit this clause, Vertica optimizes for maximum parallelism.

query-expression

Specifies the data to export. See Query Restrictions for important limitations.

Parameters

directory

The destination directory for the output files. The directory must not exist, and the current user must have permission to write it. The destination can be on any of the following file systems:

- HDFS file system
- S3 object store
- Google Cloud Storage (GCS) object store
- Azure Blob Storage object store
- Linux file system, either an NFS mount or local storage on each node

filename

If specified, all output is written to a single file of this name in the location specified by directory . While the query can be processed by multiple nodes, only a single node generates the output data. The fileSizeMB parameter is ignored, and the query cannot use partitioning in the OVER() clause.

compression

Column compression type, one of:

- Snappy
- GZIP
- Brotli
- ZSTD
- Uncompressed

Default: Snappy

rowGroupSizeMB

The uncompressed size of exported row groups, in MB, an integer value between 1 and fileSizeMB , inclusive, or unlimited if fileSizeMB is 0.

The row groups in the exported files are smaller than this value because Parquet files are compressed on write. For best performance when exporting to HDFS, set *size* to be smaller than the HDFS block size.
Row-group size affects memory consumption during export. An export thread consumes at least double the row-group size. The default value of 512MB is a compromise between writing larger row groups and allowing enough free memory for other Vertica operations. If you perform exports when the database is not otherwise under heavy load, you can improve read performance on the exported data by increasing row-group size on export. However, row groups that span multiple blocks on HDFS decrease read performance by requiring more I/O, so do not set the row-group size to be larger than your HDFS block size.

Default: 512

fileSizeMB

The maximum file size of a single output file. This value is a hint, not a hard limit. A value of 0 specifies no limit. If filename is also specified, fileSizeMB is ignored.
This value affects the size of individual output files, not the total output size. For smaller values, Vertica divides the output into more files; all data is still exported.

Default: 10GB

fileMode

For writes to HDFS only, permission to apply to all exported files. You can specify the value in Unix octal format (such as 665) or *user - group - other* format—for example, rwxr-xr-x . The value must be formatted as a string even if using the octal format.
Valid octal values range between 0 and 1777 , inclusive. See HDFS Permissions in the Apache Hadoop documentation.

When writing files to any destination other than HDFS, this parameter has no effect.

Default: 660 , regardless of the value of fs.permissions.umask-mode in hdfs-site.xml .

dirMode

For writes to HDFS only, permission to apply to all exported directories. Values follow the same rules as those for *fileMode* . Further, you must give the Vertica HDFS user full permission, at least rwx------ or 700 .
When writing files to any destination other than HDFS, this parameter has no effect.

Default: 755 , regardless of the value of fs.permissions.umask-mode in hdfs-site.xml .

int96AsTimestamp

Boolean, specifies whether to export timestamps as int96 physical type (true) or int64 physical type (false).

Default: true

Privileges

Non-superusers:

- Source table: SELECT
- Source table schema: USAGE
- Destination directory: Write

Query restrictions

You must provide an alias column label for selected column targets that are expressions.

If you partition the output, you cannot specify schema and table names in the SELECT statement. Specify only the column name.

The query can contain only a single outer SELECT statement. For example, you cannot use UNION:

```
=> EXPORT TO PARQUET(directory = '/mnt/shared_nfs/accounts/rm')
  OVER(PARTITION BY hash)
  AS
  SELECT 1 as account_id, '{}' as json, 0 hash
  UNION ALL
  SELECT 2 as account_id, '{}' as json, 1 hash;
ERROR 8975:  Only a single outer SELECT statement is supported
HINT:  Please use a subquery for multiple outer SELECT statements
```

Instead, rewrite the query to use a subquery:

```
=> EXPORT TO PARQUET(directory = '/mnt/shared_nfs/accounts/rm')
  OVER(PARTITION BY hash)
  AS
  SELECT
   account_id,
   json
  FROM
  (
    SELECT 1 as account_id, '{}' as json, 0 hash
    UNION ALL
    SELECT 2 as account_id, '{}' as json, 1 hash
  ) a;
 Rows Exported
---------------
             2
(1 row)
```

To use composite statements such as UNION, INTERSECT, and EXCEPT, rewrite them as subqueries.

Data types

EXPORT TO PARQUET converts Vertica data types to Hive data types as shown in the following table.

Vertica Data Type	Hive Data Type
INTEGER, BIGINT	BIGINT
FLOAT, DECIMAL, SMALLINT, TINYINT, CHAR, BOOLEAN	Corresponding Hive type
VARCHAR, LONG VARCHAR	VARCHAR (max 64KB) or STRING (can be read as either)
BINARY, VARBINARY, LONG VARBINARY	BINARY
DATE	DATE if supported by your version of Hive, otherwise INT96 (can be read as TIMESTAMP)
TIMESTAMP, TIMESTAMPTZ	TIMESTAMP. Vertica does not convert TIMESTAMP values to UTC. To avoid problems arising from time zones, use TIMESTAMPTZ instead of TIMESTAMP.
TIME, TIMEZ, INTERVAL	Not supported
ARRAY	ARRAY
SET	Not supported
ROW	STRUCT

Decimal precision must be <= 38.

The exported Hive types might not be identical to the Vertica types. For example, a Vertica INT is exported as a Hive BIGINT. When defining Hive external tables to read exported data, you might have to adjust column definitions.

This operation exports raw Flex columns as binary data.

Output

The export operation always creates an output directory, even if all output is written to a single file or the query produces zero rows.

Output file names follow the pattern: [*8-character-hash*]-[*nodename*]-[*thread-id*].parquet .

Column names in partition directories are lowercase.

Files exported to a local file system by any Vertica user are owned by the Vertica superuser. Files exported to HDFS or object stores are owned by the Vertica user who exported the data.

Making concurrent exports to the same output destination is an error and can produce incorrect results.

Exports to the local file system can be to an NFS mount (shared) or to the Linux file system on each node (non-shared). For details, see Exporting to the Linux file system . Exports to non-shared local file systems have the following restrictions:

- The output directory must not exist on any node.
- You must have a USER storage location or superuser privileges.
- You cannot override the permissions mode of 700 for directories and 600 for files.

Exports to object-store file systems are not atomic. Be careful to wait for the export to finish before using the data. For details, see Exporting to object stores .

Examples

The following example demonstrates exporting all columns from theT1 table in the public schema, using GZIP compression.

```
=> EXPORT TO PARQUET(directory='webhdfs:///user1/data', compression='gzip')
 AS SELECT * FROM public.T1;
```

The following example demonstrates exporting the results of a query using more than one table.

```
=> EXPORT TO PARQUET(directory='s3://DataLake/sales_by_region')
  AS SELECT sale.price, sale.date, store.region
  FROM public.sales sale
  JOIN public.vendor store ON sale.distribID = store.ID;
```

The following example demonstrates partitioning and exporting data. EXPORT TO PARQUET first partitions the data on region and then, within each partition, sorts by store.

```
=> EXPORT TO PARQUET(directory='gs://DataLake/user2/data')
 OVER(PARTITION BY store.region ORDER BY store.ID)
 AS SELECT sale.price, sale.date, store.ID
 FROM public.sales sale
 JOIN public.vendor store ON sale.distribID = store.ID;
```

The following example uses an alias column label for a selected column target that is an expression.

```
=> EXPORT TO PARQUET(directory='webhdfs:///user3/data')
 OVER(ORDER BY col1) AS SELECT col1 + col1 AS A, col2
 FROM public.T3;
```

The following example sets permissions for the output.

```
=> EXPORT TO PARQUET(directory='webhdfs:///user1/data',
    fileMode='432', dirMode='rwxrw-r-x')
 AS SELECT * FROM public.T1;
```

EXPORT TO VERTICA

Exports table data from one Vertica database to another.

Important
The source database must be no more than one major release behind the target database.

Syntax

```
EXPORT [ /*+LABEL (label-string)*/ ] TO VERTICA
  database.[schema.]target-table [ ( target-columns ) ]
  { AS SELECT query-expression | FROM [schema.]source-table[ ( source-columns ) ] }
```

Arguments

LABEL

Assigns a label to a statement to identify it for profiling and debugging.

database

The target database of the data to export. A connection to this database must already exist in the current session before starting the copy operation; otherwise Vertica returns an error. For details, see CONNECT TO VERTICA .

[*schema.*] *target-table*

The table in *database* to store the exported data. The table cannot have columns of complex data types other than native arrays.

target-columns

A comma-delimited list of columns in *target-table* in which to store the exported data.See Mapping Between Source and Target Columns , below.

query-expression

The data to export.

[*schema.*] *source-table*

The table that contains the data to export.

source-columns

A comma-delimited list of the columns in the source table to export. The table cannot have columns of complex data types.See Mapping Between Source and Target Columns , below.

Privileges

Non-superusers:

- Source table: SELECT
- Source table schema: USAGE
- Target table: INSERT
- Target table schema: USAGE

Mapping between source and target columns

If you export all table data from one database to another, EXPORT TO VERTICA can omit specifying column lists if column definitions in both tables comply with the following conditions:

- Same number of columns
- Identical column names
- Same sequence of columns
- Matching or compatible column data types
- No complex data types (ARRAY, SET, or ROW), except for native arrays

If any of these conditions is not true, the EXPORT TO VERTICA statement must include column lists that explicitly map source and target columns to each other, as follows:

- Contain the same number of columns.
- List source and target columns in the same order.
- Pair columns with the same (or compatible) data types.

Examples

See Exporting data to another database .

See also

- Handling node failure during copy/export
- COPY FROM VERTICA

GET DIRECTED QUERY

Queries system table DIRECTED_QUERIES on the specified input query, and returns details of all directed queries that map to the input query. For details about output, see Getting directed queries .

Syntax

```
GET DIRECTED QUERY input-query
```

Arguments

input-query

An input query that is associated with one or more directed queries.

Privileges

None

Examples

See Getting directed queries .

GRANT statements

GRANT statements grant privileges on database objects to users and roles .

Important

In a database with trust authentication, GRANT statements appear to work as expected but have no real effect on database security.

In this section

- GRANT (authentication)
- GRANT (database)
- GRANT (key)
- GRANT (library)
- GRANT (model)
- GRANT (procedure)
- GRANT (Resource pool)
- GRANT (Role)

- GRANT (schema)
- GRANT (sequence)
- GRANT (storage location)
- GRANT (table)
- GRANT (TLS configuration)
- GRANT (user defined extension)
- GRANT (view)

GRANT (authentication)

Associates an authentication record to one or more users and roles .

Syntax

```
GRANT AUTHENTICATION auth-method-name TO grantee[,...]
```

Parameters

auth-method-name

Name of the authentication method to associate with one or more users or roles.

grantee

Specifies who is associated with the authentication method, one of the following:

- User name
- Role
- PUBLIC : The default role of all users.

Privileges

Superuser

Examples

- Associate v_ldap authentication with user jsmith :

```
=> GRANT AUTHENTICATION v_ldap TO jsmith;
```

- Associate v_gss authentication to the role DBprogrammer :

```
=> CREATE ROLE DBprogrammer;
=> GRANT AUTHENTICATION v_gss TO DBprogrammer;
```

- Associate client authentication method v_localpwd with role PUBLIC , which is assigned by default to all users:

```
=> GRANT AUTHENTICATION v_localpwd TO PUBLIC;
```

See also

- REVOKE (authentication)
- Granting and revoking privileges

GRANT (database)

Grants database privileges to users and roles .

Syntax

```
GRANT { privilege[,...] | ALL [ PRIVILEGES ] }
  ON DATABASE db-spec
  TO grantee[,...]
  [ WITH GRANT OPTION ]
```

Parameters

privilege

The following privileges are valid for a database:

- CREATE : Create schemas.
- TEMP : Create temporary tables. By default, all users are granted this privilege through their DEFAULT role.

ALL [PRIVILEGES]

Grants all database privileges that also belong to the grantor. Grantors cannot grant privileges that they themselves lack.

The optional keyword PRIVILEGES conforms with the SQL standard.

db-spec

Specifies the current database, set to the database name or DEFAULT .

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superuser: Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

Examples

The following example grants user Fred the right to create schemas in the current database.

```
=> GRANT CREATE ON DATABASE DEFAULT TO Fred;
```

See also

- REVOKE (database)
- Granting and revoking privileges

GRANT (key)

Grants privileges on a cryptographic key to a user or role.

Important

Because certificates depend on their underlying key, DROP privileges on a key effectively act as DROP privileges on its associated certificate when used with DROP KEY...CASCADE .

To revoke granted privileges, see REVOKE (key) .

Superusers have limited access to cryptographic objects that they do not own. For details, see Database object privileges .

Syntax

```
GRANT { privilege[,...] | ALL [ PRIVILEGES ] [ EXTEND ] } ON KEY
   key_name[,...]
   TO grantee[,...]
   [ WITH GRANT OPTION ]
```

Parameters

privilege

A privilege, one of the following:

- USAGE: Allows a user to perform the following actions:
  - View the contents of the key.
  - Create or sign certificates using the key.

  USAGE on the key also gives implicit USAGE privileges on a certificate that uses it as its private key. Users can also get these privileges from ownership of the key or certificate. USAGE privileges on a certificate allow a user to perform the following actions:
  - View the contents of the certificate.
  - Add (with CREATE or ALTER) the certificate to a TLS Configuration.
  - Reuse the CA certificate when importing certificates signed by it. For example, if a user imports a chain of certificates A > B > C and have USAGE on B , the database reuses B (as opposed to creating a duplicate of B).
  - Specify that the CA certificate signed an imported certificate. For example, if certificate B signed certificate C , USAGE on B allows a user to import C and specify that it was SIGNED BY B .
- DROP
- ALTER: Allows a user to see the key and its associated certificates in their respective system tables, but not their contents.

key_name

The target key .

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superuser:

- Owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

Examples

The following example grants USAGE privileges on a private key to a user, which then allows the user to add the self-signed CA certificate to the server TLS Configuration:

```
=> CREATE KEY new_ca_key TYPE 'RSA' LENGTH 2048;
=> CREATE CA CERTIFICATE new_ca_cert
   SUBJECT '/C=US/ST=Massachusetts/L=Cambridge/O=Micro Focus/OU=Vertica/CN=Vertica example CA'
   VALID FOR 3650
   EXTENSIONS 'authorityKeyIdentifier' = 'keyid:always,issuer', 'nsComment' = 'new CA'
   KEY new_ca_key;

=> CREATE USER u1;
=> GRANT USAGE ON KEY new_ca_key TO u1;
=> GRANT ALTER ON TLS CONFIGURATION data_channel TO u1;

=> \c - u1

=> ALTER TLS CONFIGURATION data_channel ADD CA CERTIFICATES new_ca_cert;

-- clean up:
=> \c
=> ALTER TLS CONFIGURATION data_channel REMOVE CA CERTIFICATES new_ca_cert;
=> DROP KEY new_ca_key CASCADE;
=> DROP USER u1;
```

GRANT (library)

Grants privileges on one or more libraries to users and roles .

For example, when working with the Connector Framework Service, you might need to grant a user usage privileges to a library to be able to set UDSession parameters. For more information see Implementing CFS .

Syntax

```
GRANT privilege
  ON LIBRARY [[database.]schema.]library[,...]
  TO grantee[,...]
  [ WITH GRANT OPTION ]
```

Arguments

privilege

Privilege to grant, one of:

- USAGE : Grants access to functions in the specified libraries.
- DROP : Grants permission to drop libraries that the grantee created.

- ALL [PRIVILEGES] [EXTEND] : Grants all library privileges that also belong to the grantor. Grantors cannot grant privileges that they themselves lack.

You can qualify ALL with two optional keywords:

- PRIVILEGES conforms with the SQL standard.
- EXTEND extends the semantics of ALL to include DROP privileges. An unqualified ALL excludes this privilege. This option enables backward compatibility with GRANT ALL usage in pre-9.2.1 Vertica releases.

Important
To execute functions inside the library, users must also have separate EXECUTE privileges on them, and USAGE privileges on their respective schemas.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

library

The target library.

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superusers require USAGE on the schema and one of the following:

- Owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

Examples

Grant USAGE privileges on the MyFunctions library to Fred :

```
=> GRANT USAGE ON LIBRARY MyFunctions TO Fred;
```

See also

- REVOKE (library)
- Granting and revoking privileges

GRANT (model)

Grants usage privileges on a model to users and roles .

Syntax

```
GRANT { privilege[,...] | ALL [ PRIVILEGES ] [ EXTEND ] }
  ON MODEL [[database.]schema.]model-name[,...]
  TO grantee[,...]
  [ WITH GRANT OPTION ]
```

Parameters

privilege

The following privileges are valid for models:

- USAGE
- ALTER
- DROP

ALL [PRIVILEGES][EXTEND]

Grants all model privileges that also belong to the grantor. Grantors cannot grant privileges that they themselves lack.

You can qualify ALL with two optional keywords:

- PRIVILEGES conforms with the SQL standard.
- EXTEND extends the semantics of ALL to include ALTER and DROP privileges. An unqualified ALL excludes these two privileges. This option enables backward compatibility with GRANT ALL usage in pre-9.2.1 Vertica releases.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

model-name

The model on which to grant the privilege.

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superusers require USAGE on the schema and one of the following:

- Owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

Examples

This example grants USAGE privileges on the mySvmClassModel model to user1:

```
=> GRANT USAGE ON MODEL mySvmClassModel TO user1;
```

See also

- REVOKE (model)
- Managing model security

GRANT (procedure)

Grants privileges on a stored procedure or external procedure to a user or role .

Important

External procedures that you create with CREATE PROCEDURE (external) are always run with Linux dbadmin privileges. If a dbadmin or pseudosuperuser grants a non-dbadmin permission to run a procedure using GRANT (procedure) , be aware that the non-dbadmin user runs the procedure with full Linux dbadmin privileges.

Syntax

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON PROCEDURE [[database.]schema.]procedure( [arg-list] )[,...]
  TO grantee[,...]
  [ WITH GRANT OPTION ]
```

Parameters

EXECUTE

Enables grantees to run the specified *procedure* .

ALL [PRIVILEGES]

Grants all procedure privileges that also belong to the grantor. Grantors cannot grant privileges that they themselves lack.

The optional keyword PRIVILEGES conforms with the SQL standard.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

procedure

The target procedure.

arg-list

A comma-delimited list of procedure arguments, where each argument is specified as follows:

```
[ argname ] argtype
```

If the procedure is defined with no arguments, supply an empty argument list.

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superuser, one of the following:

- Owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles

Examples

Grant EXECUTE privileges on the tokenize procedure to users Bob and Jules , and to the role Operator :

```
=> GRANT EXECUTE ON PROCEDURE tokenize(varchar) TO Bob, Jules, Operator;
```

See also

- REVOKE (procedure)
- Granting and revoking privileges

GRANT (Resource pool)

Grants USAGE privileges on resource pools to users and roles . Users can access their resource pools with ALTER USER or SET SESSION RESOURCE POOL .

Syntax

```
GRANT USAGE
  ON RESOURCE POOL resource-pool[,...]
  [FOR SUBCLUSTER subcluster | FOR CURRENT SUBCLUSTER]
  TO grantee[,...]
  [ WITH GRANT OPTION ]
```

Parameters

USAGE

Enables grantees to acess the specified resource pools.

ALL [PRIVILEGES]

Grants all resource pool privileges that also belong to the grantor. Grantors cannot grant privileges that they themselves lack.

The optional keyword PRIVILEGES conforms with the SQL standard.

resource-pool

A resource pool on which to grant the specified privileges.

subcluster

The subcluster for the resource pool.

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*

- PUBLIC : Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superuser, one of the following:

- Ownership
- GRANT OPTION on the object

Examples

Grant user Joe USAGE privileges on resource pool Joe_pool .

```
=> CREATE USER Joe;
CREATE USER
=> CREATE RESOURCE POOL Joe_pool;
CREATE RESOURCE POOL
=> GRANT USAGE ON RESOURCE POOL Joe_pool TO Joe;
GRANT PRIVILEGE
```

Grant user Joe USAGE privileges on resource pool Joe_pool for subcluster sub1 .

```
=> GRANT USAGE on RESOURCE POOL Joe_pool FOR SUBCLUSTER sub1 TO Joe;
GRANT PRIVILEGE
```

See also

- REVOKE (Resource pool)
- Granting and revoking privileges

GRANT (Role)

Assigns roles to users or other roles .

> **Note**
> Granting a role does not activate the role automatically; you must enable it with the SET ROLE statement or specify it as a default role to enable it automatically.

Syntax

```
GRANT role[,...] TO grantee[,...] [ WITH ADMIN OPTION ]
```

Arguments

role

A role to grant

grantee

User or role to be granted the specified roles, one of the following:

- User name
- Role
- PUBLIC : The default role of all users.

WITH ADMIN OPTION

Gives *grantee* the privilege to grant the specified roles to other users or roles. For details, see Granting privileges .

Privileges

Non-superuser: If WITH GRANT OPTION is used, can grant the same roles to other users or roles.

Examples

See Granting database roles .

See also

REVOKE (Role)

GRANT (schema)

Grants schema privileges to users and roles. By default, only superusers and the schema owner have the following schema privileges:

- Create objects within a schema.
- Alter and drop a schema.

> **Note**
> By default, new users cannot access schema PUBLIC. You must explicitly grant all new users USAGE privileges on the PUBLIC schema.

Syntax

```
GRANT { privilege[,...] | ALL [ PRIVILEGES ] [ EXTEND ] }
  ON SCHEMA [database.]schema[,...]
  TO grantee[,...]
  [ WITH GRANT OPTION ]
```

Parameters

privilege

One of the following privileges:

- USAGE: Enables access to objects in the specified schemas. Grantees can then be granted privileges on individual objects in these schemas in order to access them, for example, with GRANT TABLE and GRANT VIEW.
- CREATE: Create and rename objects in the specified schemas, and move objects from other schemas.

You can also grant the following privileges on a schema, to be inherited by tables and their projections, and by views of that schema. If inheritance is enabled for the database and schema, these privileges are automatically granted to those objects on creation:

- SELECT: Query tables and views. SELECT privileges are granted by default to the PUBLIC role.
- INSERT: Insert rows, or and load data into tables with COPY.

  > **Note**
  > COPY FROM STDIN is allowed for users with INSERT privileges, while COPY FROM *file* requires admin privileges.

- UPDATE: Update table rows.
- DELETE: Delete table rows.
- REFERENCES: Create foreign key constraints on this table. This privilege must be set on both referencing and referenced tables.
- TRUNCATE: Truncate table contents. Non-owners of tables can also execute the following partition operations on them:
  - DROP_PARTITIONS
  - SWAP_PARTITIONS_BETWEEN_TABLES
  - MOVE_PARTITIONS_TO_TABLE
- ALTER: Modify the DDL of tables and views with ALTER TABLE and ALTER VIEW, respectively.
- DROP: Drop tables and views.

ALL [PRIVILEGES][EXTEND]

Grants USAGE AND CREATE privileges. Inherited privileges must be granted explicitly.

You can qualify ALL with two optional keywords:

- PRIVILEGES conforms with the SQL standard.
- EXTEND extends the semantics of ALL to include ALTER and DROP privileges. An unqualified ALL excludes these two privileges. This option enables backward compatibility with GRANT ALL usage in pre-9.2.1 Vertica releases.

[*database* .] *schema*

Specifies a target schema. If you specify a database, it must be the current database.

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC: Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superusers, one of the following:

- Schema owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

Examples

Grant user Joe USAGE privilege on schema online_sales .

```
=> CREATE USER Joe;
CREATE USER
=> GRANT USAGE ON SCHEMA online_sales TO Joe;
GRANT PRIVILEGE
```

See also

- REVOKE (schema)
- Granting and revoking privileges

GRANT (sequence)

Grants sequence privileges to users and roles .

Syntax

```
GRANT { privilege[,...] | ALL [ PRIVILEGES ] [ EXTEND ] }
  ON {
    SEQUENCE [[database.]schema.]sequence[,...]
    | ALL SEQUENCES IN SCHEMA [database.]schema[,...] }
  TO grantee[,...]
  [ WITH GRANT OPTION ]
```

Parameters

privilege

The following privileges are valid for sequences:

- SELECT: Execute functions CURRVAL and NEXTVAL on the specified sequences.
- ALTER: Modify a sequence's DDL with ALTER SEQUENCE
- DROP: Drop this sequence with DROP SEQUENCE .

ALL [PRIVILEGES][EXTEND]

Grants all sequence privileges that also belong to the grantor. Grantors cannot grant privileges that they themselves lack

You can qualify ALL with two optional keywords:

- PRIVILEGES conforms with the SQL standard.
- EXTEND extends the semantics of ALL to include ALTER and DROP privileges. An unqualified ALL excludes these two privileges. This option enables backward compatibility with GRANT ALL usage in pre-9.2.1 Vertica releases.

[*database.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

SEQUENCE *sequence*

Specifies the sequence on which to grant privileges.

ALL SEQUENCES IN SCHEMA *schema*

Grants the specified privileges on all sequences in schema *schema* .

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superusers require USAGE on the schema and one of the following:

- Owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

Examples

This example shows how to grant user Joe all privileges on sequence my_seq .

```
=> CREATE SEQUENCE my_seq START 100;
CREATE SEQUENCE
=> GRANT ALL PRIVILEGES ON SEQUENCE my_seq TO Joe;
GRANT PRIVILEGE
```

See also

- REVOKE (sequence)
- Granting and revoking privileges
- Creating and using named sequences

GRANT (storage location)

Grants privileges to users and roles on a USER-defined storage location. For details, see Creating storage locations .

Syntax

```
GRANT { privilege[,...] | ALL [ PRIVILEGES ] }
  ON LOCATION 'path' [ ON node ]
  TO grantee[,...]
  [ WITH GRANT OPTION ]
```

Parameters

privilege

The following privileges are valid for storage locations:

- READ : Copy data from files in the storage location into a table.
- WRITE : Export data from the database to the storage location. With WRITE privileges, grantees can also save COPY statement rejected data and exceptions files to the storage location.

ALL [PRIVILEGES]

Grants all storage location privileges that also belong to the grantor. Grantors cannot grant privileges that they themselves lack.

The optional keyword PRIVILEGES conforms with the SQL standard.

ON LOCATION ' *path* ' [ON *node*]

Specifies the path name mount point of the storage location. If qualified by ON NODE , Vertica grants access to the storage location residing on *node* .

If no node is specified, the grant operation applies to all nodes on the specified path. All nodes must be on the specified path; otherwise, the entire grant operation rolls back.

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superuser, one of the following:

- Ownership
- GRANT OPTION on the object

> **Note**
> Only a superuser can add, alter, retire, drop, and restore a location.

Examples

Create a storage location:

```
=> CREATE LOCATION '/home/dbadmin/UserStorage/BobStore' NODE 'v_mcdb_node0007' USAGE 'USER';
CREATE LOCATION
```

Grant user Bob all available privileges to the /BobStore location:

```
=> GRANT ALL ON LOCATION '/home/dbadmin/UserStorage/BobStore' TO Bob;
GRANT PRIVILEGE
```

Revoke all storage location privileges from Bob:

```
=> REVOKE ALL ON LOCATION '/home/dbadmin/UserStorage/BobStore' FROM Bob;
REVOKE PRIVILEGE
```

Grant privileges to Bob on the BobStore location again, specifying a node:

```
=> GRANT ALL ON LOCATION '/home/dbadmin/UserStorage/BobStore' ON v_mcdb_node0007 TO Bob;
GRANT PRIVILEGE
```

Revoke all storage location privileges from Bob :

```
=> REVOKE ALL ON LOCATION '/home/dbadmin/UserStorage/BobStore' ON v_mcdb_node0007 FROM Bob;
REVOKE PRIVILEGE
```

See also

- Storage functions
- REVOKE (storage location)
- Granting and revoking privileges

GRANT (table)

Grants table privileges to users and roles . Users must also be granted USAGE on the table schema .

Syntax

```
GRANT { privilege[,...] | ALL [ PRIVILEGES ] [ EXTEND ] }
  ON {
    [ TABLE ] [[database.]schema.]table[,...]
    | ALL TABLES IN SCHEMA [database.]schema[,...] }
  TO grantee[,...]
  [ WITH GRANT OPTION ]
```

Parameters

privilege

The following privileges are valid for tables:

Important

Only SELECT privileges are valid for system tables.

- SELECT: Quer y tables. SELECT privileges are granted by default to the PUBLIC role.
- INSERT: Insert table rows with INSERT , and load data with COPY .

> **Note**
> COPY FROM STDIN is allowed for users with INSERT privileges, while COPY FROM *file* requires admin privileges.

- UPDATE: Update table rows.
- DELETE: Delete table rows.
- REFERENCES: Create foreign key constraints on this table. This privilege must be set on both referencing and referenced tables.
- TRUNCATE: Truncate table contents. Non-owners of tables can also execute the following partition operations on them:
  - DROP_PARTITIONS
  - SWAP_PARTITIONS_BETWEEN_TABLES
  - MOVE_PARTITIONS_TO_TABLE
- ALTER: Modify a table's DDL with ALTER TABLE .
- DROP: Drop a table .

ALL [PRIVILEGES][EXTEND]

Invalid for system tables, grants all table privileges that also belong to the grantor. Grantors cannot grant privileges that they themselves lack.

You can qualify ALL with two optional keywords:

- PRIVILEGES conforms with the SQL standard.
- EXTEND extends the semantics of ALL to include ALTER and DROP privileges. An unqualified ALL excludes these two privileges. This option enables backward compatibility with GRANT ALL usage in pre-9.2.1 Vertica releases.

[*database.*] *schema*

Specifies a schema , by default public . If *schema* is any schema other than public , you must supply the schema name. For example:

```
myschema.thisDbObject
```

One exception applies: you can specify system tables without their schema name.
If you specify a database, it must be the current database.

TABLE *table*

Specifies the table on which to grant privileges.

> **Note**
> The table can be a global temporary table, but not a local temporary table. See Creating temporary tables .

ON ALL TABLES IN SCHEMA *schema*

Grants the specified privileges on all tables and views in schema *schema* .

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superusers require USAGE on the schema and one of the following:

- Owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

Examples

Grant user Joe all privileges on table customer_dimension :

```
=> CREATE USER Joe;
CREATE USER
=> GRANT ALL PRIVILEGES ON TABLE customer_dimension TO Joe;
GRANT PRIVILEGE
```

Grant user Joe SELECT privileges on all system tables:

```
=> GRANT SELECT on all tables in schema V_MONITOR, V_CATALOG TO Joe;
GRANT PRIVILEGE
```

See also

- REVOKE (table)
- Granting and revoking privileges

GRANT (TLS configuration)

Grants privileges on a TLS Configuration to a user or role.

To revoke granted privileges, see REVOKE (TLS configuration) .

Superusers have limited access to cryptographic objects that they do not own. For details, see Database object privileges .

Syntax

```
GRANT { privilege[,...] } ON TLS CONFIGURATION
   tls_configuration[,...]
   TO grantee[,...]
   [ WITH GRANT OPTION ]
```

Parameters

privilege

A privilege, one of the following:

- USAGE: Allows the user to set the TLS Configuration for a type of connection and view its contents in the system table TLS_CONFIGURATIONS . For details, see Security parameters .
- DROP
- ALTER

tls_configuration

The target TLS Configuration.

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superuser:

- Owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

Examples

You can grant ALTER on a TLS Configuration to a user or role to delegate management of that TLS context, which includes adding and removing certificates, setting the TLSMODE, etc. For example, the following statement grants ALTER privileges on the TLS CONFIGURATION server to the role client_server_tls_manager :

```
=> GRANT ALTER ON TLS CONFIGURATION server TO client_server_tls_manager;
```

GRANT (user defined extension)

Grants privileges on a user-defined extensions (UDx) to users and roles .

Syntax

```
GRANT { privilege[,...] | ALL [ PRIVILEGES ] [ EXTEND ] }
  ON {
     UDx-type [[database.]schema.]function( [arg-list] )[,...]
     | ALL FUNCTIONS IN SCHEMA schema[,...] }
  TO grantee[,...]
  [ WITH GRANT OPTION ]
```

Arguments

privilege

The following privileges are valid for user-defined extensions:

- EXECUTE
- ALTER
- DROP

> **Note**
> Users can only call a UDx function on which they have EXECUTE privilege, and USAGE privilege on its schema.

ALL [PRIVILEGES] [EXTEND]

Grants all function privileges that also belong to the grantor. Grantors cannot grant privileges that they themselves lack

You can qualify ALL with two optional keywords:

- PRIVILEGES conforms with the SQL standard.
- EXTEND extends the semantics of ALL to include ALTER and DROP privileges. An unqualified ALL excludes these two privileges. This option enables backward compatibility with GRANT ALL usage in pre-9.2.1 Vertica releases.

ON *UDx-type*

Type of the user-defined extension (UDx), one of the following:

- FUNCTION (scalar function)
- AGGREGATE FUNCTION
- ANALYTIC FUNCTION
- TRANSFORM FUNCTION
- FILTER
- PARSER
- SOURCE

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function

Name of the user-defined function on which to grant privileges.

ON ALL FUNCTIONS IN SCHEMA *schema*

Grants privileges on all functions in the specified schema.

arg-list

Required for all polymorphic functions, a comma-delimited list of function arguments, where each argument is specified as follows:

```
[ argname ] argtype
```

If the procedure is defined with no arguments, supply an empty argument list.

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

> **Note**
> Grantees must have USAGE privileges on the schema.

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superusers require USAGE on the schema and one of the following:

- Owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

Examples

Grant EXECUTE privileges on the myzeroifnull SQL function to users Bob and Jules , and to the role Operator . The function takes one integer argument:

```
=> GRANT EXECUTE ON FUNCTION myzeroifnull (x INT) TO Bob, Jules, Operator;
```

Grant EXECUTE privileges on all functions in the zero-schema schema to user Bob :

```
=> GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA zero-schema TO Bob;
```

Grant EXECUTE privileges on the tokenize transform function to user Bob and the role Operator :

```
=> GRANT EXECUTE ON TRANSFORM FUNCTION tokenize(VARCHAR) TO Bob, Operator;
```

Grant EXECUTE privileges on the ExampleSource() source to user Alice :

```
=> CREATE USER Alice;
=> GRANT USAGE ON SCHEMA hdfs TO Alice;
=> GRANT EXECUTE ON SOURCE ExampleSource() TO Alice;
```

Grant all privileges on the ExampleSource() source to user Alice :

```
=> GRANT ALL ON SOURCE ExampleSource() TO Alice;
```

Grant all privileges on polymorphic function Pagerank to the dbadmin role:

```
=> GRANT ALL ON TRANSFORM FUNCTION Pagerank(z varchar) to dbadmin;
```

See also

- REVOKE (user defined extension)
- Granting and revoking privileges
- Developing user-defined extensions (UDxs)

GRANT (view)

Grants view privileges to users and roles .

Syntax

```
GRANT { privilege[,...] | ALL [ PRIVILEGES ] [ EXTEND ] }
  ON [[database.]schema.]view[,...]
  TO grantee[,...]
  [ WITH GRANT OPTION ]
```

Parameters

privilege ``

The following privileges are valid for views:

- SELECT
- ALTER
- DROP

ALL [PRIVILEGES][EXTEND]

Grants all view privileges that also belong to the grantor. Grantors cannot grant privileges that they themselves lack.

You can qualify ALL with two optional keywords:

- PRIVILEGES conforms with the SQL standard.

- EXTEND extends the semantics of ALL to include ALTER and DROP privileges. An unqualified ALL excludes these two privileges. This option enables backward compatibility with GRANT ALL usage in pre-9.2.1 Vertica releases.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

view

The target view.

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

WITH GRANT OPTION

Gives *grantee* the privilege to grant the same privileges to other users or roles, and also revoke them. For details, see Granting privileges .

Privileges

Non-superusers require USAGE on the schema and one of the following:

- Owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

> **Note**
> As view owner, you can grant other users SELECT privilege on the view only if one of the following is true:
>
> - You own the view's base table.
> - You have SELECT...WITH GRANT OPTION privilege on the view's base table.

Examples

Grant user Joe all privileges on view ship .

```
=> CREATE VIEW ship AS SELECT * FROM public.shipping_dimension;
CREATE VIEW
=> GRANT ALL PRIVILEGES ON ship TO Joe;
GRANT PRIVILEGE
```

See also

REVOKE (view)

INSERT

Inserts values into all projections of the specified table. You must insert one complete tuple at a time. If no projections are associated with the target table, Vertica creates a superprojection to store the inserted values.

INSERT works for flex tables as well as regular native tables. If the table has real columns, inserted data of scalar types and native arrays of scalar types is added to both the real column and the __raw__ column. For data of complex types , the values are not added to the __raw__ column.

Syntax

```
INSERT [ /*+LABEL (label-string)*/ ] INTO [[database.]schema.]table-name
   [ ( column-list ) ]
   { DEFAULT VALUES | VALUES ( values-list )[,...] | SELECT query-expression }
```

Arguments

LABEL

Assigns a label to a statement to identify it for profiling and debugging.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table-name

The target table. You cannot invoke INSERT on a projection.

column-list

A comma-delimited list of one or more target columns in this table, listed in any order. VALUES clause values are mapped to columns in the same order. If you omit this list, Vertica maps VALUES clause values to columns according to column order in the table definition.

A list of target columns is invalid with DEFAULT VALUES .

DEFAULT VALUES

Fills all columns with their default values as specified in the table definition. If no default value is specified for a column, Vertica inserts a NULL value.

You cannot specify a list of target columns with this option.

VALUES (*values-list*)

A comma-delimited list of one or more values to insert in the target columns, where each value is one of the following:

- *expression* resolves to a value to insert in the target column. The expression must not nest other expressions, include Vertica meta-functions, or use mixed complex types. Values may include native array or ROW types if Vertica can coerce the element or field types.
- DEFAULT inserts the default value as specified in the table definition.

If no value is supplied for a column, Vertica implicitly adds a DEFAULT value, if defined. Otherwise Vertica inserts a NULL value. If the column is defined as NOT NULL , INSERT returns an error.
You can use INSERT to insert multiple rows in the target table, by specifying multiple comma-delimited VALUES lists:

```
INSERT INTO table-name
  VALUES ( values-list ), ( values-list )[,...]
```

For details, see Multi-Row INSERT below.

SELECT *query-expression*

A query that returns the rows to insert. Isolation level applies only to the SELECT clauses and works like any query. Restrictions on use of complex types apply as in other queries.

Privileges

- Table owner or user with GRANT OPTION is grantor
- INSERT privilege on table
- USAGE privilege on schema that contains the table

Committing successive table changes

Vertica follows the SQL-92 transaction model, so successive INSERT, UPDATE, and DELETE statements are included in the same transaction. You do not need to explicitly start this transaction; however, you must explicitly end it with COMMIT , or implicitly end it with COPY . Otherwise, Vertica discards all changes that were made within the transaction.

Multi-row INSERT

You can use INSERT to insert multiple rows in the target table, by specifying multiple comma-delimited VALUES lists. For example:

```
=> CREATE TABLE public.t1(a int, b int, c varchar(16));
CREATE TABLE
=> INSERT INTO t1 VALUES (1,2, 'un, deux'), (3,4, 'trois, quatre');
 OUTPUT
--------
      2
(1 row)

=> COMMIT;
COMMIT
=> SELECT * FROM t1;
 a | b |       c
---+---+---------------
 1 | 2 | un, deux
 3 | 4 | trois, quatre
(4 rows)
```

Restrictions

- Vertica does not support subqueries as the target of an INSERT statement.
- Restrictions on the use of complex types in SELECT statements apply equally to INSERT . Using complex values that cannot be coerced to the column type results in an error.
- If primary key, unique key, or check constraints are enabled for automatic enforcement in the target table, Vertica enforces those constraints when you load new data. If a violation occurs, Vertica rolls back the operation and returns an error.
- If an insert would violate a table or schema disk quota, the operation fails. For more information, see Disk quotas .

Examples

```
=> INSERT INTO t1 VALUES (101, 102, 103, 104);
=> INSERT INTO customer VALUES (10, 'male', 'DPR', 'MA', 35);
=> INSERT INTO start_time VALUES (12, 'film','05:10:00:01');
=> INSERT INTO retail.t1 (C0, C1) VALUES (1, 1001);
=> INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

Vertica does not support subqueries or nested expressions as the target of an INSERT statement. For example, the following query returns an error message:

```
=> INSERT INTO t1 (col1, col2) VALUES ('abc', (SELECT mycolumn FROM mytable));
ERROR 4821:  Subqueries not allowed in target of insert
```

You can rewrite the above query as follows:

```
=> INSERT INTO t1 (col1, col2) (SELECT 'abc', mycolumn FROM mytable);
 OUTPUT
--------
      0
(1 row)
```

The following example shows how to use INSERT...VALUES with flex tables:

```
=> CREATE FLEX TABLE flex1();
 CREATE TABLE
=> INSERT INTO flex1(a,b) VALUES (1, 'x');
 OUTPUT
--------
      1
(1 row)

=> SELECT MapToString(__raw__) FROM flex1;
          MapToString
-----------------------------------
{
"a" : "1",
"b" : "x"
}
(1 row)
```

The following example shows how to use INSERT...SELECT with flex tables:

```
=> CREATE FLEX TABLE flex2();
 CREATE TABLE
=> INSERT INTO flex2(a, b) SELECT a, b, '2016-08-10 11:10' c, 'Hello' d, 3.1415 e, f from flex1;
 OUTPUT
--------
      1
(1 row)

=> SELECT MapToString(__raw__) FROM flex2;
           MapToString
---------------------------------
{
"a" : "1",
"b" : "x",
"c" : "2016-08-10",
"d" : "Hello",
"e" : 3.1415,
"f" : null
}
(1 row)
```

The following examples use complex types:

```
=> CREATE TABLE inventory(storeID INT, product ROW(name VARCHAR, code VARCHAR));
CREATE TABLE

--- LookUpProducts() returns a row(varchar, int), which is cast to row(varchar, varchar):
=> INSERT INTO inventory(product) SELECT LookUpProducts();
 OUTPUT
--------
      5
(1 row)

--- Cannot use with select...values:
=> INSERT INTO inventory(product) VALUES(LookUpProducts());
ERROR 2631:  Column "product" is of type "row(varchar,varchar)" but expression is of type "row(varchar,int)"

--- Literal values are supported:
=> INSERT INTO inventory(product) VALUES(ROW('xbox',165));
 OUTPUT
--------
      1
(1 row)

=> SELECT product FROM inventory;
          product
------------------------------
 {"name":"xbox","code":"125"}
(1 row)
```

LOCK TABLE

Locks a table, giving the caller's session exclusive access to certain operations. Tables are automatically unlocked after the current transaction ends—that is, after COMMIT or ROLLBACK . LOCK TABLE can be useful for preventing deadlocks .

To view existing locks, see LOCKS .

> **Note**
> READ COMMITTED isolation (default) and SERIALIZABLE isolation automatically handle locks for you, and the vast majority of users can rely on them exclusively; LOCK TABLE is only for advanced users who need granular control over locks for more complex workloads.

To implement pessimistic concurrency without manually locking tables, you can use SELECT...FOR UPDATE to acquire an EXCLUSIVE (X) lock on the table.

Syntax

```
LOCK [ TABLE ] [[database.]schema.] table [,...]
  IN { lock_type } MODE
  [ NOWAIT ]
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

The table to lock.

lock-type

The type of lock , one of the following:

- SHARE
- INSERT
- INSERT VALIDATE
- SHARE INSERT
- EXCLUSIVE
- NOT DELETE
- USAGE
- OWNER

> **Note**
> LOCK TABLE does not currently support D (drop partition) locks.

NOWAIT

If specified, LOCK TABLE returns and reports an error immediately if it cannot acquire the lock. Otherwise, LOCK TABLE waits for incompatible locks to be released by their respective sessions, returning an error if the lock is not released after a certain amount of time, as defined by LockTimeout .

Privileges

Required privileges depend on the type of lock requested:

Lock	Privileges
SHARED (S)	SELECT
INSERT (I)	INSERT
SHARE INSERT	SELECT, INSERT
INSERT VALIDATE (IV)	SELECT, INSERT
EXCLUSIVE (X)	UPDATE, DELETE
NOT DELETE (T)	SELECT
USAGE	All privileges
Owner	All privileges

Examples

See Lock examples .

MERGE

Performs update and insert operations on a target table based on the results of a join with another data set, such as a table or view. The join can match a source row with only one target row; otherwise, Vertica returns an error.

If a merge would violate a table or schema disk quota, the operation fails. For more information, see Disk quotas .

The target table cannot have columns of complex data types. The source table can, so long as those columns are not included in the merge operation.

Syntax

```
MERGE [ /*+LABEL (label-string)*/ ]
   INTO [[database.]schema.]target-table [ [AS] alias ]
   USING source-dataset
   ON  join-condition matching-clause[ matching-clause ]
```

Returns

Number of target table rows updated or inserted

Arguments

LABEL

Assigns a label to a statement to identify it for profiling and debugging.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

target-table

The table on which to perform update and insert operations. MERGE takes an X (exclusive) lock on the target table during the operation. The table must not contain columns of complex types.

Important

The total number of target table columns cannot exceed 831.

source-dataset

The data to join to *target-table* , one of the following:

- [[*database* .] *schema.*] *table* [[AS] *alias*]
- [[*database* .] *schema.*] *view* [[AS] *alias*]
- (*subquery*) *sq-alias*

The specified data set typically supplies the data used to update the target table and populate new rows. You can specify an external table.

ON *join-condition*

The conditions on which to join the target table and source data set.

> **Tip**
> The Vertica query optimizer can create an optimized query plan for a MERGE statement only if the target table join column has a unique or primary key constraint. For details, see MERGE optimization .

matching-clause

One of the following clauses:

- WHEN MATCHED THEN UPDATE
- WHEN NOT MATCHED THEN INSERT

MERGE supports one instance of each clause, and must include at least one.

WHEN MATCHED THEN UPDATE

For each *target-table* row that is joined (matched) to *source-dataset* , specifies to update one or more columns:

```
WHEN MATCHED [ AND update-filter ] THEN UPDATE
 SET { column = expression }[,...]
```

update-filter optionally filters the set of matching rows. The update filter can specify any number of conditions. Vertica evaluates each matching row against this filter, and updates only the rows that evaluate to true. For details, see Update and insert filters .

Note
Vertica also supports Oracle syntax for specifying update filters:

```
WHEN MATCHED THEN UPDATE
SET { column = expression }[,...]
[ WHERE update-filter ]
```

The following requirements apply:

- A MERGE statement can contain only one WHEN MATCHED clause.
- *target-column* can only specify a column name in the target table. It cannot be qualified with a table name.

For details, see Merging table data .

WHEN NOT MATCHED THEN INSERT

For each *source-dataset* row that is not joined (not matched) to *target-table* , specifies to:

- Insert a new row into *target-table* .
- Populate each new row with the values specified in *values-list* .

```
WHEN NOT MATCHED [ AND insert-filter ] THEN INSERT
 [ ( column-list ) ] VALUES ( values-list )
```

column-list is a comma-delimited list of one or more target columns in the target table, listed in any order. MERGE maps *column-list* columns to *values-list* values in the same order, and each column-value pair must be compatible . If you omit *column-list* , Vertica maps *values-list* values to columns according to column order in the table definition.
insert-filter optionally filters the set of non-matching rows. The insert filter can specify any number of conditions. Vertica evaluates each non-matching source row against this filter. For each row that evaluates to true, Vertica inserts a new row in the target table. For details, see Update and insert filters .

Note
Vertica also supports Oracle syntax for specifying insert filters:

```
WHEN NOT MATCHED THEN INSERT
[ ( column-list ) ] VALUES ( values-list )
[ WHERE insert-filter ]
```

The following requirements apply:

- A MERGE statement can contain only one WHEN NOT MATCHED clause.
- * *column-list* * can only specify column names in the target table. It cannot be qualified with a table name.
- Insert filter conditions can only reference the source data. If any condition references the target table, Vertica returns an error.

For details, see Merging table data .

Privileges

MERGE requires the following privileges:

- SELECT permissions on the source data and INSERT , UPDATE , and DELETE permissions on the target table.
- Automatic constraint enforcement requires SELECT permissions on the table containing the constraint.
- SELECT permissions on the target table if the condition in the syntax reads data from the target table.

For example, the following GRANT statement grants user1 access to the t2 table. This allows user1 to run the MERGE statement that follows:

```
=> GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE t2 to user1;
GRANT PRIVILEGE

=>\c - user1
You are now connected as user "user1".

=> MERGE INTO t2 USING t1 ON t1.a = t2.a
WHEN MATCHED THEN UPDATE SET b = t1.b
WHEN NOT MATCHED THEN INSERT (a, b) VALUES (t1.a, t1.b);
```

Improving MERGE performance

You can improve MERGE performance in several ways:

- Design projections for optimal MERGE performance .
- Facilitate creation of optimized query plans .
- Use a source data set that is smaller than the target table.

For details, see MERGE optimization .

Constraint enforcement

If primary key, unique key, or check constraints are enabled for automatic enforcement in the target table, Vertica enforces those constraints when you load new data. If a violation occurs, Vertica rolls back the operation and returns an error.

> **Caution**
> If you run MERGE multiple times using the same target and source table, each iteration is liable to introduce duplicate values into the target columns and return with an error.

Columns prohibited from merge

The following columns cannot be specified in a merge operation; attempts to do so return with an error:

- IDENTITY columns, or columns whose default value is set to a named sequence .
- Vmap columns such as __raw__ in flex tables.
- Columns of complex types ARRAY, SET, or ROW.

Examples

See:

- Basic MERGE example
- MERGE source options
- MERGE matching clauses
- Update and insert filters

See also

- Merging table data

PROFILE

Profiles a single SQL statement.

Syntax

```
PROFILE { sql-statement }
```

Parameters

sql-statement

A query (SELECT) statement or DML statement--for example, you can profile INSERT , UPDATE , COPY , and MERGE .

Output

Writes profile summary to stderr, saves details to system catalog V_MONITOR.EXECUTION_ENGINE_PROFILES .

Privileges

The same privileges required to run the profiled statement

Description

PROFILE generates detailed information about how the target statement executes, and saves that information in the system catalog V_MONITOR.EXECUTION_ENGINE_PROFILES . Query output is preceded by a profile summary: profile identifiers transaction_id and statement_id , initiator memory for the query, and total memory required. For example:

```
=> PROFILE SELECT customer_name, annual_income FROM public.customer_dimension WHERE (customer_gender, annual_income) IN (SELECT
customer_gender, MAX(annual_income) FROM public.customer_dimension GROUP BY customer_gender);
NOTICE 4788:  Statement is being profiled
HINT:  Select * from v_monitor.execution_engine_profiles where transaction_id=45035996274683334 and statement_id=7;
NOTICE 3557:  Initiator memory for query: [on pool general: 708421 KB, minimum: 554324 KB]
NOTICE 5077:  Total memory required by query: [708421 KB]
  customer_name   | annual_income
------------------+---------------
 Emily G. Vogel   |       999998
 James M. McNulty |       999979
(2 rows)
```

Use profile identifiers to query the table for profile information on a given query.

See also

Profiling single statements

RELEASE SAVEPOINT

Destroys a savepoint without undoing the effects of commands executed after the savepoint was established.

Syntax

```
RELEASE [ SAVEPOINT ] savepoint_name
```

Parameters

savepoint_name

Specifies the name of the savepoint to destroy.

Privileges

None

Notes

Once destroyed, the savepoint is unavailable as a rollback point.

Examples

The following example establishes and then destroys a savepoint called my_savepoint. The values 101 and 102 are both inserted at commit.

```
=> INSERT INTO product_key VALUES (101);
=> SAVEPOINT my_savepoint;
=> INSERT INTO product_key VALUES (102);
=> RELEASE SAVEPOINT my_savepoint;
=> COMMIT;
```

See also

- SAVEPOINT
- ROLLBACK TO SAVEPOINT

REVOKE statements

REVOKE statements let you revoke privileges on database objects from users and roles .

Important

In a database with trust authentication, REVOKE statements appear to work as expected but have no real effect on database security.

In this section

- REVOKE (authentication)
- REVOKE (database)
- REVOKE (key)
- REVOKE (library)

- REVOKE (model)
- REVOKE (procedure)
- REVOKE (Resource pool)
- REVOKE (Role)
- REVOKE (schema)
- REVOKE (sequence)
- REVOKE (storage location)
- REVOKE (table)
- REVOKE (TLS configuration)
- REVOKE (user defined extension)
- REVOKE (view)

REVOKE (authentication)

Revokes privileges on an authentication method from users and roles .

Syntax

```
REVOKE AUTHENTICATION auth-method-name FROM grantee[,...]
```

Parameters

auth-method-name

Name of the target authentication method.

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

Privileges

Superuser

Examples

- Revoke v_ldap authentication from user jsmith :

```
=> REVOKE AUTHENTICATION v_ldap FROM jsmith;
```

- Revoke v_gss authentication from the role DBprogrammer :

```
=> REVOKE AUTHENTICATION v_gss FROM DBprogrammer;
```

- Revoke localpwd as the default client authentication method:

```
=> REVOKE AUTHENTICATION localpwd FROM PUBLIC;
```

See also

- ALTER AUTHENTICATION
- CREATE AUTHENTICATION
- DROP AUTHENTICATION
- GRANT (authentication)

REVOKE (database)

Revokes database privileges from users and roles .

Syntax

```
REVOKE [ GRANT OPTION FOR ] { privilege[,...] | ALL [ PRIVILEGES ] }
  ON DATABASE db-spec
  FROM grantee[,...]
  [ CASCADE ]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

privilege

The database privilege to revoke, one of the following:

- CREATE : Create schemas.
- TEMP : Create temporary tables.

ALL [PRIVILEGES]

Revokes all database privileges that also belong to the revoker. Users cannot revoke privileges that they themselves lack.

The optional keyword PRIVILEGES is supported to comply with the SQL standard.

ON DATABASE *db-spec*

Specifies the current database, set to the database name or DEFAULT .

`grantee`

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Privileges

Non-superuser, one of the following:

- Ownership
- GRANT OPTION on the object

Examples

Revoke user Fred 's privilege to create schemas in the current database:

```
=> REVOKE CREATE ON DATABASE DEFAULT FROM Fred;
```

Revoke user Fred 's privilege to create temporary tables in the current database:

```
=> REVOKE TEMP ON DATABASE DEFAULT FROM Fred;
```

See also

- GRANT (database)
- Granting and revoking privileges

REVOKE (key)

Revokes privileges on a cryptographic key from a user or role.

Important

Because certificates depend on their underlying key, DROP privileges on a key effectively act as DROP privileges on its associated certificate when used with DROP KEY...CASCADE .

To grant privileges on a key, see GRANT (key) .

Syntax

```
REVOKE [ GRANT OPTION FOR ] { privilege[,...] | ALL [ PRIVILEGES ] } ON KEY
   key_name[,...]
   FROM user[,...]
```

Parameters

`GRANT OPTION FOR`

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

privilege

A privilege, one of the following:

- USAGE: Allows a user to perform the following actions:
  - View the contents of the key.
  - Create or sign certificates using the key.

  USAGE on the key also gives implicit USAGE privileges on a certificate that uses it as its private key. Users can also get these privileges from ownership of the key or certificate. USAGE privileges on a certificate allow a user to perform the following actions:
  - View the contents of the certificate.
  - Add (with CREATE or ALTER) the certificate to a TLS Configuration.
  - Reuse the CA certificate when importing certificates signed by it. For example, if a user imports a chain of certificates A > B > C and have USAGE on B , the database reuses B (as opposed to creating a duplicate of B).
  - Specify that the CA certificate signed an imported certificate. For example, if certificate B signed certificate C , USAGE on B allows a user to import C and specify that it was SIGNED BY B .
- DROP
- ALTER: Allows a user to see the key and its associated certificates in their respective system tables, but not their contents.

key_name

The target key .

user

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

Privileges

Non-superuser:

- Owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

Examples

The following example revokes DROP privileges on a key (and, by extension, its associated certificate) from a user:

```
=> REVOKE USAGE ON KEY new_key FROM u1;
REVOKE PRIVILEGE
```

REVOKE (library)

Revokes library privileges from users and roles .

Syntax

```
REVOKE [ GRANT OPTION FOR ] { USAGE | ALL [ PRIVILEGES ] }
  ON LIBRARY [[database.]schema.]library[,...]
  FROM grantee[,...]
  [ CASCADE ]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

USAGE

Revokes access to the specified libraries.

Important

Privileges on functions in these libraries must be separately revoked.

ALL [PRIVILEGES]

Revokes all library privileges that also belong to the revoker. Users cannot revoke privileges that they themselves lack. The optional keyword PRIVILEGES conforms with the SQL standard.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

library

The target library.

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Examples

These commands show how to create a new library, and then grant and revoke user Fred 's USAGE privilege on that library.

```
=> CREATE LIBRARY MyFunctions AS 'home/dbadmin/my_functions.so';
=> GRANT USAGE ON LIBRARY MyFunctions TO Fred;
=> REVOKE USAGE ON LIBRARY MyFunctions FROM Fred;
```

See also

- GRANT (library)
- Granting and revoking privileges

REVOKE (model)

Revokes model privileges from users and roles .

Syntax

```
REVOKE [ GRANT OPTION FOR ] { privilege[,...] | ALL [ PRIVILEGES ] }
  ON MODEL [[database.]schema.]model-name [,...]
  FROM grantee[,...]
  [ CASCADE ]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

USAGE

One of the following privileges:

- USAGE: Usage of the specified models
- ALTER
- DROP

ALL [PRIVILEGES]

Revokes all model privileges that also belong to the revoker. Users cannot revoke privileges that they themselves lack.

The optional keyword PRIVILEGES conforms with the SQL standard.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

model-name

Name of the target model.

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Privileges

Non-superuser, one of the following:

- Ownership
- GRANT OPTION on the object

Examples

Revoke user Fred 's USAGE privilege on model mySvmClassModel :

```
=> REVOKE USAGE ON mySvmClassModel FROM Fred;
```

See also

- GRANT (model)
- Managing model security

REVOKE (procedure)

Revokes procedure privileges from users and roles .

Syntax

```
REVOKE [ GRANT OPTION FOR ] { EXECUTE | ALL PRIVILEGES }
  ON PROCEDURE [[database.]schema.]procedure( [argument-list] )[,...]
  FROM grantee[,...]
  [ CASCADE ]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

EXECUTE

Revokes grantees ability to run the specified procedures.

ALL [PRIVILEGES]

Revokes all procedure privileges that also belong to the revoker. Users cannot revoke privileges that they themselves lack.

The optional keyword PRIVILEGES conforms with the SQL standard.

[database .] schema

Database and schema . The default schema is public . If you specify a database, it must be the current database.

procedure

The target procedure.

argument-list

A comma-delimited list of procedure arguments, where each argument is specified as follows:

```
[argname] argtype
```

If the procedure is defined with no arguments, supply an empty argument list.

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Privileges

Non-superuser, one of the following:

- Ownership
- GRANT OPTION on the object

Examples

This example revokes user Bob 's execute privilege on the tokenize procedure.

```
=> REVOKE EXECUTE ON PROCEDURE tokenize(varchar) FROM Bob;
```

See also

- GRANT (procedure)
- Granting and revoking privileges

REVOKE (Resource pool)

Revokes resource pool access privileges from users and roles .

Vertica checks resource pool privileges at runtime. Revoking a user's privileges for a resource pool can have an immediate effect on the user's current session. For example, a user query might require USAGE privileges on a resource pool. If you revoke those privileges from that user, subsequent attempts by the user to execute that query fail and return with an error message.

Syntax

```
REVOKE  [ GRANT OPTION FOR ] { USAGE | ALL PRIVILEGES }
  ON RESOURCE POOL resource-pool[,...]
  [FOR SUBCLUSTER subcluster | FOR CURRENT SUBCLUSTER]
  FROM grantee[,...]
  [ CASCADE ]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

USAGE

Revokes grantee's access to the specified resource pool.

ALL PRIVILEGES

Revokes all resource pool privileges that also belong to the revoker. Users cannot revoke privileges that they themselves lack.

The optional keyword PRIVILEGES conforms with the SQL standard.

resource-pool

The target resource pool.

subcluster

The subcluster for the resource pool.

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Privileges

Non-superuser, one of the following:

- Ownership
- GRANT OPTION on the object

Examples

Revoke user Joe 's USAGE privileges on resource pool Joe_pool .

```
=> REVOKE USAGE ON RESOURCE POOL Joe_pool FROM Joe;
REVOKE PRIVILEGE
```

Revoke user Joe 's USAGE privileges on resource pool Joe_pool for subcluster sub1 .

```
=> REVOKE USAGE ON RESOURCE POOL Joe_pool FOR SUBCLUSTER sub1 FROM Joe;
REVOKE PRIVILEGE
```

See also

- GRANT (Resource pool)
- Granting and revoking privileges

REVOKE (Role)

Revokes a role from users and roles .

Syntax

```
REVOKE [ ADMIN OPTION FOR ] role[,...]
  FROM grantee[,...]
  [ CASCADE ]
```

Parameters

ADMIN OPTION FOR

Revokes from the grantees the authority to assign the specified roles to other users or roles. Current roles for grantees remain unaffected. If you omit this clause, Vertica revokes role assignment privileges and the current roles .

role

Role to revoke.

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Privileges

One of the following:

- Superuser
- Privileges grantee who was given the option (WITH ADMIN OPTION) of extending these privileges to other users

Examples

This example shows the revocation of the pseudosuperuser role from the dbadmin user:

```
=> REVOKE pseudosuperuser from dbadmin;
```

This example shows the revocation of administration access from the dbadmin user for the pseudosuperuser role. The ADMIN OPTION command does not remove the pseudosuperuser role.

```
=> REVOKE ADMIN OPTION FOR pseudosuperuser FROM dbadmin;
```

Notes

If the role you are trying to revoke was not already granted to the user, Vertica returns a NOTICE:

```
=> REVOKE commentor FROM Sue;
NOTICE 2022:  Role "commentor" was not already granted to user "Sue"
REVOKE ROLE
```

See also

- GRANT (Role)
- Granting and revoking privileges

REVOKE (schema)

Revokes schema privileges from users and roles .

Syntax

```
REVOKE [ GRANT OPTION FOR ] { privilege[,...] | ALL [ PRIVILEGES ] }
  ON SCHEMA [database.]schema[,...]
  FROM grantee[,...]
  [ CASCADE ]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

privilege

The schema privilege to revoke, one of the following:

- USAGE: Access objects in the specified schemas.
- CREATE: Create objects in the specified schemas.

You can also revoke privileges from tables and views that they inherited on creation from this schema. When you revoke inherited privileges at the schema level, Vertica automatically applies the revocation to all tables and views that inherited these privileges.

- SELECT: Query tables and views. SELECT privileges are granted by default to the PUBLIC role.
- INSERT: Insert rows, or and load data into tables with COPY .

> **Note**
> COPY FROM STDIN is allowed for users with INSERT privileges, while COPY FROM *file* requires admin privileges.

- UPDATE: Update table rows.
- DELETE: Delete table rows.
- REFERENCES: Create foreign key constraints on this table. This privilege must be set on both referencing and referenced tables.
- TRUNCATE: Truncate table contents. Non-owners of tables can also execute the following partition operations on them:
  - DROP_PARTITIONS
  - SWAP_PARTITIONS_BETWEEN_TABLES
  - MOVE_PARTITIONS_TO_TABLE
- ALTER: Modify the DDL of tables and views with ALTER TABLE and ALTER VIEW , respectively.
- DROP: Drop tables and views.

ALL [PRIVILEGES]

Revokes USAGE AND CREATE privileges. Users cannot revoke privileges that they themselves lack.

The optional keyword PRIVILEGES conforms with the SQL standard.

Important

Inherited privileges must be explicitly revoked.

[*database* .] *schema*

The schema on which to revoke privileges. If you specify a database, it must be the current database.

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Privileges

Non-superuser, one of the following:

- Ownership
- GRANT OPTION on the object

Examples

Revoke user Joe 's USAGE privilege on schema online_sales .

```
=> REVOKE USAGE ON SCHEMA online_sales FROM Joe;
REVOKE PRIVILEGE
```

See also

- GRANT (schema)
- Granting and revoking privileges

REVOKE (sequence)

Revokes sequence privileges from users and roles .

Syntax

```
REVOKE [ GRANT OPTION FOR ] { privilege[,...] | ALL [ PRIVILEGES ] }
  ON {
    SEQUENCE [[database.]schema.]sequence[,...]
    | ALL SEQUENCES IN SCHEMA [database.]schema[,...] }
  FROM grantee[,...]
  [ CASCADE ]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

privilege

One of the following privileges:

- SELECT: Execute functions CURRVAL and NEXTVAL on the specified sequences.
- ALTER: Modify a sequence's DDL with ALTER SEQUENCE
- DROP: Drop this sequence with DROP SEQUENCE .

ALL [PRIVILEGES]

Revokes all sequence privileges that also belong to the revoker. Users cannot revoke privileges that they themselves lack.

The optional keyword PRIVILEGES is supported to comply with the SQL standard.

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

SEQUENCE *sequence*

Specifies the sequence on which to revoke privileges.

ALL SEQUENCES IN SCHEMA *schema*

Revokes the specified privileges on all sequences in schema *schema* .

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Privileges

Non-superuser, one of the following:

- Ownership
- GRANT OPTION on the object

Examples

Revoke user Joe 's privileges on sequence my_seq .

```
=> REVOKE ALL PRIVILEGES ON SEQUENCE my_seq FROM Joe;
REVOKE PRIVILEGE
```

See also

- GRANT (sequence)
- Granting and revoking privileges

REVOKE (storage location)

Revokes privileges on a USER-defined storage location from users and roles . For more information, see Creating storage locations .

> **Note**
> If the storage location is dropped , Vertica automatically revokes all privileges on it.

Syntax

```
REVOKE [ GRANT OPTION FOR ] { privilege[,...] | ALL [ PRIVILEGES ] }
  ON LOCATION 'path' [ ON node ]
  FROM grantee[,...]
  [ CASCADE ]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

privilege

One of the following privileges:

- READ : Copy data from files in the storage location into a table.
- WRITE : Export data from the database to the storage location. With WRITE privileges, grantees can also save COPY statement rejected data and exceptions files to the storage location.

ALL [PRIVILEGES]

Revokes all storage location privileges that also belong to the revoker. Users cannot revoke privileges that they themselves lack.

The optional keyword PRIVILEGES is supported to comply with the SQL standard.

ON LOCATION ' *path* ' [ON *node*]

Specifies the path name mount point of the storage location. If qualified by ON NODE , Vertica revokes access to the storage location residing on *node* .

If no node is specified, the revoke operation applies to all nodes on the specified path. All nodes must be on the specified path; otherwise, the entire revoke operation rolls back.

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Privileges

Non-superuser, one of the following:

- Ownership
- GRANT OPTION on the object

Examples

See GRANT (storage location) .

See also

Granting and revoking privileges
REVOKE (table)

Revokes table privileges from users and roles .

Syntax

```
REVOKE [ GRANT OPTION FOR ] { privilege[,...] | ALL [ PRIVILEGES ] }
  ON {
    [ TABLE ] [[database.]schema.]table[,...]
    | ALL TABLES IN SCHEMA [database.]schema[,...] }
  FROM grantee[,...]
  [ CASCADE ]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

privilege

One of the following privileges:

- SELECT: Quer y tables. SELECT privileges are granted by default to the PUBLIC role.
- INSERT: Insert table rows with INSERT , and load data with COPY .

> **Note**
> COPY FROM STDIN is allowed for users with INSERT privileges, while COPY FROM *file* requires admin privileges.

- UPDATE: Update table rows.
- DELETE: Delete table rows.
- REFERENCES: Create foreign key constraints on this table. This privilege must be set on both referencing and referenced tables.
- TRUNCATE: Truncate table contents. Non-owners of tables can also execute the following partition operations on them:
  - DROP_PARTITIONS
  - SWAP_PARTITIONS_BETWEEN_TABLES
  - MOVE_PARTITIONS_TO_TABLE
- ALTER: Modify a table's DDL with ALTER TABLE .
- DROP: Drop a table .

ALL [PRIVILEGES]

Revokes all table privileges that also belong to the revoker. Users cannot revoke privileges that they themselves lack.

The optional keyword PRIVILEGES is supported to comply with the SQL standard.

[*database.*] *schema*

Specifies a schema , by default public . If *schema* is any schema other than public , you must supply the schema name. For example:

```
myschema.thisDbObject
```

One exception applies: you can specify system tables without their schema name.
If you specify a database, it must be the current database.

TABLE *table*

Specifies the table on which to revoke privileges.

ON ALL TABLES IN SCHEMA *schema*

Revokes the specified privileges on all tables and views in schema *schema* .

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Privileges

Non-superuser, one of the following:

- Ownership
- GRANT OPTION on the object

Examples

Revoke user Joe 's privileges on table customer_dimension .

```
=> REVOKE ALL PRIVILEGES ON TABLE customer_dimension FROM Joe;
REVOKE PRIVILEGE
```

See also

- GRANT (table)
- Granting and revoking privileges

REVOKE (TLS configuration)

Revokes privileges granted on one or more TLS Configurations from users and roles .

Syntax

```
REVOKE [ GRANT OPTION FOR ]
  { ALL | [ privilege[,...] ]}
  ON TLS CONFIGURATION tls_configuration[,...]
  FROM grantee [,...]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

privilege

A privilege, one of the following:

- USAGE: Allows the user to set the TLS Configuration for a type of connection and view its contents in the system table TLS_CONFIGURATIONS . For details, see Security parameters .
- DROP
- ALTER

tls_configuration

The TLS Configuration on which to revoke privileges.

grantee

Specifies who is granted privileges, one of the following:

- *user-name*
- *role*
- PUBLIC : Default role of all users

Privileges

Non-superusers require USAGE on the schema and one of the following:

- Owner
- Privileges grantee given the option (WITH GRANT OPTION) of granting privileges to other users or roles.

Examples

To revoke ALTER privileges on the TLS Configuration server from the role client_server_tls_manager :

```
=> REVOKE ALTER ON TLS CONFIGURATION server FROM client_server_tls_manager;
```

REVOKE (user defined extension)

Revokes privileges on one or more user-defined extensions from users and roles .

Syntax

```
REVOKE [ GRANT OPTION FOR ] { EXECUTE | ALL PRIVILEGES }
  ON {
     UDx-type [[database.]schema.]function-name( [argument-list] )[,...]
     | ALL FUNCTIONS IN SCHEMA schema[,...] }
  FROM grantee[,...]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

EXECUTE

Revokes grantees ability to run the specified functions.

ALL [PRIVILEGES]

Revokes all function privileges that also belong to the revoker. Users cannot revoke privileges that they themselves lack.

The optional keyword PRIVILEGES conforms with the SQL standard.

ON *UDx-type*

Specifies the function's user-defined extension (UDx) type, where *UDx-type* is one of the following:

- FUNCTION
- AGGREGATE FUNCTION
- ANALYTIC FUNCTION
- TRANSFORM FUNCTION
- FILTER
- PARSER
- SOURCE

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

function-name

The name of the user-defined function on which to revoke privileges.

ON ALL FUNCTIONS IN SCHEMA *schema*

Revokes privileges on all functions in the specified schema.

argument-list

Required for all polymorphic functions, a comma-delimited list of function arguments, where each argument is specified as follows:

```
[argname] argtype
```

If the procedure is defined with no arguments, supply an empty argument list.

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Privileges

Non-superuser, one of the following:

- Owner of the target function
- Privileges grantee who was given the option (WITH GRANT OPTION) of extending these privileges to other users

Examples

Revoke EXECUTE privileges from user Bob on function myzeroifnull :

```
=> REVOKE EXECUTE ON FUNCTION myzeroifnull (x INT) FROM Bob;
```

Revoke all privileges from user Doug on function Pagerank :

```
=> REVOKE ALL ON TRANSFORM FUNCTION Pagerank (t float) FROM Doug;
```

Revoke EXECUTE privileges on all functions in the zero-schema schema from user Bob:

```
=> REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA zero-schema FROM Bob;
```

Revoke EXECUTE privileges from user Bob on the tokenize function:

```
=> REVOKE EXECUTE ON TRANSFORM FUNCTION tokenize(VARCHAR) FROM Bob;
```

Revoke all privileges on the ExampleSource() source from user Alice:

```
=> REVOKE ALL ON SOURCE ExampleSource() FROM Alice;
```

See also

- GRANT (User Defined Extension)
- Granting and Revoking Privileges
- Developing user-defined extensions (UDxs)

REVOKE (view)

Revokes privileges on a view from users and roles .

Syntax

```
REVOKE [ GRANT OPTION FOR ] { privilege[,...] | ALL [ PRIVILEGES ] }
  ON [[database.]schema.]view[,...]
  FROM grantee[,...]
  [ CASCADE ]
```

Parameters

GRANT OPTION FOR

Revokes the grant option for the specified privileges. Current privileges for grantees remain unaffected. If you omit this clause, Vertica revokes both the grant option and current privileges.

privilege

One of the following:

- SELECT: Query the specified views.
- ALTER: Modify a view's DDL with ALTER VIEW
- DROP: Drop this view with DROP VIEW .

ALL PRIVILEGES

Revokes all privileges that pertain to views that also belong to the revoker. Users cannot revoke privileges that they themselves lack.

The optional keyword PRIVILEGES conforms with the SQL standard.

[*database.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

view

The view on which to revoke privileges.

grantee

Specifies whose privileges are revoked, one of the following:

- User name
- Role
- PUBLIC : The default role of all users

CASCADE

I f the target grantees have a grant option to extend the specified privileges to other users, CASCADE specifies to search for these users and revoke the privileges from them also.

Examples

Revoke SELECT privileges from user Joe on view test_view .

```
=> REVOKE SELECT ON test_view FROM Joe;
REVOKE PRIVILEGE
```

See also

- GRANT (view)
- Granting and revoking privileges

ROLLBACK

Ends the current transaction and discards all changes that occurred during the transaction.

Syntax

```
ROLLBACK [ WORK | TRANSACTION ]
```

Parameters

WORK | TRANSACTION

Have no effect; they are optional keywords for readability.

Privileges

None

Notes

When an operation is rolled back, any locks that are acquired by the operation are also rolled back.

ABORT is a synonym for ROLLBACK.

Examples

This example shows how to roll back from a DELETE transaction.

```
=> SELECT * FROM sample_table;
a
---
1
(1 row)

=> DELETE FROM sample_table WHERE a = 1;

=> SELECT * FROM sample_table;
a
---
(0 rows)
=> ROLLBACK;
=> SELECT * FROM sample_table;
a
---
1
(1 row)
```

This example shows how to roll back the changes you made since the BEGIN statement.

```
=> BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED READ ONLY;
BEGIN
=> ROLLBACK TRANSACTION;
ROLLBACK
```

See also

- Transactions
- Creating and rolling back transactions
- BEGIN
- COMMIT
- END
- START TRANSACTION

ROLLBACK TO SAVEPOINT

Rolls back all commands that have been entered within the transaction since the given savepoint was established.

Syntax

```
ROLLBACK TO [SAVEPOINT] savepoint_name
```

Parameters

savepoint_name

Specifies the name of the savepoint to roll back to.

Privileges

None

Notes

- The savepoint remains valid and can be rolled back to again later if needed.
- When an operation is rolled back, any locks that are acquired by the operation are also rolled back.
- ROLLBACK TO SAVEPOINT implicitly destroys all savepoints that were established after the named savepoint.

Examples

The following example rolls back the values 102 and 103 that were entered after the savepoint, my_savepoint , was established. Only the values 101 and 104 are inserted at commit.

```
=> INSERT INTO product_key VALUES (101);
=> SAVEPOINT my_savepoint;
=> INSERT INTO product_key VALUES (102);
=> INSERT INTO product_key VALUES (103);
=> ROLLBACK TO SAVEPOINT my_savepoint;
=> INSERT INTO product_key VALUES (104);
=> COMMIT;
```

See also

- RELEASE SAVEPOINT
- SAVEPOINT

SAVE QUERY

Saves an input query to associate with a custom directed query.

Syntax

```
SAVE QUERY input-query
```

Arguments

input-query

The input query to associate with a custom directed query. The input query supports only one optimizer hint, :v .

Privileges

Superuser

Usage

SAVE QUERY saves the specified input query for use by the next invocation of CREATE DIRECTED QUERY CUSTOM . CREATE DIRECTED QUERY CUSTOM pairs the saved query with its annotated query argument to create a directed query. Both statements must be issued in the same user session.

The saved query remains available until the one of the following events occurs:

- The next invocation of CREATE DIRECTED QUERY, whether invoked with CUSTOM or OPTIMIZER.
- Another invocation of SAVE QUERY.
- The session ends.

Caution

Vertica associates a saved query with a directed query without checking whether the input and annotated queries are compatible. Be careful to sequence SAVE QUERY and CREATE DIRECTED QUERY CUSTOM so the saved and directed queries are correctly matched.

Examples

See Custom directed queries .

SAVEPOINT

Creates a special mark, called a savepoint, inside a transaction. A savepoint allows all commands that are executed after it was established to be rolled back, restoring the transaction to the state it was in at the point in which the savepoint was established.

Tip

Savepoints are useful when creating nested transactions. For example, a savepoint could be created at the beginning of a subroutine. That way, the result of the subroutine could be rolled back if necessary.

Syntax

```
SAVEPOINT savepoint_name
```

Parameters

savepoint_name

Specifies the name of the savepoint to create.

Privileges

None

Notes

- Savepoints are local to a transaction and can only be established when inside a transaction block.
- Multiple savepoints can be defined within a transaction.
- If a savepoint with the same name already exists, it is replaced with the new savepoint.

Examples

The following example illustrates how a savepoint determines which values within a transaction can be rolled back. The values 102 and 103 that were entered after the savepoint, my_savepoint , was established are rolled back. Only the values 101 and 104 are inserted at commit.

```
=> INSERT INTO T1 (product_key) VALUES (101);
=> SAVEPOINT my_savepoint;
=> INSERT INTO T1 (product_key) VALUES (102);
=> INSERT INTO T1 (product_key) VALUES (103);
=> ROLLBACK TO SAVEPOINT my_savepoint;
=> INSERT INTO T1 (product_key) VALUES (104);
=> COMMIT;
=> SELECT product_key FROM T1;
.
.
.
101
104
(2 rows)
```

See also

- RELEASE SAVEPOINT
- ROLLBACK TO SAVEPOINT

SELECT

Returns a result set from one or more data sources— tables , views , joined tables , and named subqueries .

Syntax

```
[ AT epoch ] [ WITH-clause ] SELECT [ ALL | DISTINCT ]
   { * | { MATCH_COLUMNS('pattern') | expression [ [AS] alias ] }[,...] }
   [ into-table-clause ]
   [ from-clause ]
   [ where-clause ]
   [ time-series-clause ]
   [ group-by-clause[,...] ]
   [ having-clause[,...] ]
   [ match-clause ]
   [ union-clause ]
   [ intersect-clause ]
   [ except-clause ]
   [ order-by-clause [ offset-clause ]]
   [ limit-clause ]
   [ FOR UPDATE [ OF table-name[,...] ] ]
```

> **Note**
> SELECT statements can also embed various directives, or *hints* , that let you control how a given query is handled—for example, join hints such as JTYPE , which enforces the join type (merge or hash join).
>
> For details on using Vertica hints, see Hints .

Parameters

AT *epoch*

Returns data from the specified epoch, where *epoch* is one of the following:

- EPOCH LATEST : Return data up to but not including the current epoch. The result set includes data from the latest committed DML transaction.
- EPOCH *integer* : Return data up to and including the *integer* -specified epoch.
- TIME ' *timestamp* ' : Return data from the *timestamp* -specified epoch.

> **Note**
> These options are ignored if used to query temporary or external tables.

See Epochs for additional information about how Vertica uses epochs.
For details, see Historical queries .

ALL | DISTINCT

- ALL (default): Retains duplicate rows in result set or group.
- DISTINCT : Removes duplicate rows from the result set or group.

The ALL or DISTINCT qualifier must immediately follow the SELECT keyword. Only one instance of this keyword can appear in the select list.

*****

Lists all columns in the queried tables.

> **Caution**
> Selecting all columns from the queried tables can produce a very large wide set, which can adversely affect performance.

MATCH_COLUMNS (' *pattern* ')

Returns all columns in the queried tables that match *pattern* .

***expression* [[AS] *alias*]**

An expression that typically resolves to column data from the queried tables—for example, names of columns that are specified in the FROM clause; also:

- Literals (constants)
- Aggregate expressions
- CASE expressions
- SQL functions
- Subqueries in the SELECT list

You can optionally assign a temporary alias to each column expression and reference that alias elsewhere in the SELECT statement—for example, in the query predicate or ORDER BY clause. Vertica uses the alias as the column heading in query output.

FOR UPDATE

Specifies to obtain an X lock on all tables specified in the query, most often used from READ COMMITTED isolation.

FOR UPDATE requires update/delete permissions on the queried tables and cannot be issued from a read-only transaction.

Privileges

Non-superusers:

- USAGE on the schema
- SELECT on the table or view

> **Note**
> As view owner, you can grant other users SELECT privilege on the view only if one of the following is true:
>
> - You own the view's base table.
> - You have SELECT...WITH GRANT OPTION privilege on the view's base table.

Examples

When multiple clients run transactions as in the following example query, deadlocks can occur if FOR UPDATE is not used. Two transactions acquire an S lock, and when both attempt to upgrade to an X lock, they encounter deadlocks:

```
=> SELECT balance FROM accounts WHERE account_id=3476 FOR UPDATE;
   ...
=> UPDATE accounts SET balance = balance+10 WHERE account_id=3476;
=> COMMIT;
```

See also

- LOCKS
- Analytic functions
- SQL analytics
- Time series analytics
- Event series pattern matching
- Subqueries
- Joins

In this section

- EXCEPT clause
- FROM clause
- GROUP BY clause
- HAVING clause
- INTERSECT clause
- INTO TABLE clause
- LIMIT clause
- MATCH clause
- MINUS clause
- OFFSET clause
- ORDER BY clause
- TIMESERIES clause
- UNION clause
- WHERE clause
- WITH clause

EXCEPT clause

Combines two or more SELECT queries. EXCEPT returns distinct results of the left-hand query that are not also found in the right-hand query.

> **Note**
> MINUS is an alias for EXCEPT.

Syntax

```
SELECT
  EXCEPT except-query[...]
  [ ORDER BY { column-name  | ordinal-number } [ ASC | DESC ] [,...] ]
  [ LIMIT { integer | ALL } ]
  [ OFFSET integer ]
```

Notes

- Use the EXCEPT clause to filter out specific results from a SELECT statement. The EXCEPT query operates on the results of two or more SELECT queries. It returns only those rows in the left-hand query that are not also present in the right-hand query.
- Vertica evaluates multiple EXCEPT clauses in the same SELECT query from left to right, unless parentheses indicate otherwise.
- You cannot use the ALL keyword with an EXCEPT query.
- The results of each SELECT statement must be union compatible. Each statement must return the same number of columns, and the corresponding columns must have compatible data types. For example, you cannot use the EXCEPT clause on a column of type INTEGER and a column of type VARCHAR. If statements do not meet these criteria, Vertica returns an error.

  > **Note**
  > The Data type coercion chart lists the data types that can be cast to other data types. If one data type can be cast to the other, those two

> data types are compatible.

- You can use EXCEPT in FROM, WHERE, and HAVING clauses.
- You can order the results of an EXCEPT operation by including an ORDER BY operation in the statement. When you write the ORDER BY list, specify the column names from the leftmost SELECT statement, or specify integers that indicate the position of the columns by which to sort.
- The rightmost ORDER BY, LIMIT, or OFFSET clauses in an EXCEPT query do not need to be enclosed in parentheses, because the rightmost query specifies that Vertica perform the operation on the results of the EXCEPT operation. Any ORDER BY, LIMIT, or OFFSET clauses contained in SELECT queries that appear earlier in the EXCEPT query must be enclosed in parentheses.
- Vertica supports EXCEPT noncorrelated subquery predicates. For example:

```
=> SELECT * FROM T1
  WHERE T1.x IN
    (SELECT MAX(c1) FROM T2
     EXCEPT
       SELECT MAX(cc1) FROM T3
     EXCEPT
       SELECT MAX(d1) FROM T4);
```

Examples

Consider the following three tables:

Company_A

```
 Id  | emp_lname |      dept      | sales
-----+-----------+----------------+-------
1234 | Stephen   | auto parts     |  1000
5678 | Alice     | auto parts     |  2500
9012 | Katherine | floral         |   500
3214 | Smithson  | sporting goods |  1500
(4 rows)
```

Company_B

```
 Id  | emp_lname |    dept     | sales
-----+-----------+-------------+-------
4321 | Marvin    | home goods  |   250
8765 | Bob       | electronics | 20000
9012 | Katherine | home goods  |   500
3214 | Smithson  | home goods  |  1500
(4 rows)
```

Company_C

```
 Id  | emp_lname |      dept      | sales
-----+-----------+----------------+-------
3214 | Smithson  | sporting goods |  1500
5432 | Madison   | sporting goods |   400
7865 | Cleveland | outdoor        |  1500
1234 | Stephen   | floral         |  1000
(4 rows)
```

The following query returns the IDs and last names of employees that exist in Company_A, but not in Company_B:

```
=> SELECT id, emp_lname FROM Company_A
   EXCEPT
   SELECT id, emp_lname FROM Company_B;
 id  | emp_lname
-----+-----------
1234 | Stephen
5678 | Alice
(2 rows)
```

The following query sorts the results of the previous query by employee last name:

```
=> SELECT id, emp_lname FROM Company_A
   EXCEPT
   SELECT id, emp_lname FROM Company_B
   ORDER BY emp_lname ASC;
 id  | emp_lname
------+-----------
 5678 | Alice
 1234 | Stephen
(2 rows)
```

If you order by the column position, the query returns the same results:

```
=> SELECT id, emp_lname FROM Company_A
   EXCEPT
   SELECT id, emp_lname FROM Company_B
   ORDER BY 2 ASC;
 id  | emp_lname
------+-----------
 5678 | Alice
 1234 | Stephen
(2 rows)
```

The following query returns the IDs and last names of employees that exist in Company_A, but not in Company_B or Company_C:

```
=> SELECT id, emp_lname FROM Company_A
   EXCEPT
   SELECT id, emp_lname FROM Company_B
   EXCEPT
   SELECT id, emp_lname FROM Company_C;
 id  | emp_lname
------+-----------
 5678 | Alice
(1 row)
```

The following query shows the results of mismatched data types:

```
=> SELECT id, emp_lname FROM Company_A
   EXCEPT
   SELECT emp_lname, id FROM Company_B;
ERROR 3429:  For 'EXCEPT', types int and varchar are inconsistent
DETAIL:  Columns: id and emp_lname
```

Using the VMart example database, the following query returns information about all Connecticut-based customers who bought items through stores and whose purchases amounted to more than $500, except for those customers who paid cash:

```
=> SELECT customer_key, customer_name FROM public.customer_dimension
    WHERE customer_key IN (SELECT customer_key FROM store.store_sales_fact
      WHERE sales_dollar_amount > 500
      EXCEPT
      SELECT customer_key FROM store.store_sales_fact
      WHERE tender_type = 'Cash')
    AND customer_state = 'CT';
 customer_key |    customer_name
--------------+---------------------
      15084 | Doug V. Lampert
      21730 | Juanita F. Peterson
      24412 | Mary U. Garnett
      25840 | Ben Z. Taylor
      29940 | Brian B. Dobisz
      32225 | Ruth T. McNulty
      33127 | Darlene Y. Rodriguez
      40000 | Steve L. Lewis
      44383 | Amy G. Jones
      46495 | Kevin H. Taylor
(10 rows)
```

See also

- SELECT
- INTERSECT clause
- UNION clause
- Subqueries

FROM clause

A comma-separated list of data sources to query.

Syntax

```
FROM dataset[,...] [ TABLESAMPLE(percent) ]
```

Parameters

dataset ``

A set of data to query, one of the following:

- Table reference
- Joined tables
- View
- Named subquery : *subquery* [AS] *name*

`TABLESAMPLE( *percent* )`

Specifies to return a random sampling of records, where *percent* specifies the approximate sampling size. The *percent* value must be between 0 and 100, exclusive, and can include decimal values. The number of records returned is not guaranteed to be the exact percentage specified.

All rows of the data have equal opportunities to be selected. Vertica performs sampling before applying other query filters.

Examples

Count all records in customer_dimension table:

```
=> SELECT COUNT(*) FROM customer_dimension;
 COUNT
-------
 50000
(1 row)
```

Return a small sampling of rows in table customer_dimension :

```
=> SELECT customer_name, customer_state FROM customer_dimension TABLESAMPLE(0.5) WHERE customer_state='IL';
   customer_name    | customer_state
--------------------+----------------
 Amy Y. McNulty     | IL
 Daniel C. Nguyen   | IL
 Midori O. Greenwood | IL
 Meghan U. Lampert  | IL
 Tiffany Y. Lang    | IL
 Laura S. King      | IL
 Steve T. Nguyen    | IL
 Craig S. Webber    | IL
 Luigi A. Lewis     | IL
 Mark W. Williams   | IL
(10 rows)
```

In this section

- Joined-table
- Table-reference

Joined-table

Specifies how to join tables.

Syntax

```
table-reference [ join-type ] JOIN table-reference [ TABLESAMPLE(percent) ] [ ON join-predicate ]
```

Arguments

table-reference

A table name, optionally qualified .

join-type

One of the following:

- INNER (default). INNER JOIN is equivalent to a query that specifies its join predicate in a WHERE clause.
- LEFT [OUTER]
- RIGHT [OUTER]
- FULL [OUTER]
- NATURAL
- CROSS

TABLESAMPLE(*percent*)

Use simple random sampling to return an approximate percentage of records. The percentage value must be greater than 0 and less than 100. All rows in the total potential return set are equally eligible to be included in the sampling. Vertica performs this sampling before other filters in the query are applied. The number of records returned is not guaranteed to be exactly *percent* .

The TABLESAMPLE option is valid only with user-defined tables and Data Collector (DC) tables. Views and system tables are not supported.

ON *join-predicate*

Specifies the columns to join on . Invalid for NATURAL and CROSS joins, required for all other join types.

Alternative JOIN syntax options

Vertica supports two older join syntax conventions:

- Table joins specified by join predicate in a WHERE clause
- Table joins specified by a USING clause

For details, see Join Syntax .

Examples

The following SELECT statement qualifies its JOIN clause with the TABLESAMPLE option:

```
=> SELECT user_id.id, user_name.name FROM user_name TABLESAMPLE(50)
    JOIN user_id TABLESAMPLE(50) ON user_name.id = user_id.id;
 id  |  name
-----+--------
 489 | Markus
2234 | Cato
 763 | Pompey
(3 rows)
```

Table-reference

Syntax

```
[[database.]schema.]table[ [AS] alias]
```

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

A table in the logical schema.

[AS] *alias*

A temporary name used for references to *table* .

GROUP BY clause

Use the GROUP BY clause with aggregate functions in a SELECT statement to collect data across multiple records. Vertica groups the results into one or more sets of rows that match an expression.

The GROUP BY clause without aggregates is similar to using SELECT DISTINCT .

ROLLUP is an extension to the GROUP BY clause. ROLLUP performs subtotal aggregations.

Syntax

```
GROUP BY [/*+GBYTYPE(algorithm)*/] { expression | aggregate-expression }[,...]
```

Arguments

/*+ GBYTYPE (*algorithm*)*/

Specifies which algorithm has precedence for implementing this GROUP BY clause, over the algorithm the Vertica query optimizer might otherwise choose. You can set *algorithm* to one of the following values:

- HASH : GROUPBY HASH algorithm
- PIPE : GROUPBY PIPELINED algorithm

For more information about both algorithms, see GROUP BY implementation options .

expression

Any expression, including constants and column references in the tables specified in the FROM clause . For example:

```
column,... column, (expression)
```

aggregate-expression

An ordered list of columns, expressions, CUBE, GROUPING SETS , or ROLLUP aggregates.

You can include CUBE and ROLLUP aggregates within a GROUPING SETS aggregate. CUBE and ROLLUP aggregates can result in a large amount of output. In that case, use GROUPING SETS to return only certain results.
You cannot include any aggregates within a CUBE or ROLLUP expression.

You can append multiple GROUPING SETS , CUBE , or ROLLUP aggregates in the same query. For example:

```
GROUP BY a,b,c,d, ROLLUP(a,b)
GROUP BY a,b,c,d, CUBE((a,b),c,d)
GROUP BY a,b,c,d, CUBE(a,b), ROLLUP (c,d)
GROUP BY ROLLUP(a), CUBE(b), GROUPING SETS(c)
GROUP BY a,b,c,d, GROUPING SETS ((a,d),(b,c),CUBE(a,b))
GROUP BY a,b,c,d, GROUPING SETS ((a,d),(b,c),(a,b),(a),(b),())
```

Usage considerations

- *expression* cannot include aggregate functions . However, you can use the GROUP BY clause with CUBE, GROUPING SETS, and ROLLUP to return summary values for each group.
- When you create a GROUP BY clause, you must include all non-aggregated columns that appear in the SELECT list.
- If the GROUP BY clause includes a WHERE clause, Vertica ignores all rows that do not satisfy the WHERE clause.

Examples

This example shows how to use the WHERE clause with GROUP BY . In this case, the example retrieves all employees whose last name begins with S, and ignores all rows that do not meet this criteria. The G ROUP BY clause uses the ILIKE function to retrieve only last names beginning with S. The aggregate function SUM computes the total vacation days for each group.

```
=> SELECT employee_last_name, SUM(vacation_days)
   FROM employee_dimension
   WHERE employee_last_name ILIKE 'S%'
   GROUP BY employee_last_name;
 employee_last_name | SUM
--------------------+------
 Sanchez            | 2892
 Smith              | 2672
 Stein              | 2660
(3 rows)
```

The GROUP BY clause in the following example groups results by vendor region, and vendor region's biggest deal:

```
=> SELECT vendor_region, MAX(deal_size) AS "Biggest Deal"
   FROM vendor_dimension
   GROUP BY vendor_region;
 vendor_region | Biggest Deal
---------------+--------------
 East          |       990889
 MidWest       |       699163
 NorthWest     |        76101
 South         |       854136
 SouthWest     |       609807
 West          |       964005
(6 rows)
```

The following query modifies the previous one with a HAVING clause, which specifies to return only groups whose maximum deal size exceeds $900,000:

```
=> SELECT vendor_region, MAX(deal_size) as "Biggest Deal"
   FROM vendor_dimension
   GROUP BY vendor_region
   HAVING MAX(deal_size) > 900000;
 vendor_region | Biggest Deal
---------------+--------------
 East          |       990889
 West          |       964005
(2 rows)
```

You can use the GROUP BY clause with one-dimensional arrays of scalar types. In the following example, grants is an ARRAY[VARCHAR] and grant_values is an ARRAY[INT].

```
=> SELECT department, grants, SUM(apply_sum(grant_values))
   FROM employees
   GROUP BY grants, department;
 department |          grants          |  SUM
------------+--------------------------+--------
 Physics    | ["US-7376","DARPA-1567"] | 235000
 Astronomy  | ["US-7376","DARPA-1567"] |   9000
 Physics    | ["US-7376"]              |  30000
(3 rows)
```

The GROUP BY clause without aggregates is similar to using SELECT DISTINCT . For example, the following two queries return the same results:

```
=> SELECT DISTINCT household_id FROM customer_dimension;
=> SELECT household_id FROM customer_dimension GROUP BY household_id;
```

See also

- CUBE aggregate
- GROUP_ID
- GROUPING
- GROUPING_ID
- GROUPING SETS aggregate
- ROLLUP

In this section

- CUBE aggregate
- GROUPING SETS aggregate
- ROLLUP aggregate

CUBE aggregate

Automatically performs all possible aggregations of the specified columns, as an extension to the GROUP BY clause.

You can use the ROLLUP clause with three grouping functions:

- GROUPING
- GROUP_ID
- GROUPING_ID

Syntax

```
GROUP BY group-expression[,...]
```

Parameters

group-expression ``

One or both of the following:

- An expression that is not an aggregate or a grouping function that includes constants and column references in FROM -specified tables. For example:

  ```
  column1, (column2+1), column3+column4
  ```

- A multilevel expression, one of the following:
  - ROLLUP
  - CUBE
  - GROUPING SETS

Restrictions

- GROUP BY CUBE does not order data. If you want to sort data, use the ORDER BY clause . The ORDER BY clause must come *after* the GROUP BY clause.
- You can use CUBE inside a GROUPING SETS expression, but not inside a ROLLUP expression or another CUBE expression.

Levels of CUBE aggregation

If *n* is the number of grouping columns, CUBE creates 2^n levels of aggregations. For example:

CUBE (A, B, C) creates all possible groupings, resulting in eight groups:

- (A, B, C)
- (A, B)
- (A, C)
- (B, C)
- (A)
- (B)
- (C)
- ()

If you increase the number of CUBE columns, the number of CUBE groupings increases exponentially. The CUBE query may be resource intensive and produce combinations that are not of interest. In that case, consider using the GROUPING SETS aggregate , which allows you to choose specific groupings.

Examples

Using CUBE to return all groupings

Suppose you have a table that contains information about family expenses for books and electricity:

```
=> SELECT * FROM expenses ORDER BY Category, Year;
 Year |  Category   | Amount
------+-------------+--------
2005  | Books       |  39.98
2007  | Books       |  29.99
2008  | Books       |  29.99
2005  | Electricity | 109.99
2006  | Electricity | 109.99
2007  | Electricity | 229.98
```

To aggregate the data by both Category and Year using the CUBE aggregate:

```
=> SELECT Category, Year, SUM(Amount) FROM expenses
   GROUP BY CUBE(Category, Year) ORDER BY 1, 2, GROUPING_ID();
 Category    | Year |  SUM
-------------+------+--------
 Books       | 2005 |  39.98
 Books       | 2007 |  29.99
 Books       | 2008 |  29.99
 Books       |      |  99.96
 Electricity | 2005 | 109.99
 Electricity | 2006 | 109.99
 Electricity | 2007 | 229.98
 Electricity |      | 449.96
             | 2005 | 149.97
             | 2006 | 109.99
             | 2007 | 259.97
             | 2008 |  29.99
             |      | 549.92
```

The results include subtotals for each category and year, and a grand total ($549.92).

Using CUBE with the HAVING clause

This example shows how you can restrict the GROUP BY results, use the HAVING clause with the CUBE aggregate. This query returns only the category totals and the full total:

```
=> SELECT Category, Year, SUM(Amount) FROM expenses
   GROUP BY CUBE(Category,Year) HAVING GROUPING(Year)=1;
 Category    | Year |  SUM
-------------+------+--------
 Books       |      |  99.96
 Electricity |      | 449.96
             |      | 549.92
```

The next query returns only the aggregations for the two categories for each year. The GROUPING ID function specifies to omit the grand total ($549.92):

```
=> SELECT Category, Year, SUM (Amount) FROM expenses
  GROUP BY CUBE(Category,Year) HAVING GROUPING_ID(Category,Year)<2
  ORDER BY 1, 2, GROUPING_ID();
 Category   | Year |  SUM
------------+------+--------
 Books      | 2005 |  39.98
 Books      | 2007 |  29.99
 Books      | 2008 |  29.99
 Books      |      |  99.96
 Electrical | 2005 | 109.99
 Electrical | 2006 | 109.99
 Electrical | 2007 | 229.98
 Electrical |      | 449.96
```

See also

- Data aggregation
- GROUP BY clause
- GROUP_ID
- GROUPING
- GROUPING_ID
- GROUPING SETS aggregate
- ROLLUP aggregate

GROUPING SETS aggregate

The GROUPING SETS aggregate is an extension to the GROUP BY clause that automatically performs subtotal aggregations on groupings that you specify.

You can use the GROUPING SETS clause with three grouping functions:

- GROUPING
- GROUP_ID
- GROUPING_ID

To sort data, use the ORDER BY clause. The ORDER BY clause must follow the GROUP BY clause.

Syntax

```
GROUP BY group-expression[,...]
```

Parameters

group-expression ``

One or both of the following:

- An expression that is not an aggregate or a grouping function that includes constants and column references in FROM -specified tables. For example:
  column1, (column2+1), column3+column4
- A multilevel expression, one of the following:
  - ROLLUP
  - CUBE
  - GROUPING SETS

Defining the groupings

GROUPING SETS allows you to specify exactly which groupings you want in the results. You can also concatenate the groupings as follows:

The following example clauses result in the groupings shown.

This clause...	Defines groupings...
...GROUP BY GROUPING SETS(A,B,C,D)...	(A), (B), (C), (D)
...GROUP BY GROUPING SETS((A),(B),(C),(D))...	(A), (B), (C), (D)

...GROUP BY GROUPING SETS((A,B,C,D))...	(A, B, C, D)
...GROUP BY GROUPING SETS(A,B),GROUPING SETS(C,D)...	(A, C), (B, C), (A, D), (B, C)
...GROUP BY GROUPING SETS((A,B)),GROUPING SETS(C,D)...	(A, B, C), (A, B, D)
...GROUP BY GROUPING SETS(A,B),GROUPING SETS(ROLLUP(C,D))...	(A,B), (A,B,C), (A,B,C,D)
...GROUP BY A,B,C,GROUPING SETS(ROLLUP(C, D))...	(A, B, C, D), (A, B, C), (A, B, C) The clause contains two groups (A, B, C). In the HAVING clause, use the GROUP_ID function as a predicate, to eliminate the second grouping.

Example: selecting groupings

This example shows how to select only those groupings you want. Suppose you want to aggregate on columns only, and you do not need the grand total. The first query omits the total. In the second query, you add () to the GROUPING SETS list to get the total. Use the ORDER BY clause to sort the results by grouping:

```
=> SELECT Category, Year, SUM(Amount) FROM expenses
   GROUP BY GROUPING SETS((Category, Year), (Year))
   ORDER BY 1, 2, GROUPING_ID();
  Category  | Year |  SUM
------------+------+--------
 Books      | 2005 |  39.98
 Books      | 2007 |  29.99
 Books      | 2008 |  29.99
 Electrical | 2005 | 109.99
 Electrical | 2006 | 109.99
 Electrical | 2007 | 229.98
            | 2005 | 149.97
            | 2006 | 109.99
            | 2007 | 259.97
            | 2008 |  29.99
=> SELECT Category, Year, SUM(Amount) FROM expenses
   GROUP BY GROUPING SETS((Category, Year), (Year), ())
   ORDER BY 1, 2, GROUPING_ID();
  Category  | Year |  SUM
------------+------+--------
 Books      | 2005 |  39.98
 Books      | 2007 |  29.99
 Books      | 2008 |  29.99
 Electrical | 2005 | 109.99
 Electrical | 2006 | 109.99
 Electrical | 2007 | 229.98
            | 2005 | 149.97
            | 2006 | 109.99
            | 2007 | 259.97
            | 2008 |  29.99
            |      | 549.92
```

See also

- Data aggregation
- CUBE aggregate
- GROUPING
- GROUP_ID
- GROUPING_ID

- GROUP BY clause
- ROLLUP aggregate

ROLLUP aggregate

Automatically performs subtotal aggregations as an extension to the GROUP BY clause. ROLLUP performs these aggregations across multiple dimensions, at different levels, within a single SQL query.

You can use the ROLLUP clause with three grouping functions:

- GROUPING
- GROUP_ID
- GROUPING_ID

Syntax

ROLLUP *grouping-expression* [,...]

Parameters

group-expression

One or both of the following:

- An expression that is not an aggregate or a grouping function that includes constants and column references in FROM -specified tables. For example:
  column1, (column2+1), column3+column4
- A multilevel expression, one of the following:
  - ROLLUP
  - CUBE
  - GROUPING SETS

Restrictions

GROUP BY ROLLUP does not sort results. To sort data, an ORDER BY clause must follow the GROUP BY clause.

Levels of aggregation

If *n* is the number of grouping columns, ROLLUP creates *n* +1 levels of subtotals and grand total. Because ROLLUP removes the right-most column at each step, specify column order carefully.

Suppose that ROLLUP(A, B, C) creates four groups:

- (A, B, C)
- (A, B)
- (A)
- ()

Because ROLLUP removes the right-most column at each step, there are no groups for (A, C) and (B, C) .

If you enclose two or more columns in parentheses, GROUP BY treats them as a single entity. For example:

- ROLLUP(A, B, C) creates four groups:

```
(A, B, C)
(A, B)
(A)
()
```

- ROLLUP((A, B), C) treats (A, B) as a single entity and creates three groups:

```
(A, B, C)
(A, B)
()
```

Example: aggregating the full data set

The following example shows how to use the GROUP BY clause to determine family expenses for electricity and books over several years. The SUM aggregate function computes the total amount of money spent in each category per year.

Suppose you have a table that contains information about family expenses for books and electricity:

```
=> SELECT * FROM expenses ORDER BY Category, Year;
 Year |  Category   | Amount
------+-------------+--------
2005  | Books       |  39.98
2007  | Books       |  29.99
2008  | Books       |  29.99
2005  | Electricity | 109.99
2006  | Electricity | 109.99
2007  | Electricity | 229.98
```

For the expenses table, ROLLUP computes the subtotals in each category between 2005–2007:

- Books: $99.96
- Electricity: $449.96
- Grand total: $549.92.

Use the ORDER BY clause to sort the results:

```
=> SELECT Category, Year, SUM(Amount) FROM expenses
   GROUP BY ROLLUP(Category, Year) ORDER BY 1,2, GROUPING_ID();
 Category    | Year |  SUM
-------------+------+--------
 Books       | 2005 |  39.98
 Books       | 2007 |  29.99
 Books       | 2008 |  29.99
 Books       |      |  99.96
 Electricity | 2005 | 109.99
 Electricity | 2006 | 109.99
 Electricity | 2007 | 229.98
 Electricity |      | 449.96
             |      | 549.92
```

Example: using ROLLUP with the HAVING clause

This example shows how to use the HAVING clause with ROLLUP to restrict the GROUP BY results. The following query produces only those ROLLUP categories where year is subtotaled, based on the expression in the GROUPING function:

```
=> SELECT Category, Year, SUM(Amount) FROM expenses
   GROUP BY ROLLUP(Category,Year) HAVING GROUPING(Year)=1
   ORDER BY 1, 2, GROUPING_ID();
 Category    | Year |  SUM
-------------+------+--------
 Books       |      |  99.96
 Electricity |      | 449.96
             |      | 549.92
```

The next example rolls up on (Category , Year), but not on the full results. The GROUPING_ID function specifies to aggregate less than three levels:

```
=> SELECT Category, Year, SUM(Amount) FROM expenses
   GROUP BY ROLLUP(Category,Year) HAVING GROUPING_ID(Category,Year)<3
   ORDER BY 1, 2, GROUPING_ID();
 Category    | Year |  SUM
-------------+------+--------
 Books       | 2005 |  39.98
 Books       | 2007 |  29.99
 Books       | 2008 |  29.99
 Books       |      |  99.96
 Electricity | 2005 | 109.99
 Electricity | 2006 | 109.99
 Electricity | 2007 | 229.98
 Electricity |      | 449.96
```

See also

- Data aggregation

- CUBE aggregate
- GROUPING
- GROUP_ID
- GROUPING_ID
- GROUP BY clause
- GROUPING SETS aggregate

HAVING clause

Filters the results of a GROUP BY clause . Semantically, the HAVING clause occurs after the GROUP BY operation. It was added to the SQL standard because a WHERE clause cannot specify aggregate functions .

Syntax

```
HAVING condition[,...]
```

Parameters

condition

Unambiguously references a grouping column, unless the reference appears in an aggregate function.

Examples

The following example returns the employees with salaries greater than $800,000:

```
=> SELECT employee_last_name, MAX(annual_salary) as highest_salary FROM employee_dimension
   GROUP BY employee_last_name HAVING MAX(annual_salary) > 800000 ORDER BY highest_salary DESC;
 employee_last_name | highest_salary
--------------------+----------------
 Sanchez            |         992363
 Vogel              |         983634
 Vu                 |         977716
 Lewis              |         957949
 Taylor             |         953373
 King               |         937765
 Gauthier           |         927335
 Garnett            |         903104
 Bauer              |         901181
 Jones              |         885395
 Rodriguez          |         861647
 Young              |         846657
 Greenwood          |         837543
 Overstreet         |         831317
 Garcia             |         811231
(15 rows)
```

INTERSECT clause

Calculates the intersection of the results of two or more SELECT queries. INTERSECT returns distinct values by both the query on the left and right sides of the INTERSECT operand.

Syntax

```
select-stmt
  INTERSECT query[...]
  [ order-by-clause  [ offset-clause ]]
  [ limit-clause ]
```

Notes

- Use the INTERSECT clause to return all elements that are common to the results of all the SELECT queries. The INTERSECT query operates on the results of two or more SELECT queries. INTERSECT returns only the rows that are returned by all the specified queries.
- You cannot use the ALL keyword with an INTERSECT query.
- The results of each SELECT query must be union compatible; they must return the same number of columns, and the corresponding columns must have compatible data types. For example, you cannot use the INTERSECT clause on a column of type INTEGER and a column of type VARCHAR. If the SELECT queries do not meet these criteria, Vertica returns an error.

> **Note**
> The Data type coercion chart lists the data types that can be cast to other data types. If one data type can be cast to the other, those two data types are compatible.

- Order the results of an INTERSECT operation by using an ORDER BY clause. In the ORDER BY list, specify the column names from the leftmost SELECT statement or specify integers that indicate the position of the columns by which to sort.
- You can use INTERSECT in FROM, WHERE, and HAVING clauses.
- The rightmost ORDER BY, LIMIT, or OFFSET clauses in an INTERSECT query do not need to be enclosed in parentheses because the rightmost query specifies that Vertica perform the operation on the results of the INTERSECT operation. Any ORDER BY, LIMIT, or OFFSET clauses contained in SELECT queries that appear earlier in the INTERSECT query must be enclosed in parentheses.
- The order by column names is from the first select.
- Vertica supports INTERSECT noncorrelated subquery predicates. For example:

```
=> SELECT * FROM T1
  WHERE T1.x IN
    (SELECT MAX(c1) FROM T2
     INTERSECT
       SELECT MAX(cc1) FROM T3
     INTERSECT
       SELECT MAX(d1) FROM T4);
```

Examples

Consider the following three tables:

Company_A

```
id      emp_lname     dept          sales
------+------------+----------------+-------
1234  | Stephen    | auto parts     | 1000
5678  | Alice      | auto parts     | 2500
9012  | Katherine  | floral         |  500
3214  | Smithson   | sporting goods | 1500
```

Company_B

```
id      emp_lname     dept        sales
------+------------+-------------+-------
4321  | Marvin     | home goods  |   250
9012  | Katherine  | home goods  |   500
8765  | Bob        | electronics | 20000
3214  | Smithson   | home goods  |  1500
```

Company_C

```
 id  | emp_lname |      dept      | sales
------+-----------+----------------+-------
 3214 | Smithson  | sporting goods |  1500
 5432 | Madison   | sporting goods |   400
 7865 | Cleveland | outdoor        |  1500
 1234 | Stephen   | floral         |  1000
```

The following query returns the IDs and last names of employees that exist in both Company_A and Company_B:

```
=> SELECT id, emp_lname FROM Company_A
   INTERSECT
   SELECT id, emp_lname FROM Company_B;
 id   | emp_lname
------+-----------
 3214 | Smithson
 9012 | Katherine
(2 rows)
```

The following query returns the same two employees in descending order of sales:

```
=> SELECT id, emp_lname, sales FROM Company_A
   INTERSECT
   SELECT id, emp_lname, sales FROM Company_B
   ORDER BY sales DESC;
  id  | emp_lname | sales
------+-----------+-------
 3214 | Smithson  |  1500
 9012 | Katherine |   500
(2 rows)
```

The following query returns the employee who works for both companies whose sales in Company_B are greater than 1000:

```
=> SELECT id, emp_lname, sales FROM Company_A
   INTERSECT
   (SELECT id, emp_lname, sales FROM company_B WHERE sales > 1000)
   ORDER BY sales DESC;
  id  | emp_lname | sales
------+-----------+-------
 3214 | Smithson  |  1500
(1 row)
```

In the following query returns the ID and last name of the employee who works for all three companies:

```
=> SELECT id, emp_lname FROM Company_A
    INTERSECT
   SELECT id, emp_lname FROM Company_B
   INTERSECT
   SELECT id, emp_lname FROM Company_C;
  id  | emp_lname
------+-----------
 3214 | Smithson
(1 row)
```

The following query shows the results of a mismatched data types; these two queries are not union compatible:

```
=> SELECT id, emp_lname FROM Company_A
    INTERSECT
   SELECT emp_lname, id FROM Company_B;
ERROR 3429:  For 'INTERSECT', types int and varchar are inconsistent
DETAIL:  Columns: id and emp_lname
```

Using the VMart example database, the following query returns information about all Connecticut-based customers who bought items online and whose purchase amounts were between $400 and $500:

```
=> SELECT customer_key, customer_name from public.customer_dimension
     WHERE customer_key IN (SELECT customer_key
      FROM online_sales.online_sales_fact
      WHERE sales_dollar_amount > 400
      INTERSECT
      SELECT customer_key FROM online_sales.online_sales_fact
      WHERE sales_dollar_amount > 500)
    AND customer_state = 'CT' ORDER BY customer_key;
 customer_key |     customer_name
--------------+-----------------------
           39 | Sarah S. Winkler
           44 | Meghan H. Overstreet
           70 | Jack X. Cleveland
          103 | Alexandra I. Vu
          110 | Matt . Farmer
          173 | Mary R. Reyes
          188 | Steve G. Williams
          233 | Theodore V. McNulty
          250 | Marcus E. Williams
          294 | Samantha V. Young
          313 | Meghan P. Pavlov
          375 | Sally N. Vu
          384 | Emily R. Smith
          387 | Emily L. Garcia
...
```

The previous query and the next one are equivalent, and return the same results:

```
=> SELECT customer_key,customer_name FROM public.customer_dimension
     WHERE customer_key IN (SELECT customer_key
    FROM online_sales.online_sales_fact
      WHERE sales_dollar_amount > 400
      AND sales_dollar_amount < 500)
  AND customer_state = 'CT' ORDER BY customer_key;
```

See also

- SELECT
- EXCEPT clause
- UNION clause
- Subqueries

INTO TABLE clause

Creates a table from a query result set.

Syntax

Permanent table:

```
INTO [TABLE] [[database.]schema.]table
```

Temporary table:

```
INTO [scope] TEMP[ORARY] [TABLE] [[database.]schema.]table
  [ ON COMMIT { DELETE | PRESERVE } ROWS ]
```

Parameters

scope

Specifies visibility of a temporary table definition:

- GLOBAL (default): The table definition is visible to all sessions, and persists until you explicitly drop the table.
- LOCAL : The table definition is visible only to the session in which it is created, and is dropped when the session ends.

Regardless of this setting, retention of temporary table data is set by the keywords ON COMMIT DELETE ROWS and ON COMMIT PRESERVE ROWS (see below).

For more information, see Creating temporary tables .

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table

The name of the table to create.

ON COMMIT { DELETE | PRESERVE } ROWS

Specifies whether data is transaction- or session-scoped:

- DELETE (default) marks the temporary table for transaction-scoped data. Vertica removes all table data after each commit.
- PRESERVE marks the temporary table for session-scoped data, which is preserved beyond the lifetime of a single transaction. Vertica removes all table data when the session ends.

Examples

The following SELECT statement has an INTO TABLE clause that creates table newTable from customer_dimension :

```
=> SELECT * INTO TABLE newTable FROM customer_dimension;
```

The following SELECT statement creates temporary table newTempTable . By default, temporary tables are created at a global scope, so its definition is visible to other sessions and persists until it is explicitly dropped. No customer_dimension data is copied into the new table, and Vertica issues a warning accordingly:

```
=> SELECT * INTO TEMP TABLE newTempTable FROM customer_dimension;
WARNING 4102:  No rows are inserted into table "public"."newTempTable" because
  ON COMMIT DELETE ROWS is the default for create temporary table
HINT:  Use "ON COMMIT PRESERVE ROWS" to preserve the data in temporary table
CREATE TABLE
```

The following SELECT statement creates local temporary table newTempTableLocal . This table is visible only to the session in which it was created, and is automatically dropped when the session ends. The INTO TABLE clause includes ON COMMIT PRESERVE ROWS , so Vertica copies all selection data into the new table:

```
=> SELECT * INTO LOCAL TEMP TABLE newTempTableLocal ON COMMIT PRESERVE ROWS
    FROM customer_dimension;
CREATE TABLE
```

LIMIT clause

Specifies the maximum number of result set rows to return, either from the entire result set, or from windows of a partitioned result set.

Syntax

Applied to entire result set:

```
LIMIT { num-rows | ALL }
```

Applied to windows of a partitioned result set:

```
LIMIT num-rows OVER ( PARTITION BY column-expr-x, ORDER BY column-expr-y [ASC | DESC]
```

Parameters

num-rows

The maximum number of rows to return.

ALL

Returns all rows, valid only when LIMIT is applied to the entire result set.

OVER()

Specifies how to partition and sort input data with respect to the current row. The input data is the result set that the query returns after it evaluates FROM, WHERE, GROUP BY, and HAVING clauses.

For details, see Using LIMIT with Window Partitioning below.

Limiting returned rows

LIMIT specifies to return only top-k rows from the queried dataset. Row precedence is determined by the query's ORDER BY clause .

Important
The following dependencies apply:

- Always use an ORDER BY clause with LIMIT. Otherwise, the query returns an undefined subset of the result set. The ORDER BY clause must precede LIMIT.
- When a SELECT statement specifies both LIMIT and OFFSET , Vertica first processes the OFFSET, and then applies LIMIT to the remaining rows.

For example, the following query returns the first 10 rows of data in table customer_dimension , as ordered by columns store_region and number_of_employees :

```
=> SELECT store_region, store_city||', '||store_state location, store_name, number_of_employees
    FROM store.store_dimension WHERE number_of_employees <= 12 ORDER BY store_region, number_of_employees LIMIT 10;
 store_region |    location    | store_name | number_of_employees
--------------+----------------+------------+---------------------
 East         | Stamford, CT   | Store219   |                  12
 East         | New Haven, CT  | Store66    |                  12
 East         | New York, NY   | Store122   |                  12
 MidWest      | South Bend, IN | Store134   |                  10
 MidWest      | Evansville, IN | Store30    |                  11
 MidWest      | Green Bay, WI  | Store27    |                  12
 South        | Mesquite, TX   | Store124   |                  10
 South        | Cape Coral, FL | Store18    |                  11
 South        | Beaumont, TX   | Store226   |                  11
 South        | Houston, TX    | Store33    |                  11
(10 rows)
```

Using LIMIT with window partitioning

You can use LIMIT to apply window partitioning on query results, and limit the number of rows that are returned in each window:

```
SELECT ... FROM dataset LIMIT num-rows OVER ( PARTITION BY column-expr-x, ORDER BY column-expr-y [ASC | DESC] )
```

where querying *dataset* returns *num-rows* rows in each *column-expr-x* partition with the highest or lowest values of *column-expr-y* .

For example, the following statement queries table store.store_dimension and specifies window partitioning on the result set. LIMIT is set to 2, so each window partition can display no more than two rows. The OVER clause specifies to partition the result set by store_region , where each partition window displays for one region the two stores with the smallest number of employees:

```
=> SELECT store_region, store_city||', '||store_state location, store_name, number_of_employees FROM store.store_dimension
    LIMIT 2 OVER (PARTITION BY store_region ORDER BY number_of_employees ASC);
 store_region |      location       | store_name | number_of_employees
--------------+---------------------+------------+---------------------
 West         | Norwalk, CA         | Store43    |                  10
 West         | Lancaster, CA       | Store95    |                  11
 East         | Stamford, CT        | Store219   |                  12
 East         | New York, NY        | Store122   |                  12
 SouthWest    | North Las Vegas, NV | Store170   |                  10
 SouthWest    | Phoenix, AZ         | Store228   |                  11
 NorthWest    | Bellevue, WA        | Store200   |                  19
 NorthWest    | Portland, OR        | Store39    |                  22
 MidWest      | South Bend, IN      | Store134   |                  10
 MidWest      | Evansville, IN      | Store30    |                  11
 South        | Mesquite, TX        | Store124   |                  10
 South        | Beaumont, TX        | Store226   |                  11
(12 rows)
```

MATCH clause

A SQL extension that lets you screen large amounts of historical data in search of event patterns, the MATCH clause provides subclasses for analytic partitioning and ordering and matches rows from the result table based on a pattern you define.

You specify a pattern as a regular expression, which is composed of event types defined in the DEFINE subclause, where each event corresponds to a row in the input table. Then you can search for the pattern within a sequence of input events. Pattern matching returns the contiguous sequence of rows that conforms to PATTERN subclause. For example, pattern P (A B* C) consist of three event types: A, B, and C. When Vertica finds a match in the input table, the associated pattern instance must be an event of type A followed by 0 or more events of type B, and an event of type C.

Pattern matching is particularly useful for clickstream analysis where you might want to identify users' actions based on their Web browsing behavior (page clicks). For details, see Event series pattern matching .

Syntax

```
MATCH ( [ PARTITION BY table-column ] ORDER BY table-column
  DEFINE event-name AS boolean-expr [,...]
  PATTERN pattern-name AS ( regexp)
  [ rows-match-clause ] )
```

Arguments

PARTITION BY

Defines the window data scope in which the pattern, defined in the PATTERN subclause, is matched. The partition clause partitions the data by matched patterns defined in the PATTERN subclause. For each partition, data is sorted by the ORDER BY clause. If the partition clause is omitted, the entire data set is considered a single partition.

ORDER BY

Defines the window data scope in which the pattern, defined in the PATTERN subclause, is matched. For each partition, the order clause specifies how the input data is ordered for pattern matching.

> **Note**
> The ORDER BY clause is mandatory.

DEFINE

Defines the boolean expressions that make up the event types in the regular expressions. For example:

```
DEFINE
 Entry    AS RefURL  NOT ILIKE '%website2.com%' AND PageURL ILIKE
                          '%website2.com%',
 Onsite   AS PageURL ILIKE     '%website2.com%' AND Action='V',

 Purchase AS PageURL ILIKE     '%website2.com%' AND Action='P'
```

The DEFINE subclause accepts a maximum of 52 events. See Event series pattern matching for examples.

event-name

Name of the event to evaluate for each row—in the earlier example, Entry, Onsite, Purchase .

> **Note**
> Event names are case insensitive and follow the same naming conventions as those used for tables and columns.

boolean-expr

Expression that returns true or false. boolean_expr can include Logical operators and relational (comparison) operators. For example:

```
Purchase AS PageURL ILIKE '%website2.com%' AND Action = 'P'
```

PATTERN *pattern-name*

Name of the pattern defined in the PATTERN subclause; for example, P is the pattern name defined below:

```
PATTERN P AS (...)
```

A PATTERN is a *search pattern* that is comprised of a name and a regular expression.

> **Note**
> Vertica supports one pattern per query.

regexp

A regular expression comprised of event types defined in the DEFINE subclause and one or more quantifiers below. When Vertica evaluates the MATCH clause, the regular expression identifies the rows that meet the expression criteria.

- * : Match 0 or more times
- *? : Match 0 or more times, not greedily
- + : Match 1 or more times
- +? : Match 1 or more times, not greedily
- ? : Match 0 or 1 time
- ?? : Match 0 or 1 time, not greedily
- *+ : Match 0 or more times, possessive
- ++ : Match 1 or more times, possessive
- ?+ : Match 0 or 1 time, possessive
- | : Alternation. Matches expression before or after the vertical bar. Similar to a Boolean OR.

rows-match-clause

Specifies how to resolve more than one event evaluating to true for a single row, one of the following:

- ROWS MATCH ALL EVENTS : If more than one event evaluates to true for a single row, Vertica returns this error :

```
ERROR: pattern events must be mutually exclusive
HINT:  try using ROWS MATCH FIRST EVENT
```

- ROWS MATCH FIRST EVENT : If more than one event evaluates to true for a given row, Vertica uses the first event in the SQL statement for that row.

Pattern semantic evaluation

- The semantic evaluating ordering of the SQL clauses is: FROM -> WHERE -> PATTERN MATCH -> SELECT.
- Data is partitioned as specified in the PARTITION BY clause. If the partition clause is omitted, the entire data set is considered a single partition.
- For each partition, the order clause specifies how the input data is ordered for pattern matching.
- Events are evaluated for each row. A row could have 0, 1, or *N* events evaluate to true. If more than one event evaluates to true for the same row, Vertica returns a run-time error unless you specify ROWS MATCH FIRST EVENT. If you specify ROWS MATCH FIRST EVENT and more than one event evaluates to TRUE for a single row, Vertica chooses the event that was defined first in the SQL statement to be the event it uses for the row.
- Vertica performs pattern matching by finding the contiguous sequence of rows that conforms to the pattern defined in the PATTERN subclause.

For each match, Vertica outputs the rows that contribute to the match. Rows not part of the match (do not satisfy one or more predicates) are not output.

- Vertica reports only non-overlapping matches. If an overlap occurs, Vertica chooses the first match found in the input stream. After finding the match, Vertica looks for the next match, starting at the end of the previous match.
- Vertica reports the longest possible match, not a subset of a match. For example, consider pattern: A *B with input: AAAB. Because A uses the greedy regular expression quantifier (*), Vertica reports all A inputs (AAAB), not AAB, AB, or B.

Notes and restrictions

- DISTINCT and GROUP BY/HAVING clauses are not allowed in pattern match queries.
- The following expressions are not allowed in the DEFINE subclause:
  - Subqueries, such as DEFINE X AS c IN ELECT c FROM table
  - Analytic functions, such as DEFINE X AS c < LEA1) OVER (ORDER BY 1)
  - Aggregate functions, such as DEFINE X AS c < MA1)
- You cannot use the same pattern name to define a different event; for example, the following is not allowed for X:

```
DEFINE   X AS c1 <  3
 X AS c1  >= 3
```

- Used with MATCH clause, Vertica MATCH clause functions provide additional data about the patterns it finds. For example, you can use the functions to return values representing the name of the event that matched the input row, the sequential number of the match, or a partition-wide unique identifier for the instance of the pattern that matched.

Examples

For examples, see Event series pattern matching .

See also

- MATCH clause functions
- EVENT_NAME
- MATCH_ID

- PATTERN_ID

In this section

- Event series pattern matching

Event series pattern matching

The SQL MATCH clause syntax lets you screen large amounts of historical data in search of event patterns. You specify a pattern as a regular expression and can then search for the pattern within a sequence of input events. MATCH provides subclauses for analytic data partitioning and ordering, and the pattern matching occurs on a contiguous set of rows.

Pattern matching is particularly useful for clickstream analysis where you might want to identify users' actions based on their Web browsing behavior (page clicks). A typical online clickstream funnel is:

Company home page -> product home page -> search -> results -> purchase online

Using this clickstream funnel, you can search for a match on the user's sequence of web clicks and identify that user:

- Landed on the company home page
- Navigated to the product page
- Ran a search
- Clicked a link from the search results
- Made a purchase

Clickstream funnel schema

The examples in this topic use this clickstream funnel and the following clickstream_log table schema:

```
=> CREATE TABLE clickstream_log (
  uid INT,             --user ID
  sid INT,             --browsing session ID, produced by previous sessionization computation
  ts TIME,              --timestamp that occurred during the user's page visit
  refURL VARCHAR(20),  --URL of the page referencing PageURL
  pageURL VARCHAR(20), --URL of the page being visited
  action CHAR(1)       --action the user took after visiting the page ('P' = Purchase, 'V' = View)
);

INSERT INTO clickstream_log VALUES (1,100,'12:00','website1.com','website2.com/home', 'V');
INSERT INTO clickstream_log VALUES (1,100,'12:01','website2.com/home','website2.com/floby', 'V');
INSERT INTO clickstream_log VALUES (1,100,'12:02','website2.com/floby','website2.com/shamwow', 'V');
INSERT INTO clickstream_log values (1,100,'12:03','website2.com/shamwow','website2.com/buy', 'P');
INSERT INTO clickstream_log values (2,100,'12:10','website1.com','website2.com/home', 'V');
INSERT INTO clickstream_log values (2,100,'12:11','website2.com/home','website2.com/forks', 'V');
INSERT INTO clickstream_log values (2,100,'12:13','website2.com/forks','website2.com/buy', 'P');
COMMIT;
```

Here's the clickstream_log table's output:

```
=> SELECT * FROM clickstream_log;
 uid | sid |    ts    |        refURL        |       pageURL        | action
-----+-----+----------+----------------------+----------------------+--------
   1 | 100 | 12:00:00 | website1.com         | website2.com/home    | V
   1 | 100 | 12:01:00 | website2.com/home    | website2.com/floby   | V
   1 | 100 | 12:02:00 | website2.com/floby   | website2.com/shamwow | V
   1 | 100 | 12:03:00 | website2.com/shamwow | website2.com/buy     | P
   2 | 100 | 12:10:00 | website1.com         | website2.com/home    | V
   2 | 100 | 12:11:00 | website2.com/home    | website2.com/forks   | V
   2 | 100 | 12:13:00 | website2.com/forks   | website2.com/buy     | P
(7 rows)
```

Examples

This example includes the Vertica MATCH clause functions to analyze users' browsing history over website2.com. It identifies patterns where the user performed the following tasks:

- Landed on website2.com from another web site (Entry)

- Browsed to any number of other pages (Onsite)
- Made a purchase (Purchase)

In the following statement, pattern P (Entry Onsite* Purchase) consist of three event types: Entry, Onsite, and Purchase. When Vertica finds a match in the input table, the associated pattern instance must be an event of type Entry followed by 0 or more events of type Onsite, and an event of type Purchase

```
=> SELECT uid,
      sid,
      ts,
      refurl,
      pageurl,
      action,
      event_name(),
      pattern_id(),
      match_id()
FROM clickstream_log
MATCH
 (PARTITION BY uid, sid ORDER BY ts
  DEFINE
    Entry    AS RefURL  NOT ILIKE '%website2.com%' AND PageURL ILIKE '%website2.com%',
    Onsite   AS PageURL ILIKE     '%website2.com%' AND Action='V',
    Purchase AS PageURL ILIKE     '%website2.com%' AND Action = 'P'
  PATTERN
    P AS (Entry Onsite* Purchase)
  ROWS MATCH FIRST EVENT);
```

In the output below, the first four rows represent the pattern for user 1's browsing activity, while the following three rows show user 2's browsing habits.

```
 uid | sid |    ts    |        refurl        |       pageurl        | action | event_name | pattern_id | match_id
-----+-----+----------+----------------------+----------------------+--------+------------+------------+----------
   1 | 100 | 12:00:00 | website1.com         | website2.com/home    | V      | Entry      |          1 |        1
   1 | 100 | 12:01:00 | website2.com/home    | website2.com/floby   | V      | Onsite     |          1 |        2
   1 | 100 | 12:02:00 | website2.com/floby   | website2.com/shamwow | V      | Onsite     |          1 |        3
   1 | 100 | 12:03:00 | website2.com/shamwow | website2.com/buy     | P      | Purchase   |          1 |        4
   2 | 100 | 12:10:00 | website1.com         | website2.com/home    | V      | Entry      |          1 |        1
   2 | 100 | 12:11:00 | website2.com/home    | website2.com/forks   | V      | Onsite     |          1 |        2
   2 | 100 | 12:13:00 | website2.com/forks   | website2.com/buy     | P      | Purchase   |          1 |        3
(7 rows)
```

See also

- MATCH clause
- MATCH clause functions

MINUS clause

MINUS is an alias for EXCEPT .

OFFSET clause

Omits a specified number of rows from the beginning of the result set.

Syntax

```
OFFSET rows
```

Parameters

start-row

Specifies the first row to include in the result set. All preceding rows are omitted.

Dependencies

- Use an ORDER BY clause with OFFSET. Otherwise, the query returns an undefined subset of the result set.
- OFFSET must follow the ORDER BY clause in a SELECT statement or UNION clause.

- When a SELECT statement or UNION clause specifies both LIMIT and OFFSET, Vertica first processes the OFFSET statement, and then applies the LIMIT statement to the remaining rows.

Examples

The following query returns 14 rows from the customer_dimension table:

```
=> SELECT customer_name, customer_gender FROM customer_dimension
   WHERE occupation='Dancer' AND customer_city = 'San Francisco' ORDER BY customer_name;
    customer_name     | customer_gender
----------------------+-----------------
 Amy X. Lang          | Female
 Anna H. Li           | Female
 Brian O. Weaver      | Male
 Craig O. Pavlov      | Male
 Doug Z. Goldberg     | Male
 Harold S. Jones      | Male
 Jack E. Perkins      | Male
 Joseph W. Overstreet | Male
 Kevin . Campbell     | Male
 Raja Y. Wilson       | Male
 Samantha O. Brown    | Female
 Steve H. Gauthier    | Male
 William . Nielson    | Male
 William Z. Roy       | Male
(14 rows)
```

If you modify the previous query to specify an offset of 8 (OFFSET 8), Vertica skips the first eight rows of the previous result set. The query returns the following results:

```
=> SELECT customer_name, customer_gender FROM customer_dimension
   WHERE occupation='Dancer' AND customer_city = 'San Francisco' ORDER BY customer_name OFFSET 8;
   customer_name   | customer_gender
-------------------+-----------------
 Kevin . Campbell  | Male
 Raja Y. Wilson    | Male
 Samantha O. Brown | Female
 Steve H. Gauthier | Male
 William . Nielson | Male
 William Z. Roy    | Male
(6 rows)
```

ORDER BY clause

Sorts a query result set on one or more columns or column expressions. Vertica uses the current locale and collation sequence to compare and sort string values.

> **Note**
> Vertica projection data is always stored sorted by the ASCII (binary) collating sequence.

Syntax

```
ORDER BY expression [ ASC | DESC ] [,...]
```

Parameters

expression

One of the following:

- Name or ordinal number of a SELECT list item. The ordinal number refers to the position of the result column, counting from the left beginning at one. Use them to order by a column whose name is not unique. Ordinal numbers are invalid for an ORDER BY clause of an analytic function's OVER clause .
- Arbitrary expression formed from columns that do not appear in the SELECT list
- CASE expression.

> **Note**
> You cannot use DISTINCT on a collection column if it is also included in the sort order.

ASC | DESC

Specifies whether to sort values in ascending or descending order. NULL values are either first or last in the sort order, depending on data type:

- INTEGER, INT, DATE/TIME: NULL has the smallest value.
- FLOAT, BOOLEAN, CHAR, VARCHAR, ARRAY, SET: NULL has the largest value

Examples

The follow example returns all the city and deal size for customer Metamedia, sorted by deal size in descending order.

```
=> SELECT customer_city, deal_siz FROM customer_dimension WHERE customer_name = 'Metamedia'
   ORDER BY deal_size DESC;
  customer_city   | deal_size
------------------+-----------
 El Monte         |   4479561
 Athens           |   3815416
 Ventura          |   3792937
 Peoria           |   3227765
 Arvada           |   2671849
 Coral Springs    |   2643674
 Fontana          |   2374465
 Rancho Cucamonga |   2214002
 Wichita Falls    |   2117962
 Beaumont         |   1898295
 Arvada           |   1321897
 Waco             |   1026854
 Joliet           |    945404
 Hartford         |    445795
(14 rows)
```

The following example uses a transform function. It returns an error because the ORDER BY column is not in the window partition.

```
=> CREATE TABLE t(geom geometry(200), geog geography(200));
=> SELECT PolygonPoint(geom) OVER(PARTITION BY geom)
   AS SEL_0 FROM t ORDER BY geog;
ERROR 2521: Cannot specify anything other than user defined transforms and partitioning expressions in the ORDER BY list
```

The following example, using the same table, corrects this error.

```
=> SELECT PolygonPoint(geom) OVER(PARTITION BY geom)
   AS SEL_0 FROM t ORDER BY geom;
```

The following example uses an array in the ORDER BY clause.

```
=> SELECT * FROM employees
   ORDER BY grant_values;
 id | department |          grants           |  grant_values
----+------------+---------------------------+----------------
 36 | Astronomy  | ["US-7376","DARPA-1567"] | [5000,4000]
 36 | Physics    | ["US-7376","DARPA-1567"] | [10000,25000]
 33 | Physics    | ["US-7376"]              | [30000]
 42 | Physics    | ["US-7376","DARPA-1567"] | [65000,135000]
(4 rows)
```

TIMESERIES clause

Provides gap-filling and interpolation (GFI) computation, an important component of time series analytics computation. See Time series analytics for details and examples.

Syntax

```
TIMESERIES slice-time AS 'length-and-time-unit-expr' OVER (
 [ PARTITION BY column-expr[,...] ] ORDER BY time-expr ) [ ORDER BY table-column[,...] ]
```

Parameters

slice-time

A time column produced by the TIMESERIES clause, which stores the time slice start times generated from gap filling.

Note: This parameter is an alias, so you can use any name that an alias would take.

length-and-time-unit-expr

An INTERVAL DAY TO SECOND literal that specifies the length of time unit of time slice computation. For example:

`TIMESERIES slice_time AS '3 seconds' ...

OVER()

Specifies partitioning and ordering for the function. OVER() also specifies that the time series function operates on a query result set—that is, the rows that are returned after the FROM , WHERE , GROUP BY , and HAVING clauses are evaluated.

PARTITION BY (*column-expr* `[,...] )`

Partitions the data by the specified column expressions. Gap filling and interpolation is performed on each partition separately.

ORDER BY *time-expr*

Sorts the data by the TIMESTAMP expression *time-expr* , which computes the time information of the time series data.

> **Note**
> The TIMESERIES clause requires an ORDER BY operation on the timestamp column.

Notes

If the *window-partition-clause* is not specified in TIMESERIES OVER(), for each defined time slice, exactly one output record is produced; otherwise, one output record is produced per partition per time slice. Interpolation is computed there.

Given a query block that contains a TIMESERIES clause, the following are the semantic phases of execution (after evaluating the FROM and the optional WHERE clauses):

1. Compute *time-expression.*
2. Perform the same computation as the TIME_SLICE() function on each input record based on the result of *time-exp* and ' *length-and-time-unit-expr'* .
    1. Perform gap filling to generate time slices missing from the input.
    2. Name the result of this computation as *slice_time* , which represents the generated "time series" column (alias) after gap filling.
3. Partition the data by *expression* , *slice-time* . For each partition, do step 4.
4. Sort the data by *time-expr* . Interpolation is computed here.

There is semantic overlap between the TIMESERIES clause and the TIME_SLICE function with the following key differences:

- TIMESERIES only supports the interval qualifier DAY TO SECOND ; it does not allow YEAR TO MONTH .
- Unlike TIME_SLICE , the time slice length and time unit expressed in * length-and-time-unit-expr *must be constants so gaps in the time slices are well-defined.
- TIMESERIES performs gap filling; the TIME_SLICE function does not.
- TIME_SLICE can return the start or end time of a time slice, depending on the value of its fourth input parameter (*start-or-end*). TIMESERIES , on the other hand, always returns the start time of each time slice. To output the end time of each time slice, write a SELECT statement like the following:

```
=> SELECT slice_time + <slice_length>;
```

Restrictions

- When the TIMESERIES clause occurs in a SQL query block, only the following clauses can be used in the same query block:
    - SELECT
    - FROM
    - WHERE
    - ORDER BY

  GROUP BY and HAVING clauses are not allowed. If a GROUP BY operation is needed before or after gap-filling and interpolation (GFI), use a subquery and place the GROUP BY In the outer query. For example:

```
=> SELECT symbol, AVG(first_bid) as avg_bid FROM (
     SELECT symbol, slice_time, TS_FIRST_VALUE(bid1) AS first_bid
     FROM Tickstore
     WHERE symbol IN ('MSFT', 'IBM')
     TIMESERIES slice_time AS '5 seconds' OVER (PARTITION BY symbol ORDER BY ts)
     ) AS resultOfGFI
GROUP BY symbol;
```

- When the TIMESERIES clause is present in the SQL query block, the SELECT list can include only the following:
  - Time series aggregate functions such as TS_FIRST_VALUE and TS_LAST_VALUE
  - *slice_time* column
  - PARTITION BY expressions
  - TIME_SLICE function

  For example, the following two queries return a syntax error because bid1 is not a PARTITION BY or GROUP BY column:

```
=> SELECT bid, symbol, TS_FIRST_VALUE(bid) FROM Tickstore
  TIMESERIES slice_time AS '5 seconds' OVER (PARTITION BY symbol ORDER BY ts);
  ERROR:  column "Tickstore.bid" must appear in the PARTITION BY list of Timeseries clause or be used in a Timeseries Output function
=> SELECT bid, symbol, AVG(bid) FROM Tickstore
  GROUP BY symbol;
  ERROR:  column "Tickstore.bid" must appear in the GROUP BY clause or be used in an aggregate function
```

Examples

For examples, see Gap filling and interpolation (GFI) .

See also

- TIME_SLICE
- TS_FIRST_VALUE
- TS_LAST_VALUE
- Gap filling and interpolation (GFI)

UNION clause

Combines the results of multiple SELECT statements. You can include UNION in FROM , WHERE , and HAVING clauses.

Syntax

```
select { UNION [ ALL | DISTINCT ] select }[...]
  [ order-by-clause  [ offset-clause ]]
  [ limit-clause ]
```

Arguments

select

A SELECT statement that returns one or more rows, depending on whether you specify keywords DISTINCT or ALL.

The first SELECT statement can include the LABEL hint. Vertica ignores LABEL hints in subsequent SELECT statements.
Each SELECT statement can specify its own ORDER BY , LIMIT , and OFFSET clauses. A SELECT statement with one or more of these clauses must be enclosed by parentheses. See also: ORDER BY, LIMIT, and OFFSET Clauses in UNION .

DISTINCT , ALL

How to return duplicate rows:

- DISTINCT (default) returns only unique rows.
- ALL concatenates all rows, including duplicates. For best performance, use UNION ALL.

UNION ALL supports columns of complex types; UNION DISTINCT does not.

Requirements

- Each row of the UNION result set must be in the result set of at least one of its SELECT statements.
- Each SELECT statement must specify the same number of columns.
- Data types of corresponding SELECT statement columns must be compatible , otherwise Vertica returns an error.

ORDER BY, LIMIT, and OFFSET clauses in UNION

A UNION statement can specify its own ORDER BY , LIMIT , and OFFSET clauses, as in the following example:

```
=> SELECT id, emp_name FROM company_a UNION ALL SELECT id, emp_name FROM company_b ORDER BY emp_name LIMIT 2;
 id  | emp_name
------+----------
 5678 | Alice
 8765 | Bob
(2 rows)
```

Each SELECT statement in a UNION clause can specify its own ORDER BY, LIMIT, and OFFSET clauses. Vertica processes the SELECT statement clauses before it processes the UNION clauses. In the following example, Vertica processes the individual queries and then concatenates the two result sets:

```
=> (SELECT id, emp_name FROM company_a ORDER BY emp_name LIMIT 2)
   UNION ALL
   (SELECT id, emp_name FROM company_b ORDER BY emp_name LIMIT 2);
 id  | emp_name
------+-----------
 5678 | Alice
 9012 | Katherine
 8765 | Bob
 9012 | Katherine
(4 rows)
```

The following requirements and restrictions determine how Vertica processes a UNION clause that contains ORDER BY , LIMIT , and OFFSET clauses:

- A UNION's ORDER BY clause must specify columns from the first (leftmost) SELECT statement.
- ORDER BY must precede LIMIT and OFFSET.
- When a SELECT or UNION statement specifies both LIMIT and OFFSET, Vertica first processes the OFFSET statement, and then applies the LIMIT statement to the remaining rows.

UNION in non-correlated subqueries

Vertica supports UNION in noncorrelated subquery predicates . For example:

```
=> SELECT DISTINCT customer_key, customer_name FROM public.customer_dimension
   WHERE customer_key IN
     (SELECT customer_key FROM store.store_sales_fact WHERE sales_dollar_amount > 500
      UNION ALL
      SELECT customer_key FROM online_sales.online_sales_fact WHERE sales_dollar_amount > 500)
   AND customer_state = 'CT';
 customer_key |     customer_name
--------------+-------------------------
         7021 | Luigi T. Dobisz
         1971 | Betty V. Dobisz
        46284 | Ben C. Gauthier
        33885 | Tanya Y. Taylor
         5449 | Sarah O. Robinson
        29059 | Sally Z. Fortin
        11200 | Foodhope
        15582 | John J. McNulty
        24638 | Alexandra F. Jones
 ...
```

UNION ALL with complex types

You can use UNION ALL with complex types. Consider a table with the following definition:

```
=> CREATE TABLE restaurants(
    name VARCHAR, cuisine VARCHAR,
    locations ARRAY[ROW(city VARCHAR(50), state VARCHAR(2)),50],
    menu ARRAY[ROW(item VARCHAR(50), price FLOAT),100] );
```

Suppose you are in a new city looking for a place to eat. The database has information about the following restaurants:

```
=> SELECT name, cuisine FROM restaurants
   WHERE CONTAINS(locations,ROW('Pittsburgh', 'PA'));
       name        | cuisine
-------------------+----------
 Bakersfield Tacos | Mexican
 Bob's pizzeria    | Italian
 Succulent Steaks  | American
 Sushi House       | Asian
 Villa Milano      | Italian
(5 rows)
```

Suppose you are hungry for Italian food. If you cannot have Italian, you want something inexpensive. The following query uses two SELECT clauses from the same table, one finding menu items for Italian restaurants and one finding menu items under $10:

```
=> WITH menu_entries AS
    (SELECT name, cuisine,
          EXPLODE(menu USING PARAMETERS skip_partitioning=true) AS (idx, menu_entry)
     FROM restaurants WHERE CONTAINS(locations,ROW('Pittsburgh', 'PA')))
   SELECT name, cuisine, menu_entry FROM menu_entries WHERE cuisine = 'Italian'
   UNION ALL
   SELECT name, cuisine, menu_entry FROM menu_entries WHERE menu_entry.price <= 10;
       name        | cuisine |                menu_entry
-------------------+---------+-------------------------------------------
 Bob's pizzeria    | Italian | {"item":"cheese pizza","price":8.25}
 Bob's pizzeria    | Italian | {"item":"spinach pizza","price":10.5}
 Villa Milano      | Italian | {"item":"pasta carbonara","price":24.99}
 Villa Milano      | Italian | {"item":"eggplant parmesan","price":23.49}
 Villa Milano      | Italian | {"item":"herbed salmon","price":28.99}
 Bakersfield Tacos | Mexican | {"item":"veggie taco","price":9.95}
 Bob's pizzeria    | Italian | {"item":"cheese pizza","price":8.25}
(7 rows)
```

You cannot use LIMIT OVER with UNION ALL if the selected columns are of complex types. In this case, the statement returns an error like "Multi-value expressions are not supported in this context". You can still use LIMIT OVER in a single SELECT statement by using parentheses to make the scoping explicit.

Examples

The examples that follow use these two tables:

```
=> SELECT * FROM company_a;
  ID    emp_name       dept       sales
------+------------+-------------+-------
1234  | Stephen    | auto parts  | 1000
5678  | Alice      | auto parts  | 2500
9012  | Katherine  | floral      |  500

=> SELECT * FROM company_b;
  ID    emp_name       dept       sales
------+------------+-------------+-------
4321  | Marvin     | home goods  |   250
9012  | Katherine  | home goods  |   500
8765  | Bob        | electronics | 20000
```

The following query finds all employee IDs and names from the two tables. The UNION statement uses DISTINCT to combine unique IDs and last names of employees. Katherine works for both companies, so she appears only once in the result set. DISTINCT is the default and can be omitted:

```
=> SELECT id, emp_name FROM company_a
   UNION DISTINCT SELECT id, emp_name FROM company_b ORDER BY id;
 id  | emp_name
------+----------
 1234 | Stephen
 4321 | Marvin
 5678 | Alice
 8765 | Bob
 9012 | Katherine
(5 rows)
```

If the UNION statement instead uses ALL, the query returns two records for Katherine:

```
=> SELECT id, emp_name FROM company_a
   UNION ALL SELECT id, emp_name FROM company_b ORDER BY id;
 id  | emp_name
------+----------
 1234 | Stephen
 5678 | Alice
 9012 | Katherine
 4321 | Marvin
 9012 | Katherine
 8765 | Bob
(6 rows)
```

The following query returns the top two salespeople in each company. Each SELECT statement specifies its own ORDER BY and LIMIT clauses, and the UNION statement concatenates the result sets as returned by each query:

```
=> (SELECT id, emp_name, sales FROM company_a ORDER BY sales DESC LIMIT 2)
   UNION ALL
   (SELECT id, emp_name, sales FROM company_b ORDER BY sales DESC LIMIT 2);
 id  |  emp_name | sales
------+-----------+-------
 8765 | Bob       | 20000
 5678 | Alice     |  2500
 1234 | Stephen   |  1000
 9012 | Katherine |   500
(4 rows)
```

The following query returns all employees in both companies with an overall ordering. The ORDER BY clause is part of the UNION statement:

```
=> SELECT id, emp_name, sales FROM company_a
   UNION
   SELECT id, emp_name, sales FROM company_b
   ORDER BY sales;
 id  |  emp_name | sales
------+-----------+-------
 4321 | Marvin    |   250
 9012 | Katherine |   500
 1234 | Stephen   |  1000
 5678 | Alice     |  2500
 8765 | Bob       | 20000
(5 rows)
```

The following query groups total sales by department within each company. Each SELECT statement has its own GROUP BY clause. UNION combines the aggregate results from each query:

```
=> (SELECT 'Company A' as company, dept, SUM(sales) FROM company_a
    GROUP BY dept)
    UNION
   (SELECT 'Company B' as company, dept, SUM(sales) FROM company_b
    GROUP BY dept)
    ORDER BY 1;
 company   |    dept     | sum
-----------+-------------+-------
 Company A | auto parts  |  3500
 Company A | floral      |   500
 Company B | electronics | 20000
 Company B | home goods  |   750
(4 rows)
```

See also

- SELECT
- EXCEPT clause
- INTERSECT clause
- Subqueries

WHERE clause

Specifies which rows to include in a query's result set.

Syntax

```
WHERE boolean-expression [ subquery ]...
```

Arguments

boolean-expression

An expression that returns true or false. The result set only includes rows that evaluate to true. The expression can include boolean operators and the following predicate elements:

- BETWEEN
- Boolean
- IN
- LIKE
- NULL

Use parentheses to group expressions, predicates, and boolean operators. For example:

```
... WHERE NOT (A=1 AND B=2) OR C=3;
```

Examples

The following example returns the names of all customers in the Eastern region whose name starts with the string Amer :

```
=> SELECT DISTINCT customer_name
   FROM customer_dimension
   WHERE customer_region = 'East'
   AND customer_name ILIKE 'Amer%';
 customer_name
---------------
 Americare
 Americom
 Americore
 Americorp
 Ameridata
 Amerigen
 Amerihope
 Amerimedia
 Amerishop
 Ameristar
 Ameritech
(11 rows)
```

WITH clause

A WITH clause defines one or more named common table expressions (CTEs), where each CTE encapsulates a result set that can be referenced by another CTE in the same WITH clause, or by the primary query. Vertica can evaluate WITH clauses in two ways:

- Inline expansion (default): Vertica evaluates each WITH clause every time it is referenced by the primary query.
- Materialization : Vertica evaluates each WITH clause once, stores results in a temporary table, and references this table as often as the query requires.

In both cases, WITH clauses can help simplify complicated queries and avoid statement repetition.

Syntax

```
WITH [ /*+ENABLE_WITH_CLAUSE_MATERIALIZATION */ ] [ RECURSIVE ] {
  cte-identifier [ ( column-aliases ) ] AS (
  [ subordinate-WITH-clause ]
  query-expression )
} [,...]
```

Arguments

/*+ENABLE_WITH_CLAUSE_MATERIALIZATION*/

Enables materialization of all queries in the current WITH clause. Otherwise, materialization is set by configuration parameter WithClauseMaterialization, by default set to 0 (disabled). If WithClauseMaterialization is disabled, materialization is automatically cleared when the primary query of the WITH clause returns. For details, see Materialization of WITH clause .

RECURSIVE

Specifies to iterate over the WITH clause's own result set, through repeated execution of an embedded UNION or UNION ALL statement. For details, see WITH clause recursion .

cte-identifier

Identifies a common table expression (CTE) within a WITH clause. This identifier is available to CTEs of the same WITH clause, and of parent and child WITH clauses (if any). CTE identifiers of the outermost (primary) WITH clause are also available to the primary query.

All CTE identifiers of the same WITH clause must be unique. For example, the following WITH clause defines two CTEs, so they require unique identifiers: regional_sales and top_regions :

```
WITH
-- query sale amounts for each region
  regional_sales AS (SELECT ... ),
  top_regions AS ( SELECT ... )
  )
```

column-aliases

A comma-delimited list of result set column aliases. The list of aliases must map to all column expressions in the CTE query. If omitted, result set columns can only be referenced by the names used in the query.

In the following example, the revenue CTE specifies two column aliases: vkey and total_revenue . These map to column vendor_key and aggregate expression SUM(total_order_cost) , respectively. The primary query references these aliases:

```
WITH revenue ( vkey, total_revenue ) AS (
  SELECT vendor_key, SUM(total_order_cost)
  FROM store.store_orders_fact
  GROUP BY vendor_key ORDER BY vendor_key)

SELECT v.vendor_name, v.vendor_address, v.vendor_city, r.total_revenue
FROM vendor_dimension v JOIN revenue r ON v.vendor_key = r.vkey
WHERE r.total_revenue = (SELECT MAX(total_revenue) FROM revenue )
ORDER BY vendor_name;
```

subordinate-WITH-clause

A WITH clause that is nested within the current one. CTEs of this WITH clause can only reference CTEs of the same clause, and of parent and child WITH clauses.

Important

The primary query can only reference CTEs in the outermost WITH clause. It cannot reference the CTEs of any nested WITH clause.

query-expression

The query of a given CTE.

Restrictions

WITH clauses only support SELECT and INSERT statements. They do not support UPDATE or DELETE statements.

Examples

> **Note**
> For examples that show usage of recursive WITH clauses, see WITH clause recursion .

Single WITH clause with single CTE

The following SQL defines a WITH clause with one CTE, revenue , which aggregates data in table store.store_orders_fact . The primary query references the WITH clause result set twice: in its JOIN clause and predicate:

```
-- define WITH clause
WITH revenue ( vkey, total_revenue ) AS (
    SELECT vendor_key, SUM(total_order_cost)
    FROM store.store_orders_fact
    GROUP BY vendor_key ORDER BY 1)
-- End WITH clause

-- primary query
SELECT v.vendor_name, v.vendor_address, v.vendor_city, r.total_revenue
FROM vendor_dimension v JOIN revenue r ON v.vendor_key = r.vkey
WHERE r.total_revenue = (SELECT MAX(total_revenue) FROM revenue )
ORDER BY vendor_name;
  vendor_name    | vendor_address | vendor_city | total_revenue
-----------------+----------------+-------------+---------------
 Frozen Suppliers | 471 Mission St | Peoria      |     49877044
(1 row)
```

Single WITH clause and multiple CTEs

In the following example, the WITH clause contains two CTEs:

- regional_sales totals sales for each region
- top_regions uses the result set from regional_sales to identify the three regions with the highest sales:

The primary query aggregates sales by region and departments in the top_regions result set:

```
WITH
-- query sale amounts for each region
  regional_sales (region, total_sales) AS (
      SELECT sd.store_region, SUM(of.total_order_cost) AS total_sales
      FROM store.store_dimension sd JOIN store.store_orders_fact of ON sd.store_key = of.store_key
      GROUP BY store_region ),
-- query previous result set
  top_regions AS (
      SELECT region, total_sales
      FROM regional_sales ORDER BY total_sales DESC LIMIT 3
    )

-- primary query
-- aggregate sales in top_regions result set
SELECT sd.store_region AS region, pd.department_description AS department, SUM(of.total_order_cost) AS product_sales
FROM store.store_orders_fact of
JOIN store.store_dimension sd ON sd.store_key = of.store_key
JOIN public.product_dimension pd ON of.product_key = pd.product_key
WHERE sd.store_region IN (SELECT region FROM top_regions)
```

```
WHERE sales_region IN (SELECT region FROM top_regions)
GROUP BY ROLLUP (region, department) ORDER BY region, product_sales DESC, GROUPING_ID();

 region  |          department          | product_sales
---------+------------------------------+---------------
 East    |                              |    1716917786
 East    | Meat                         |     189837962
 East    | Produce                      |     170607880
 East    | Photography                  |     162271618
 East    | Frozen Goods                 |     141077867
 East    | Gifts                        |     137604397
 East    | Bakery                       |     136497842
 East    | Liquor                       |     130410463
 East    | Canned Goods                 |     128683257
 East    | Cleaning supplies            |     118996326
 East    | Dairy                        |     118866901
 East    | Seafood                      |     109986665
 East    | Medical                      |     100404891
 East    | Pharmacy                     |      71671717
 MidWest |                              |    1287550770
 MidWest | Meat                         |     141446607
 MidWest | Produce                      |     125156100
 MidWest | Photography                  |     122666753
 MidWest | Frozen Goods                 |     105893534
 MidWest | Gifts                        |     103088595
 MidWest | Bakery                       |     102844467
 MidWest | Canned Goods                 |      97647270
 MidWest | Liquor                       |      97306898
 MidWest | Cleaning supplies            |      90775242
 MidWest | Dairy                        |      89065443
 MidWest | Seafood                      |      82541528
 MidWest | Medical                      |      76674814
 MidWest | Pharmacy                     |      52443519
 West    |                              |    2159765937
 West    | Meat                         |     235841506
 West    | Produce                      |     215277204
 West    | Photography                  |     205949467
 West    | Frozen Goods                 |     178311593
 West    | Bakery                       |     172824555
 West    | Gifts                        |     172134780
 West    | Liquor                       |     164798022
 West    | Canned Goods                 |     163330813
 West    | Cleaning supplies            |     148776443
 West    | Dairy                        |     145244575
 West    | Seafood                      |     139464407
 West    | Medical                      |     126184049
 West    | Pharmacy                     |      91628523
         |                              |    5164234493
(43 rows)
```

INSERT statement that includes WITH clause

The following SQL uses a WITH clause to insert data from a JOIN query into table total_store_sales :

```
CREATE TABLE total_store_sales (store_key int, region VARCHAR(20), store_sales numeric (12,2));

INSERT INTO total_store_sales
WITH store_sales AS (
        SELECT sd.store_key, sd.store_region::VARCHAR(20), SUM (of.total_order_cost)
        FROM store.store_dimension sd JOIN store.store_orders_fact of ON sd.store_key = of.store_key
        GROUP BY sd.store_region, sd.store_key ORDER BY sd.store_region, sd.store_key)
SELECT * FROM store_sales;

=> SELECT * FROM total_store_sales ORDER BY region, store_key;
 store_key |  region   | store_sales
-----------+-----------+-------------
         2 | East      | 47668303.00
         6 | East      | 48136354.00
        12 | East      | 46673113.00
        22 | East      | 48711211.00
        24 | East      | 48603836.00
        31 | East      | 46836469.00
        36 | East      | 48461449.00
        37 | East      | 48018279.00
        41 | East      | 48713084.00
        44 | East      | 47808362.00
        49 | East      | 46990023.00
        50 | East      | 47643329.00
         9 | MidWest   | 46851087.00
        15 | MidWest   | 48787354.00
        27 | MidWest   | 48497620.00
        29 | MidWest   | 47639234.00
        30 | MidWest   | 49013483.00
        38 | MidWest   | 48856012.00
        42 | MidWest   | 47297912.00
        45 | MidWest   | 48544521.00
        46 | MidWest   | 48887255.00
         4 | NorthWest | 47580215.00
        39 | NorthWest | 47136892.00
        47 | NorthWest | 48477574.00
         8 | South     | 48131455.00
        13 | South     | 47605422.00
        17 | South     | 46054367.00
...
(50 rows)
```

In this section

- Inline expansion of WITH clause
- Materialization of WITH clause
- WITH clause recursion

Inline expansion of WITH clause

By default, Vertica uses inline expansion to evaluate WITH clauses. Vertica evaluates each WITH clause every time it is referenced by the primary query. Inline expansion often works best if the query does not reference the same WITH clause multiple times, or if some local optimizations are possible after inline expansion.

Example

The following example shows a WITH clause that is a good candidate for inline expansion. The WITH clause is used in a query that obtains order information for all 2007 orders shipped between December 01-07:

```
-- Begin WITH
WITH store_orders_fact_new AS(
   SELECT * FROM store.store_orders_fact WHERE date_shipped between '2007-12-01' and '2007-12-07')
-- End WITH
-- Begin primary query
SELECT store_key, product_key, product_version, SUM(quantity_ordered*unit_price) AS total_price
FROM store_orders_fact_new
GROUP BY store_key, product_key, product_version
ORDER BY total_price DESC;

 store_key | product_key | product_version | total_price
-----------+-------------+-----------------+-------------
       232 |        1855 |               2 |       29008
       125 |        8500 |               4 |       28812
       139 |        3707 |               2 |       28812
       212 |        3203 |               1 |       28000
       236 |        8023 |               4 |       27548
       123 |       10598 |               2 |       27146
        34 |        8888 |               4 |       27100
       203 |        2243 |               1 |       27027
       117 |       13932 |               2 |       27000
        84 |         768 |               1 |       26936
       123 |        1038 |               1 |       26885
       106 |       18932 |               1 |       26864
        93 |       10395 |               3 |       26790
       162 |       13073 |               1 |       26754
        15 |        3679 |               1 |       26675
        52 |        5957 |               5 |       26656
       190 |        8114 |               3 |       26611
         5 |        7772 |               1 |       26588
       139 |        6953 |               3 |       26572
       202 |       14735 |               1 |       26404
       133 |        2740 |               1 |       26312
       198 |        8545 |               3 |       26287
       221 |        7582 |               2 |       26280
       127 |        9468 |               3 |       26224
        63 |        8115 |               4 |       25960
       171 |        2088 |               1 |       25650
       250 |       11210 |               3 |       25608
...
```

Vertica processes the query as follows:

1. Expands the WITH clause reference to store_orders_fact_new within the primary query.
2. After expanding the WITH clause, evaluates the primary query.

Materialization of WITH clause

When materialization is enabled, Vertica evaluates each WITH clause once, stores results in a temporary table, and references this table as often as the query requires. Vertica drops the temporary table after primary query execution completes.

> **Note**
> If the primary query returns with an error, temporary tables might be dropped only after the client's session ends.

Materialization can facilitate better performance when WITH clauses are complex—for example, when the WITH clauses contain JOIN and GROUP BY clauses, and are referenced multiple times in the primary query.

If materialization is enabled, WITH statements perform an auto-commit of the user transaction. This occurs even when using EXPLAIN with the WITH statement.

Enabling WITH clause materialization

WITH materialization is set by configuration parameter WithClauseMaterialization, by default set to 0 (disabled). You can enable and disable materialization by setting WithClauseMaterialization at database and session levels, with ALTER DATABASE and ALTER SESSION , respectively:

- Database:
  ```
  => ALTER DATABASE db-spec SET PARAMETER WithClauseMaterialization={ 0 | 1 };
  => ALTER DATABASE db-spec CLEAR PARAMETER WithClauseMaterialization;
  ```
- Session: Parameter setting remains in effect until you explicitly clear it, or the session ends.
  ```
  => ALTER SESSION SET PARAMETER WithClauseMaterialization={ 0 | 1 };
  => ALTER SESSION CLEAR PARAMETER WithClauseMaterialization;
  ```

You can also enable WITH materialization for individual queries with the hint ENABLE_WITH_CLAUSE_MATERIALIZATION . Materialization is automatically cleared when the query returns. For example:

```
=> WITH /*+ENABLE_WITH_CLAUSE_MATERIALIZATION */ revenue AS (
    SELECT vendor_key, SUM(total_order_cost) AS total_revenue
    FROM store.store_orders_fact
    GROUP BY vendor_key ORDER BY 1)
   ...
```

Processing WITH clauses using EE5 temp relations

By default, when WITH clause queries are reused, Vertica saves those WITH clause query outputs in EE5 temp relations. However, this option can be changed. EE5 temp relation support is set by configuration parameter EnableWITHTempRelReuseLimit, which can be set in the following ways:

- 0: Disables this feature.
- 1: Force-saves all WITH clause queries into EE5 temp relations, whether or not they are reused.
- 2 (default): Saves only reused WITH clause queries into EE5 temp relations.
- 3 or more: Saves WITH clause queries into EE5 temp relations only when they are used at least this number of times.

EnableWITHTempRelReuseLimit can be set at database and session levels, with ALTER DATABASE and ALTER SESSION , respectively. When WithClauseMaterialization is set to 1, that setting overrides any EnableWITHTempRelReuseLimit settings.

Note that for WITH queries with complex types, temp relations are disabled.

Example

The following example shows a WITH clause that is a good candidate for materialization. The query obtains data for the vendor who has the highest combined order cost for all orders:

```
-- Enable materialization
=> ALTER SESSION SET PARAMETER WithClauseMaterialization=1;

-- Define WITH clause
=> WITH revenue AS (
    SELECT vendor_key, SUM(total_order_cost) AS total_revenue
    FROM store.store_orders_fact
    GROUP BY vendor_key ORDER BY 1)
-- End WITH clause

-- Primary query
=> SELECT vendor_name, vendor_address, vendor_city, total_revenue
FROM vendor_dimension v, revenue r
WHERE v.vendor_key = r.vendor_key AND total_revenue = (SELECT MAX(total_revenue) FROM revenue )
ORDER BY vendor_name;
   vendor_name    | vendor_address | vendor_city | total_revenue
------------------+----------------+-------------+--------------
 Frozen Suppliers | 471 Mission St | Peoria      |     49877044
(1 row)
```

Vertica processes this query as follows:

1. WITH clause revenue evaluates its SELECT statement from table store.store_orders_fact .
2. Results of the revenue clause are stored in a local temporary table.

3. Whenever the revenue clause statement is referenced, the results stored in the table are used.
4. The temporary table is dropped when query execution is complete.

WITH clause recursion

A WITH clause that includes the RECURSIVE option iterates over its own output through repeated execution of a UNION or UNION ALL query. Recursive queries are useful when working with self-referential data—hierarchies such as manager-subordinate relationships, or tree-structured data such as taxonomies.

The configuration parameter WithClauseRecursionLimit —by default set to 8—sets the maximum depth of recursion. You can set this parameter at database and session scopes with ALTER DATABASE and ALTER SESSION, respectively. Recursion continues until it reaches the configured maximum depth, or until the last iteration returns with no data.

You specify a recursive WITH clause as follows:

```
WITH [ /*+ENABLE_WITH_CLAUSE_MATERIALIZATION*/ ] RECURSIVE
  cte-identifier [ ( column-aliases ) ] AS (
    non-recursive-term
    UNION [ ALL ]
    recursive-term
  )
```

Non-recursive and recursive terms are separated by UNION or UNION ALL:

- The *non-recursive-term* query sets its result set in *cte-identifier* , which is subject to recursion in *recursive-term* .
- The UNION statement's *recursive-term* recursively iterates over its own output. When recursion is complete, the results of all iterations are compiled and set in *cte-identifier* .

For example:

```
=> ALTER SESSION SET PARAMETER WithClauseRecursionLimit=4; -- maximum recursion depth = 4
=> WITH RECURSIVE nums (n) AS (
  SELECT 1 -- non-recursive (base) term
  UNION ALL
    SELECT n+1 FROM nums -- recursive term
 )
SELECT n FROM nums; -- primary query
```

This simple query executes as follows:

1. Executes the WITH RECURSIVE clause:
   - Evaluates the non-recursive term *SELECT 1* , and places the result set—1—in nums .
   - Iterates over the UNION ALL query (*SELECT n+1*) until the number of iterations is greater than the configuration parameter WithClauseRecursionLimit.
   - Combines the results of all UNION queries and sets the result set in nums , and then exits to the primary query.
2. Executes the primary query *SELECT n FROM nums* :

```
 n
---
 1
 2
 3
 4
 5
(5 rows)
```

In this case , WITH RECURSIVE clause exits after four iterations as per WithClauseRecursionLimit. If you restore WithClauseRecursionLimit to its default value of 8, then the clause exits after eight iterations:

```
=> ALTER SESSION CLEAR PARAMETER WithClauseRecursionLimit;
=> WITH RECURSIVE nums (n) AS (
   SELECT 1
   UNION ALL
     SELECT n+1 FROM nums
  )
SELECT n FROM nums;
 n
---
 1
 2
 3
 4
 5
 6
 7
 8
 9
(9 rows)
```

Important

Be careful to set WithClauseRecursionLimit only as high as needed to traverse the deepest hierarchies. Vertica sets no limit on this parameter; however, a high value can incur considerable overhead that adversely affects performance and exhausts system resources.

If a high recursion count is required, then consider enabling materialization. For details, see WITH RECURSIVE Materialization .

Restrictions

The following restrictions apply:

- The SELECT list of a non-recursive term cannot include the wildcard * (asterisk) or the function MATCH_COLUMNS .
- A recursive term can reference the target CTE only once.
- Recursive reference cannot appear within an outer join.
- Recursive reference cannot appear within a subquery.
- WITH clauses do not support UNION options ORDER BY, LIMIT, and OFFSET.

Examples

A small software company maintains the following data on employees and their managers:

```
=> SELECT * FROM personnel.employees ORDER BY emp_id;
 emp_id |   fname   |   lname   | section_id |    section_name     |  section_leader  | leader_id
--------+-----------+-----------+------------+---------------------+------------------+-----------
      0 | Stephen   | Mulligan  |          0 |                     |                  |
      1 | Michael   | North     |        201 | Development         | Zoe Black        |         3
      2 | Megan     | Berry     |        202 | QA                  | Richard Chan     |        18
      3 | Zoe       | Black     |        101 | Product Development | Renuka Patil     |        24
      4 | Tim       | James     |        203 | IT                  | Ebuka Udechukwu  |        17
      5 | Bella     | Tucker    |        201 | Development         | Zoe Black        |         3
      6 | Alexandra | Climo     |        202 | QA                  | Richard Chan     |        18
      7 | Leonard   | Gray      |        203 | IT                  | Ebuka Udechukwu  |        17
      8 | Carolyn   | Henderson |        201 | Development         | Zoe Black        |         3
      9 | Ryan      | Henderson |        201 | Development         | Zoe Black        |         3
     10 | Frank     | Tucker    |        205 | Sales               | Benjamin Glover  |        29
     11 | Nathan    | Ferguson  |        102 | Sales Marketing     | Eric Redfield    |        28
     12 | Kevin     | Rampling  |        101 | Product Development | Renuka Patil     |        24
     13 | Tuy Kim   | Duong     |        201 | Development         | Zoe Black        |         3
     14 | Dwipendra | Sing      |        204 | Tech Support        | Sarah Feldman    |        26
     15 | Dylan     | Wijman    |        206 | Documentation       | Kevin Rampling   |        12
     16 | Tamar     | Sasson    |        207 | Marketing           | Nathan Ferguson  |        11
     17 | Ebuka     | Udechukwu |        101 | Product Development | Renuka Patil     |        24
     18 | Richard   | Chan      |        101 | Product Development | Renuka Patil     |        24
     19 | Maria     | del Rio   |        201 | Development         | Zoe Black        |         3
     20 | Hua       | Song      |        204 | Tech Support        | Sarah Feldman    |        26
     21 | Carmen    | Lopez     |        204 | Tech Support        | Sarah Feldman    |        26
     22 | Edgar     | Mejia     |        206 | Documentation       | Kevin Rampling   |        12
     23 | Riad      | Salim     |        201 | Development         | Zoe Black        |         3
     24 | Renuka    | Patil     |        100 | Executive Office    | Stephen Mulligan |         0
     25 | Rina      | Dsouza    |        202 | QA                  | Richard Chan     |        18
     26 | Sarah     | Feldman   |        101 | Product Development | Renuka Patil     |        24
     27 | Max       | Mills     |        102 | Sales Marketing     | Eric Redfield    |        28
     28 | Eric      | Redfield  |        100 | Executive Office    | Stephen Mulligan |         0
     29 | Benjamin  | Glover    |        102 | Sales Marketing     | Eric Redfield    |        28
     30 | Dominic   | King      |        205 | Sales               | Benjamin Glover  |        29
     32 | Ryan      | Metcalfe  |        206 | Documentation       | Kevin Rampling   |        12
     33 | Piers     | Paige     |        201 | Development         | Zoe Black        |         3
     34 | Nicola    | Kelly     |        207 | Marketing           | Nathan Ferguson  |        11
(34 rows)
```

You can query this data for employee-manager relationships through WITH RECURSIVE. For example, the following query's WITH RECURSIVE clause gets employee-manager relationships for employee Eric Redfield, including all employees who report directly and indirectly to him:

```
WITH RECURSIVE managers (employeeID, employeeName, sectionID, section, lead, leadID)
 AS (SELECT emp_id, fname||' '||lname, section_id, section_name, section_leader, leader_id
     FROM personnel.employees WHERE fname||' '||lname = 'Eric Redfield'
 UNION
    SELECT emp_id, fname||' '||lname AS employee_name, section_id, section_name, section_leader, leader_id FROM personnel.employees e
     JOIN managers m ON m.employeeID = e.leader_id)
 SELECT employeeID, employeeName, lead AS 'Reports to', section, leadID from managers ORDER BY sectionID, employeeName;
```

The WITH RECURSIVE clause defines the CTE managers , and then executes in two phases:

1. The non-recursive term populates managers with data that it queries from personnel.employees .
2. The recursive term's UNION query iterates over its own output until, on the fourth cycle, it finds no more data. The results of all iterations are then compiled and set in managers , and the WITH CLAUSE exits to the primary query.

The primary query returns three levels of data from managers —one for each recursive iteration:

```
 employeeID |  employeeName   |    Reports to    |     section      | leadID
------------+-----------------+------------------+------------------+--------
         28 | Eric Redfield   | Stephen Mulligan | Executive Office |      0   — Level 0
         29 | Benjamin Glover | Eric Redfield    | Sales Marketing  |     28  ┐
         27 | Max Mills       | Eric Redfield    | Sales Marketing  |     28  ├ Level 1
         11 | Nathan Ferguson | Eric Redfield    | Sales Marketing  |     28  ┘
         30 | Dominic  King   | Benjamin Glover  | Sales            |     29  ┐
         10 | Frank Tucker    | Benjamin Glover  | Sales            |     29  ├ Level 2
         34 | Nicola Kelly    | Nathan Ferguson  | Marketing        |     11  ┘
         16 | Tamar Sasson    | Nathan Ferguson  | Marketing        |     11
(8 rows)
```

Similarly, the following query iterates over the same data to get all employee-manager relationships for employee Richard Chan, who is one level lower in the company chain of command:

```
WITH RECURSIVE managers (employeeID, employeeName, sectionID, section, lead, leadID)
 AS (SELECT emp_id, fname||' '||lname, section_id, section_name, section_leader, leader_id
     FROM personnel.employees WHERE fname||' '||lname = 'Richard Chan'
 UNION
    SELECT emp_id, fname||' '||lname AS employee_name, section_id, section_name, section_leader, leader_id FROM personnel.employees e
     JOIN managers m ON m.employeeID = e.leader_id)
 SELECT employeeID, employeeName, lead AS 'Reports to', section, leadID from managers ORDER BY sectionID, employeeName;
```

The WITH RECURSIVE clause executes as before, except this time it finds no more data after two iterations and exits. Accordingly, the primary query returns two levels of data from managers :

```
 employeeID |  employeeName   |  Reports to  |       section        | leadID
------------+-----------------+--------------+----------------------+--------
         18 | Richard Chan    | Renuka Patil | Product Development  |     24   — Level 0
          6 | Alexandra Climo | Richard Chan | QA                   |     18  ┐
          2 | Megan Berry     | Richard Chan | QA                   |     18  ├ Level 1
         25 | Rina Dsouza     | Richard Chan | QA                   |     18  ┘
(4 rows)
```

WITH RECURSIVE materialization

By default, materialization is disabled. In this case, Vertica rewrites the WITH RECURSIVE query into subqueries, as many as necessary for the required level of recursion.

If recursion is very deep, the high number of query rewrites is liable to incur considerable overhead that adversely affects performance and exhausts system resources. In this case, consider enabling materialization, either with the configuration parameter WithClauseMaterialization , or the hint ENABLE_WITH_CLAUSE_MATERIALIZATION . In either case, intermediate result sets from all recursion levels are written to local temporary tables. When recursion is complete, the intermediate results in all temporary tables are compiled and passed on to the primary query.

> **Note**
> If materialization is not possible, you can improve throughput on a resource pool that handles recursive queries by setting its EXECUTIONPARALLELISM parameter to 1.

SET statements

SET statements let you change how the database operates, such as changing the autocommit settings or the resource pool your session uses.

In this section

- SET DATESTYLE
- SET ESCAPE_STRING_WARNING
- SET INTERVALSTYLE
- SET LOCALE
- SET ROLE
- SET SEARCH_PATH
- SET SESSION AUTOCOMMIT
- SET SESSION CHARACTERISTICS AS TRANSACTION
- SET SESSION GRACEPERIOD
- SET SESSION IDLESESSIONTIMEOUT
- SET SESSION MEMORYCAP
- SET SESSION MULTIPLEACTIVERESULTSETS
- SET SESSION RESOURCE_POOL
- SET SESSION RUNTIMECAP
- SET SESSION TEMPSPACECAP
- SET STANDARD_CONFORMING_STRINGS
- SET TIME ZONE

SET DATESTYLE

Specifies how to format date/time output for the current session. Use SHOW DATESTYLE to verify the current output settings.

Syntax

```
SET DATESTYLE TO { arg | 'arg' }[, arg | 'arg' ]
```

Parameters

SET DATESTYLE has a single parameter, which can be set to one or two arguments that specify date ordering and style. Each argument can be specified singly or in combination with the other; if combined, they can be specified in any order.

The following table describes each style and the date ordering arguments it supports:

Date style arguments	Order arguments	Example
ISO (ISO 8601/SQL standard)	n/a	2016-03-16 00:00:00
GERMAN	n/a	16.03.2016 00:00:00
SQL	MDY	03/16/2016 00:00:00
	DMY (default)	16/03/2016 00:00:00
POSTGRES	MDY (default)	Wed Mar 16 00:00:00 2016
	DMY	Wed 16 Mar 00:00:00 2016

Vertica ignores the order argument for date styles ISO and GERMAN . If the date style is SQL or POSTGRES , the order setting determines whether dates are output in MDY or DMY order. Neither SQL nor POSTGRES support YMD order. If you specify YMD for SQL or POSTGRES , Vertica ignores it and uses their default MDY order.

Date styles and ordering can also affect how Vertica interprets input values. For more information, see Date/time literals .

Privileges

None

Input dependencies

In some cases, input format can determine output, regardless of date style and order settings:

- Vertica ISO output for DATESTYLE is ISO long form, but several input styles are accepted. If the year appears first in the input, YMD is used for input and output, regardless of the DATESTYLE value.
- INTERVAL input and output share the same format, with the following exceptions:
  - Units like CENTURY or WEEK are converted to years and days.
  - AGO is converted to the appropriate sign.

  If the date style is set to ISO, output follows this format:

  ```
  [ quantity unit [...] ] [ days ] [ hours:minutes:seconds ]
  ```

Examples

```
=> CREATE TABLE t(a DATETIME);
CREATE TABLE
=> INSERT INTO t values ('3/16/2016');
 OUTPUT
--------
      1
(1 row)

=> SHOW DATESTYLE;
   name    | setting
-----------+----------
 datestyle | ISO, MDY
(1 row)

=> SELECT * FROM t;
          a
----------------------
 2016-03-16 00:00:00
(1 row)

=> SET DATESTYLE TO German;
SET
=> SHOW DATESTYLE;
   name    |   setting
-----------+-------------
 datestyle | German, DMY
(1 row)

=> SELECT * FROM t;
          a
----------------------
 16.03.2016 00:00:00
(1 row)

=> SET DATESTYLE TO SQL;
SET
=> SHOW DATESTYLE;
   name    | setting
-----------+----------
 datestyle | SQL, DMY
(1 row)

=> SELECT * FROM t;
          a
----------------------
 16/03/2016 00:00:00
(1 row)

=> SET DATESTYLE TO Postgres, MDY;
SET
=> SHOW DATESTYLE;
   name    |    setting
-----------+---------------
 datestyle | Postgres, MDY
(1 row)

=> SELECT * FROM t;
           a
--------------------------
 Wed Mar 16 00:00:00 2016
(1 row)
```

SET ESCAPE_STRING_WARNING

Issues a warning when a backslash is used in a string literal during the current session .

Syntax

```
SET ESCAPE_STRING_WARNING TO { ON | OFF }
```

Parameters

ON

[Default] Issues a warning when a back slash is used in a string literal.

Tip: Organizations that have upgraded from earlier versions of Vertica can use this as a debugging tool for locating backslashes that used to be treated as escape characters, but are now treated as literals.

OFF

Ignores back slashes within string literals.

Privileges

None

Notes

- This statement works under vsql only.
- Turn off standard conforming strings before you turn on this parameter.

> **Tip**
> To set escape string warnings across all sessions, use the EscapeStringWarnings configuration parameter. See the Internationalization parameters .

Examples

The following example shows how to turn OFF escape string warnings for the session.

```
=> SET ESCAPE_STRING_WARNING TO OFF;
```

See also

- SET STANDARD_CONFORMING_STRINGS

SET INTERVALSTYLE

Specifies whether to include units in interval output for the current session .

Syntax

```
SET INTERVALSTYLE TO [ plain | units ]
```

Parameters

plain

(default) Sets the default interval output to omit units.

units

Enables interval output to include subtype unit identifiers . When INTERVALSTYLE is set to units, the DATESTYLE parameter controls output. If you enable units and they do not display in the output, check the DATESTYLE parameter value, which must be set to ISO or POSTGRES for interval units to display.

Privileges

None

Examples

See Setting interval unit display .

SET LOCALE

Specifies locale for the current session .

You can also set the current locale with the vsql command \locale .

Syntax

```
SET LOCALE TO ICU-locale-identifier
```

Parameters

locale-identifier

Specifies the ICU locale identifier to use, by default set to:

```
en_US@collation=binary
```

If set to an empty string, Vertica sets locale to en_US_POSIX .

The following requirements apply:

- Vertica only supports the COLLATION keyword.
- Single quotes are mandatory to specify collation.

Privileges

None

Commonly used locales

For details on identifier options, see About locale . For a complete list of locale identifiers, see the ICU Project .

de_DE
German (Germany)

en_GB
English (Great Britain)

es_ES
Spanish (Spain)

fr_FR
French (France)

pt_BR
Portuguese (Brazil)

pt_PT
Portuguese (Portugal)

ru_RU
Russian (Russia)

ja_JP
Japanese (Japan)

zh_CN
Chinese (China, simplified Han)

zh_Hant_TW
Chinese (Taiwan, traditional Han)

Examples

Set session locale to en_GB :

```
=> SET LOCALE TO en_GB;
INFO 2567:  Canonical locale: 'en_GB'
Standard collation: 'LEN'
English (United Kingdom)
SET
```

Use the short form of a locale:

```
=> SET LOCALE TO LEN;
INFO 2567:  Canonical locale: 'en'
Standard collation: 'LEN'
English
SET
```

Specify collation:

```
=> SET LOCALE TO 'tr_tr@collation=standard';
INFO 2567:  Canonical locale: 'tr_TR@collation=standard'
Standard collation: 'LTR'
Turkish (Turkey, collation=standard)  Türkçe (Türkiye, Sıralama=standard)
SET
```

See also

- Implement locales for international data sets
- About locale

SET ROLE

Enables a role for the user's current session. The user can access privileges that have been granted to the role. Enabling a role has no effect on roles that are currently enabled.

> **Tip**
> Use SHOW AVAILABLE ROLES to list granted roles.

Syntax

```
SET ROLE roles-expression
```

Parameters

roles-expression

Specifies what roles are the default roles for this user, with one of the following expressions:

- NONE (default): Disables all roles.
- *roles-list* : A comma-delimited list of roles to enable. You can only set roles that are currently granted to you.
- ALL [EXCEPT *roles-list*] : Enables all roles currently granted to this user, excluding any comma-delimited roles specified in the optional EXCEPT clause.
- DEFAULT : Enables all default roles . Default roles are, by definition, enabled automatically, but this option might be useful for re-enabling them if they are disabled with SET ROLE NONE.

Privileges

None

Examples

This example shows the following:

- SHOW AVAILABLE_ROLES; lists the roles available to the user, but not enabled.
- SET ROLE applogs; enables the applogs role for the user.
- SHOW ENABLED_ROLES; lists the applogs role as enabled (SET) for the user.
- SET ROLE appuser; enables the appuser role for the user.
- SHOW ENABLED_ROLES now lists both applogs and appuser as enabled roles for the user.
- SET ROLE NONE disables all the users' enabled roles .
- SHOW ENABLED_ROLES shows that no roles are enabled for the user.

```
=> SHOW AVAILABLE_ROLES;
     name       |          setting
----------------+----------------------------
 available roles | applogs, appadmin, appuser
(1 row)

=> SET ROLE applogs;
SET
=> SHOW ENABLED_ROLES;
    name      | setting
--------------+---------
 enabled roles | applogs
(1 row)

=> SET ROLE appuser;
SET
=> SHOW ENABLED_ROLES;
    name      |     setting
--------------+------------------
 enabled roles | applogs, appuser
(1 row)

=> SET ROLE NONE;
SET


=> SHOW ENABLED_ROLES;
    name      | setting
--------------+---------
 enabled roles |
(1 row)
```

Set User Default Roles

Though the DBADMIN user is normally responsible for setting a user's default roles, as a user you can set your own role. For example, if you run SET ROLE NONE all of your enabled roles are disabled. Then it was determined you need access to role1 as a default role. The DBADMIN uses ALTER USER to assign you a default role:

```
=> ALTER USER user1 default role role1;
```

This example sets role1 as user1's default role because the DBADMIN assigned this default role using ALTER USER.

```
user1 => SET ROLE default;
user1 => SHOW ENABLED_ROLES;
   name      | setting
-------------------------
enabled roles |   role1
(1 row)
```

Set All Roles as Default

This example makes all roles granted to user1 default roles:

```
user1 => SET ROLE all;
user1 => show enabled roles;
   name      |   setting
-----------------------------------
enabled roles | role1, role2, role3
(1 row)
```

Set All Roles as Default With EXCEPT

This example makes all the roles granted to the user default roles with the exception of role1.

```
user1 => set role all except role1;
user1 => SHOW ENABLED_ROLES
    name     | setting
-----------------------------
enabled roles | role2, role3
(1 row)
```

SET SEARCH_PATH

Specifies the order in which Vertica searches schemas when a SQL statement specifies a table name that is unqualified by a schema name. SET SEARCH_PATH overrides the current session's search path, which is initially set from the user profile. This search path remains in effect until the next SET SEARCH_PATH statement, or the session ends. For details, see Setting search paths .

To view the current search path, use SHOW SEARCH_PATH .

Syntax

```
SET SEARCH_PATH { TO | = } { schema-list | DEFAULT }
```

Parameters

schema-list

A comma-delimited list of schemas that indicates the order in which Vertica searches schemas for a table whose name is unqualified by a schema name.

If the search path includes a schema that does not exist, or for which the user lacks access privileges, Vertica silently skips over that schema.

DEFAULT

Sets the search path to the database default:

```
"$user", public, v_catalog, v_monitor, v_internal
```

Privileges

None

Examples

Show the current search path:

```
=> SHOW SEARCH_PATH;
   name     |                    setting
------------+-------------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
(1 row)
```

Reset the search path to schemas store and public :

```
=> SET SEARCH_PATH TO store, public;
=> SHOW SEARCH_PATH;
   name     |                   setting
------------+-------------------------------------------------
 search_path | store, public, v_catalog, v_monitor, v_internal
(1 row)
```

Reset the search path to the database default settings:

```
=> SET SEARCH_PATH TO DEFAULT;
SET
=> SHOW SEARCH_PATH;
   name     |                    setting
------------+-------------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
(1 row)
```

SET SESSION AUTOCOMMIT

Sets whether statements automatically commit their transactions on completion. This statement is primarily used by the client drivers to enable and disable autocommit, you should never have to directly call it.

Syntax

```
SET SESSION AUTOCOMMIT TO { ON | OFF }
```

Parameters

ON

Enable autocommit. Statements automatically commit their transactions when they complete. This is the default setting for connections made using the Vertica client libraries.

OFF

Disable autocommit. Transactions are not automatically committed. This is the default for interactive sessions (connections made through vsql).

Privileges

None

Examples

This examples show how to set AUTOCOMMIT to 'on' and then to 'off'.

```
=> SET SESSION AUTOCOMMIT TO on;
SET
=> SET SESSION AUTOCOMMIT TO off;
SET
```

See also

- Client libraries

SET SESSION CHARACTERISTICS AS TRANSACTION

Sets the isolation level and access mode of all transactions that start after this statement is issued.

A transaction retains its isolation level until it completes, even if the session's isolation level changes during the transaction. Vertica internal processes (such as the Tuple Mover and refresh operations) and DDL operations always run at the SERIALIZABLE isolation level to ensure consistency.

Syntax

```
SET SESSION CHARACTERISTICS AS TRANSACTION settings
```

settings

One or both of the following:

-

ISOLATION LEVEL *argument*

- READ ONLY | READ WRITE

ISOLATION LEVEL arguments

The ISOLATION LEVEL clause determines what data the transaction can access when other transactions run concurrently. You cannot change the isolation level after the first query (SELECT) or DML statement (INSERT, DELETE, UPDATE) if a transaction has run.

Set ISOLATION LEVEL to one of the following arguments:

SERIALIZABLE

Sets the strictest level of SQL transaction isolation. This level emulates transactions serially, rather than concurrently. It holds locks and blocks write operations until the transaction completes.

Applications that use SERIALIZABLE must be prepared to retry transactions in the event of serialization failures. This isolation level is not recommended for normal query operations.
Setting the transaction isolation level to SERIALIZABLE does not apply to temporary tables. Temporary tables are isolated by their transaction scope.

REPEATABLE READ

Automatically converted to SERIALIZABLE .

READ COMMITTED

Default, allows concurrent transactions.

READ UNCOMMITTED

Automatically converted to READ COMMITTED .

READ WRITE/READ ONLY

You can set the transaction access mode with one of the following:

READ WRITE

Default, allows read/write access to SQL statements.

READ ONLY

Disallows SQL statements that require write access:

- INSERT, UPDATE, DELETE, and COPY operations on any non-temporary table.
- CREATE, ALTER, and DROP
- GRANT, REVOKE
- EXPLAIN if the SQL statement to explain requires write access.

> **Note**
> Setting the transaction session mode to read-only does not prevent all write operations.

Privileges

None

Viewing session transaction characteristics

SHOW TRANSACTION_ISOLATION and SHOW TRANSACTION_READ_ONLY show the transaction settings for the current session:

```
=> SHOW TRANSACTION_ISOLATION;
        name         |   setting
----------------------+--------------
 transaction_isolation | SERIALIZABLE
(1 row)

=> SHOW TRANSACTION_READ_ONLY;
        name         | setting
----------------------+---------
 transaction_read_only | true
(1 row)
```

SET SESSION GRACEPERIOD

Sets how long a session socket remains blocked while awaiting client input or output for a given query. If the socket is blocked for a continuous period that exceeds the grace period setting, the server shuts down the socket and throws a fatal error. The session is then terminated. If no grace period is set, the query can maintain its block on the socket indefinitely.

Vertica applies a session's grace period and RUNTIMECAP settings independently. If no grace period is set, a query can continue to block indefinitely on a session socket, regardless of the query's RUNTIMECAP setting.

Syntax

```
SET SESSION GRACEPERIOD duration
```

Parameters

duration

Specifies how long a query can block on any session socket, one of the following:

- ' *interval* ' : Specifies as an interval the maximum grace period for current session queries, up to 20 days.
- =DEFAULT : Sets the grace period for queries in this session to the user's GRACEPERIOD value. A new session is initially set to this value.
- NONE : Valid only for superusers, removes any grace period previously set on session queries.

Privileges

- Superusers can increase session grace period to any value, regardless of database or node settings.
- Non-superusers can only set the session grace period to a value equal to or lower than their own user setting. If no grace period is explicitly set for a user, the grace period for that user is inherited from the node or database settings.

Examples

See Handling session socket blocking in the Administrator's Guide.

SET SESSION IDLESESSIONTIMEOUT

Sets the maximum amount of time that a session can remain idle before it exits.

> **Note**
> An idle session has no queries running.

Syntax

```
SET SESSION IDLESESSIONTIMEOUT duration
```

Parameters

duration

Specifies the amount of time a session can remain idle before it exits:

- NONE (default): No idle timeout set on the session.
- ' *interval* ' : Specifies as an interval the maximum amount of time a session can remain idle.
- =DEFAULT : Sets the idle timeout period for this session to the user's IDLESESSIONTIMEOUT value.

Privileges

- Superusers can increase the time a session can remain idle to any value, regardless of database or node settings.
- Non-superusers can only set the session idle time to a value equal to or lower than their own user setting. If no session idle time is explicitly set for a user, the session idle time for that user is inherited from the node or database settings.

Examples

See Managing client connections in the Administrator's Guide.

SET SESSION MEMORYCAP

Limits how much memory can be allocated to any request in the current session . This limit only applies to the current session; it does not limit the total amount of memory used by multiple sessions.

Syntax

```
SET SESSION MEMORYCAP limit
```

Parameters

limit

One of the following:

- ' *max-expression* ': A string value that specifies the memory limit, one of the following:
  - *int* % — Expresses the maximum as a percentage of total memory available to the Resource manager , where *int* is an integer value between 0 and 100.For example:
    MEMORYCAP '40%'
  - *int* {K|M|G|T} — Expresses memory allocation in kilobytes, megabytes, gigabytes, or terabytes. For example:
    MEMORYCAP '10G'
- =DEFAULT : Sets the memory cap for queries in this session to the user's MEMORYCAP value.A new session is initially set to this value.
- NONE : Removes the memory cap for this session.

Privileges

- Superusers can increase session memory cap to any value.
- Non-superusers can only set the session memory cap to a value equal to or lower than their own user setting.

Examples

Set the session memory cap to 2 gigabytes:

```
=> SET SESSION MEMORYCAP '2G';
SET
=> SHOW MEMORYCAP;
   name    | setting
-----------+---------
 memorycap | 2097152
(1 row)
```

Revert the memory cap to the default setting as specified in the user profile:

```
=> SET MEMORYCAP=DEFAULT;
SET
=> SHOW MEMORYCAP;
   name    | setting
-----------+---------
 memorycap | 2013336
(1 row)
```

See also

Managing workloads

SET SESSION MULTIPLEACTIVERESULTSETS

Enables or disable the execution of multiple active result sets (MARS) on a single JDBC connection. Using this option requires an active JDBC connection.

Syntax

```
SET SESSION MULTIPLEACTIVERESULTSETS TO { ON | OFF }
```

Parameters

ON

Enable MultipleActiveResultSets.Allows you to execute multiple result sets on a single connection.

OFF

Disable MultipleActiveResultSets. Allows only one active result set per connection.(Default value.)

Privileges

None

Examples

This example shows how you can set MultipleActiveResultSets to on and then to off:

```
=> SET SESSION MULTIPLEACTIVERESULTSETS TO on;
SET
=> SET SESSION MULTIPLEACTIVERESULTSETS TO off;
SET
```

SET SESSION RESOURCE_POOL

Associates the user session with the specified resource pool.

Syntax

```
SET SESSION RESOURCE_POOL = { pool-name | DEFAULT }
```

Parameters

pool-name

The name of an existing resource pool to associate with the current session. Non-superusers must have USAGE privileges on the specified resource pool.

DEFAULT

Sets the session's resource pool to the resource pool assigned to this user .

Privileges

None

Examples

This example sets ceo_pool as the session resource pool:

```
=> SET SESSION RESOURCE_POOL = ceo_pool;
SET
```

See also

- ALTER RESOURCE POOL
- CREATE RESOURCE POOL
- CREATE USER
- DROP RESOURCE POOL
- GRANT (Resource pool)
- SET SESSION MEMORYCAP
- Managing workloads

SET SESSION RUNTIMECAP

Sets the maximum amount of time queries and stored procedures can run in a given session. If a query or stored procedure exceeds its session's RUNTIMECAP , Vertica terminates it and returns an error. You cannot increase the RUNTIMECAP beyond the limit that is set in your user profile .

> **Note**
> Vertica does not strictly enforce session RUNTIMECAP settings. If you time a query or stored procedure, you might discover that it runs longer than the RUNTIMECAP setting.

Syntax

```
SET SESSION RUNTIMECAP duration
```

Parameters

duration

Specifies how long a given query can run in the current session, one of the following:

- NONE (default): Removes a runtime limit for all current session queries.
- ' *interval* ' : Specifies as an interval the maximum runtime for current session queries, up to one year—for example, 1 minute or 100 seconds .
- =DEFAULT : Sets maximum runtime for queries in this session to the user's RUNTIMECAP value.

Privileges

- Superusers can increase session RUNTIMECAP to any value.
- Non-superusers can only set the session RUNTIMECAP to a value equal to or lower than their own user RUNTIMECAP .

Examples

Set the maximum query runtime for the current session to 10 minutes:

```
=> SET SESSION RUNTIMECAP '10 minutes';
```

Revert the session RUNTIMECAP to your user default setting:

```
=> SET SESSION RUNTIMECAP =DEFAULT;
SET
=> SHOW RUNTIMECAP;
   name    | setting
-----------+-----------
 runtimecap | UNLIMITED
(1 row)
```

Set the RUNTIMECAP to 1 SECOND and run an anonymous procedure with an infinite loop :

```
=> SET SESSION RUNTIMECAP '1 SECOND';
SET

=> DO $$
BEGIN
   LOOP
   END LOOP;
END;
$$;

ERROR 0:  Query exceeded maximum runtime
HINT:  Change the maximum runtime using SET SESSION RUNTIMECAP
```

See also

- Setting a runtime limit for queries
- Managing workloads

SET SESSION TEMPSPACECAP

Sets the maximum amount of temporary file storage that any request issued by the session can consume. If a query's execution plan requires more storage space than the session TEMPSPACECAP, it returns an error.

Syntax

```
SET SESSION TEMPSPACECAP limit
```

Arguments

limit

The maximum amount of temporary file storage to allocate to the current session, one of the following:

- NONE (default): Unlimited temporary storage
- = DEFAULT : Session TEMPSPACECAP is set to the user's TEMPSPACECAP value.
- String that specifies the storage limit, one of the following:
  - *int* % expresses the maximum as a percentage of total temporary storage available to the Resource Manager, where *int* is an integer value between 0 and 100. For example:
    ```
    SET SESSION TEMPSPACECAP '40%';
    ```
  - *int* {K|M|G|T} expresses storage allocation in kilobytes, megabytes, gigabytes, or terabytes. For example:
    ```
    SET SESSION TEMPSPACECAP '10G';
    ```

Privileges

Non-superusers:

- Restricted to setting only their own sessions
- Session TEMPSPACECAP cannot be greater than their own TEMPSPACECAP .

Examples

Set the session TEMPSPACECAP to 20 gigabytes:

```
=> SET SESSION TEMPSPACECAP '20G';
SET
=> SHOW TEMPSPACECAP;
    name     | setting
-------------+----------
 tempspacecap | 20971520
(1 row)
```

> **Note**
> SHOW displays the TEMPSPACECAP in kilobytes.

Set the session TEMPSPACECAP to unlimited:

```
=> SET SESSION TEMPSPACECAP NONE;
SET
=> SHOW TEMPSPACECAP;
    name     | setting
-------------+-----------
 tempspacecap | UNLIMITED
(1 row)
```

See also

- ALTER USER
- CREATE USER
- Managing workloads

SET STANDARD_CONFORMING_STRINGS

Specifies whether to treat backslashes as escape characters for the current session. By default, Vertica conforms to the SQL standard and supports SQL:2008 string literals within Unicode escapes.

Syntax

```
SET STANDARD_CONFORMING_STRINGS TO { ON | OFF }
```

Parameters

ON

(Default) Treat ordinary string literals ('...') as backslashes () literally. This means that backslashes are treated as string literals and not as escape characters.

OFF

Treat backslashes as escape characters.

Privileges

None

Requirements

- This statement works under vsql only.
- Standard-conforming strings must be ON to use Unicode-style string literals (U&'\nnnn').

> **Tip**
> You can set conforming strings across all sessions by setting the configuration parameter StandardConformingStrings with ALTER DATABASE...SET PARAMETER .

Examples

Turn off conforming strings for the session:

```
=> SET STANDARD_CONFORMING_STRINGS TO OFF;
```

Verify the current setting:

```
=> SHOW STANDARD_CONFORMING_STRINGS;
           name            | setting
---------------------------+---------
 standard_conforming_strings | off
(1 row)
```

Turn on conforming strings for the session:

```
=> SET STANDARD_CONFORMING_STRINGS TO ON;
```

See also

- SET ESCAPE_STRING_WARNING
- Standard-Conforming Strings and Escape Characters

SET TIME ZONE

Changes the TIME ZONE run-time parameter for the current session . Use SHOW TIMEZONE to show the session's current time zone.

If you set the timezone using POSIX format, the timezone abbreviation you use overrides the default timezone abbreviation. If the date style is set to POSTGRES, the timezone abbreviation you use is also used when converting a timestamp to a string.

Syntax

```
SET TIME ZONE [TO] { value | 'value' }
```

> **Note**
> Vertica treats literals TIME ZONE and TIMEZONE as synonyms.

Parameters

value

One of the following:

- A time zone literal supported by Vertica. To view the default list of valid literals, see the files in the following directory:
  ```
  /opt/vertica/share/timezonesets
  ```
- A signed integer representing an offset from UTC in hours
- A time zone literal with a signed integer offset. For example:
  ```
  => SET TIME ZONE TO 'America/New York -3'; -- equivalent to Pacific time
  ```

  > **Note**
  > Only valid timezone+offset combinations are meaningful as arguments to SET TIME ZONE. However, Vertica does not return an error for meaningless combinations—for example, America/NewYork + 150 .

- An interval value
- Constants LOCAL and DEFAULT , which respectively set the time zone to the one specified in environment variable TZ , or if TZ is undefined, to the operating system time zone.

Only valid (timezone+offset) combination are acceptable as parameter for this function.

Privileges

None

Examples

```
=> SET TIME ZONE TO DEFAULT;
=> SET TIME ZONE TO 'PST8PDT'; -- Berkeley, California
=> SET TIME ZONE TO 'Europe/Rome'; -- Italy
=> SET TIME ZONE TO '-7'; -- UDT offset equivalent to PDT
=> SET TIME ZONE TO INTERVAL '-08:00 HOURS';
```

See also

Using time zones with Vertica

In this section

- Time zone names for setting TIME ZONE

Time zone names for setting TIME ZONE

The time zone names listed below are valid settings for the SQL time zone (the TIME ZONE run-time parameter).

> **Note**
> Time zone and daylight-saving rules are controlled by individual governments and are subject to change. For the latest information, see Sources for Time Zone and Daylight Saving Time Data .

These names are not the same as the names shown in /opt/vertica/share/timezonesets , which are recognized by Vertica in date/time input values. The TIME ZONE names listed below imply a local Daylight Saving Time rule, where date/time input names represent a fixed offset from UTC.

In many cases, the same zone has several names. These are grouped together. The list is sorted primarily by commonly used zone names.

In addition to the names in the list, Vertica accepts time zone names as one of the following:

- *STDoffset*
- *STDoffsetDST*

where *STD* is a zone abbreviation, *offset* is a numeric offset in hours west from UTC, and *DST* is an optional Daylight Saving Time zone abbreviation, assumed to stand for one hour ahead of the given offset.

For example, if EST5EDT were not already a recognized zone name, Vertica accepts it as functionally equivalent to USA East Coast time. When a Daylight Saving Time zone name is present, Vertica assumes it uses USA time zone rules, so this feature is of limited use outside North America.

> **Caution**
> Be aware that this provision can lead to silently accepting invalid input, as there is no check on the reasonableness of the zone abbreviations. For example, SET TIME ZONE TO FOOBANKO works, leaving the system effectively using a rather peculiar abbreviation for GMT.

Time zones

- Africa:
  - Africa/Abidjan
  - Africa/Accra
  - Africa/Addis_Ababa
  - Africa/Algiers
  - Africa/Asmera
  - Africa/Bamako
  - Africa/Bangui
  - Africa/Banjul
  - Africa/Bissau
  - Africa/Blantyre
  - Africa/Brazzaville
  - Africa/Bujumbura
  - Africa/Cairo Egypt
  - Africa/Casablanca
  - Africa/Ceuta
  - Africa/Conakry
  - Africa/Dakar
  - Africa/Dar_es_Salaam
  - Africa/Djibouti
  - Africa/Douala
  - Africa/El_Aaiun
  - Africa/Freetown
  - Africa/Gaborone
  - Africa/Harare
  - Africa/Johannesburg
  - Africa/Kampala
  - Africa/Khartoum
  - Africa/Kigali
  - Africa/Kinshasa
  - Africa/Lagos
  - Africa/Libreville
  - Africa/Lome
  - Africa/Luanda
  - Africa/Lubumbashi
  - Africa/Lusaka
  - Africa/Malabo
  - Africa/Maputo
  - Africa/Maseru

- Africa/Mbabane
- Africa/Mogadishu
- Africa/Monrovia
- Africa/Nairobi
- Africa/Ndjamena
- Africa/Niamey
- Africa/Nouakchott
- Africa/Ouagadougou
- Africa/Porto-Novo
- Africa/Sao_Tome
- Africa/Timbuktu
- Africa/Tripoli Libya
- Africa/Tunis
- Africa/Windhoek

- America
  - America/Adak America/Atka US/Aleutian
  - America/Anchorage SystemV/YST9YDT US/Alaska
  - America/Anguilla
  - America/Antigua
  - America/Araguaina
  - America/Aruba
  - America/Asuncion
  - America/Bahia
  - America/Barbados
  - America/Belem
  - America/Belize
  - America/Boa_Vista
  - America/Bogota
  - America/Boise
  - America/Buenos_Aires
  - America/Cambridge_Bay
  - America/Campo_Grande
  - America/Cancun
  - America/Caracas
  - America/Catamarca
  - America/Cayenne
  - America/Cayman
  - America/Chicago CST6CDT SystemV/CST6CDT US/Central
  - America/Chihuahua
  - America/Cordoba America/Rosario
  - America/Costa_Rica
  - America/Cuiaba
  - America/Curacao
  - America/Danmarkshavn
  - America/Dawson
  - America/Dawson_Creek
  - America/Denver MST7MDT SystemV/MST7MDT US/Mountain America/Shiprock Navajo
  - America/Detroit US/Michigan
  - America/Dominica
  - America/Edmonton Canada/Mountain
  - America/Eirunepe
  - America/El_Salvador
  - America/Ensenada America/Tijuana Mexico/BajaNorte
  - America/Fortaleza
  - America/Glace_Bay
  - America/Godthab
  - America/Goose_Bay
  - America/Grand_Turk
  - America/Grenada
  - America/Guadeloupe
  - America/Guatemala

- America/Guayaquil
- America/Guyana
- America/Halifax Canada/Atlantic SystemV/AST4ADT
- America/Havana Cuba
- America/Hermosillo
- America/Indiana/Indianapolis
- America/Indianapolis
- America/Fort_Wayne EST SystemV/EST5 US/East-Indiana
- America/Indiana/Knox America/Knox_IN US/Indiana-Starke
- America/Indiana/Marengo
- America/Indiana/Vevay
- America/Inuvik
- America/Iqaluit
- America/Jamaica Jamaica
- America/Jujuy
- America/Juneau
- America/Kentucky/Louisville America/Louisville
- America/Kentucky/Monticello
- America/La_Paz
- America/Lima
- America/Los_Angeles PST8PDT SystemV/PST8PDT US/Pacific US/Pacific- New
- America/Maceio
- America/Managua
- America/Manaus Brazil/West
- America/Martinique
- America/Mazatlan Mexico/BajaSur
- America/Mendoza
- America/Menominee
- America/Merida
- America/Mexico_City Mexico/General
- America/Miquelon
- America/Monterrey
- America/Montevideo
- America/Montreal
- America/Montserrat
- America/Nassau
- America/New_York EST5EDT SystemV/EST5EDT US/Eastern
- America/Nipigon
- America/Nome
- America/Noronha Brazil/DeNoronha
- America/North_Dakota/Center
- America/Panama
- America/Pangnirtung
- America/Paramaribo
- America/Phoenix MST SystemV/MST7 US/Arizona
- America/Port-au-Prince
- America/Port_of_Spain
- America/Porto_Acre America/Rio_Branco Brazil/Acre
- America/Porto_Velho
- America/Puerto_Rico SystemV/AST4
- America/Rainy_River
- America/Rankin_Inlet
- America/Recife
- America/Regina Canada/East-Saskatchewan Canada/Saskatchewan SystemV/CST6
- America/Santiago Chile/Continental
- America/Santo_Domingo
- America/Sao_Paulo Brazil/East
- America/Scoresbysund
- America/St_Johns Canada/Newfoundland
- America/St_Kitts
- America/St_Lucia

- America/St_Thomas America/Virgin
- America/St_Vincent
- America/Swift_Current
- America/Tegucigalpa
- America/Thule
- America/Thunder_Bay
- America/Toronto Canada/Eastern
- America/Tortola
- America/Vancouver Canada/Pacific
- America/Whitehorse Canada/Yukon
- America/Winnipeg Canada/Central
- America/Yakutat
- America/Yellowknife

- Antarctica
  - Antarctica/Casey
  - Antarctica/Davis
  - Antarctica/DumontDUrville
  - Antarctica/Mawson
  - Antarctica/McMurdo
  - Antarctica/South_Pole
  - Antarctica/Palmer
  - Antarctica/Rothera
  - Antarctica/Syowa
  - Antarctica/Vostok
- Asia
  - Asia/Aden
  - Asia/Almaty
  - Asia/Amman
  - Asia/Anadyr
  - Asia/Aqtau
  - Asia/Aqtobe
  - Asia/Ashgabat Asia/Ashkhabad
  - Asia/Baghdad
  - Asia/Bahrain
  - Asia/Baku
  - Asia/Bangkok
  - Asia/Beirut
  - Asia/Bishkek
  - Asia/Brunei
  - Asia/Calcutta
  - Asia/Choibalsan
  - Asia/Chongqing Asia/Chungking
  - Asia/Colombo
  - Asia/Dacca Asia/Dhaka
  - Asia/Damascus
  - Asia/Dili
  - Asia/Dubai
  - Asia/Dushanbe
  - Asia/Gaza
  - Asia/Harbin
  - Asia/Hong_Kong Hongkong
  - Asia/Hovd
  - Asia/Irkutsk
  - Asia/Jakarta
  - Asia/Jayapura
  - Asia/Jerusalem Asia/Tel_Aviv Israel
  - Asia/Kabul
  - Asia/Kamchatka
  - Asia/Karachi
  - Asia/Kashgar
  - Asia/Katmandu

- Asia/Krasnoyarsk
- Asia/Kuala_Lumpur
- Asia/Kuching
- Asia/Kuwait
- Asia/Macao Asia/Macau
- Asia/Magadan
- Asia/Makassar Asia/Ujung_Pandang
- Asia/Manila
- Asia/Muscat
- Asia/Nicosia Europe/Nicosia
- Asia/Novosibirsk
- Asia/Omsk
- Asia/Oral
- Asia/Phnom_Penh
- Asia/Pontianak
- Asia/Pyongyang
- Asia/Qatar
- Asia/Qyzylorda
- Asia/Rangoon
- Asia/Riyadh
- Asia/Riyadh87 Mideast/Riyadh87
- Asia/Riyadh88 Mideast/Riyadh88
- Asia/Riyadh89 Mideast/Riyadh89
- Asia/Saigon
- Asia/Sakhalin
- Asia/Samarkand
- Asia/Seoul ROK
- Asia/Shanghai PRC
- Asia/Singapore Singapore
- Asia/Taipei ROC
- Asia/Tashkent
- Asia/Tbilisi
- Asia/Tehran Iran
- Asia/Thimbu Asia/Thimphu
- Asia/Tokyo Japan
- Asia/Ulaanbaatar Asia/Ulan_Bator
- Asia/Urumqi
- Asia/Vientiane
- Asia/Vladivostok
- Asia/Yakutsk
- Asia/Yekaterinburg
- Asia/Yerevan

- Atlantic
  - Atlantic/Azores
  - Atlantic/Bermuda
  - Atlantic/Canary
  - Atlantic/Cape_Verde
  - Atlantic/Faeroe
  - Atlantic/Madeira
  - Atlantic/Reykjavik Iceland
  - Atlantic/South_Georgia
  - Atlantic/St_Helena
  - Atlantic/Stanley
- Australia
  - Australia/ACT
  - Australia/Canberra
  - Australia/NSW
  - Australia/Sydney
  - Australia/Adelaide
  - Australia/South
  - Australia/Brisbane

    - Australia/Queensland
    - Australia/Broken_Hill
    - Australia/Yancowinna
    - Australia/Darwin
    - Australia/North
    - Australia/Hobart
    - Australia/Tasmania
    - Australia/LHI
    - Australia/Lord_Howe
    - Australia/Lindeman
    - Australia/Melbourne
    - Australia/Victoria
    - Australia/Perth Australia/West
- CET
- EET
- Etc/GMT
    - GMT
    - GMT+0
    - GMT-0
    - GMT0
    - Greenwich
    - Etc/Greenwich
    - Etc/GMT+0...Etc/GMT+12
    - Etc/GMT-0...Etc/GMT-14
- Europe
    - Europe/Amsterdam
    - Europe/Andorra
    - Europe/Athens
    - Europe/Belfast
    - Europe/Belgrade
    - Europe/Ljubljana
    - Europe/Sarajevo
    - Europe/Skopje
    - Europe/Zagreb
    - Europe/Berlin
    - Europe/Brussels
    - Europe/Bucharest
    - Europe/Budapest
    - Europe/Chisinau Europe/Tiraspol
    - Europe/Copenhagen
    - Europe/Dublin Eire
    - Europe/Gibraltar
    - Europe/Helsinki
    - Europe/Istanbul Asia/Istanbul Turkey
    - Europe/Kaliningrad
    - Europe/Kiev
    - Europe/Lisbon Portugal
    - Europe/London GB GB-Eire
    - Europe/Luxembourg
    - Europe/Madrid
    - Europe/Malta
    - Europe/Minsk
    - Europe/Monaco
    - Europe/Moscow W-SU
    - Europe/Oslo
    - Arctic/Longyearbyen
    - Atlantic/Jan_Mayen
    - Europe/Paris
    - Europe/Prague Europe/Bratislava
    - Europe/Riga
    - Europe/Rome Europe/San_Marino Europe/Vatican

    - Europe/Samara
    - Europe/Simferopol
    - Europe/Sofia
    - Europe/Stockholm
    - Europe/Tallinn
    - Europe/Tirane
    - Europe/Uzhgorod
    - Europe/Vaduz
    - Europe/Vienna
    - Europe/Vilnius
    - Europe/Warsaw Poland
    - Europe/Zaporozhye
    - Europe/Zurich
- Factory
- Indian
    - Indian/Antananarivo
    - Indian/Chagos
    - Indian/Christmas
    - Indian/Cocos
    - Indian/Comoro
    - Indian/Kerguelen
    - Indian/Mahe
    - Indian/Maldives
    - Indian/Mauritius
    - Indian/Mayotte
    - Indian/Reunion
- MET
- Pacific
    - Pacific/Apia
    - Pacific/Auckland NZ
    - Pacific/Chatham NZ-CHAT
    - Pacific/Easter
    - Chile/EasterIsland
    - Pacific/Efate
    - Pacific/Enderbury
    - Pacific/Fakaofo
    - Pacific/Fiji
    - Pacific/Funafuti
    - Pacific/Galapagos
    - Pacific/Gambier SystemV/YST9
    - Pacific/Guadalcanal
    - Pacific/Guam
    - Pacific/Honolulu HST SystemV/HST10 US/Hawaii
    - Pacific/Johnston
    - Pacific/Kiritimati
    - Pacific/Kosrae
    - Pacific/Kwajalein Kwajalein
    - Pacific/Majuro
    - Pacific/Marquesas
    - Pacific/Midway
    - Pacific/Nauru
    - Pacific/Niue
    - Pacific/Norfolk
    - Pacific/Noumea
    - Pacific/Pago_Pago
    - Pacific/Samoa US/Samoa
    - Pacific/Palau
    - Pacific/Pitcairn SystemV/PST8
    - Pacific/Ponape
    - Pacific/Port_Moresby
    - Pacific/Rarotonga

    - Pacific/Saipan
    - Pacific/Tahiti
    - Pacific/Tarawa
    - Pacific/Tongatapu
    - Pacific/Truk
    - Pacific/Wake
    - Pacific/Wallis
    - Pacific/Yap
- UCT Etc
- UCT
- UTC
    - Universal Zulu
    - Etc/UTC
    - Etc/Universal
    - Etc/Zulu
- WET

SHOW

Shows run-time parameters for the current session.

Syntax

```
SHOW { parameter | ALL }
```

Parameters

ALL

Shows all run-time settings.

AUTOCOMMIT

Returns on/off to indicate whether statements automatically commit their transactions when they complete.

AVAILABLE ROLES

Lists all roles available to the user.

DATESTYLE

Shows the current style of date values. See SET DATESTYLE .

ENABLED ROLES

Shows the roles enabled for the current session. See SET ROLE .

ESCAPE_STRING_WARNING

Returns on/off to indicate whether warnings are issued when backslash escapes are found in strings. See SET ESCAPE_STRING_WARNING .

GRACEPERIOD

Shows the session GRACEPERIOD set by SET SESSION GRACEPERIOD .

IDLESESSIONTIMEOUT

Shows how long the session can remain idle before it times out.

INTERVALSTYLE

Shows whether units are output when printing intervals. See SET INTERVALSTYLE .

LOCALE

Shows the current locale. See SET LOCALE .

MEMORYCAP

Shows the maximum amount of memory that any request use. See SET MEMORYCAP .

MULTIPLEACTIVERESULTSETS

Returns on/off to indicate whether multiple active result sets on one connection are allowed. See SET SESSION MULTIPLEACTIVERESULTSETS .

RESOURCE POOL

Shows the resource pool that the session is using. See SET RESOURCE POOL .

RUNTIMECAP

Shows the maximum amount of time that queries can run in the session. See SET RUNTIMECAP .

SEARCH_PATH

Shows the order in which Vertica searches schemas. See SET SEARCH_PATH . For example:

```
=> SHOW SEARCH_PATH;
   name     |                    setting
-------------+-----------------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
(1 row)
```

STANDARD_CONFORMING_STRINGS

Shows whether backslash escapes are enabled for the session. See SET STANDARD_CONFORMING_STRINGS .

TEMPSPACECAP

Shows the maximum amount of temporary file space that queries can use in the session. See SET TEMPSPACECAP .

TIMEZONE

Shows the timezone set in the current session. See SET TIMEZONE .

TRANSACTION_ISOLATION

Shows the current transaction isolation setting, as described in SET SESSION CHARACTERISTICS AS TRANSACTION . For example:

```
=> SHOW TRANSACTION_ISOLATION;
        name          |    setting
-----------------------+----------------
 transaction_isolation | READ COMMITTED
(1 row)
```

TRANSACTION_READ_ONLY

Returns true/false to indicate the current read-only setting, as described in SET SESSION CHARACTERISTICS AS TRANSACTION . For example:

```
=> SHOW TRANSACTION_READ_ONLY;
        name          | setting
-----------------------+---------
 transaction_read_only | false
(1 row)
```

Privileges

None

Examples

Display all current runtime parameter settings:

```
=> SHOW ALL;
          name            |                        setting
------------------------------+-------------------------------------------------------------
 locale                     | en_US@collation=binary (LEN_KBINARY)
 autocommit                 | off
 standard_conforming_strings | on
 escape_string_warning      | on
 multipleactiveresultsets   | off
 datestyle                  | ISO, MDY
 intervalstyle              | plain
 timezone                   | America/New_York
 search_path                | "$user", public, v_catalog, v_monitor, v_internal, v_func
 transaction_isolation      | READ COMMITTED
 transaction_read_only      | false
 resource_pool              | general
 memorycap                  | UNLIMITED
 tempspacecap               | UNLIMITED
 runtimecap                 | UNLIMITED
 idlesessiontimeout         | UNLIMITED
 graceperiod                | UNLIMITED
 enabled roles              | dbduser*, dbadmin*, pseudosuperuser*
 available roles            | dbduser*, dbadmin*, pseudosuperuser*
(19 rows)
```

SHOW CURRENT

Displays active configuration parameter values that are set at all levels. Vertica first checks values set at the session level. If a value is not set for a configuration parameter at the session level, Vertica next checks if the value is set for the node where you are logged in, and then checks the database level. If no values are set, SHOW CURRENT shows the default value for the configuration parameter. If the configuration parameter requires a restart to take effect, the active values shown might differ from the set values.

Syntax

```
SHOW CURRENT { parameter-name[,...] | ALL }
```

Parameters

parameter-name

Names of configuration parameters to show.

ALL

Shows all configuration parameters set at all levels.

Privileges

Non-superuser: SHOW CURRENT ALL returns masked parameter settings. Attempts to view specific parameter settings return an error.

Examples

Show configuration parameters and their settings at all levels.

```
=> SHOW CURRENT ALL;
level    |    name                  |setting
---------+--------------------------+---------
DEFAULT  | ActivePartitionCount     | 1
DEFAULT  | AdvanceAHMInterval       | 180
DEFAULT  | AHMBackupManagement      | 0
DATABASE | AnalyzeRowCountInterval  | 3600
SESSION  | ForceUDxFencedMode       | 1
NODE     | MaxClientSessions        | 0
...
```

SHOW DATABASE

Displays configuration parameter values that are set for the database.

Important

You can also get detailed information on configuration parameters, including their current and default values, by querying system table CONFIGURATION_PARAMETERS .

> **Note**
> If the configuration parameter is set but requires a database restart to take effect, the value shown might differ from the active value.

Syntax

```
SHOW DATABASE db-spec { parameter-name[,...] | ALL }
```

Parameters

db-spec

Specifies the current database, set to the database name or DEFAULT .

parameter-name

Names of one or more configuration parameters to show.Non-superusers can only specify parameters whose settings are not masked by SHOW DATABASE...ALL , otherwise Vertica returns an error.

If you specify a single parameter that is not set, SHOW DATABASE returns an empty row for that parameter.
To obtain the names of database-level parameters, query system table CONFIGURATION_PARAMETERS .

ALL

Shows all configuration parameters set at the database level.For non-superusers, Vertica masks settings of security parameters, which only superusers can access.

Privileges

- Superuser: Shows all database parameter settings.
- Non-superuser: Masks all security parameter settings, which only superusers can access. To determine which parameters require superuser privileges, query system table CONFIGURATION_PARAMETERS .

Examples

Show to a non-superuser all configuration parameters that are set on the database:

```
=> SHOW DATABASE DEFAULT ALL;
             name              | setting
-------------------------------+----------
 AllowNumericOverflow          | 1
 CopyFaultTolerantExpressions  | 1
 GlobalHeirUsername            | ********
 MaxClientSessions             | 50
 NumericSumExtraPrecisionDigits | 0
(6 rows)
```

Show settings for two configuration parameters:

```
=> SHOW DATABASE DEFAULT AllowNumericOverflow, NumericSumExtraPrecisionDigits;
             name              | setting
-------------------------------+---------
 AllowNumericOverflow          | 1
 NumericSumExtraPrecisionDigits | 0
(2 rows)
```

SHOW NODE

Displays configuration parameter values that are set for a node. If you specify a parameter that is not set, SHOW NODE returns an empty row for that parameter.

> **Note**
> If the configuration parameter is set but requires a database restart to take effect, the value shown might differ from the active value.

Syntax

```
SHOW NODE node-name { parameter-name [,...] | ALL }
```

Parameters

node-name

Name of the target node.

parameter-name

Names of one or more node-level configuration parameters. To obtain the names of node-level parameters, query system table CONFIGURATION_PARAMETERS .

ALL

Shows all configuration parameters set at the node level.

Privileges

None

Examples

View all configuration parameters and their settings for node v_vmart_node0001 :

```
=> SHOW NODE v_vmart_node0001 ALL;
          name           | setting
-------------------------+---------
 DefaultIdleSessionTimeout | 5 hour
 MaxClientSessions         | 20
```

SHOW SESSION

Displays configuration parameter values that are set for the current session. If you specify a parameter that is not set, SHOW SESSION returns an empty row for that parameter.

> **Note**
> If the configuration parameter is set but requires a database restart to take effect, the value shown might differ from the active value.

Syntax

```
SHOW SESSION { ALL | UDPARAMETER ALL }
```

Parameters

ALL

Shows all Vertica configuration parameters set at the session level.

UDPARAMETER ALL

Shows all parameters defined by user-defined extensions. These parameters are not shown in the CONFIGURATION_PARAMETERS table.

Privileges

None

Examples

View all Vertica configuration parameters and their settings for the current session. User-defined parameters are not included:

```
=> SHOW SESSION ALL;
name                        |                      setting
----------------------------+-------------------------------------------------
locale                      | en_US@collation=binary (LEN_KBINARY)
autocommit                  | off
standard_conforming_strings | on
escape_string_warning       | on
datestyle                   | ISO, MDY
intervalstyle               | plain
timezone                    | America/New_York
search_path                 | "$user", public, v_catalog, v_monitor, v_internal
transaction_isolation       | READ COMMITTED
transaction_read_only       | false
resource_pool               | general
memorycap                   | UNLIMITED
tempspacecap                | UNLIMITED
runtimecap                  | UNLIMITED
enabled roles               | dbduser*, dbadmin*, pseudosuperuser*
available roles             | dbduser*, dbadmin*, pseudosuperuser*
ForceUDxFencedMode          | 1
(17 rows)
```

SHOW USER

Displays configuration parameter settings for database users. To get the names of user-level parameters, query system table CONFIGURATION_PARAMETERS :

```
SELECT parameter_name, allowed_levels FROM configuration_parameters
    WHERE allowed_levels ilike '%USER%' AND parameter_name ilike '%depot%';
   parameter_name     |     allowed_levels
------------------------+------------------------
 UseDepotForWrites      | SESSION, USER, DATABASE
 DepotOperationsForQuery | SESSION, USER, DATABASE
 UseDepotForReads       | SESSION, USER, DATABASE
(3 rows)
```

Syntax

```
SHOW USER { user-name | ALL } [PARAMETER] { cfg-parameter [,...] | ALL }
```

Parameters

***user-name* | ALL**

Show parameter settings for the specified user, or for all users.

[PARAMETER] *parameter-list*

A comma-delimited list of user-level configuration parameters.

PARAMETER ALL

Show all configuration parameters that are set for the specified users.

Privileges

Non-superusers: Can view only their own configuration parameter settings.

Examples

The following example shows configuration parameter settings for two users, Yvonne and Ahmed:

```
=> SELECT user_name FROM v_catalog.users WHERE user_name != 'dbadmin';
 user_name
-----------
 Ahmed
 Yvonne
(2 rows)
```

```
=> SHOW USER Yvonne PARAMETER ALL;
  user  |       parameter        | setting
--------+-------------------------+---------
 Yvonne | DepotOperationsForQuery | Fetches
(1 row)

=> ALTER USER Yvonne SET PARAMETER UseDepotForWrites = 0;
ALTER USER
=> SHOW USER Yvonne PARAMETER ALL;
  user  |       parameter        | setting
--------+-------------------------+---------
 Yvonne | DepotOperationsForQuery | Fetches
 Yvonne | UseDepotForWrites       | 0
(2 rows)

=> ALTER USER Ahmed SET PARAMETER DepotOperationsForQuery = 'Fetches';
ALTER USER
=> SHOW USER ALL PARAMETER ALL;
  user  |       parameter        | setting
--------+-------------------------+---------
 Ahmed  | DepotOperationsForQuery | Fetches
 Yvonne | DepotOperationsForQuery | Fetches
 Yvonne | UseDepotForWrites       | 0
(3 rows)
```

See also

ALTER USER
START TRANSACTION

Starts a transaction block.

Syntax

```
START TRANSACTION [ isolation_level ]
```

where *isolation_level* is one of:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }READ { ONLY | WRITE }
```

Parameters

Isolation level, described in the following table, determines what data the transaction can access when other transactions are running concurrently. The isolation level cannot be changed after the first query (SELECT) or DML statement (INSERT , DELETE , UPDATE) has run. A transaction retains its isolation level until it completes, even if the session's isolation level changes during the transaction. Vertica internal processes (such as the Tuple Mover and refresh operations) and DDL operations always run at the SERIALIZABLE isolation level to ensure consistency.

WORK | TRANSACTION

Have no effect; they are optional keywords for readability.

ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }

- SERIALIZABLE: Sets the strictest level of SQL transaction isolation. This level emulates transactions serially, rather than concurrently. It holds locks and blocks write operations until the transaction completes. Not recommended for normal query operations.
- REPEATABLE READ: Automatically converted to SERIALIZABLE by Vertica.
- READ COMMITTED (Default): Allows concurrent transactions. Use READ COMMITTED isolation for normal query operations, but be aware that there is a subtle difference between them. See Transactions for more information.
- READ UNCOMMITTED: Automatically converted to READ COMMITTED by Vertica.

READ {WRITE | ONLY}

Determines whether the transaction is read/write or read-only. Read/write is the default.

Setting the transaction session mode to read-only disallows the following SQL commands, but does not prevent all disk write operations:

- INSERT, UPDATE, DELETE, and COPY if the table they would write to is not a temporary table
- All CREATE, ALTER, and DROP commands
- GRANT, REVOKE, and EXPLAIN if the command it would run is among those listed.

Privileges

None

Notes

BEGIN performs the same function as START TRANSACTION.

Examples

This example shows how to start a transaction.

```
= > START TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
START TRANSACTION
=> CREATE TABLE sample_table (a INT);
CREATE TABLE
=> INSERT INTO sample_table (a) VALUES (1);
OUTPUT
--------
1
(1 row)
```

See also

- Transactions
- Creating and rolling back transactions
- COMMIT
- END
- ROLLBACK

TRUNCATE TABLE

Removes all storage associated with a table, while leaving the table definition intact. TRUNCATE TABLE auto-commits the current transaction after statement execution and cannot be rolled back.

TRUNCATE TABLE also performs the following actions:

- Removes all table history preceding the current epoch. After TRUNCATE TABLE returns, AT EPOCH queries on the truncated table return nothing.
- Drops all table - and partition -level statistics.

Syntax

```
TRUNCATE TABLE [[database.]schema.]table-name
```

Parameters

[*database.*] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table-name

The name of the anchor table or temporary table to truncate. You cannot truncate an external table.

Privileges

Non-superuser:

- Table owner
- USAGE privileges on table schema

Examples

See Truncating tables .

See also

- DELETE
- DROP TABLE

UPDATE

Replaces the values of the specified columns in all rows for which a specified condition is true. All other columns and rows in the table are unchanged. If successful, UPDATE returns the number of rows updated. A count of 0 indicates no rows matched the condition.

Important

The Vertica implementation of UPDATE differs from traditional databases. It does not delete data from disk storage; it writes two rows, one with new data and one marked for deletion. Rows marked for deletion remain available for historical queries.

Syntax

```
UPDATE [[database.]schema.]table-reference [AS] alias
  SET set-expression [,...]
  [ FROM from-list ]
  [ where-clause ]
```

> **Note**
> UPDATE statements can also embed the following hints :
>
> - General: ALLNODES , EARLY_MATERIALIZATION , LABEL , SKIP_STATISTICS , VERBATIM
> - Join: SYNTACTIC_JOIN , DISTRIB , GBYTYPE , JTYPE , UTYPE
> - Projection: PROJS , SKIP_PROJS

Parameters

[*database* .] *schema*

Database and schema . The default schema is public . If you specify a database, it must be the current database.

table-reference

A table, one of the following:

- An optionally qualified table name with optional table aliases, column aliases, and outer joins.
- An outer join table.

You cannot update a projection.

alias

A temporary name used to reference the table.

SET *set-expression*

The columns to update from one or more set expressions. Each SET clause expression specifies a target column and its new value as follows:

```
column-name = { expression | DEFAULT }
```

where:

- *column-name* is any column that does not have primary key or foreign key referential integrity constraints and is not of a complex type. Native arrays are permitted.
- *expression* specifies a value to assign to the column. The expression can use the current values of this and other table columns. For example:

```
=> UPDATE T1 SET C1 = C1+1
```

- DEFAULT sets *column-name* to its default value, or is ignored if no default value is defined for this column.

UPDATE only modifies the columns specified by the SET clause. Unspecified columns remain unchanged.

FROM *from-list*

A list of table expressions, allowing columns from other tables to appear in the WHERE condition and the UPDATE expressions. This is similar to the list of tables that can be specified in the FROM clause of a SELECT command.

The FROM clause can reference the target table as follows:

```
FROM DEFAULT [join-type] JOIN table-reference [ ON join-predicate ]
```

DEFAULT specifies the table to update. This keyword can be used only once in the FROM clause, and it cannot be used elsewhere in the UPDATE statement.

Privileges

Table owner or user with GRANT OPTION is grantor.

- UPDATE privilege on table
- USAGE privilege on schema that contains the table
- SELECT privilege on the table when executing an UPDATE statement that references table column values in a WHERE or SET clause

Subqueries and joins

UPDATE supports subqueries and joins, which is useful for updating values in a table based on values that are stored in other tables. For details, see Subqueries in UPDATE and DELETE statements.

Committing successive table changes

Vertica follows the SQL-92 transaction model, so successive INSERT, UPDATE, and DELETE statements are included in the same transaction. You do not need to explicitly start this transaction; however, you must explicitly end it with COMMIT, or implicitly end it with COPY. Otherwise, Vertica discards all changes that were made within the transaction.

Restrictions

- You cannot update an immutable table.
- You cannot update columns of complex types except for native arrays.
- If the joins specified in the FROM clause or WHERE predicate produce more than one copy of the row in the target table, the new value of the row in the table is chosen arbitrarily.
- If primary key, unique key, or check constraints are enabled for automatic enforcement in the target table, Vertica enforces those constraints when you load new data. If a violation occurs, Vertica rolls back the operation and returns an error.
- If an update would violate a table or schema disk quota, the operation fails. For more information, see Disk quotas.

Examples

In the fact table, modify the price column value for all rows where the cost column value is greater than 100:

```
=> UPDATE fact SET price = price - cost * 80 WHERE cost > 100;
```

In the retail.customer table, set the state column to NH when the CID column value is greater than 100:

```
=> UPDATE retail.customer SET state = 'NH' WHERE CID > 100;
```

To use table aliases in UPDATE queries, consider the following two tables:

```
=> SELECT * FROM result_table;
 cust_id |      address
---------+--------------------
      20 | Lincoln Street
      30 | Beach Avenue
      30 | Booth Hill Road
      40 | Mt. Vernon Street
      50 | Hillside Avenue
(5 rows)
=> SELECT * FROM new_addresses;
 new_cust_id | new_address
-------------+--------------
          20 | Infinite Loop
          30 | Loop Infinite
          60 | New Addresses
(3 rows)
```

The following query and subquery use table aliases to update the address column in result_table (alias r) with the new address from the corresponding column in the new_addresses table (alias n):

```
=> UPDATE result_table r
   SET address=n.new_address
   FROM new_addresses n
   WHERE r.cust_id = n.new_cust_id;
```

result_table shows the address field updates made for customer IDs 20 and 30:

```
=> SELECT * FROM result_table ORDER BY cust_id;
 cust_id |     address
---------+------------------
      20 | Infinite Loop
      30 | Loop Infinite
      30 | Loop Infinite
      40 | Mt. Vernon Street
      50 | Hillside Avenue
(5 rows)
```

You cannot use UPDATE to update individual elements of native arrays. Instead, replace the entire array value. The following example uses ARRAY_CAT to add an element to an array column:

```
=> SELECT * FROM singers;
 lname  | fname |                  bands
--------+-------+-----------------------------------------------
 Cher   |       | ["Sonny and Cher"]
 Jagger | Mick  | ["Rolling Stones"]
 Slick  | Grace | ["Jefferson Airplane","Jefferson Starship"]
(3 rows)

=> UPDATE singers SET bands=ARRAY_CAT(bands,ARRAY['something new'])
    WHERE lname='Cher';
 OUTPUT
--------
      1
(1 row)

=> SELECT * FROM singers;
 lname  | fname |                  bands
--------+-------+-----------------------------------------------
 Jagger | Mick  | ["Rolling Stones"]
 Slick  | Grace | ["Jefferson Airplane","Jefferson Starship"]
 Cher   |       | ["Sonny and Cher","something new"]
(3 rows)
```

Vertica system tables

Vertica provides system tables that let you monitor your database and evaluate settings of its objects. You can query these tables just as you do other tables, depending on privilege requirements.

See also

- Using system tables
- Monitoring Vertica

In this section

- V_CATALOG schema
- V_MONITOR schema

V_CATALOG schema

The system tables in this section reside in the v_catalog schema. These tables provide information (metadata) about the objects in a database; for example, tables, constraints, users, projections, and so on.

In this section

- ACCESS_POLICY
- ALL_TABLES
- AUDIT_MANAGING_USERS_PRIVILEGES
- CA_BUNDLES
- CATALOG_SUBSCRIPTION_CHANGES
- CATALOG_SYNC_STATE
- CATALOG_TRUNCATION_STATUS
- CERTIFICATES
- CLIENT_AUTH
- CLIENT_AUTH_PARAMS
- CLUSTER_LAYOUT
- COLUMNS
- COMMENTS
- COMPLEX_TYPES
- CONSTRAINT_COLUMNS
- CRYPTOGRAPHIC_KEYS
- DATABASES
- DIRECTED_QUERIES
- DUAL
- ELASTIC_CLUSTER
- EPOCHS
- FAULT_GROUPS
- FOREIGN_KEYS
- GRANTS
- HCATALOG_COLUMNS
- HCATALOG_SCHEMATA
- HCATALOG_TABLE_LIST
- HCATALOG_TABLES
- ICEBERG_COLUMNS
- INHERITED_PRIVILEGES
- INHERITING_OBJECTS
- KEYWORDS
- LARGE_CLUSTER_CONFIGURATION_STATUS
- LICENSE_AUDITS
- LICENSES
- LOAD_BALANCE_GROUPS
- LOG_PARAMS
- LOG_QUERIES
- LOG_TABLES
- MATERIALIZE_FLEXTABLE_COLUMNS_RESULTS
- MODELS
- NETWORK_ADDRESSES

- NODE_SUBSCRIPTION_CHANGE_PHASES
- NODE_SUBSCRIPTIONS
- NODES
- ODBC_COLUMNS
- PASSWORD_AUDITOR
- PASSWORDS
- PRIMARY_KEYS
- PROFILE_PARAMETERS
- PROFILES
- PROJECTION_CHECKPOINT_EPOCHS
- PROJECTION_COLUMNS
- PROJECTION_DELETE_CONCERNS
- PROJECTIONS
- REGISTERED_MODELS
- RESOURCE_POOL_DEFAULTS
- RESOURCE_POOLS
- ROLES
- ROUTING_RULES
- SCHEDULER_TIME_TABLE
- SCHEMATA
- SEQUENCES
- SESSION_SUBSCRIPTIONS
- SHARDS
- STORAGE_LOCATIONS
- STORED_PROC_TRIGGERS
- SUBCLUSTER_RESOURCE_POOL_OVERRIDES
- SUBCLUSTERS
- SYSTEM_COLUMNS
- SYSTEM_TABLES
- TABLE_CONSTRAINTS
- TABLES
- TEXT_INDICES
- TYPES
- USER_AUDITS
- USER_CLIENT_AUTH
- USER_CONFIGURATION_PARAMETERS
- USER_FUNCTION_PARAMETERS
- USER_FUNCTIONS
- USER_PROCEDURES
- USER_SCHEDULES
- USER_TRANSFORMS
- USERS
- VIEW_COLUMNS
- VIEW_TABLES
- VIEWS

ACCESS_POLICY

Provides information about existing access policies .

Column Name	Data Type	Description
ACCESS_POLICY_OID	INTEGER	The unique identifier for the access policy.
TABLE_NAME	VARCHAR	Name of the table specified in the access policy.
IS_POLICY_ENABLED	BOOLEAN	Whether the access policy is enabled.

POLICY_TYPE	VARCHAR	The type of access policy assigned to the table: • Column Policy • Row Policy
EXPRESSION	VARCHAR	The expression used when creating the access policy.
COLUMN_NAME	VARCHAR	The column to which the access policy is assigned. Row policies apply to all columns in the table.
TRUST_GRANTS	BOOLEAN	If true, GRANT statements override the access policy when determining whether a user can perform DML operations on the column or row.

Privileges

By default, only the superuser can view this table. Superusers can grant non-superusers access to this table with the following statement. Non-superusers can only see rows for tables that they own:

```
=> GRANT SELECT ON access_policy TO PUBLIC
```

Examples

The following query returns all access policies on table public.customer_dimension :

```
=> \x
=> SELECT policy_type, is_policy_enabled, table_name, column_name, expression FROM access_policy WHERE table_name =
'public.customer_dimension';
-[ RECORD 1 ]-----+-----------------------------------------------------------------------------------------
policy_type       | Column Policy
is_policy_enabled | Enabled
table_name        | public.customer_dimension
column_name       | customer_address
expression        | CASE WHEN enabled_role('administrator') THEN customer_address ELSE '**************' END
```

ALL_TABLES

Provides summary information about tables in a Vertica database.

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	The name of the schema that contains the table.
TABLE_ID	INTEGER	Catalog-assigned numeric value that uniquely identifies the table.
TABLE_NAME	VARCHAR	The table name.
TABLE_TYPE	VARCHAR	The type of table, which can be one of the following: • TABLE • SYSTEM TABLE • VIEW • GLOBAL TEMPORARY • LOCAL TEMPORARY
REMARKS	VARCHAR	A brief comment about the table. You define this field by using the COMMENT ON TABLE and COMMENT ON VIEW commands.

Examples

```
onenode=> SELECT DISTINCT table_name, table_type FROM all_tables
          WHERE table_name ILIKE 't%';
      table_name       |  table_type
-----------------------+--------------
 types                 | SYSTEM TABLE
 trades                | TABLE
 tuple_mover_operations | SYSTEM TABLE
 tables                | SYSTEM TABLE
 tuning_recommendations | SYSTEM TABLE
 testid                | TABLE
 table_constraints     | SYSTEM TABLE
 transactions          | SYSTEM TABLE
(8 rows)
onenode=> SELECT table_name, table_type FROM all_tables
          WHERE table_name ILIKE 'my%';
 table_name | table_type
------------+------------
 mystocks   | VIEW
(1 row)
=> SELECT * FROM all_tables LIMIT 4;
-[ RECORD 1 ]-------------------------------------------
schema_name | v_catalog
table_id    | 10206
table_name  | all_tables
table_type  | SYSTEM TABLE
remarks     | A complete listing of all tables and views
-[ RECORD 2 ]-------------------------------------------
schema_name | v_catalog
table_id    | 10000
table_name  | columns
table_type  | SYSTEM TABLE
remarks     | Table column information
-[ RECORD 3 ]-------------------------------------------
schema_name | v_catalog
table_id    | 10054
table_name  | comments
table_type  | SYSTEM TABLE
remarks     | User comments on catalog objects
-[ RECORD 4 ]-------------------------------------------
schema_name | v_catalog
table_id    | 10134
table_name  | constraint_columns
table_type  | SYSTEM TABLE
remarks     | Table column constraint information
```

AUDIT_MANAGING_USERS_PRIVILEGES

Provides summary information about privileges, creating, modifying, and deleting users, and authentication changes. This table is a join of LOG_PARAMS , LOG_QUERIES , and LOG_TABLES filtered on the Managing_Users_Privileges category.

Column Name	Data Type	Description
ISSUED_TIME	VARCHAR	The time at which the query was executed.
USER_NAME	VARCHAR	Name of the user who issued the query at the time Vertica recorded the session.
USER_ID	INTEGER	Numeric representation of the user who ran the query.
HOSTNAME	VARCHAR	The hostname, IP address, or URL of the database server.

SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
AUDIT_TYPE	VARCHAR	The type of operation for the audit: • Query • Parameter • Table
AUDIT_TAG_NAME	VARCHAR	The tag name for the specific parameter, query, or table.
REQUEST_TYPE	VARCHAR	The type of query request. Examples include, but are not limited to: • QUERY • DDL • LOAD • UTILITY • TRANSACTION • PREPARE • EXECUTE • SET • SHOW
REQUEST_ID	INTEGER	The ID of the privilege request.
SUBJECT	VARCHAR	The name of the table or parameter that was queried or the subject of a query.
REQUEST	VARCHAR	Lists the privilege request.
SUCCESS	VARCHAR	Indicates whether or not the operation was successful.
CATEGORY	VARCHAR	The audit parent category, Managing_Users_Privileges.

CA_BUNDLES

Stores certificate authority (CA) bundles created by CREATE CA BUNDLE .

Column Name	Data Type	Description
OID	INTEGER	The object identifier.
NAME	VARCHAR	The name of the CA bundle.
OWNER	INTEGER	The OID of the owner of the CA bundle.
CERTIFICATES	INTEGER	The OIDs of the CA certificates inside the CA bundle.

Privileges

- See CA bundle OID, name, and owner: Superuser or owner of the CA bundle.
- See CA bundle contents: Owner of the bundle

Joining with CERTIFICATES

CA_BUNDLES only stores OIDs. Since operations on CA bundles require certificate and owner names, you can use the following query to map bundles to certificate and owner names:

```
=> SELECT user_name AS owner_name,
      owner     AS owner_oid,
      b.name    AS bundle_name,
      c.name    AS cert_name
FROM   (SELECT name,
               STRING_TO_ARRAY(certificates) :: array[INT] AS certs
        FROM   ca_bundles) b
      LEFT JOIN certificates c
             ON CONTAINS(b.certs, c.oid)
      LEFT JOIN users
             ON user_id = owner
ORDER  BY 1;

 owner_name |     owner_oid     | bundle_name  | cert_name
------------+-------------------+--------------+-----------
 dbadmin    | 45035996273704962 | ca_bundle    | root_ca
 dbadmin    | 45035996273704962 | ca_bundle    | ca_cert
(2 rows)
```

See also

- Managing CA bundles
- ALTER CA BUNDLE
- DROP CA BUNDLE

CATALOG_SUBSCRIPTION_CHANGES

Lists the changes made to catalog subscriptions.

Column Name	Data Type	Description
EVENT_TIMESTAMP	TIMESTAMP	The time a catalog subscription changed.
SESSION_ID	VARCHAR	A unique numeric ID assigned by the Vertica catalog, which identifies the session for which profiling information is captured. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
USER_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog, which identifies the user.
USER_NAME	VARCHAR	The user who made changes to the subscriptions.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any. If a session is active but no transaction has begun, TRANSACTION_ID returns NULL.
SHARD_NAME	VARCHAR	The name of the shard.
SHARD_OID	INTEGER	The OID of the shard.
SUBSCRIBER_NODE_NAME	VARCHAR	The node name or names subscribed to the shard.
SUBSCRIBER_NODE_OID	INTEGER	The OID of the subscribing node or nodes.
OLD_STATE	VARCHAR	The previous state of the node subscription.
NEW_STATE	VARCHAR	The current state of the node subscription.
WAS_PRIMARY	BOOLEAN	Defines whether the node was the primary subscriber.
IS_PRIMARY	BOOLEAN	Defines whether the node is currently the primary subscriber.

CATALOG_VERSION	INTEGER	The version of the catalog at the time of the subscription change.

CATALOG_SYNC_STATE

Shows when an Eon Mode database node synchronized its catalog to communal storage.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
SYNC_CATALOG_VERSION	INTEGER	The version number of the catalog being synchronized.
EARLIEST_CHECKPOINT_VERSION	INTEGER	The earliest checkpoint that is maintained in communal storage.
SYNC_TRAILING_INTERVAL	INTEGER	The difference between the global catalog version and the synchronized catalog version for a node.
LAST_SYNC_AT	TIMESTAMPTZ	The date and time the last time the catalog was synchronized.

CATALOG_TRUNCATION_STATUS

Indicates how up to date the catalog is on communal storage. It is completely up to date when the current catalog version is the same as the catalog truncation version.

The catalog truncation version (CTV) is the version that Vertica cluster uses when it revives after a crash, shutdown, or hibernation. A cluster has only one CTV for all nodes in a cluster.

Column Name	Data Type	Description
CURRENT_CATALOG_VERSION	INTEGER	The version number of the catalog currently on the cluster.
TRUNCATION_CATALOG_VERSION	INTEGER	The version number as of the last time the catalog was synced on communal storage.

CERTIFICATES

Stores certificates created by CREATE CERTIFICATE .

Column Name	Data Type	Description
OID	INTEGER	The object identifier.
NAME	VARCHAR	The name of the certificate.
OWNER	INTEGER	The owner of the object.
SIGNED_BY	INTEGER	The OID of the signing certificate.
PRIVATE_KEY	INTEGER	The OID of the certificate's private key .
START_DATE	TIMESTAMPTZ	When the certificate becomes valid.
EXPIRATION_DATE	TIMESTAMPTZ	When the certificate expires.
ISSUER	VARCHAR	The signing CA.
SUBJECT	VARCHAR	The entity for which the certificate is issued.
SERIAL	VARCHAR	The certificate's serial number.

x509v3_EXTENSIONS	VARCHAR	Lists additional attributes specified during the certificate's creation. For more information on extensions, see the OpenSSL documentation .
CERTIFICATE_TEXT	VARCHAR	The contents of the certificate.

Examples

See Generating TLS certificates and keys .

CLIENT_AUTH

Provides information about client authentication methods.

Higher values indicate higher priorities. Vertica tries to authenticate a user with an authentication method in order of priority from highest to lowest. For example:

- A priority of 10 is higher than a priority of 5.
- A priority 0 is the lowest possible value.

Column Name	**Data Type**	**Description**
AUTH_OID	INTEGER	Unique identifier for the authentication method.
AUTH_NAME	VARCHAR	User-given name of the authentication method.
IS_AUTH_ENABLED	BOOLEAN	Indicates if the authentication method is enabled.
AUTH_HOST_TYPE	VARCHAR	The authentication host type, one of the following: • LOCAL • HOST • HOSTSSL • HOSTNOSSL
AUTH_HOST_ADDRESS	VARCHAR	If AUTH_HOST_TYPE is HOST, AUTH_HOST_ADDRESS is the IP address (or address range) of the remote host.
AUTH_METHOD	VARCHAR	Authentication method to be used. Valid values: • IDENT • GSS • HASH • LDAP • REJECT • TLS • TRUST
AUTH_PARAMETERS	VARCHAR	The parameter names and values assigned to the authentication method.
AUTH_PRIORITY	INTEGER	The priority specified for the authentication. Authentications with higher values are used first.
METHOD_PRIORITY	INTEGER	The priority of this authentication based on the AUTH_METHOD. Vertica only considers METHOD_PRIORITY when deciding between multiple authentication methods of equal AUTH_PRIORITY.

ADDRESS_PRIORITY	INTEGER	The priority of this authentication based on the specificity of the AUTH_HOST_ADDRESS, if any. More specific IP addresses (fewer zeros) are used first. Vertica only considers ADDRESS_PRIORITY when deciding between multiple authentication methods of equal AUTH_PRIORITY and METHOD_PRIORITY.
IS_FALLTHROUGH_ENABLED	Boolean	Whether authentication fallthrough is enabled.

Examples

This example shows how to get information about each client authentication method that you created:

```
=> SELECT * FROM client_auth;
    auth_oid   |  auth_name  | is_auth_enabled | auth_host_type | auth_host_address | auth_method | auth_parameters | auth_priority | method_priority |
address_priority
-------------------+-------------+-----------------+----------------+-------------------+-------------+-----------------+---------------+-----------------+------------------
 45035996274059694 | v_gss       | True            | HOST           | 0.0.0.0/0         | GSS         |                 |             0 |             5 |               96
 45035996274059696 | v_trust     | True            | LOCAL          |                   | TRUST       |                 |             0 |             0 |                0
 45035996274059698 | v_ldap      | True            | HOST           | 10.19.133.123/    | LDAP        |                 |             0 |             5 |              128
 45035996274059700 | RejectNoSSL | True            | HOSTNOSSL      | 0.0.0.0/0         | REJECT      |                 |             0 |            10 |               96
 45035996274059702 | v_hash      | True            | LOCAL          |                   | HASH        |                 |             0 |             2 |                0
 45035996274059704 | v_tls       | True            | HOSTSSL        | 1.1.1.1/0         | TLS         |                 |             0 |             5 |               96
(6 rows)
```

See also

- Client authentication
- Authentication record priority

CLIENT_AUTH_PARAMS

Provides information about client authentication methods that have parameter values assigned.

Column Name	Data Type	Description
AUTH_OID	INTEGER	A unique identifier for the authentication method.
AUTH_NAME	VARCHAR	Name that you defined for the authentication method.
AUTH_PARAMETER_NAME	VARCHAR	Parameter name required by the authentication method. Some examples are: • system_users • binddn_prefix • host
AUTH_PARAMETER_VALUE	VARCHAR	Value of the specified parameter.

Examples

This example shows how to retrieve parameter names and values for all authentication methods that you created. The authentication methods that have parameters are:

- v_ident
- v_ldap
- v_ldap1

```
=> SELECT * FROM CLIENT_AUTH_PARAMS;
    auth_oid     |   auth_name   | auth_parameter_name |    auth_parameter_value
-----------------+---------------+---------------------+-----------------------------
 45035996273741304 | v_ident       | system_users        | root
 45035996273741332 | v_gss         |                     |
 45035996273741350 | v_password    |                     |
 45035996273741368 | v_trust       |                     |
 45035996273741388 | v_ldap        | host                | ldap://172.16.65.177
 45035996273741388 | v_ldap        | binddn_prefix       | cn=
 45035996273741388 | v_ldap        | binddn_suffix       | ,dc=qa_domain,dc=com
 45035996273741406 | RejectNoSSL   |                     |
 45035996273741424 | RejectWithSSL |                     |
 45035996273741450 | v_md5         |                     |
 45035996273904044 | l_tls         |                     |
 45035996273906566 | v_hash        |                     |
 45035996273910432 | v_ldap1       | host                | ldap://172.16.65.177
 45035996273910432 | v_ldap1       | basedn              | dc=qa_domain,dc=com
 45035996273910432 | v_ldap1       | binddn              | cn=Manager,dc=qa_domain,dc=com
 45035996273910432 | v_ldap1       | bind_password       | secret
 45035996273910432 | v_ldap1       | search_attribute    | cn
(17 rows)
```

CLUSTER_LAYOUT

Shows the relative position of the actual arrangement of the nodes participating in the cluster and the fault groups (in an Enterprise Mode database) or subclusters (in an Eon Mode database) that affect them. Ephemeral nodes are not shown in the cluster layout ring because they hold no resident data.

Column Name	Data Type	Description
CLUSTER_POSITION	INTEGER	Position of the node in the cluster ring, counting forward from 0. **Note** An output value of 0 has no special meaning other than there are no nodes in position before the node assigned 0.
NODE_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog that identifies the node.
NODE_NAME	VARCHAR	The name of the node in the cluster ring. Only permanent nodes participating in database activity appear in the cluster layout. Ephemeral nodes are not shown in the output.
FAULT_GROUP_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog that identifies the fault group. This column can only have a value in an Enterprise Mode database. **Note** This value matches the FAULT_GROUP.MEMBER_ID value, but only if this node is in a fault group; otherwise the value is NULL.
FAULT_GROUP_NAME	VARCHAR	The name of the fault group for the node. This column can only have a value in an Enterprise Mode database.

FAULT_GROUP_TIER	INTEGER	The node's depth in the fault group tree hierarchy. For example if the node: • Is not in a fault group, output is null • Is in the top level fault group, output is 0 • Is in a fault group's child, output is 1 • Is a fault group's grandchild, output is 2 This column can only have a value in an Enterprise Mode database.
SUBCLUSTER_ID	INTEGER	Unique identifier for the subcluster. This column only has a value in an Eon Mode database.
SUBCLUSTER_NAME	VARCHAR	The name of the subcluster containing the node. This column only has a value in an Eon Mode database.

See also

Large cluster

COLUMNS

Provides table column information. For columns of Iceberg external tables, see ICEBERG_COLUMNS .

Column Name	Data Type	Description
TABLE_ID	INTEGER	Catalog-assigned numeric value that uniquely identifies the table.
TABLE_SCHEMA	VARCHAR	Name of the table's schema.
TABLE_NAME	VARCHAR	Name of the table containing the column.
IS_SYSTEM_TABLE	BOOLEAN	Whether the table is a system table.
COLUMN_ID	VARCHAR	Catalog-assigned VARCHAR value that uniquely identifies a table column.
COLUMN_NAME	VARCHAR	Name of the column.
DATA_TYPE	VARCHAR	Column data type. Arrays of primitive types show the name ARRAY[*type*] . Other complex types show the inline name of the type, which matches the type_name value in the COMPLEX_TYPES table. For example: _ct_45035996273833610 .
DATA_TYPE_ID	INTEGER	Catalog-assigned unique numeric ID of the data type.
DATA_TYPE_LENGTH	INTEGER	Maximum allowable length of the data type.
CHARACTER_MAXIMUM_LENGTH	VARCHAR	Maximum allowable length of a VARCHAR column.
NUMERIC_PRECISION	INTEGER	Number of significant decimal digits for a NUMERIC column.
NUMERIC_SCALE	INTEGER	Number of fractional digits for a NUMERIC column.
DATETIME_PRECISION	INTEGER	Declared precision for a TIMESTAMP column, or NULL if no precision was declared.
INTERVAL_PRECISION	INTEGER	Number of fractional digits retained in the seconds field of an INTERVAL column.
ORDINAL_POSITION	INTEGER	Column position respective to other columns in the table.
IS_NULLABLE	BOOLEAN	Whether the column can contain NULL values.

COLUMN_DEFAULT	VARCHAR	Expression set on a column with the DEFAULT constraint.
COLUMN_SET_USING	VARCHAR	Expression set on a column with the SET USING constraint.
IS_IDENTITY	BOOLEAN	Whether the column is an IDENTITY column.

Examples

```
=> SELECT table_schema, table_name, column_name, data_type, is_nullable
   FROM columns WHERE table_schema = 'store'
   AND data_type = 'Date';
 table_schema |    table_name     |      column_name       | data_type | is_nullable
--------------+-------------------+------------------------+-----------+-------------
 store        | store_dimension   | first_open_date        | Date      | f
 store        | store_dimension   | last_remodel_date      | Date      | f
 store        | store_orders_fact | date_ordered           | Date      | f
 store        | store_orders_fact | date_shipped           | Date      | f
 store        | store_orders_fact | expected_delivery_date | Date      | f
 store        | store_orders_fact | date_delivered         | Date      | f
6 rows)
```

In the following query, datetime_precision is NULL because the table definition declares no precision:

```
=> CREATE TABLE c (c TIMESTAMP);
CREATE TABLE
=> SELECT table_name, column_name, datetime_precision FROM columns
   WHERE table_name = 'c';
 table_name | column_name | datetime_precision
------------+-------------+--------------------
 c          | c           |
(1 row)
```

In the following example, timestamp precision is set:

```
=> DROP TABLE c;
=> CREATE TABLE c (c TIMESTAMP(4));
CREATE TABLE
=> SELECT table_name, column_name, datetime_precision FROM columns
   WHERE table_name = 'c';
 table_name | column_name | datetime_precision
------------+-------------+--------------------
 c          | c           |                  4
```

An IDENTITY column sequence is defined in a table's DDL. Column values automatically increment as new rows are added. The following query returns identity columns:

```
=> CREATE TABLE employees (employeeID IDENTITY, fname varchar(36), lname varchar(36));
CREATE TABLE
=> SELECT table_name, column_name, is_identity FROM columns WHERE is_identity = 't';
 table_name | column_name | is_identity
------------+-------------+-------------
 employees  | employeeID  | t
(1 row)
```

You can query the SEQUENCES table to get detailed information about an IDENTITY column sequence:

```
=> SELECT sequence_schema, sequence_name, identity_table_name, sequence_id FROM sequences WHERE identity_table_name ='employees';
 sequence_schema |      sequence_name       | identity_table_name |    sequence_id
-----------------+--------------------------+---------------------+-------------------
 public          | employees_employeeID_seq | employees           | 45035996273848816
(1 row)
```

For details about sequences and IDENTITY columns, see Sequences .

COMMENTS

Returns information about comments associated with objects in the database.

Column Name	Data Type	Description
COMMENT_ID	INTEGER	Comment's internal ID number
OBJECT_ID	INTEGER	Internal ID number of the object associated with the comment.
OBJECT_TYPE	VARCHAR	Type of object associated with the comment, one of the following: • COLUMN • CONSTRAINT • FUNCTION • LIBRARY • NODE • PROJECTION • SCHEMA • SEQUENCE • TABLE • VIEW
OBJECT_SCHEMA	VARCHAR	Schema that contains the object.
OBJECT_NAME	VARCHAR	Name of the object associated with the comment.
OWNER_ID	VARCHAR	Internal ID of the object's owner.
OWNER_NAME	VARCHAR	Object owner's name.
CREATION_TIME	TIMESTAMPTZ	When the comment was created.
LAST_MODIFIED_TIME	TIMESTAMPTZ	When the comment was last modified.
COMMENT	VARCHAR	Comment text.

Caution

Queries on this table can be slow, as it obtains much of its data by querying other Vertica catalog tables.

COMPLEX_TYPES

Contains information about inlined complex types .

Each complex type in each external table has a unique type internally, even if the types are structurally the same (like two different ROW(int,int) cases). This inlined type is created when the table using it is created and is automatically dropped when the table is dropped. Inlined complex types cannot be shared or reused in other tables.

Each row in the COMPLEX_TYPES table represents one component (field) in one complex type. A ROW produces one row per field, an ARRAY produces one, and a MAP produces two.

Arrays of primitive types used in native (ROS) tables are not included in the COMPLEX_TYPES table. They are included instead in the TYPES table.

This table does not include complex types in Iceberg tables .

Column Name	Data Type	Description

TYPE_ID	INTEGER	A unique identifier for the inlined complex type.
TYPE_KIND	VARCHAR	The specific kind of complex type: row, array, or map.
TYPE_NAME	VARCHAR	The generated name of this type. All names begin with _ct_ followed by a number.
FIELD_ID	INTEGER	A unique identifier for the field.
FIELD_NAME	VARCHAR	The name of the field, if specified in the table definition, or a generated name beginning with "f".
FIELD_TYPE_NAME	VARCHAR	The type of the field's value.
FIELD_POSITION	INTEGER	The field's position in its containing complex type (0-based).
FIELD_LENGTH	INTEGER	Number of bytes in the field value, or -1 if the value is not a scalar type.
CHARACTER_MAXIMUM_LENGTH	INTEGER	Maximum allowable length of the column.
NUMERIC_PRECISION	INTEGER	Number of significant decimal digits.
NUMERIC_SCALE	INTEGER	Number of fractional digits.
DATETIME_PRECISION	INTEGER	For TIMESTAMP data type, returns the declared precision; returns NULL if no precision was declared.
INTERVAL_PRECISION	INTEGER	Number of fractional digits retained in the seconds field.

Examples

The following example shows the type and field values after defining a single external table.

```
=> CREATE EXTERNAL TABLE warehouse(
   name VARCHAR, id_map MAP<INT,VARCHAR>,
   data row(record INT, total FLOAT, description VARCHAR(100)),
   prices ARRAY[INT], comment VARCHAR(200), sales_total FLOAT, storeID INT)
 AS COPY FROM ... PARQUET;

=> SELECT type_id,type_kind,type_name,field_id,field_name,field_type_name,field_position
   FROM COMPLEX_TYPES ORDER BY type_id,field_name;

     type_id      | type_kind |       type_name       | field_id | field_name  | field_type_name | field_position
------------------+-----------+-----------------------+----------+-------------+-----------------+----------------
 45035996274278280 | Map       | _ct_45035996274278280 |        6 | key         | int             |              0
 45035996274278280 | Map       | _ct_45035996274278280 |        9 | value       | varchar(80)     |              1
 45035996274278282 | Row       | _ct_45035996274278282 |        9 | description | varchar(80)     |              2
 45035996274278282 | Row       | _ct_45035996274278282 |        6 | record      | int             |              0
 45035996274278282 | Row       | _ct_45035996274278282 |        7 | total       | float           |              1
 45035996274278284 | Array     | _ct_45035996274278284 |        6 |             | int             |              0
(6 rows)
```

CONSTRAINT_COLUMNS

Records information about table column constraints.

Column Name	Data Type	Description
CONSTRAINT_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog that identifies the constraint.

TABLE_SCHEMA	VARCHAR	Name of the schema that contains this table.
TABLE_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog that identifies the table.
TABLE_NAME	VARCHAR	Name of the table in which the column resides.
COLUMN_NAME	VARCHAR	Name of the column that is constrained. For check constraints, if more than one column is referenced, each appears as a separate row.
CONSTRAINT_NAME	VARCHAR	Constraint name for which information is listed.
CONSTRAINT_TYPE	CHAR	The constraint type, one of the following: • c : check • f : foreign • n : not null • p : primary • u : unique
IS_ENABLED	BOOLEAN	Indicates whether a constraint for a primary key, unique key, or check constraint is currently enabled.
REFERENCE_TABLE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies thereferenced table
REFERENCE_TABLE_SCHEMA	VARCHAR	Schema name for which information is listed.
REFERENCE_TABLE_NAME	VARCHAR	References the TABLE_NAME column in the PRIMARY_KEY table.
REFERENCE_COLUMN_NAME	VARCHAR	References the COLUMN_NAME column in the PRIMARY_KEY table.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

CRYPTOGRAPHIC_KEYS

Stores private keys created by CREATE KEY .

Column Name	Data Type	Description
OID	INTEGER	The object identifier.
NAME	VARCHAR	Name of the key.
OWNER	INTEGER	The owner of the object.
TYPE	INTEGER	The type of key. • 0 = AES • 1 = RSA
LENGTH	INTEGER	The size of the key in bits.
HAS_PASSWORD	BOOLEAN	Whether the key has a password.
KEY	VARCHAR	The private key.

Examples

See Generating TLS certificates and keys .

DATABASES

Provides information about the databases in this Vertica installation.

Column Name	Data Type	Description
DATABASE_ID	INTEGER	The database's internal ID number
DATABASE_NAME	VARCHAR	The database's name
OWNER_ID	INTEGER	The database owner's ID
OWNER_NAME	INTEGER	The database owner's name
START_TIME	TIMESTAMPTZ	The date and time the database last started
COMPLIANCE_MESSAGE	VARCHAR	Message describing the current state of the database's license compliance
EXPORT_SUBNET	VARCHAR	Can be either of the following: • The subnet (on the public network) used by the database for import/export • The public address of the subnet that the Vertica native load balancing uses for load balancing
LOAD_BALANCE_POLICY	VARCHAR	The current native connection load balance policy, which controls whether client connection requests are redirected to other hosts in the database. See About native connection load balancing .
BACKEND_ADDRESS_FAMILY	VARCHAR	The Internet Protocol (IP) addressing standard used for internode communications. This value is either ipv4 or ipv6.
BRANCH_NAME	VARCHAR	This column is no longer used.

Examples

This example queries the databases table from a master database.

```
=> SELECT * FROM DATABASES;
-[ RECORD 1 ]----------+---------------------------------------------------------------
database_id            | 45035996273704976
database_name          | VMart
owner_id               | 45035996273704962
owner_name             | dbadmin
start_time             | 2017-10-22 05:16:22.066961-04
compliance_message     | The database is in compliance with respect to raw data size.
export_subnet          | 0
load_balance_policy    | none
backend_address_family | ipv4
branch_name            |
```

DIRECTED_QUERIES

Returns information about directed queries.

Column Name	Data Type	Description

QUERY_NAME	VARCHAR	Directed query's unique identifier, used by statements such as ACTIVATE DIRECTED QUERY . How this identifier is set depends on how it was created: • Direct call to CREATE DIRECTED QUERY : Set by the user-supplied *query-name* argument. • Call to SAVE_PLANS : Concatenated from the following strings: save_plans_*query-label*_*query-number*_*save-plans-version* where: • *query-label* is a LABEL hint embedded in the input query associated with this directed query. If theinput query contains no label, then this string is set to nolabel . • *query-number* is an integer in a continuous sequence between 0 and *budget-query* , which uniquely identifies this directed query from others in the same SAVE_PLANS-generated set. • *[save-plans-version](/sql-reference/system-tables/v-catalog-schema/directed-queries.html#SAVE_PLANS_VERSION)* identifies the set of directed queries to which this directed query belongs.
IS_ACTIVE	BOOLEAN	Whether the directed query is active .
VERTICA_VERSION	VARCHAR	Vertica version installed when this directed query was created.
COMMENT	VARCHAR	User-supplied or optimizer-generated comment on a directed query, up to 128 characters.
SAVE_PLANS_VERSION	INTEGER	One of the following: • 0: Generated by a direct call to CREATE DIRECTED QUERY . • >0: Identifies a set of directed queries that were generated by the same call to SAVE_PLANS. All directed queries of the set share the same SAVE_PLANS_VERSION integer, which increments by 1 the previous highest SAVE_PLANS_VERSION setting. Use this identifier to activate , deactivate , and drop a set of directed queries.
USERNAME	VARCHAR	User that created this directed query.
CREATION_DATE	VARCHAR	When the directed query was created.
SINCE_DATE	VARCHAR	Populated by SAVE_PLANS-generated directed queries, the earliest timestamp of input queries eligible to be saved as directed queries.
INPUT_QUERY	VARCHAR	Input query associated with this directed query. Multiple directed queries can map to the same input query.
ANNOTATED_QUERY	VARCHAR	Query with embedded hints that was paired with the input query of this directed query, where the hints encapsulated the query plan saved with CREATE DIRECTED QUERY .
DIGEST	INTEGER	Hash of saved query plan data, used by the optimizer to map identical input queries to the same active directed query.

Privileges

Superuser

Truncated query results

Query results for the fields INPUT_QUERY and ANNOTATED_QUERY are truncated after ~32K characters. You can get the full content of both fields in two ways:

- Use the statement GET DIRECTED QUERY .
- Use EXPORT_CATALOG to export directed queries.

DUAL

DUAL is a single-column "dummy" table with one record whose value is X; for example:

```
=> SELECT * FROM DUAL;
 dummy
-------
 X
(1 row)
```

You can write the following types of queries:

```
=> SELECT 1 FROM dual;
 ?column?
----------
        1
(1 row)
=> SELECT current_timestamp, current_user FROM dual;
          ?column?           | current_user
-------------------------------+--------------
 2010-03-08 12:57:32.065841-05 | release
(1 row)
=> CREATE TABLE t1(col1 VARCHAR(20), col2 VARCHAR(2));
=> INSERT INTO T1(SELECT 'hello' AS col1, 1 AS col2 FROM dual);)
=> SELECT * FROM t1;
 col1  | col2
-------+------
 hello | 1
(1 row
```

Restrictions

You cannot create projections for DUAL.

ELASTIC_CLUSTER

Returns information about cluster elasticity, such as whether Elastic cluster is running.

Column Name	Data Type	Description
SCALING_FACTOR	INTEGER	This value is only meaningful when you enable local segments. SCALING_FACTOR influences the number of local segments on each node. Initially—before a rebalance runs—there are *scaling_factor* number of local segments per node. A large SCALING_FACTOR is good for rebalancing a potentially wide range of cluster configurations quickly. However, too large a value might lead to ROS pushback , particularly in a database with a table with a large number of partitions. See SET_SCALING_FACTOR for more details.
MAXIMUM_SKEW_PERCENT	INTEGER	This value is only meaningful when you enable local segments. MAXIMUM_SKEW_PERCENT is the maximum amount of skew a rebalance operation tolerates, which preferentially redistributes local segments; however, if after doing so the segment ranges of any two nodes differs by more than this amount, rebalance will separate and distribute storage to even the distribution.
SEGMENT_LAYOUT	VARCHAR	Current, offset=0, segment layout. New segmented projections will be created with this layout, with segments rotated by the corresponding offset. Existing segmented projections will be rebalanced into an offset of this layout.
LOCAL_SEGMENT_LAYOUT	VARCHAR	Similar to SEGMENT_LAYOUT but includes details that indicate the number of local segments, their relative size and node assignment.
VERSION	INTEGER	Number that gets incremented each time the cluster topology changes (nodes added, marked ephemeral, marked permanent, etc). Useful for monitoring active and past rebalance operations.
IS_ENABLED	BOOLEAN	True if Elastic Cluster is enabled, otherwise false.

IS_LOCAL_SEGMENT_ENABLED	BOOLEAN	True if local segments are enabled, otherwise false.
IS_REBALANCE_RUNNING	BOOLEAN	True if rebalance is currently running, otherwise false.

Privileges

Superuser

See also

- ENABLE_ELASTIC_CLUSTER
- Elastic cluster

EPOCHS

For the most recently closed epochs, lists the date and time of the close and the corresponding epoch number of the closed epoch. The EPOCHS table may return a varying number of rows depending on current commit activities.

Column Name	Data Type	Description
EPOCH_CLOSE_TIME	DATETIME	The date and time that the epoch closed.
EPOCH_NUMBER	INTEGER	The epoch number of the closed epoch.

Examples

```
=> SELECT * FROM EPOCHS;
      epoch_close_time       | epoch_number
-------------------------------+--------------
 2018-11-12 16:05:15.552571-05 |           16
(1 row)
```

Querying for historical data

If you need historical data about epochs and corresponding date information, query the DC_TRANSACTION_ENDS table.

```
=> select dc.end_epoch,min(dc.time),max(dc.time) from dc_transaction_ends dc group by end_epoch;
 end_epoch |              min              | max
-----------+-------------------------------+-------------------------------
       214 |  2018-10-12 08:05:47.02075-04 | 2018-10-15 10:22:24.015292-04
       215 | 2018-10-15 10:22:47.015172-04 | 2018-10-15 13:00:44.888984-04
                         ...
       226 | 2018-10-15 15:03:47.015235-04 | 2018-10-15 20:37:34.346667-04
       227 | 2018-10-15 20:37:47.008137-04 | 2018-10-16 07:39:00.29917-04
       228 | 2018-10-16 07:39:47.012411-04 | 2018-10-16 08:16:01.470232-04
       229 | 2018-10-16 08:16:47.018899-04 | 2018-10-16 08:21:13.854348-04
       230 | 2018-10-16 08:21:47.013767-04 | 2018-10-17 12:21:09.224094-04
       231 |  2018-10-17 12:21:09.23193-04 | 2018-10-17 15:11:59.338777-04
```

See also

- Epoch management parameters
- Epoch functions

FAULT_GROUPS

View the fault groups and their hierarchy in the cluster.

Column Name	Data Type	Description
MEMBER_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog that identifies the fault group.
MEMBER_TYPE	VARCHAR	The type of fault group. Values can be either NODE or FAULT GROUP .

MEMBER_NAME	VARCHAR	Name associated with this fault group. Values will be the node name or the fault group name.
PARENT_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog that identifies the parent fault group. The parent fault group can contain: • Nodes • Other fault groups • Nodes and other fault groups
PARENT_TYPE	VARCHAR	The type of parent fault group, where the default/root parent is the DATABASE object. Can be one of the following objects: • FAULT GROUP • DATABASE
PARENT_NAME	VARCHAR	The name of the fault group that contains nodes or other fault groups or both nodes and fault groups.
IS_AUTOMATICALLY_GENERATED	BOOLEAN	If true, denotes whether Vertica Analytic Database created fault groups for you to manage the fault tolerance of control nodes in large cluster configurations. If false, denotes that you created fault groups manually. See Fault Groups for more information

Examples

Show the current hierarchy of fault groups in the cluster:

```
vmartdb=> SELECT member_type, member_name, parent_type, CASE
        WHEN parent_type = 'DATABASE' THEN ''
        ELSE parent_name END FROM fault_groups
        ORDER BY member_name;
 member_type | member_name          | parent_type | parent_name
-------------+----------------------+-------------+-------------
 NODE        | v_vmart_node0001     | FAULT GROUP | two
 NODE        | v_vmart_node0002     | FAULT GROUP | two
 NODE        | v_vmart_node0003     | FAULT GROUP | three
 FAULT GROUP | one                  | DATABASE    |
 FAULT GROUP | three                | DATABASE    |
 FAULT GROUP | two                  | FAULT GROUP | one
```

View the distribution of the segment layout:

```
vmartdb=> SELECT segment_layout from elastic_cluster;
                        segment_layout
-------------------------------------------------------------------------------
 v_vmart_node0001[33.3%] v_vmart_node0003[33.3%] v_vmart_node0004[33.3%]
(1 row)
```

See also

- High availability with fault groups
- Fault groups

FOREIGN_KEYS

Provides foreign key information.

Column Name	Data Type	Description
CONSTRAINT_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog, which identifies the constraint.
CONSTRAINT_NAME	VARCHAR	The constraint name for which information is listed.

COLUMN_NAME	VARCHAR	The name of the column that is constrained.
ORDINAL_POSITION	VARCHAR	The position of the column within the key. The numbering of columns starts at 1.
TABLE_NAME	VARCHAR	The table name for which information is listed.
REFERENCE_TABLE_NAME	VARCHAR	References the TABLE_NAME column in the PRIMARY_KEY table.
CONSTRAINT_TYPE	VARCHAR	The constraint type, f , for foreign key.
REFERENCE_COLUMN_NAME	VARCHAR	References the COLUMN_NAME column in the PRIMARY_KEY table.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.
REFERENCE_TABLE_SCHEMA	VARCHAR	References the TABLE_SCHEMA column in the PRIMARY_KEY table.

Examples

```
mydb=> SELECT
         constraint_name,
         table_name,
         ordinal_position,
         reference_table_name
       FROM foreign_keys ORDER BY 3;
      constraint_name      |    table_name     | ordinal_position | reference_table_name
---------------------------+-------------------+------------------+----------------------
 fk_store_sales_date       | store_sales_fact  |                1 | date_dimension
 fk_online_sales_saledate  | online_sales_fact |                1 | date_dimension
 fk_store_orders_product   | store_orders_fact |                1 | product_dimension
 fk_inventory_date         | inventory_fact    |                1 | date_dimension
 fk_inventory_product      | inventory_fact    |                2 | product_dimension
 fk_store_sales_product    | store_sales_fact  |                2 | product_dimension
 fk_online_sales_shipdate  | online_sales_fact |                2 | date_dimension
 fk_store_orders_product   | store_orders_fact |                2 | product_dimension
 fk_inventory_product      | inventory_fact    |                3 | product_dimension
 fk_store_sales_product    | store_sales_fact  |                3 | product_dimension
 fk_online_sales_product   | online_sales_fact |                3 | product_dimension
 fk_store_orders_store     | store_orders_fact |                3 | store_dimension
 fk_online_sales_product   | online_sales_fact |                4 | product_dimension
 fk_inventory_warehouse    | inventory_fact    |                4 | warehouse_dimension
 fk_store_orders_vendor    | store_orders_fact |                4 | vendor_dimension
 fk_store_sales_store      | store_sales_fact  |                4 | store_dimension
 fk_store_orders_employee  | store_orders_fact |                5 | employee_dimension
 fk_store_sales_promotion  | store_sales_fact  |                5 | promotion_dimension
 fk_online_sales_customer  | online_sales_fact |                5 | customer_dimension
 fk_store_sales_customer   | store_sales_fact  |                6 | customer_dimension
 fk_online_sales_cc        | online_sales_fact |                6 | call_center_dimension
 fk_store_sales_employee   | store_sales_fact  |                7 | employee_dimension
 fk_online_sales_op        | online_sales_fact |                7 | online_page_dimension
 fk_online_sales_shipping  | online_sales_fact |                8 | shipping_dimension
 fk_online_sales_warehouse | online_sales_fact |                9 | warehouse_dimension
 fk_online_sales_promotion | online_sales_fact |               10 | promotion_dimension
(26 rows)
```

GRANTS

Returns information about privileges that are explicitly granted on database objects. Information about inherited privileges is not included.

> **Note**
> While an ADMIN OPTION granted to users through roles is not viewable directly from this table, you can view it and a summary of privileges data

with vsql meta-commands \z and \dp .

Column Name	Data Type	Description
GRANTEE	VARCHAR	The user being granted permission.
GRANTEE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theuser granted permissions.
GRANT_ID	INTEGER	Catalog-assigned integer value that uniquely identifies thegrant operation.
GRANTOR	VARCHAR	The user granting the permission.
GRANTOR_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theuser who performed the grant operation.
OBJECT_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theobject granted.
OBJECT_NAME	VARCHAR	The name of the object that is being granted privileges. Note that for schema privileges, the schema name appears in the OBJECT_NAME column instead of the OBJECT_SCHEMA column.
OBJECT_SCHEMA	VARCHAR	The name of the schema that is being granted privileges.
OBJECT_TYPE	VARCHAR	The object type on which the grant was applied—for example, ROLE, SCHEMA, DATABASE, RESOURCEPOOL.
PRIVILEGES_DESCRIPTION	VARCHAR	Lists the privileges granted on an object—for example INSERT, SELECT. An asterisk in PRIVILEGES_DESCRIPTION output shows that the privilege grant included WITH GRANT OPTION .

Examples

The following query shows the privileges that are granted to user Rob or role R1. An asterisk (*) appended to a privilege indicates that the user can grant the privilege to other users:

```
=> SELECT grantor,privileges_description,object_name,object_type,grantee FROM grants WHERE grantee='Rob' OR grantee='R1';
 grantor |  privileges_description   | object_name | object_type  | grantee
---------+--------------------------+-------------+--------------+---------
dbadmin | USAGE                     | general     | RESOURCEPOOL | Rob
dbadmin | USAGE, CREATE             | s1          | SCHEMA       | Rob
dbadmin | INSERT*, SELECT*, UPDATE* | t1          | TABLE        | Rob
dbadmin | SELECT                    | t1          | TABLE        | R1
dbadmin | USAGE                     | s1          | SCHEMA       | R1
dbadmin |                           | R1          | ROLE         | Rob
(6 rows)
```

See also

- HAS_ROLE
- ROLES
- USERS
- Database users and privileges

HCATALOG_COLUMNS

Describes the columns of all tables available through the HCatalog Connector. Each row in this table corresponds to to a column in a table accessible through the HCatalog Connector. See Using the HCatalog Connector for more information.

Column Name	Data Type	Description
TABLE_SCHEMA	VARCHAR(128)	The name of the Vertica Analytic Database schema that contains the table containing this column

HCATALOG_SCHEMA	VARCHAR(128)	The name of the Hive schema or database that contains the table containing this column
TABLE_NAME	VARCHAR(128)	The name of the table that contains the column
IS_PARTITION_COLUMN	BOOLEAN	Whether the table is partitioned on this column
COLUMN_NAME	VARCHAR(128)	The name of the column
HCATALOG_DATA_TYPE	VARCHAR(128)	The Hive data type of this column
DATA_TYPE	VARCHAR(128)	The Vertica Analytic Database data type of this column
DATA_TYPE_ID	INTEGER	Numeric ID of the column's Vertica Analytic Database data type
DATA_TYPE_LENGTH	INTEGER	The number of bytes used to store this data type
CHARACTER_MAXIMUM_LENGTH	INTEGER	For string data types, the maximum number of characters it can hold
NUMERIC_PRECISION	INTEGER	For numeric types, the precision of the values in the column
NUMERIC_SCALE	INTEGER	For numeric data types, the scale of the values in the column
DATETIME_PRECISION	INTEGER	For datetime data types, the precision of the values in the column
INTERVAL_PRECISION	INTEGER	For interval data types, the precision of the values in the column
ORDINAL_POSITION	INTEGER	The position of the column within the table

Privileges

No explicit permissions are required; however, users see only the records that correspond to schemas they have permissions to access.

Notes

If you are using WebHCat instead of HiveServer2, querying this table results in one web service call to the WebHCat server for each table in each HCatalog schema. If you need to perform multiple queries on this table in a short period of time, consider creating a copy of the table using a CREATE TABLE AS statement to improve performance. The copy does not reflect any changes made to the schema of the Hive tables after it was created, but it is much faster to query.

Examples

The following example demonstrates finding the column information for a specific table:

```
=> SELECT * FROM HCATALOG_COLUMNS WHERE table_name = 'hcatalogtypes'
-> ORDER BY ordinal_position;
-[ RECORD 1 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | intcol
hcatalog_data_type       | int
data_type                | int
data_type_id             | 6
data_type_length         | 8
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 1
-[ RECORD 2 ]------------+-----------------
table  schema            | hcat
```

```
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | floatcol
hcatalog_data_type       | float
data_type                | float
data_type_id             | 7
data_type_length         | 8
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 2
-[ RECORD 3 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | doublecol
hcatalog_data_type       | double
data_type                | float
data_type_id             | 7
data_type_length         | 8
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 3
-[ RECORD 4 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | charcol
hcatalog_data_type       | string
data_type                | varchar(65000)
data_type_id             | 9
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 4
-[ RECORD 5 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | varcharcol
hcatalog_data_type       | string
data_type                | varchar(65000)
data_type_id             | 9
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
```

```
ordinal_position         | 5
-[ RECORD 6 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | boolcol
hcatalog_data_type       | boolean
data_type                | boolean
data_type_id             | 5
data_type_length         | 1
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 6
-[ RECORD 7 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | timestampcol
hcatalog_data_type       | string
data_type                | varchar(65000)
data_type_id             | 9
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 7
-[ RECORD 8 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | varbincol
hcatalog_data_type       | binary
data_type                | varbinary(65000)
data_type_id             | 17
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 8
-[ RECORD 9 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | bincol
hcatalog_data_type       | binary
data_type                | varbinary(65000)
data_type_id             | 17
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
```

```
numeric_scale          |
datetime_precision     |
interval_precision     |
ordinal_position       | 9
```

See also

- HCATALOG_SCHEMATA
- HCATALOG_TABLES
- HCATALOG_TABLE_LIST

HCATALOG_SCHEMATA

Lists all of the schemas defined using the HCatalog Connector. See Using the HCatalog Connector .

Unlike other HCatalog Connector-related system tables, this table makes no calls to Hive, so querying incurs very little overhead.

Column Name	Data Type	Description
SCHEMA_ID	INTEGER	The Vertica Analytic Database ID number for the schema
SCHEMA_NAME	VARCHAR(128)	The name of the schema defined in the Vertica Analytic Database catalog
SCHEMA_OWNER_ID	INTEGER	The ID number of the user who owns the Vertica Analytic Database schema
SCHEMA_OWNER	VARCHAR(128)	The username of the Vertica Analytic Database schema's owner
CREATE_TIME	TIMESTAMPTZ	The date and time the schema as created
HOSTNAME	VARCHAR(128)	The host name or IP address of the database server that holds the Hive metadata
PORT	INTEGER	The port number on which the metastore database listens for connections
HIVESERVER2_HOSTNAME	VARCHAR(128)	The host name or IP address of the HiveServer2 server for the Hive database
WEBSERVICE_HOSTNAME	VARCHAR(128)	The host name or IP address of the WebHCat server for the Hive database, if used
WEBSERVICE_PORT	INTEGER	The port number on which the WebHCat server listens for connections
WEBHDFS_ADDRESS	VARCHAR (128)	The host and port ("host:port") for the WebHDFS service, used for reading ORC and Parquet files
HCATALOG_SCHEMA_NAME	VARCHAR(128)	The name of the schema or database in Hive to which the Vertica Analytic Database schema is mapped/
HCATALOG_USER_NAME	VARCHAR(128)	The username the HCatalog Connector uses to authenticate itself to the Hive database.
HCATALOG_CONNECTION_TIMEOUT	INTEGER	The number of seconds the HCatalog Connector waits for a successful connection to the HiveServer or WebHCat server. A value of 0 means wait indefinitely.
HCATALOG_SLOW_TRANSFER_LIMIT	INTEGER	The lowest data transfer rate (in bytes per second) from the HiveServer2 or WebHCat server that the HCatalog Connector accepts.
HCATALOG_SLOW_TRANSFER_TIME	INTEGER	The number of seconds the HCatalog Connector waits before enforcing the data transfer rate lower limit by breaking the connection and terminating the query.
SSL_CONFIG	VARCHAR(128)	The path of the Hadoop ssl-client.xml configuration file, if using HiveServer2 with SSL wire encryption.
CUSTOM_PARTITIONS	BOOLEAN	Whether the Hive schema uses custom partition locations.

Privileges

No explicit permissions are required; however, users see only the records that correspond to schemas they have permissions to access.

See also

- HCATALOG_COLUMNS
- HCATALOG_TABLE_LIST
- HCATALOG_TABLES

HCATALOG_TABLE_LIST

A concise list of all tables contained in all Hive schemas and databases available through the HCatalog Connector. See Using the HCatalog Connector .

Column Name	Data Type	Description
TABLE_SCHEMA_ID	INTEGER	Internal ID number for the schema containing the table
TABLE_SCHEMA	VARCHAR(128)	Name of the Vertica Analytic Database schema through which the table is available
HCATALOG_SCHEMA	VARCHAR(128)	Name of the Hive schema or database containing the table
TABLE_NAME	VARCHAR(128)	The name of the table
HCATALOG_USER_NAME	VARCHAR(128)	Name of Hive user used to access the table

Privileges

No explicit permissions are required; however, users see only the records that correspond to schemas they have permissions to access.

Notes

- Querying this table results in one call to HiveServer2 for each Hive schema defined using the HCatalog Connector. This means that the query usually takes longer than querying other system tables.
- Querying this table is faster than querying HCATALOG_TABLES. Querying HCATALOG_TABLE_LIST only makes one HiveServer2 call per HCatalog schema versus one call per table for HCATALOG_TABLES.

Examples

The following example demonstrates defining a new HCatalog schema then querying HCATALOG_TABLE_LIST. Note that one table defined in a different HCatalog schema also appears. HCATALOG_TABLE_LIST lists all of the tables available in any of the HCatalog schemas:

```
=> CREATE HCATALOG SCHEMA hcat WITH hostname='hcathost'
-> HCATALOG_SCHEMA='default' HCATALOG_DB='default' HCATALOG_USER='hcatuser';
CREATE SCHEMA
=> \x
Expanded display is on.
=> SELECT * FROM v_catalog.hcatalog_table_list;
-[ RECORD 1 ]------+------------------
table_schema_id    | 45035996273748980
table_schema       | hcat
hcatalog_schema    | default
table_name         | weblogs
hcatalog_user_name | hcatuser
-[ RECORD 2 ]------+------------------
table_schema_id    | 45035996273748980
table_schema       | hcat
hcatalog_schema    | default
table_name         | tweets
hcatalog_user_name | hcatuser
-[ RECORD 3 ]------+------------------
table_schema_id    | 45035996273748980
table_schema       | hcat
hcatalog_schema    | default
table_name         | messages
hcatalog_user_name | hcatuser
-[ RECORD 4 ]------+------------------
table_schema_id    | 45035996273864948
table_schema       | hiveschema
hcatalog_schema    | default
table_name         | weblogs
hcatalog_user_name | hcatuser
```

See also

- HCATALOG_COLUMNS
- HCATALOG_SCHEMATA
- HCATALOG_TABLES

HCATALOG_TABLES

Returns a detailed list of all tables made available through the HCatalog Connector. See Using the HCatalog Connector .

Column Name	Data Type	Description
TABLE_SCHEMA_ID	INTEGER	ID number of the schema
TABLE_SCHEMA	VARCHAR(128)	The name of the Vertica Analytic Database schema through which the table is available
HCATALOG_SCHEMA	VARCHAR(128)	The name of the Hive schema or database that contains the table
TABLE_NAME	VARCHAR(128)	The name of the table
HCATALOG_USER_NAME	VARCHAR(128)	The name of the HCatalog user whose credentials are used to access the table's data
MIN_FILE_SIZE_BYTES	INTEGER	The file size of the table's smallest data file, if using WebHCat; null if using HiveServer2
TOTAL_NUMBER_FILES	INTEGER	The number of files used to store this table's data in HDFS
LOCATION	VARCHAR(8192)	The URI for the directory containing this table's data, normally an HDFS URI
LAST_UPDATE_TIME	TIMESTAMPTZ	The last time data in this table was updated, if using WebHCat; null if using HiveServer2

OUTPUT_FORMAT	VARCHAR(128)	The Hive SerDe class used to output data from this table
LAST_ACCESS_TIME	TIMESTAMPTZ	The last time data in this table was accessed, if using WebHCat; null if using HiveServer2
MAX_FILE_SIZE_BYTES	INTEGER	The size of the largest data file for this table, if using WebHCat; null if using HiveServer2
IS_PARTITIONED	BOOLEAN	Whether this table is partitioned
PARTITION_EXPRESSION	VARCHAR(128)	The expression used to partition this table
TABLE_OWNER	VARCHAR(128)	The Hive user that owns this table in the Hive database, if using WebHCat; null if using HiveServer2
INPUT_FORMAT	VARCHAR(128)	The SerDe class used to read the data from this table
TOTAL_FILE_SIZE_BYTES	INTEGER	Total number of bytes used by all of this table's data files
HCATALOG_GROUP	VARCHAR(128)	The permission group assigned to this table, if using WebHCat; null if using HiveServer2
PERMISSION	VARCHAR(128)	The Unix file permissions for this group, as shown by the ls -l command, if using WebHCat; null if using HiveServer2

Privileges

No explicit permissions are required; however, users see only the records that correspond to schemas they have permissions to access.

See also

- HCATALOG_SCHEMATA
- HCATALOG_COLUMNS
- HCATALOG_TABLE_LIST

ICEBERG_COLUMNS

Provides column information for Iceberg external tables . The information in this table is drawn from the Iceberg metadata files at query time.

Column Name	Data Type	Description
TABLE_ID	INTEGER	Catalog-assigned numeric value that uniquely identifies the table.
TABLE_SCHEMA	VARCHAR	Name of the table's schema.
TABLE_NAME	VARCHAR	Name of the table containing the column.
COLUMN_ID	VARCHAR	Catalog-assigned VARCHAR value that uniquely identifies a table column.
COLUMN_NAME	VARCHAR	Name of the column.
DATA_TYPE	VARCHAR	Column data type.
DATA_TYPE_ID	INTEGER	Catalog-assigned unique numeric ID of the data type.
DATA_TYPE_LENGTH	INTEGER	Maximum allowable length of the data type.
CHARACTER_MAXIMUM_LENGTH	VARCHAR	Maximum allowable length of a VARCHAR column.
NUMERIC_PRECISION	INTEGER	Number of significant decimal digits for a NUMERIC column.
NUMERIC_SCALE	INTEGER	Number of fractional digits for a NUMERIC column.

DATETIME_PRECISION	INTEGER	Declared precision for a TIMESTAMP column, or NULL if no precision was declared.
INTERVAL_PRECISION	INTEGER	Number of fractional digits retained in the seconds field of an INTERVAL column.
IS_NULLABLE	BOOLEAN	Whether the column can contain NULL values.
WRITE_DEFAULT	VARCHAR	Field value for any records written after the field was added to the schema, if the writer does not supply the field's value.
INITIAL_DEFAULT	VARCHAR	Field value for all records that were written before the field was added to the schema.

INHERITED_PRIVILEGES

Provides summary information about privileges inherited by tables and views from GRANT statements on parent schemas, excluding inherited grant options .

For information about explicitly granted permissions, see system table GRANTS .

> **Note**
> Inherited privileges are not displayed if privilege inheritance is disabled at the database level.

Column Name	Data Type	Description
OBJECT_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theobject inheriting the privileges.
SCHEMA_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theparent schema.
OBJECT_SCHEMA	VARCHAR	Name of the parent schema of a table or view.
OBJECT_NAME	VARCHAR	Name of the table or view.
OBJECT_TYPE	VARCHAR	Table or view.
PRIVILEGES_DESCRIPTION	VARCHAR	Lists the privileges inherited on an object. An asterisk (*) appended to a privilege indicates that the user can grant the privilege to other users by granting the privilege on the parent schema.
PRINCIPAL	VARCHAR	Name of the role or user inheriting the privileges in the row.
PRINCIPAL_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theuser inheriting the privileges.
GRANTOR	VARCHAR	User that granted the privileges on the parent schema to the principal.
GRANTOR_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theuser who performed the grant operation.
GRANT_ID	INTEGER	Catalog-assigned integer value that uniquely identifies thegrant operation.

Examples

The following query returns the privileges that the tables and views inherit from their parent schema, customers.

```
=> SELECT object_schema,object_name,object_type,privileges_description,principal,grantor FROM inherited_privileges WHERE
object_schema='customers';
 object_schema |  object_name  | object_type |                    privileges_description                    | principal | grantor
---------------+---------------+-------------+--------------------------------------------------------------+-----------+---------
 customers     | cust_info     | Table       | INSERT, SELECT, UPDATE, DELETE, ALTER, REFERENCES, DROP, TRUNCATE | dbadmin   | dbadmin
 customers     | shipping_info | Table       | INSERT, SELECT, UPDATE, DELETE, ALTER, REFERENCES, DROP, TRUNCATE | dbadmin   | dbadmin
 customers     | cust_set      | View        | SELECT, ALTER, DROP                                          | dbadmin   | dbadmin
 customers     | cust_info     | Table       | SELECT                                                       | Val       | dbadmin
 customers     | shipping_info | Table       | SELECT                                                       | Val       | dbadmin
 customers     | cust_set      | View        | SELECT                                                       | Val       | dbadmin
 customers     | cust_info     | Table       | INSERT                                                       | Pooja     | dbadmin
 customers     | shipping_info | Table       | INSERT                                                       | Pooja     | dbadmin
(8 rows)
```

See also

- INHERITING_OBJECTS
- Database users and privileges
- GET_PRIVILEGES_DESCRIPTION

INHERITING_OBJECTS

Provides information about which tables and views inherit privileges from which schemas.

For information about the specific privileges inherited from schemas and their associated GRANT statements, see the INHERITED_PRIVILEGES table.

Column Name	Data Type	Description
OBJECT_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theobject inheriting the privileges.
SCHEMA_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theparent schema.
OBJECT_SCHEMA	VARCHAR	Name of the parent schema of a table or view.
OBJECT_NAME	VARCHAR	Name of the table or view.
OBJECT_TYPE	VARCHAR	Table or view.

Examples

The following query returns the tables and views that inherit their privileges from their parent schema, customers.

```
=> SELECT * FROM inheriting_objects WHERE object_schema='customers';
    object_id     |     schema_id     | object_schema |  object_name  | object_type
------------------+-------------------+---------------+---------------+-------------
 45035996273980908 | 45035996273980902 | customers     | cust_info     | table
 45035996273980984 | 45035996273980902 | customers     | shipping_info | table
 45035996273980980 | 45035996273980902 | customers     | cust_set      | view
(3 rows)
```

See also

- INHERITED_PRIVILEGES
- Inherited privileges
- Database users and privileges
- GET_PRIVILEGES_DESCRIPTION

KEYWORDS

Identifies Vertica reserved and non-reserved keywords.

Column Name	Data Type	Description
KEYWORD	VARCHAR	Vertica-reserved or non-reserved keyword.

RESERVED	VARCHAR	Indicates whether a keyword is reserved or non-reserved: • R: reserved • N: non-reserved

Examples

The following query gets all reserved keywords that begin with B:

```
=> SELECT * FROM keywords WHERE reserved = 'R' AND keyword ilike 'B%';
 keyword | reserved
---------+----------
 BETWEEN | R
 BIGINT  | R
 BINARY  | R
 BIT     | R
 BOOLEAN | R
 BOTH    | R
(6 rows)
```

See also

Keywords

LARGE_CLUSTER_CONFIGURATION_STATUS

Shows the current cluster nodes and control node (spread hosts) designations in the Catalog so you can see if they match.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The name of the node in the cluster.
SPREAD_HOST_NAME	VARCHAR	The host name of the control node (the host that manages control message responsibilities)
CONTROL_NODE_NAME	VARCHAR	The name of the control node

See also

Large Cluster

LICENSE_AUDITS

Lists the results of Vertica's license automatic compliance audits. See How Vertica Calculates Database Size .

Column Name	Data Type	Description
DATABASE_SIZE_BYTES	INTEGER	The estimated raw data size of the database
LICENSE_SIZE_BYTES	INTEGER	The licensed data allowance
USAGE_PERCENT	FLOAT	Percentage of the licensed allowance used
AUDIT_START_TIMESTAMP	TIMESTAMPTZ	When the audit started
AUDIT_END_TIMESTAMP	TIMESTAMPTZ	When the audit finished
CONFIDENCE_LEVEL_PERCENT	FLOAT	The confidence level of the size estimate
ERROR_TOLERANCE_PERCENT	FLOAT	The error tolerance used for the size estimate
USED_SAMPLING	BOOLEAN	Whether data was randomly sampled (if false, all of the data was analyzed)

CONFIDENCE_INTERVAL_LOWER_BOUND_BYTES	INTEGER	The lower bound of the data size estimate within the confidence level
CONFIDENCE_INTERVAL_UPPER_BOUND_BYTES	INTEGER	The upper bound of the data size estimate within the confidence level
SAMPLE_COUNT	INTEGER	The number of data samples used to generate the estimate
CELL_COUNT	INTEGER	The number of cells in the database
AUDITED_DATA	VARCHAR	The type of data audited, which includes regular (non-flex), flex, external, and total data

LICENSES

For all licenses, provides information on license types, the dates for which licenses are valid, and the limits the licenses impose.

Column Name	Data Type	Description
LICENSE_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the license.
NAME	VARCHAR	The license's name. (The license name in this column could be represented by a long license key.)
LICENSEE	VARCHAR	The entity to which the product is licensed.
START_DATE	VARCHAR	The start date for which the license is valid.
END_DATE	VARCHAR	The end date until which the license is valid (or "Perpetual" if the license has no expiration).
LICENSETYPE	VARCHAR	The type of the license (for example, Premium Edition).
PARENT	VARCHAR	The parent license (field is blank if there is no parent).
SIZE	VARCHAR	The size limit for data on the license.
IS_SIZE_LIMIT_ENFORCED	BOOLEAN	Indicates whether the license includes enforcement of data and node limits, where t is true and f is false.
NODE_RESTRICTION	VARCHAR	The node limit the license imposes.
CONFIGURED_ID	INTEGER	A long license key.

LOAD_BALANCE_GROUPS

Lists the objects contained by all load balance groups. Each row in this table represents a single object that is a member of a load balance group. If a load balance group does not contain any objects, it appears once in this table with its type column set to 'Empty Group.'

Column Name	Data Type	Description
NAME	VARCHAR	The name of the load balance group
POLICY	VARCHAR	The policy that sets how the group chooses the node for a connection. Contains one of the following: • ROUNDROBIN • RANDOM • NONE

FILTER	VARCHAR	The IP address range in CIDR format to select the members of a fault group that are included in the load balance group. This column only has a value if the TYPE column is 'Fault Group' or 'Subcluster.'
TYPE	VARCHAR	The type of object contained in the load balance group. Contains one of: • Fault Group • Subcluster • Network Address Group • Empty Group
OBJECT_NAME	VARCHAR	The name of the fault group or network address included in the load balance group. This column is NULL if the group contains no objects.

Examples

```
=> SELECT * FROM LOAD_BALANCE_GROUPS;
    name     |  policy   |  filter  |        type         | object_name
-------------+-----------+----------+---------------------+-------------
 group_1     | ROUNDROBIN |          | Network Address Group | node01
 group_1     | ROUNDROBIN |          | Network Address Group | node02
 group_2     | ROUNDROBIN |          | Empty Group           |
 group_all   | ROUNDROBIN |          | Network Address Group | node01
 group_all   | ROUNDROBIN |          | Network Address Group | node02
 group_all   | ROUNDROBIN |          | Network Address Group | node03
 group_fault_1 | RANDOM   | 0.0.0.0/0 | Fault Group          | fault_1
(7 rows)
```

See also

- NETWORK_ADDRESSES
- ALTER LOAD BALANCE GROUP
- ALTER NETWORK ADDRESS
- ALTER ROUTING RULE
- CREATE LOAD BALANCE GROUP

LOG_PARAMS

Provides summary information about changes to configuration parameters related to authentication and security run in your database.

Column Name	Data Type	Description
ISSUED_TIME	VARCHAR	The time at which the query was executed.
USER_NAME	VARCHAR	Name of the user who issued the query at the time Vertica recorded the session.
USER_ID	INTEGER	Numeric representation of the user who ran the query.
HOSTNAME	VARCHAR	The hostname, IP address, or URL of the database server.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
AUDIT_TYPE	VARCHAR	The type of operation for the audit, in this case, Parameter.
AUDIT_TAG_NAME	VARCHAR	The tag for the specific parameter.
REQUEST_TYPE	VARCHAR	The type of query request.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session.

SUBJECT	VARCHAR	The new value of the parameter.
REQUEST	VARCHAR	Lists the query request.
SUCCESS	VARCHAR	Indicates whether or not the operation was successful.
CATEGORY	VARCHAR	The audit parent category, such as Authentication.

Examples

The following example queries the LOG_PARAMS system table and shows only the most recent configuration parameter for this user under the Authentication category:

```
=> SELECT * FROM log_params limit 1;

-------------------------------------------------------------------------------------------------

issued_time   | 2018-02-12 13:41:20.837452-05
user_name     | dbadmin
user_id       | 45035996273704962
hostname      | ::1:50690
session_id    | v_vmart_node0001-341751:0x13878
audit_type    | Param
audit_tag_name| SecurityAlgorithm
request_type  | UTILITY
request_id    | 8
subject       | MD5
request       | select set_config_parameter('SecurityAlgorithm','MD5',null);
success       | t
category      | Authentication
(1 row)
```

LOG_QUERIES

Provides summary information about some queries related to authentication and security run in your database.

Column Name	**Data Type**	**Description**
ISSUED_TIME	VARCHAR	The time at which the query was executed.
USER_NAME	VARCHAR	Name of the user who issued the query at the time Vertica recorded the session.
USER_ID	INTEGER	Numeric representation of the user who ran the query.
HOSTNAME	VARCHAR	The hostname, IP address, or URL of the database server.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
AUDIT_TYPE	VARCHAR	The type of operation for the audit, in this case, Query.
AUDIT_TAG_NAME	VARCHAR	The tag for the specific query.

REQUEST_TYPE	VARCHAR	The type of query request. Examples include, but are not limited to: • QUERY • DDL • LOAD • UTILITY • TRANSACTION • PREPARE • EXECUTE • SET • SHOW
REQUEST_ID	INTEGER	The ID of the query request.
SUBJECT	VARCHAR	The subject of the query.
REQUEST	VARCHAR	Lists the query request.
SUCCESS	VARCHAR	Indicates whether or not the operation was successful.
CATEGORY	VARCHAR	The audit parent category, such as Managing_Users_Privileges.

Examples

The following example queries the LOG_QUERIES system table and shows only the most recent query for this user under the Managing_Users_Privileges category:

```
=> SELECT * FROM log_queries limit 1;
-------------------------------------------------------------------------------

issued_time   | 2018-01-22 10:36:55.634349-05
user_name     | dbadmin
user_id       | 45035996273704962
hostname      |
session_id    | v_vmart_node0001-237210:0x37e1d
audit_type    | Query
audit_tag_name| REVOKE ROLE
request_type  | DDL
request_id    | 2
subject       |
request       | revoke all privileges from Joe;
success       | f
category      | Managing_Users_Privileges
(1 row)
```

LOG_TABLES

Provides summary information about queries on system tables.

Column Name	**Data Type**	**Description**
ISSUED_TIME	VARCHAR	Time of query execution.
USER_NAME	VARCHAR	Name of user who issued the query at the time Vertica recorded the session.
USER_ID	INTEGER	Numeric representation of the user who ran the query.
HOSTNAME	VARCHAR	The hostname, IP address, or URL of the database server.

SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
AUDIT_TYPE	VARCHAR	The type of operation for the audit, in this case, Table.
AUDIT_TAG_NAME	VARCHAR	The tag for the specific table.
REQUEST_TYPE	VARCHAR	The type of query request. In this case, QUERY.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session.
SUBJECT	VARCHAR	The name of the table that was queried.
REQUEST	VARCHAR	Lists the query request.
SUCCESS	VARCHAR	Indicates whether or not the operation was successful.
CATEGORY	VARCHAR	The audit parent category—for example, Views , Security , and Managing_Users_Privileges .

Examples

The following example shows recent queries on configuration parameters:

```
dbadmin=> SELECT issued_time, audit_type, request_type, subject, request, category FROM log_tables
   WHERE category ilike '%Managing_Config_Parameters%' ORDER BY issued_time DESC LIMIT 4;
-[ RECORD 1 ]+----------------------------------------------------------------------------------
issued_time  | 2020-05-14 14:14:53.453552-04
audit_type   | Table
request_type | QUERY
subject      | vs_nodes
request      | SELECT * from vs_nodes order by name  limit 1;
category     | Managing_Config_Parameters
-[ RECORD 2 ]+----------------------------------------------------------------------------------
issued_time  | 2020-05-14 14:14:27.546474-04
audit_type   | Table
request_type | QUERY
subject      | vs_nodes
request      | SELECT * from vs_nodes order by name ;
category     | Managing_Config_Parameters
-[ RECORD 3 ]+----------------------------------------------------------------------------------
issued_time  | 2020-05-14 08:54:32.86881-04
audit_type   | Table
request_type | QUERY
subject      | vs_parameters_mismatch
request      | select * from configuration_parameters where parameter_name = 'MaxDepotSizePercent';
category     | Managing_Config_Parameters
-[ RECORD 4 ]+----------------------------------------------------------------------------------
issued_time  | 2020-05-14 08:54:32.86881-04
audit_type   | Table
request_type | QUERY
subject      | vs_nodes
request      | select * from configuration_parameters where parameter_name = 'MaxDepotSizePercent';
category     | Managing_Config_Parameters
```

MATERIALIZE_FLEXTABLE_COLUMNS_RESULTS

Contains the results of running the MATERIALIZE_FLEXTABLE_COLUMNS function. The table contains information about keys that the function evaluated. It does not contain information about all keys.

Column Name	**Data Type**	**Description**

TABLE_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the table.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.
TABLE_NAME	VARCHAR	The table name for which information is listed.
CREATION_TIME	VARCHAR	Timestamp when the key was materialized.
KEY_NAME	VARCHAR	Name of the key from the VMap column that was materialized.
STATUS	VARCHAR	Status of the materialized column, one of the following: • ADDED • EXISTS • ERROR
MESSAGE	BOOLEAN	Message associated with the status in the previous column, one of the following: • Added successfully • Column of same name already exists in table definition • Add operation failed • No data type guess provided to add column

Examples

```
=> \x
Expanded display is on.
=> SELECT table_id, table_schema, table_name, key_name, status, message FROM MATERIALIZE_FLEXTABLE_COLUMNS_RESULTS
WHERE table_name = 'mountains_hybrid';
-[ RECORD 1 ]+-------------------------------------------------
table_id     | 45035996273708192
table_schema | public
table_name   | mountains_hybrid
key_name     | type
status       | ADDED
message      | Added successfully
-[ RECORD 2 ]+-------------------------------------------------
table_id     | 45035996273708192
table_schema | public
table_name   | mountains_hybrid
key_name     | height
status       | ADDED
message      | Added successfully
-[ RECORD 3 ]+-------------------------------------------------
table_id     | 45035996273708192
table_schema | public
table_name   | mountains_hybrid
key_name     | name
status       | EXISTS
message      | Column of same name already exists in table definition
```

MODELS

Lists details about the machine-learning models in the database.

Column Name	Data Type	Description
MODEL_ID	INTEGER	The model's internal ID.

MODEL_NAME	VARCHAR(128)	The name of the model.
SCHEMA_ID	INTEGER	The schema's internal ID.
SCHEMA_NAME	VARCHAR(128)	The name of the schema.
OWNER_ID	INTEGER	The model owner's ID.
OWNER_NAME	VARCHAR(128)	The user who created the model.
CATEGORY	VARCHAR(128)	The type of model. By default, models created in Vertica are assigned to the Vertica_Models category.
MODEL_TYPE	VARCHAR(128)	The type of algorithm used to create the model.
IS_COMPLETE	VARCHAR(128)	Denotes whether the model is complete and ready for use in machine learning functions. This field is usually false when the model is being trained. Once the training is complete, the field is set to true.
CREATE_TIME	TIMESTAMPTZ	The time the model was created.
SIZE	INTEGER	The size of the model in bytes.

Examples

```
=> SELECT * FROM models;
-[ RECORD 1 ]----------------------------
model_id    | 45035996273714020
model_name  | myLinearRegModel
schema_id   | 45035996273704980
schema_name | public
owner_id    | 45035996273704962
owner_name  | dbadmin
category    | VERTICA_MODELS
model_type  | LINEAR_REGRESSION
is_complete | t
create_time | 2018-01-22 11:13:35.018412-05
size        | 1671
```

NETWORK_ADDRESSES

Lists information about the network addresses defined in your database using the CREATE NETWORK ADDRESS statement.

Column Name	Data Type	Description
NAME	VARCHAR	The name of the network address.
NODE	VARCHAR	The name of the node that owns the network address.
ADDRESS	VARCHAR	The network address's IP address. This address can be either in IPv4 or IPv6 format.
PORT	INT	The network address's port number.
ADDRESS_FAMILY	VARCHAR	The format of the network address's IP address. This values is either 'ipv4' or 'ipv6'.
IS_ENABLED	BOOLEAN	Whether the network address is enabled. You can disable network addresses to prevent their use. If the address is disabled, the value in this column is False.
IS_AUTO_DETECTED	BOOLEAN	Whether Vertica created the network address automatically.

Examples

```
=> \x
Expanded display is on.

=> SELECT * FROM v_catalog.network_addresses;
-[ RECORD 1 ]----+-----------------
name             | node01
node             | v_vmart_node0001
address          | 10.20.100.247
port             | 5433
address_family   | ipv4
is_enabled       | t
is_auto_detected | f
-[ RECORD 2 ]----+-----------------
name             | node02
node             | v_vmart_node0002
address          | 10.20.100.248
port             | 5433
address_family   | ipv4
is_enabled       | t
is_auto_detected | f
-[ RECORD 3 ]----+-----------------
name             | node03
node             | v_vmart_node0003
address          | 10.20.100.249
port             | 5433
address_family   | ipv4
is_enabled       | t
is_auto_detected | f
```

See also

- LOAD_BALANCE_GROUPS
- ALTER LOAD BALANCE GROUP
- ALTER NETWORK ADDRESS
- ALTER ROUTING RULE
- CREATE LOAD BALANCE GROUP

NODE_SUBSCRIPTION_CHANGE_PHASES

In an Eon Mode database, stores information about changes to node's shard subscriptions.

Column Name	Data Type	Description
node_name	VARCHAR	Name of the node
subscription_change_type	VARCHAR	The change being made to the subscription
session_id	INTEGER	ID of the session in which the change was initiated
transaction_id	INTEGER	ID of the transaction in which the change was initiated
user_id	INTEGER	ID of user that initiated the change
user_name	VARCHAR	Name of user that initiated the change
subscription_oid	INTEGER	Session object ID
subscriber_node_oid	INTEGER	Object ID of node that requested the subscription
subscriber_node_name	VARCHAR	Name of the node that requested the subscription

shard_oid	INTEGER	Object ID of the shard to which the node is subscribed
shard_name	VARCHAR	Name of the shard to which the node is subscribed
min_time	TIMESTAMPTZ	Start time of the subscription change
max_time	TIMESTAMPTZ	Completion time of the subscription change
source_node_oid	INTEGER	Object ID of the node from which catalog objects were fetched
source_node_name	VARCHAR	Name of the node from which catalog objects were fetched
num_objs_affected	INTEGER	Number of catalog objects affected by the subscription change
action	VARCHAR	Description of the action taken
new_content_size	INTEGER	Total size of the catalog objects that were fetched for the subscription change
phase_limit_reached	BOOLEAN	Reached maximum number of retries?
START_TIME	TIMESTAMPTZ	When the subscription change started
END_TIME	TIMESTAMPTZ	When the subscription change was finished
retried	BOOLEAN	Retry of subscription phase?
phase_result	VARCHAR	Outcome of the subscription change, one of the following: • Success • Failure

Examples

```
=> SELECT NODE_NAME, SUBSCRIPTION_CHANGE_TYPE, SHARD_NAME,
   ACTION  FROM node_subscription_change_phases
   ORDER BY start_time ASC LIMIT 10;

     NODE_NAME       | SUBSCRIPTION_CHANGE_TYPE | SHARD_NAME  |         ACTION
---------------------+--------------------------+-------------+------------------------
 v_verticadb_node0001 | CREATE SUBSCRIPTION      | segment0007 | COLLECT SHARD METADATA
 v_verticadb_node0001 | CREATE SUBSCRIPTION      | segment0010 | COLLECT SHARD METADATA
 v_verticadb_node0001 | CREATE SUBSCRIPTION      | segment0004 | COLLECT SHARD METADATA
 v_verticadb_node0001 | CREATE SUBSCRIPTION      | segment0005 | COLLECT SHARD METADATA
 v_verticadb_node0001 | CREATE SUBSCRIPTION      | replica     | COLLECT SHARD METADATA
 v_verticadb_node0001 | CREATE SUBSCRIPTION      | segment0005 | COLLECT SHARD METADATA
 v_verticadb_node0001 | CREATE SUBSCRIPTION      | segment0006 | COLLECT SHARD METADATA
 v_verticadb_node0001 | CREATE SUBSCRIPTION      | segment0008 | COLLECT SHARD METADATA
 v_verticadb_node0001 | CREATE SUBSCRIPTION      | segment0011 | COLLECT SHARD METADATA
 v_verticadb_node0001 | CREATE SUBSCRIPTION      | segment0002 | COLLECT SHARD METADATA
```

NODE_SUBSCRIPTIONS

Eon Mode only

Lists information about database node subscriptions to shards.

Column Name	**Data Type**	**Description**
SUBSCRIPTION_OID	INTEGER	Subscription OID

NODE_OID	INTEGER	Subscribed node OID
NODE_NAME	VARCHAR	Name of the node
SHARD_OID	INTEGER	OID of the shard to which the node is subscribed
SHARD_NAME	VARCHAR	Name of the shard to which the node is subscribed
SUBSCRIPTION_STATE	VARCHAR	Node's current subscription state
FROM _VERSION	INTEGER	Deprecated
IS_PRIMARY	BOOLEAN	Specifies whether the node is currently the primary subscriber .
IS_RESUBSCRIBING	BOOLEAN	Indicates whether a subscription is resubscribing to a node: • t (true): A subscription is resubscribing, only applies to PENDING subscriptions created during the cluster or node startup. • f (false): A subscription is not resubscribing, applies to PENDING subscriptions created with REBALANCE_SHARDS that transitioned to an ACTIVE state.
CREATOR_TID	INTEGER	ID of transaction that created this subscription
SUBSCRIBED_TO_METADATA_AT	INTEGER	Deprecated
IS_PARTICIPATING_PRIMARY	BOOLEAN	Whether this node is the participating primary subscriber for the shard. If true, the node listed in NODE_NAME is the only one that reads from and writes to communal storage for this shard in the subcluster. Other nodes in the subcluster that subscribe to the same shard receive data from this node via peer-to-peer transfers.

Examples

The following example queries the NODE_SUBSCRIPTIONS table in a database with two three-node subclusters (a primary and a secondary) in a 12-shard database.

```
=> SELECT node_name, shard_name, subscription_state, is_primary,
          is_participating_primary AS is_p_primary
       FROM NODE_SUBSCRIPTIONS ORDER BY node_name, shard_name;

      node_name       | shard_name  | subscription_state | is_primary | is_p_primary
----------------------+-------------+--------------------+------------+--------------
 v_verticadb_node0001 | replica     | ACTIVE             | t          | t
 v_verticadb_node0001 | segment0001 | ACTIVE             | t          | t
 v_verticadb_node0001 | segment0003 | ACTIVE             | f          | f
 v_verticadb_node0001 | segment0004 | ACTIVE             | t          | t
 v_verticadb_node0001 | segment0006 | ACTIVE             | f          | f
 v_verticadb_node0001 | segment0007 | ACTIVE             | t          | t
 v_verticadb_node0001 | segment0009 | ACTIVE             | f          | f
 v_verticadb_node0001 | segment0010 | ACTIVE             | t          | t
 v_verticadb_node0001 | segment0012 | ACTIVE             | f          | f
 v_verticadb_node0002 | replica     | ACTIVE             | f          | t
 v_verticadb_node0002 | segment0001 | ACTIVE             | f          | f
 v_verticadb_node0002 | segment0002 | ACTIVE             | t          | t
 v_verticadb_node0002 | segment0004 | ACTIVE             | f          | f
 v_verticadb_node0002 | segment0005 | ACTIVE             | t          | t
 v_verticadb_node0002 | segment0007 | ACTIVE             | f          | f
 v_verticadb_node0002 | segment0008 | ACTIVE             | t          | t
 v_verticadb_node0002 | segment0010 | ACTIVE             | f          | f
 v_verticadb_node0002 | segment0011 | ACTIVE             | t          | t
 v_verticadb_node0003 | replica     | ACTIVE             | f          | t
 v_verticadb_node0003 | segment0002 | ACTIVE             | f          | f
 v_verticadb_node0003 | segment0003 | ACTIVE             | t          | t
 v_verticadb_node0003 | segment0005 | ACTIVE             | f          | f
 v_verticadb_node0003 | segment0006 | ACTIVE             | t          | t
 v_verticadb_node0003 | segment0008 | ACTIVE             | f          | f
 v_verticadb_node0003 | segment0009 | ACTIVE             | t          | t
 v_verticadb_node0003 | segment0011 | ACTIVE             | f          | f
 v_verticadb_node0003 | segment0012 | ACTIVE             | t          | t
 v_verticadb_node0004 | replica     | ACTIVE             | f          | t
 v_verticadb_node0004 | segment0001 | ACTIVE             | f          | t
 v_verticadb_node0004 | segment0003 | ACTIVE             | f          | f
 v_verticadb_node0004 | segment0004 | ACTIVE             | f          | t
 v_verticadb_node0004 | segment0006 | ACTIVE             | f          | f
 v_verticadb_node0004 | segment0007 | ACTIVE             | f          | t
 v_verticadb_node0004 | segment0009 | ACTIVE             | f          | f
 v_verticadb_node0004 | segment0010 | ACTIVE             | f          | t
 v_verticadb_node0004 | segment0012 | ACTIVE             | f          | f
 v_verticadb_node0005 | replica     | ACTIVE             | f          | t
 v_verticadb_node0005 | segment0001 | ACTIVE             | f          | f
 v_verticadb_node0005 | segment0002 | ACTIVE             | f          | t
 v_verticadb_node0005 | segment0004 | ACTIVE             | f          | f
 v_verticadb_node0005 | segment0005 | ACTIVE             | f          | t
 v_verticadb_node0005 | segment0007 | ACTIVE             | f          | f
 v_verticadb_node0005 | segment0008 | ACTIVE             | f          | t
 v_verticadb_node0005 | segment0010 | ACTIVE             | f          | f
 v_verticadb_node0005 | segment0011 | ACTIVE             | f          | t
 v_verticadb_node0006 | replica     | ACTIVE             | f          | t
 v_verticadb_node0006 | segment0002 | ACTIVE             | f          | f
 v_verticadb_node0006 | segment0003 | ACTIVE             | f          | t
 v_verticadb_node0006 | segment0005 | ACTIVE             | f          | f
 v_verticadb_node0006 | segment0006 | ACTIVE             | f          | t
 v_verticadb_node0006 | segment0008 | ACTIVE             | f          | f
 v_verticadb_node0006 | segment0009 | ACTIVE             | f          | t
 v_verticadb_node0006 | segment0011 | ACTIVE             | f          | f
 v_verticadb_node0006 | segment0012 | ACTIVE             | f          | t
(54 rows)
```

NODES

Lists details about the nodes in the database.

Column Name	Data Type	Description
NODE_NAME	VARCHAR(128)	The name of the node.
NODE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies the node.
NODE_STATE	VARCHAR(128)	The node's current state, one of the following: • UP • DOWN • READY • UNSAFE • SHUTDOWN • SHUTDOWN_ERROR • RECOVERING • RECOVER_ERROR • RECOVERED • INITIALIZING • STANDBY • NEEDS_CATCHUP
IS_PRIMARY	BOOLEAN	Whether the node is a primary or secondary node. Primary nodes are the only ones Vertica considers when determining the K-Safety of an Eon Mode database. The node inherits this property from the subcluster that contains it.
IS_READONLY	BOOLEAN	Whether the node is in read-only mode or not. This column is TRUE if the Eon Mode database is read-only due to the loss of quorum or primary shard coverage . See Database Read-Only Mode .
NODE_ADDRESS	VARCHAR(80)	The host address of the node.
NODE_ADDRESS_FAMILY	VARCHAR(10)	The IP Version of the node_address . For example, ipv4 .
EXPORT_ADDRESS	VARCHAR(8192)	The IP address of the node (on the public network) used for import/export operations and native load-balancing.
EXPORT_ADDRESS_FAMILY	VARCHAR(10)	The IP Version of the export_address . For example, ipv4 .
CATALOG_PATH	VARCHAR(8192)	The absolute path to the catalog on the node.
NODE_TYPE	VARCHAR(9)	The type of the node. For more information on the types of nodes, refer to Setting node type .
IS_EPHEMERAL	BOOLEAN	(Deprecated) True if this node has been marked as ephemeral. (in preparation for removing it from the cluster).
STANDING_IN_FOR	VARCHAR(128)	The name of the node that this node is currently replacing.
SUBCLUSTER_NAME	VARCHAR(128)	In an Eon Mode database, the name of the subcluster that contains the node. Nodes belong to exactly one subcluster.
SANDBOX	VARCHAR(128)	In an Eon Mode database, the name, if any, of the sandbox to which the node belongs. NULL if the node is not a member of an active sandbox.
LAST_MSG_FROM_NODE_AT	TIMESTAMPTZ	The date and time the last message was received from this node.

NODE_DOWN_SINCE	TIMESTAMPTZ	The amount of time that the replaced node has been unavailable.
BUILD_INFO	VARCHAR(128)	The version of the Vertica server binary the node is running.

Example

```
=> SELECT NODE_NAME, NODE_STATE, IS_PRIMARY, IS_READONLY, NODE_TYPE,
       SUBCLUSTER_NAME FROM NODES ORDER BY NODE_NAME ASC;
    NODE_NAME      | NODE_STATE | IS_PRIMARY | IS_READONLY | NODE_TYPE |  SUBCLUSTER_NAME
--------------------+------------+------------+-------------+-----------+-------------------
 v_verticadb_node0001 | UP       | t          | f           | PERMANENT | default_subcluster
 v_verticadb_node0002 | UP       | t          | f           | PERMANENT | default_subcluster
 v_verticadb_node0003 | UP       | t          | f           | PERMANENT | default_subcluster
 v_verticadb_node0004 | UP       | f          | f           | PERMANENT | analytics
 v_verticadb_node0005 | UP       | f          | f           | PERMANENT | analytics
 v_verticadb_node0006 | UP       | f          | f           | PERMANENT | analytics
(6 rows)
```

ODBC_COLUMNS

Provides table column information. The format is defined by the ODBC standard for the ODBC SQLColumns metadata. Details on the ODBC SQLColumns format are available in the ODBC specification: http://msdn.microsoft.com/en-us/library/windows/desktop/ms711683%28v=vs.85%29.aspx .

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	Name of the schema in which the column resides. If the column does not reside in a schema, this field is empty.
TABLE_NAME	VARCHAR	Name of the table in which the column resides.
COLUMN_NAME	VARCHAR	Name of the column.
DATA_TYPE	INTEGER	Data type of the column. This can be an ODBC SQL data type or a driver-specific SQL data type. This column corresponds to the ODBC_TYPE column in the TYPES table.
DATA_TYPE_NAME	VARCHAR	Driver-specific data type name.
COLUMN_SIZE	INTEGER	ODBC-defined data size of the column.
BUFFER_LENGTH	INTEGER	Transfer octet length of a column is the maximum number of bytes returned to the application when data is transferred to its default C data type. See http://msdn.microsoft.com/en-us/library/windows/desktop/ms713979%28v=vs.85%29.aspx
DECIMAL_DIGITS	INTEGER	Total number of significant digits to the right of the decimal point. This value has no meaning for non-decimal data types.
NUM_PREC_RADIX	INTEGER	Radix Vertica reports decimal_digits and columns_size as. This value is always 10, because it refers to a number of decimal digits, rather than a number of bits.
NULLABLE	BOOLEAN	Whether the column can contain null values.
REMARKS	VARCHAR	Textual remarks for the column.
COLUMN_DEFAULT	VARCHAR	Default value of the column.
SQL_TYPE_ID	INTEGER	SQL data type of the column.

SQL_DATETIME_SUB	VARCHAR	Subtype for a datetime data type. This value has no meaning for non-datetime data types.
CHAR_OCTET_LENGTH	INTEGER	Maximum length of a string or binary data column.
ORDINAL_POSITION	INTEGER	Position of the column in the table definition.
IS_NULLABLE	VARCHAR	Values can be YES or NO, determined by the value of the NULLABLE column.
IS_IDENTITY	BOOLEAN	Whether the column is an IDENTITY column.

PASSWORD_AUDITOR

Stores information about user accounts, account expirations, and password hashing algorithms .

Column Name	Data Type	Description
USER_ID	INTEGER	Unique ID for the user.
USER_NAME	VARCHAR	Name of the user.
ACCTEXPIRED	BOOLEAN	Indicates if the user's password expires. 'f' indicates that it does not expire. 't' indicates that it does expire.
SECURITY_ALGORITHM	VARCHAR	User-level security algorithm for hash authentication. Valid values: • 'NONE' (Default. Algorithm specified by SYSTEM_SECURITY_ALGORITHM is used.) • 'SHA512' • 'MD5'
SYSTEM_SECURITY_ALGORITHM	VARCHAR	System-level security algorithm for hash authentication. Valid values: • 'SHA512' (Default) • 'MD5'
EFFECTIVE_SECURITY_ALGORITHM	VARCHAR	The resulting security algorithm, depending on the values of SECURTY_ALGORITHM and SYSTEM_SECURITY_ALGORITHM. For details, see Password hashing algorithm .
CURRENT_SECURITY_ALGORITHM	VARCHAR	The security algorithm used to hash the user's current password. This can differ from the EFFECTIVE_SECURITY_ALGORITHM if a user hasn't reset their password since a change in the EFFECTIVE_SECURITY_ALGORITHM. Valid values: • 'NONE' (Default. Algorithm specified by SYSTEM_SECURITY_ALGORITHM is used.) • 'SHA512' • 'MD5'

PASSWORDS

Contains information on current user passwords. This table also includes information on past passwords if any Profiles have PASSWORD_REUSE_TIME or PASSWORD_REUSE_MAX parameters set. See CREATE PROFILE for details.

Column Name	Data Type	Description
USER_ID	INTEGER	The ID of the user who owns the password.
USER_NAME	VARCHAR	The name of the user who owns the password.
PASSWORD	VARCHAR	The hashed password.
PASSWORD_CREATE_TIME	DATETIME	The date and time when the password was created.
IS_CURRENT_PASSWORD	BOOLEAN	Denotes whether this is the user's current password. Non-current passwords are retained to enforce password reuse limitations.
PROFILE_ID	INTEGER	The ID number of the profile to which the user is assigned.
PROFILE_NAME	VARCHAR	The name of the profile to which the user is assigned.
PASSWORD_REUSE_MAX	VARCHAR	The number password changes that must take place before an old password can be reused.
PASSWORD_REUSE_TIME	VARCHAR	The amount of time that must pass before an old password can be reused.
SALT	VARCHAR	A hex string used to hash the password.

Examples

The following query returns the SHA-512 hashed password and salt of user 'u1'.

```
=> SELECT user_name, password, salt FROM passwords WHERE user_name='u1';
 user_name |                          password                          |               salt
-----------+------------------------------------------------------------+----------------------------------
 u1        | sha512f3f802f1c56e2530cd9c3164cc7b8002ba444c0834160f10 | f05e9d859fb441f9f612f8a787bfc872
(1 row)
```

PRIMARY_KEYS

Provides primary key information.

Column Name	Data Type	Description
CONSTRAINT_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog, which identifies the constraint.
CONSTRAINT_NAME	VARCHAR	The constraint name for which information is listed.
COLUMN_NAME	VARCHAR	The column name for which information is listed.
ORDINAL_POSITION	VARCHAR	The position of the column within the key. The numbering of columns starts at 1.
TABLE_NAME	VARCHAR	The table name for which information is listed.
CONSTRAINT_TYPE	VARCHAR	The constraint type, p , for primary key.
IS_ENABLED`	BOOLEAN	Indicates if a table column constraint for a PRIMARY KEY is enabled by default. Can be t (True) or f (False).
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.

PROFILE_PARAMETERS

Defines what information is stored in profiles.

Column Name	Data Type	Description
PROFILE_ID	INTEGER	The ID of the profile to which this parameter belongs.
PROFILE_NAME	VARCHAR	The name of the profile to which this parameter belongs.
PARAMETER_TYPE	VARCHAR	The policy type of this parameter (password_complexity , password_security , etc.)
PARAMETER_NAME	VARCHAR	The name of the parameter.
PARAMETER_LIMIT	VARCHAR	The parameter's value.

PROFILES

Provides information about password policies that you set using the CREATE PROFILE statement.

Column Name	Data Type	Description
PROFILE_ID	INTEGER	Unique identifier for the profile.
PROFILE_NAME	VARCHAR	Profile name.
PASSWORD_LIFE_TIME	VARCHAR	Number of days before the user's password expires. After expiration, the user is forced to change passwords during login or warned that their password has expired if password_grace_time is set to a value other than zero or unlimited.
PASSWORD_MIN_LIFE_TIME	VARCHAR	The number of days a password must be set before it can be reset.
PASSWORD_MIN_CHAR_CHANGE	VARCHAR	The minimum number of characters that must be different from the previous password when performing a password reset.
PASSWORD_GRACE_TIME	VARCHAR	Number of days users are allowed to log in after their passwords expire. During the grace time, users are warned about their expired passwords when they log in. After the grace period, the user is forced to change passwords if he or she hasn't already.
PASSWORD_REUSE_MAX	VARCHAR	Number of password changes that must occur before the current password can be reused.
PASSWORD_REUSE_TIME	VARCHAR	Number of days that must pass after setting a password before it can be used again.
FAILED_LOGIN_ATTEMPTS	VARCHAR	Number of consecutive failed login attempts that triggers Vertica to lock the account.
PASSWORD_LOCK_TIME	VARCHAR	Number of days an account is locked after being locked due to too many failed login attempts.
PASSWORD_MAX_LENGTH	VARCHAR	Maximum number of characters allowed in a password.
PASSWORD_MIN_LENGTH	VARCHAR	Minimum number of characters required in a password.
PASSWORD_MIN_LETTERS	VARCHAR	The minimum number of letters (either uppercase or lowercase) required in a password.
PASSWORD_MIN_LOWERCASE_LETTERS	VARCHAR	The minimum number of lowercase.
PASSWORD_MIN_UPPERCASE_LETTERS	VARCHAR	The minimum number of uppercase letters required in a password.
PASSWORD_MIN_DIGITS	VARCHAR	The minimum number of digits required in a password.

PASSWORD_MIN_SYMBOLS	VARCHAR	The minimum of symbols (for example, !, #, $, etc.) required in a password.

Notes

Non-superusers querying this table see only the information for the profile to which they are assigned.

See also

- CREATE PROFILE
- ALTER PROFILE

PROJECTION_CHECKPOINT_EPOCHS

Provides details on checkpoint epochs, applies only to Enterprise Mode.

Column Name	Data Type	Description
NODE_ID	INTEGER	Unique numeric identifier of this projection's node.
NODE_NAME	VARCHAR	Name of this projection's node.
PROJECTION_SCHEMA_ID	INTEGER	Unique numeric identifier of the projection schema.
PROJECTION_SCHEMA	VARCHAR	Name of the projection schema.
PROJECTION_ID	INTEGER	Unique numeric identifier of this projection.
PROJECTION_NAME	VARCHAR	Name of this projection.
IS_UP_TO_DATE	BOOLEAN	Specifies whether the projection is up to date and available to participate in query execution.
CHECKPOINT_EPOCH	INTEGER	Checkpoint epoch of the projection on the corresponding node. Data up to and including this epoch is in persistent storage, and can be recovered in the event of node failure.
WOULD_RECOVER	BOOLEAN	Determines whether data up to and including CHECKPOINT_EPOCH can be used to recover from an unclean shutdown : • t : CHECKPOINT_EPOCH is less than the cluster's Last Good Epoch, so data up to and including this epoch can be used during recovery. • f : Vertica must use Last Good Epoch to recover data for this projection. See also: GET_LAST_GOOD_EPOCH
IS_BEHIND_AHM	BOOLEAN	Specifies whether CHECKPOINT_EPOCH is less than the AHM (ancient history mark). If set to t (true), data for this projection cannot rolled back. See also: GET_AHM_EPOCH

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

Examples

```
=> SELECT node_name, projection_schema, projection_name, is_up_to_date, checkpoint_epoch FROM projection_checkpoint_epochs
   WHERE projection_name ilike 't1_b%' ORDER BY projection_name, node_name;
   node_name     | projection_schema | projection_name | is_up_to_date | checkpoint_epoch
-----------------+-------------------+-----------------+---------------+------------------
 v_vmart_node0001 | public           | t1_b1           | t             |              965
 v_vmart_node0002 | public           | t1_b1           | t             |              965
 v_vmart_node0003 | public           | t1_b1           | t             |              965
 v_vmart_node0001 | public           | t1_b0           | t             |              965
 v_vmart_node0002 | public           | t1_b0           | t             |              965
 v_vmart_node0003 | public           | t1_b0           | t             |              965
(6 rows)

dbadmin=> INSERT INTO t1 VALUES (100, 101, 102);
 OUTPUT
--------
      1
(1 row)

dbadmin=> COMMIT;
COMMIT
dbadmin=> SELECT node_name, projection_schema, projection_name, is_up_to_date, checkpoint_epoch FROM projection_checkpoint_epochs
          WHERE projection_name ILIKE 't1_b%' ORDER BY projection_name, node_name;
   node_name     | projection_schema | projection_name | is_up_to_date | checkpoint_epoch
-----------------+-------------------+-----------------+---------------+------------------
 v_vmart_node0001 | public           | t1_b1           | t             |              966
 v_vmart_node0002 | public           | t1_b1           | t             |              966
 v_vmart_node0003 | public           | t1_b1           | t             |              966
 v_vmart_node0001 | public           | t1_b0           | t             |              966
 v_vmart_node0002 | public           | t1_b0           | t             |              966
 v_vmart_node0003 | public           | t1_b0           | t             |              966
(6 rows)
```

PROJECTION_COLUMNS

Provides information about projection columns, such as encoding type, sort order, type of statistics, and the time at which columns statistics were last updated.

Column Name	Data Type	Description
PROJECTION_ID	INTEGER	Catalog-assigned numeric value that uniquely identifies the projection.
PROJECTION_NAME	VARCHAR	The projection name for which information is listed.
PROJECTION_COLUMN_NAME	VARCHAR	The projection column name.
COLUMN_POSITION	INTEGER	The ordinal position of a projection's column used in the CREATE PROJECTION statement.
SORT_POSITION	INTEGER	The projection's column sort specification, as specified in CREATE PROJECTION .. ORDER BY clause. If the column is not included in the projection's sort order, SORT_POSITION output is NULL.
COLUMN_ID	INTEGER	A unique numeric object ID (OID) that identifies the associated projection column object and is assigned by the Vertica catalog. This field is helpful as a key to other system tables.
DATA_TYPE	VARCHAR	Matches the corresponding table column data type (see V_CATALOG.COLUMNS). DATA_TYPE is provided as a complement to ENCODING_TYPE .

ENCODING_TYPE	VARCHAR	The encoding type defined on the projection column.
ACCESS_RANK	INTEGER	The access rank of the projection column. See the ACCESSRANK parameter in the CREATE PROJECTION statement for more information.
GROUP_ID	INTEGER	A unique numeric ID (OID) that identifies the group and is assigned by the Vertica catalog.
TABLE_SCHEMA	VARCHAR	The name of the schema in which the projection is stored.
TABLE_ID	INTEGER	Catalog-assigned numeric value that uniquely identifies the table.
TABLE_NAME	VARCHAR	The table name that contains the projection.
TABLE_COLUMN_ID	VARCHAR	Catalog-assigned VARCHAR value that uniquely identifies a table column.
TABLE_COLUMN_NAME	VARCHAR	The projection's corresponding table column name.
STATISTICS_TYPE	VARCHAR	The type of statistics the column contains: • NONE : No statistics • ROWCOUNT : Set by ANALYZE_ROW_COUNT • FULL : Set by running ANALYZE_STATISTICS
STATISTICS_UPDATED_TIMESTAMP	TIMESTAMPTZ	The time at which the columns statistics were last updated by ANALYZE_STATISTICS . By querying this column, along with STATISTICS_TYPE and PROJECTION_COLUMN_NAME , you can identify projection columns whose statistics need updating. See also system table PROJECTIONS .
IS_EXPRESSION	BOOLEAN	Indicates whether this projection column is calculated with an expression. For aggregate columns, IS_EXPRESSION is always true.
IS_AGGREGATE	BOOLEAN	Indicates whether the column is an aggregated column in a live aggregate projection. IS_AGGREGATE is always false for Top-K projection columns.
PARTITION_BY_POSITION	INTEGER	Position of that column in the PARTITION BY and GROUP BY clauses, if applicable.
ORDER_BY_POSITION	INTEGER	Set only for Top-K projections , specifies the column's position in the ORDER BY clause, as defined in the projection definition's window partition clause . If the column is omitted from the ORDER BY clause, ORDER_BY_POSITION output is NULL.
ORDER_BY_TYPE	INTEGER	Type of sort order: • ASC NULLS FIRST • ASC NULLS LAST • DESC NULLS FIRST • DESC NULLS LAST
COLUMN_EXPRESSION	VARCHAR	Expression that calculates the column value.

Examples

See Statistics Data in PROJECTION_COLUMNS

See also

- PROJECTIONS
- ANALYZE_STATISTICS
- CREATE PROJECTION

PROJECTION_DELETE_CONCERNS

Lists projections whose design are liable to cause performance issues when deleting data. This table is generated by calling the EVALUATE_DELETE_PERFORMANCE function. See Optimizing DELETE and UPDATE for more information.

Column Name	Data Type	Description
PROJECTION_ID	INTEGER	The ID number of the projection
PROJECTION_SCHEMA	VARCHAR	The schema containing the projection
PROJECTION_NAME	VARCHAR	The projection's name
CREATION_TIME	TIMESTAMPTZ	When the projection was created
LAST_MODIFIED_TIME	TIMESTAMPTZ	When the projection was last modified
COMMENT	VARCHAR	A comment describing the potential delete performance issue.

PROJECTIONS

Provides information about projections.

Column Name	Data Type	Description
PROJECTION_SCHEMA_ID	INTEGER	A unique numeric ID that identifies the specific schema that contains the projection and is assigned by the Vertica catalog.
PROJECTION_SCHEMA	VARCHAR	The name of the schema that contains the projection.
PROJECTION_ID	INTEGER	A unique numeric ID that identifies the projection and is assigned by the Vertica catalog.
PROJECTION_NAME	VARCHAR	The projection name for which information is listed.
PROJECTION_BASENAME	VARCHAR	The base name used for other projections: • For auto-created projections, identical to ANCHOR_TABLE_NAME . • For a manually-created projection, the name specified in the CREATE PROJECTION statement.
OWNER_ID	INTEGER	A unique numeric ID that identifies the projection owner and is assigned by the Vertica catalog.
OWNER_NAME	VARCHAR	The name of the projection's owner.
ANCHOR_TABLE_ID	INTEGER	The unique numeric identification (OID) of the projection's anchor table.
ANCHOR_TABLE_NAME	VARCHAR	The name of the projection's anchor table.
NODE_ID	INTEGER	A unique numeric ID (OID) for any nodes that contain any unsegmented projections.
NODE_NAME	VARCHAR	The names of any nodes that contain the projection. This column returns information for unsegmented projections only.
IS_PREJOIN	BOOLEAN	Deprecated, always set to f (false).
CREATED_EPOCH	INTEGER	The epoch in which the projection was created.

CREATE_TYPE	VARCHAR	The method in which the projection was created: • CREATE PROJECTION : A custom projection created using CREATE PROJECTION. • CREATE TABLE : A superprojection that was automatically created when its associated table was created using CREATE TABLE. • ALTER TABLE : The system automatically created the key projection in response to a non-empty table. • CREATE TABLE WITH PROJ CLAUSE : A superprojection that was automatically created using CREATE TABLE. • DELAYED_CREATION : A superprojection that was automatically created when data was loaded for the first time into a new table. • DESIGNER : A projection created by Database Designer. • SYSTEM TABLE : A projection that was automatically created for a system table. Rebalancing does not change the CREATE_TYPE value for a projection.
VERIFIED_FAULT_TOLERANCE	INTEGER	The projection K-safe value. This value can be greater than the database K-safety value (if more replications of a projection exist than are required to meet the database K-safety). This value cannot be less than the database K-safe setting.
IS_UP_TO_DATE	BOOLEAN	Specifies whether projection data is up to date . Only up-to-date projections are available to participate in query execution.
HAS_STATISTICS	BOOLEAN	Specifies whether there are statistics for any column in the projection. HAS_STATISTICS returns true only when all non-epoch columns for a table or table partition have full statistics. For details, see Collecting table statistics and Collecting partition statistics . **Note** Projections that have no data never have full statistics. Query system table PROJECTION_STORAGE to determine whether your projection contains data.
IS_SEGMENTED	BOOLEAN	Specifies whether the projection is segmented.
SEGMENT_EXRESSION	VARCHAR	The segmentation expression used for the projection. In the following example for the clicks_agg projection, the following values: hash(clicks.user_id, (clicks.click_time)::date) indicate that the projection was created with the following expression: SEGMENTED BY HASH(clicks.user_id, (clicks.click_time)::date)
SEGMENT_RANGE	VARCHAR	The percentage of projection data stored on each node, according to the segmentation expression. For example, segmenting a projection by the HASH function on all nodes results in a SEGMENT_RANGE value such as the following: implicit range: v_vmart_node0002[33.3%] v_vmart_node0003[33.3%] v_vmart_node0001[33.3%]
IS_SUPER_PROJECTION	BOOLEAN	Specifies whether a projection is a superprojection.
IS_KEY_CONSTRAINT_PROJECTION	BOOLEAN	Indicates whether a projection is a key constraint projection: • t : A key constraint projection that validates a key constraint. Vertica uses the projection to efficiently enforce at least one enabled key constraint. • f : Not a projection that validates a key constraint.

HAS_EXPRESSIONS	BOOLEAN	Specifies whether this projection has expressions that define the column values. HAS_EXPRESSIONS is always true for live aggregate projections.
IS_AGGREGATE_PROJECTION	BOOLEAN	Specifies whether this projection is a live aggregate projection.
AGGREGATE_TYPE	VARCHAR	Specifies the type of live aggregate projection: • GROUPBY • TOPK
IS_SHARED	BOOLEAN	Indicates whether the projection is located on shared storage.

PARTITION_RANGE_MIN
PARTITION_RANGE_MAX
	VARCHAR	Populated only if a projection specifies a partition range , the lowest and highest partition keys of the range. For example, following projection defines a range of orders that were placed since the first of the year:
		`=> CREATE PROJECTION ytd_orders AS SELECT * FROM store_orders ORDER BY order_date`
		`  ON PARTITION RANGE BETWEEN date_trunc('year',now())::date AND NULL;`
		Given that range, columns PARTITION_RANGE_MIN and PARTITION_RANGE_MAX contain the following values:
		`=> SELECT projection_name partition_range_min, partition_range_min, partition_range_max FROM projections WHERE projection_name ILIKE 'ytd_orders%';`
		`partition_range_min | partition_range_min | partition_range_max`
		`----------+----------+----------`
		`ytd_orders_b1 | 2021-01-01 | infinity`
		`ytd_orders_b0 | 2021-01-01 | infinity`
		`(2 rows)`

PARTITION_RANGE_MIN_EXPRESSION
PARTITION_RANGE_MAX_EXPRESSION
	VARCHAR	Populated only if a projection specifies partition range , the minimum and maximum range expressions as defined in the projection DDL. For example, following projection defines a range of orders that were placed since the third quarter of last year:
		`=> CREATE PROJECTION q3_td AS SELECT * FROM store_orders ORDER BY order_date`
		`  ON PARTITION RANGE BETWEEN add_months(date_trunc('year',now()), -3)::date AND NULL;`
		Given that definition, PARTITION_RANGE_MIN_EXPRESSION and PARTITION_RANGE_MAX_EXPRESSION are set as follows:
		`=> SELECT projection_name, partition_range_min_expression, partition_range_max_expression FROM projections WHERE projection_name ILIKE 'Q3_td%';`
		`projection_name | partition_range_min_expression | partition_range_max_expression`
		`----------+----------+----------`
		`q3_td_b1 | add_months(date_trunc('year', now()), (-3)) | NULL`
		`q3_td_b0 | add_months(date_trunc('year', now()), (-3)) | NULL`

See also

PROJECTION_COLUMNS

REGISTERED_MODELS

Lists details about registered machine learning models in the database. The table lists only registered models for which the caller has USAGE privileges.

For a table that lists all models, registered and unregistered, see MODELS .

Column Name	Data Type	Description

REGISTERED_NAME	VARCHAR	The abstract name to which the model is registered. This REGISTERED_NAME can represent a group of models for a higher-level application, where each model in the group has a unique version number.
REGISTERED_VERSION	INTEGER	The unique version number of the model under its specified REGISTERED_NAME .
STATUS	INTEGER	The status of the registered model, one of the following: • under_review : Status assigned to newly registered models. • staging : Model is targeted for A/B testing against the model currently in production. • production : Model is in production for its specified application. Only one model can be in production for a given *registered_name* at one time. • archived : Status of models that were previously in production. Archived models can be returned to production at any time. • declined : Model is no longer in consideration for production. • unregistered : Model is removed from the versioning environment. The model does not appear in the REGISTERED_MODELS system table.
REGISTERED_TIME	VARCHAR	The time at which the model was registered.
MODEL_ID	INTEGER	The model's internal ID.
SCHEMA_NAME	VARCHAR	The name of the schema that contains the model.
MODEL_NAME	VARCHAR	The name of the model. [*schema_name.*] *model_name* can be used to uniquely identify a model, as can the combination of its REGISTERED_NAME and REGISTERED_VERSION .
MODEL_TYPE	VARCHAR	The type of algorithm used to create the model.
CATEGORY	VARCHAR	The category of the model, one of the following: • VERTICA_MODELS • PMML • TENSORFLOW By default, models created in Vertica are assigned to the VERTICA_MODELS category.

Example

If a user with the MLSUPERVISOR role queries REGISTERED_MODELS, all registered models are listed:

```
=> SELECT * FROM REGISTERED_MODELS;
 registered_name | registered_version |    status    |       registered_time        |     model_id     | schema_name |    model_name    |      model_type       |
category
-----------------+--------------------+--------------+------------------------------+------------------+-------------+------------------+-----------------------+----------------
 linear_reg_app  |                  2 | UNDER_REVIEW | 2023-01-29 09:09:00.082166-04 | 45035996273714020 | public      | linear_reg_spark1 |
PMML_REGRESSION_MODEL | PMML
 linear_reg_app  |                  1 | PRODUCTION   | 2023-01-24 06:19:04.553102-05 | 45035996273850350 | public      | native_linear_reg |
LINEAR_REGRESSION     | VERTICA_MODELS
 logistic_reg_app |                  2 | PRODUCTION   | 2023-01-25 08:45:11.279013-02 | 45035996273855542 | public      | log_reg_cgd       |
LOGISTIC_REGRESSION   | VERTICA_MODELS
 logistic_reg_app |                  1 | ARCHIVED     | 2023-01-22 04:29:25.990626-02 | 45035996273853740 | public      | log_reg_bfgs      |
LOGISTIC_REGRESSION   | VERTICA_MODELS
(4 rows)
```

See also

- REGISTER_MODEL
- CHANGE_MODEL_STATUS
- Model versioning
- MODEL_STATUS_HISTORY

RESOURCE_POOL_DEFAULTS

Returns default parameter settings for built-in and user-defined resource pools. Use ALTER RESOURCE POOL to restore resource pool parameters to their default settings.

For information about valid parameters for built-in resource pools and their default settings, see Built-in resource pools configuration .

To obtain a resource pool's current settings, query system table RESOURCE_POOLS .

Privileges

None

RESOURCE_POOLS

Displays settings for built-in and user-defined resource pools. For information about defining resource pools, see CREATE RESOURCE POOL and ALTER RESOURCE POOL .

Column Name	Data Type	Description
POOL_ID	INTEGER	Unique identifier for the resource pool
NAME	VARCHAR	The name of the resource pool.
SUBCLUSTER_OID	INTEGER	Unique identifier for a subcluster-specific resource pool. For global resource pools, 0 is returned.
SUBCLUSTER_NAME	VARCHAR	Specifies the subcluster that the subcluster-specific resource pool belongs to.If there are subcluster-specific resource pools with the same name on separate subclusters, multiple entries are returned. For global resource pools, this column is blank.
IS_INTERNAL	BOOLEAN	Specifies whether this pool is a built-in pool .
MEMORYSIZE	VARCHAR	The amount of memory allocated to this resource pool.
MAXMEMORYSIZE	VARCHAR	Value assigned as the maximum size this resource pool can grow by borrowing memory from the GENERAL pool.
MAXQUERYMEMORYSIZE	VARCHAR	The maximum amount of memory allocated by this pool to process any query.
EXECUTIONPARALLELISM	INTEGER	Limits the number of threads used to process any single query issued in this resource pool.
PRIORITY	INTEGER	Specifies priority of queries in this pool when they compete for resources in the GENERAL pool.
RUNTIMEPRIORITY	VARCHAR	The run-time priority defined for this pool, indicates how many run-time resources (CPU, I/O bandwidth) the Resource Manager should dedicate to running queries in the resource pool. Valid values are: • HIGH • MEDIUM (default) • LOW These values are relative to each other. Queries with a HIGH run-time priority are given more CPU and I/O resources than those with a MEDIUM or LOW run-time priority.
RUNTIMEPRIORITYTHRESHOLD	INTEGER	Limits in seconds how soon a query must finish before the Resource Manager assigns to it the resource pool's RUNTIMEPRIORITY setting.

QUEUETIMEOUT	INTEGER INTERVAL	The maximum length of time requests can wait for resources to become available before being rejected, specified in seconds or as an interval . This value is set by the pool's QUEUETIMEOUT parameter.
PLANNEDCONCURRENCY	INTEGER	The preferred number of queries that execute concurrently in this resource pool, specified by the pool's PLANNEDCONCURRENCY parameter.
MAXCONCURRENCY	INTEGER	The maximum number of concurrent execution slots available to the resource pool, specified by the pool MAXCONCURRENCY parameter.
RUNTIMECAP	INTERVAL	The maximum time a query in the pool can execute.
SINGLEINITIATOR	BOOLEAN	Set for backward compatibility.
CPUAFFINITYSET	VARCHAR	The set of CPUs on which queries associated with this pool are executed. For example: • 0, 2-4 : Specifies CPUs 0, 2, 3, and 4 • 25% : A percentage of available CPUs, rounded down to whole CPUs.
CPUAFFINITYMODE	VARCHAR	Specifies whether to share usage of the CPUs assigned to this resource pool by CPUAFFINITYSET , one of the following: • SHARED : Queries that run in this pool share its CPUAFFINITYSET CPUs with other Vertica resource pools. • EXCLUSIVE : Dedicates CPUAFFINITYSET CPUs to this resource pool only, and excludes other Vertica resource pools. If CPUAFFINITYSET is set as a percentage, then that percentage of CPU resources available to Vertica is assigned solely for this resource pool. • ANY : Queries in this resource pool can run on any CPU.
CASCADETO	VARCHAR	A secondary resource pool for executing queries that exceed the RUNTIMECAP setting of this resource pool.
CASCADETOSUBCLUSTERPOOL	BOOLEAN	Specifies whether this resource pool cascades to a subcluster-level resource pool.

ROLES

Contains the names of all roles the user can access, along with any roles that have been assigned to those roles.

> **Tip**
> You can also use the function HAS_ROLE to see if a role is available to a user.

Column Name	Data Type	Description
ASSIGNED_ROLES	VARCHAR	The names of any roles that have been granted to this role. By enabling the role, the user also has access to the privileges of these additional roles. **Note** An asterisk (*) appended to a role in this column indicates that the user can grant the role to other users.
NAME	VARCHAR	The name of a role that the user can access.

ROLE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies therole.
LDAP_DN	VARCHAR	Indicates whether or not the Vertica Analytic Database role maps to an LDAP Link group. When the column is set to dn , the Vertica role maps to LDAP Link.
LDAP_URI_HASH	VARCHAR	The URI hash number for the LDAP role.
IS_ORPHANED_FROM_LDAP	VARCHAR	Indicates if the role is disconnected (orphaned) from LDAP, valid values are: t - role is orphaned f - role is not orphaned For more information see Troubleshooting LDAP link issues

See also

- GRANTS
- HAS_ROLE
- USERS

ROUTING_RULES

Lists the routing rules that map incoming IP addresses to a load balancing group.

Column Name	Data Type	Description
NAME	VARCHAR	The name of the routing rule.
SOURCE_ADDRESS	VARCHAR	The IP address range in CIDR format that this rule applies to.
DESTINATION_NAME	VARCHAR	The load balance group that handles connections for this rule.

Examples

```
=> SELECT * FROM routing_rules;
-[ RECORD 1 ]----+-----------------
name             | internal_clients
source_address   | 192.168.1.0/24
destination_name | group_1
-[ RECORD 2 ]----+-----------------
name             | etl_rule
source_address   | 10.20.100.0/24
destination_name | group_2
-[ RECORD 3 ]----+-----------------
name             | subnet_192
source_address   | 192.0.0.0/8
destination_name | group_all
-[ RECORD 4 ]----+--------------
name             | all_ipv6
source_address   | 0::0/0
destination_name | default_ipv6
```

See also

SCHEDULER_TIME_TABLE

Contains information about scheduled tasks .

This table is local to the active scheduler node (ASN) and is only populated when queried from that node. To get the ASN, use ACTIVE_SCHEDULER_NODE :

```
=> SELECT active_scheduler_node();
 active_scheduler_node
-----------------------
 initiator
(1 row)
```

Column Name	Data Type	Description
SCHEDULE_NAME	VARCHAR	The name of the schedule.
SCHEMA_NAME	VARCHAR	The schedule's schema.
OWNER	VARCHAR	The owner of the schedule.
ATTACHED_TRIGGER	VARCHAR	The trigger attached to the schedule . For details, see Scheduled execution .
ENABLED	BOOLEAN	Whether the schedule is enabled.
DATE_TIME_TYPE	VARCHAR	The format for the scheduled event, one of the following: • CRON • DATE_TIME_LIST
DATE_TIME_STRING	VARCHAR	The string used to schedule the event, one of the following: • A cron expression • A comma-separated list of timestamps

Examples

To view scheduled tasks, execute the following statement on the ASN:

```
=> SELECT * FROM scheduler_time_table;
 schedule_name |       attached_trigger       | scheduled_execution_time
---------------+------------------------------+--------------------------
daily_1am      | log_user_actions             | 2022-06-01 01:00:00-00
logging_2022   | refresh_logs                 | 2022-06-01 12:00:00-00
```

SCHEMATA

Provides information about schemas in the database.

Column Name	Data Type	Description
SCHEMA_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the specific schema.
SCHEMA_NAME	VARCHAR	Schema name for which information is listed.
SCHEMA_OWNER_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the owner who created the schema.
SCHEMA_OWNER	VARCHAR	Name of the owner who created the schema.
SYSTEM_SCHEMA_CREATOR	VARCHAR	Creator information for system schema or NULL for non-system schema
CREATE_TIME	TIMESTAMPTZ	Time when the schema was created.
IS_SYSTEM_SCHEMA	BOOLEAN	Indicates whether the schema was created for system use, where *t* is true and *f* is false.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

SEQUENCES

Displays information about sequences .

Column Name	Data Type	Description
SEQUENCE_SCHEMA	VARCHAR	Sequence schema
SEQUENCE_NAME	VARCHAR	One of the following: • User-assigned name of a sequence created with CREATE SEQUENCE • Auto-generated name assigned by Vertica for the sequence of an IDENTITY table column
OWNER_NAME	VARCHAR	One of the following: • Owner of the named sequence • Owner of the table where an IDENTITY column is defined
IDENTITY_TABLE_NAME	VARCHAR	Set only for IDENTITY column sequences, name of the column table
SESSION_CACHE_COUNT	INTEGER	Count of values cached in a session
ALLOW_CYCLE	BOOLEAN	Whether values cycle when a sequence reaches its minimum or maximum value, as set by CREATE SEQUENCE parameter CYCLE|NO CYCLE
OUTPUT_ORDERED	BOOLEAN	Values guaranteed to be ordered, always false
INCREMENT_BY	INTEGER	Value by which sequences are incremented or decremented
MINIMUM	INTEGER	Minimum value the sequence can generate.
MAXIMUM	INTEGER	Maximum value the sequence can generate.
CURRENT_VALUE	INTEGER	How many sequence numbers are distributed among all cluster nodes.
SEQUENCE_SCHEMA_ID	INTEGER	Unique numeric catalog ID of the sequence schema
SEQUENCE_ID	INTEGER	Unique numeric catalog ID of the sequence
OWNER_ID	INTEGER	Unique numeric catalog ID of the user who created the sequence
IDENTITY_TABLE_ID	INTEGER	Set only for IDENTITY column sequences, unique numeric catalog ID of the column table

Examples

Create a sequence:

```
=> CREATE SEQUENCE my_seq MAXVALUE 5000 START 150;
CREATE SEQUENCE
```

Return information about this sequence:

```
=> SELECT sequence_schema, sequence_name, owner_name, session_cache_count, increment_by, current_value FROM sequences;
 sequence_schema |      sequence_name       | owner_name | session_cache_count | increment_by | current_value
-----------------+--------------------------+------------+---------------------+--------------+---------------
 public          | my_seq                   | dbadmin    |              250000 |            1 |           149
(2 rows)
```

IDENTITY columns are sequences that are defined in a table's DDL. IDENTITY column values automatically increment as new rows are added. To identify IDENTITY columns and their tables, query the system table COLUMNS :

```
=> CREATE TABLE employees (employeeID IDENTITY, fname varchar(36), lname varchar(36));
CREATE TABLE
=> SELECT table_name, column_name, is_identity FROM columns WHERE is_identity = 't';
 table_name | column_name | is_identity
------------+-------------+-------------
 employees  | employeeID  | t
(1 row)
```

Query SEQUENCES to get detailed information about the IDENTITY column sequence in employees :

```
=> SELECT sequence_schema, sequence_name, identity_table_name, sequence_id FROM sequences
    WHERE identity_table_name ='employees';
 sequence_schema |      sequence_name       | identity_table_name |    sequence_id
-----------------+--------------------------+---------------------+-------------------
 public          | employees_employeeID_seq | employees           | 45035996273848816
(1 row)
```

Use the vsql command \ds to list all named and IDENTITY column sequences. The following results show the two sequences created previously:

```
=> \ds
                                  List of Sequences
 Schema |         Sequence         | CurrentValue | IncrementBy | Minimum |       Maximum       | AllowCycle | Comment
--------+--------------------------+--------------+-------------+---------+---------------------+------------+---------
 public | employees_employeeID_seq |            0 |           1 |       1 | 9223372036854775807 | f          |
 public | my_seq                   |          149 |           1 |       1 |                5000 | f          |
(2 rows)
```

Note

The CurrentValue of both sequences is one less than its start number—0 and 149 for the IDENTITY column employeeID and named sequence my_seq , respectively:

- employeeID 's start number—by default set to 1 because the table DDL did not specify otherwise—is set to 0 because no rows have yet been added to the employees table.
- my_seq is set to 149 because NEXTVAL has not yet been called on it.

SESSION_SUBSCRIPTIONS

In an Eon Mode database, lists the shard subscriptions for all nodes, and whether the subscriptions are used to resolve queries for the current session. Nodes that will participate in resolving queries in this session have TRUE in their IS_PARTICIPATING column.

Column Name	Data Type	Description
NODE_OID	INTEGER	The OID of the subscribing node.
NODE_NAME	VARCHAR	The name of the subscribing node.
SHARD_OID	INTEGER	The OID of the shard the node subscribes to.
SHARD_NAME	VARCHAR	The name of the shard the node subscribes to.
IS_PARTICIPATING	BOOLEAN	Whether this subscription is used when resolving queries in this session.
IS_COLLABORATING	BOOLEAN	Whether this subscription is used to collaborate with a participating node when executing queries . This value is only true when queries are using elastic crunch scaling.

Examples

The following example demonstrates listing the subscriptions that are either participating or collaborating in the current session:

```
=> SELECT node_name, shard_name, is_collaborating, is_participating
      FROM V_CATALOG.SESSION_SUBSCRIPTIONS
      WHERE is_participating = TRUE OR is_collaborating = TRUE
      ORDER BY shard_name, node_name;
     node_name       | shard_name  | is_collaborating | is_participating
---------------------+-------------+------------------+------------------
 v_verticadb_node0004 | replica     | f                | t
 v_verticadb_node0005 | replica     | f                | t
 v_verticadb_node0006 | replica     | t                | f
 v_verticadb_node0007 | replica     | f                | t
 v_verticadb_node0008 | replica     | t                | f
 v_verticadb_node0009 | replica     | t                | f
 v_verticadb_node0007 | segment0001 | f                | t
 v_verticadb_node0008 | segment0001 | t                | f
 v_verticadb_node0005 | segment0002 | f                | t
 v_verticadb_node0009 | segment0002 | t                | f
 v_verticadb_node0004 | segment0003 | f                | t
 v_verticadb_node0006 | segment0003 | t                | f
(12 rows)
```

SHARDS

Lists the shards in your database.

Column Name	Data Type	Description
SHARD_OID	INTEGER	The OID of the shard.
SHARD_NAME	VARCHAR	The name of the shard.
SHARD_TYPE	VARCHAR	The type of the shard.
LOWER_HASH_BOUND	VARCHAR	The lower hash bound of the shard.
UPPER_HASH_BOUND	VARCHAR	The upper hash bound of the shard.
IS_REPLICATED	BOOLEAN	Defines if the shard is replicated.
HAS_OBJECTS	BOOLEAN	Defines if the shard contains objects.

Examples

```
=> SELECT * FROM SHARDS;
-[ RECORD 1 ]----+------------------
shard_oid        | 45035996273704980
shard_name       | replica
shard_type       | Replica
lower_hash_bound |
upper_hash_bound |
is_replicated    | t
has_objects      | t
...
```

STORAGE_LOCATIONS

Provides information about storage locations, including IDs, labels, and status.

Column Name	Data Type	Description
LOCATION_ID	INTEGER	Catalog-assigned integer value that uniquely identifies thestorage location.
NODE_NAME	VARCHAR	The node name on which the storage location exists.
LOCATION_PATH	VARCHAR	The path where the storage location is mounted.
LOCATION_USAGE	VARCHAR	The type of information stored in the location: • DATA: Only data is stored in the location. • TEMP: Only temporary files that are created during loads or queries are stored in the location. • DATA,TEMP: Both types of files are stored in the location. • USER: The storage location can be used by users without their own credentials or dbadmin access. Users gain access to data by being granted access to the user storage location. • CATALOG : The area is used for the Vertica catalog. This usage is set internally and cannot be removed or changed.
SHARING_TYPE	VARCHAR	How this location is shared among database nodes, if it is: • SHARED: The path used by the storage location is used by all nodes. See the SHARED parameter to CREATE LOCATION . • COMMUNAL: the location is used for communal storage in Eon Mode. • NONE: The location is not shared among nodes.
IS_RETIRED	BOOLEAN	Whether the storage location has been retired. This column has a value of t (true) if the location is retired, or f (false) if it is not.
LOCATION_LABEL	VARCHAR	The label associated with a specific storage location, added with the ALTER_LOCATION_LABEL function.
RANK	INTEGER	The Access rank value either assigned or supplied to the storage location, as described in Prioritizing column access speed .
THROUGHPUT	INTEGER	The throughput performance of the storage location, measured in MB/sec. You can get location performance values using MEASURE_LOCATION_PERFORMANCE , and set them with the SET_LOCATION_PERFORMANCE function.
LATENCY	INTEGER	The measured latency of the storage location as number of data seeks per second. You can get location performance values using MEASURE_LOCATION_PERFORMANCE , and set them with the SET_LOCATION_PERFORMANCE function.
MAX_SIZE	INTEGER	Maximum size of the storage location in bytes.
DISK_PERCENT	VARCHAR	Maximum percentage of available node disk space that this storage location can use, set only if depot size is defined as a percentage, otherwise blank.

Privileges

Superuser

See also

- DISK_STORAGE
- MEASURE_LOCATION_PERFORMANCE
- SET_LOCATION_PERFORMANCE
- STORAGE_POLICIES
- STORAGE_USAGE
- Storage functions

STORED_PROC_TRIGGERS

Contains information about Triggers .

Column Name	Data Type	Description
TRIGGER_NAME	VARCHAR	The name of the trigger.
SCHEMA_NAME	VARCHAR	The trigger's schema.
OWNER	VARCHAR	The owner of the trigger.
PROCEDURE_NAME	VARCHAR	The name of the stored procedure .
PROCEDURE_ARGS	INTEGER	The number of formal parameters in the stored procedure.
ENABLED	BOOLEAN	Whether the trigger is enabled.

Examples

To view triggers:

```
=> SELECT * FROM stored_proc_triggers;
 trigger_name  | schema_name |  owner  | procedure_name | procedure_args | enabled
---------------+-------------+---------+----------------+----------------+---------
raise_trigger  | public      | dbadmin | raiseXY        | 2              | t
```

SUBCLUSTER_RESOURCE_POOL_OVERRIDES

Displays subcluster-level overrides of settings for built-in global resource pools.

Column Name	Data Type	Description
POOL_OID	INTEGER	Unique identifier for the resource pool with settings overrides.
NAME	VARCHAR	The name of the built-in resource pool.
SUBCLUSTER_OID	INTEGER	Unique identifier for the subcluster with settings that override the global resource pool settings.
SUBCLUSTER_NAME	VARCHAR	The name of the subcluster with settings that overrides the global resource pool settings.
MEMORYSIZE	VARCHAR	The amount of memory allocated to the global resource pool.
MAXMEMORYSIZE	VARCHAR	Value assigned as the maximum size this resource pool can grow by borrowing memory from the GENERAL pool.
MAXQUERYMEMORYSIZE	VARCHAR	The maximum amount of memory allocated by this pool to process any query.

SUBCLUSTERS

This table lists all of the subclusters defined in the database. It contains an entry for each node in the database listing which subcluster it belongs to. Any subcluster that does not contain a node has a single entry in this table with empty NODE_NAME and NODE_OID columns. This table is only populated if the database is running in Eon Mode.

Column Name	Data Type	Description
SUBCLUSTER_OID	INTEGER	Unique identifier for the subcluster.
SUBCLUSTER_NAME	VARCHAR	The name of the subcluster.

NODE_OID	INTEGER	The catalog-assigned ID of the node.
NODE_NAME	VARCHAR	The name of the node.
PARENT_OID	INTEGER	The unique ID of the parent of the node (the database).
PARENT_NAME	VARCHAR	The name of the parent of the node (the database name).
IS_DEFAULT	BOOLEAN	Whether the subcluster is the default cluster .
IS_PRIMARY	BOOLEAN	Whether the subcluster is a primary subcluster .
CONTROL_SET_SIZE	INTEGER	The number of control nodes defined for this subcluster. This value is -1 when the large cluster feature is not enabled, or when every node in the subcluster must be a control node. See Large cluster for more information.

Examples

```
=> \x
Expanded display is on.
dbadmin=> SELECT * FROM SUBCLUSTERS;
-[ RECORD 1 ]----+--------------------
subcluster_oid   | 45035996273704978
subcluster_name  | default_subcluster
node_oid         | 45035996273704982
node_name        | v_verticadb_node0001
parent_oid       | 45035996273704976
parent_name      | verticadb
is_default       | t
is_primary       | t
control_set_size | -1
-[ RECORD 2 ]----+--------------------
subcluster_oid   | 45035996273704978
subcluster_name  | default_subcluster
node_oid         | 45035996273840970
node_name        | v_verticadb_node0002
parent_oid       | 45035996273704976
parent_name      | verticadb
is_default       | t
is_primary       | t
control_set_size | -1
-[ RECORD 3 ]----+--------------------
subcluster_oid   | 45035996273704978
subcluster_name  | default_subcluster
node_oid         | 45035996273840974
node_name        | v_verticadb_node0003
parent_oid       | 45035996273704976
parent_name      | verticadb
is_default       | t
is_primary       | t
control_set_size | -1
```

See also

- CRITICAL_SUBCLUSTERS
- ALTER SUBCLUSTER
- DEMOTE_SUBCLUSTER_TO_SECONDARY
- PROMOTE_SUBCLUSTER_TO_PRIMARY
- SHUTDOWN_SUBCLUSTER

SYSTEM_COLUMNS

Provides table column information for SYSTEM_TABLES .

Column Name	Data Type	Description
TABLE_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog, which identifies the table.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.
TABLE_NAME	VARCHAR	The table name for which information is listed.
IS_SYSTEM_TABLE	BOOLEAN	Indicates whether the table is a system table, where *t* is true and *f* is false.
COLUMN_ID	VARCHAR	Catalog-assigned VARCHAR value that uniquely identifies a table column.
COLUMN_NAME	VARCHAR	The column name for which information is listed in the database.
DATA_TYPE	VARCHAR	The data type assigned to the column; for example VARCHAR(16).
DATA_TYPE_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog, which identifies the data type.
DATA_TYPE_LENGTH	INTEGER	The maximum allowable length of the data type.
CHARACTER_MAXIMUM_LENGTH	INTEGER	The maximum allowable length of the column.
NUMERIC_PRECISION	INTEGER	The number of significant decimal digits.
NUMERIC_SCALE	INTEGER	The number of fractional digits.
DATETIME_PRECISION	INTEGER	For TIMESTAMP data type, returns the declared precision; returns null if no precision was declared.
INTERVAL_PRECISION	INTEGER	The number of fractional digits retained in the seconds field.
ORDINAL_POSITION	INTEGER	The position of the column respective to other columns in the table.
IS_NULLABLE	BOOLEAN	Indicates whether the column can contain null values, where *t* is true and *f* is false.
COLUMN_DEFAULT	VARCHAR	The default value of a column, such as empty or expression.

SYSTEM_TABLES

Returns a list of all system table names.

Column Name	Data Type	Description
TABLE_SCHEMA_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theschema.
TABLE_SCHEMA	VARCHAR	The schema name in which the system table resides, one of the following: • V_CATALOG • V_MONITOR
TABLE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies thetable.
TABLE_NAME	VARCHAR	The name of the system table.

TABLE_DESCRIPTION	VARCHAR	A description of the system table's purpose.
IS_SUPERUSER_ONLY	BOOLEAN	Specifies whether the table is accessible only by superusers by default. If false, then the PUBLIC role has the privileges to access the system table.
IS_MONITORABLE	BOOLEAN	Specifies whether the table is accessible by a user with the SYSMONITOR role enabled.
IS_ACCESSIBLE_DURING_LOCKDOWN	BOOLEAN	Specifies whether RESTRICT_SYSTEM_TABLES_ACCESS revokes privileges from PUBLIC on the system table. These privileges can be restored with RELEASE_SYSTEM_TABLES_ACCESS . In general, this field is set to t (true) for system tables that contain information that is typically needed by most users, such as TYPES . Conversely, this field is set to f (false) for tables with data that should be restricted during lockdown, such as database settings and user information.

TABLE_CONSTRAINTS

Provides information about table constraints.

Column Name	Data Type	Description
CONSTRAINT_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theconstraint.
CONSTRAINT_NAME	VARCHAR	The name of the constraint, if specified as UNIQUE, FOREIGN KEY, NOT NULL, PRIMARY KEY, or CHECK.
CONSTRAINT_SCHEMA_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theschema containing the constraint.
CONSTRAINT_KEY_COUNT	INTEGER	The number of constraint keys.
FOREIGN_KEY_COUNT	INTEGER	The number of foreign keys.
TABLE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies thetable.
TABLE_NAME	VARCHAR	The name of the table that contains the UNIQUE, FOREIGN KEY, NOT NULL, or PRIMARY KEY constraint
FOREIGN_TABLE_ID	INTEGER	The unique object ID of the foreign table referenced in a foreign key constraint (zero if not a foreign key constraint).
CONSTRAINT_TYPE	CHAR	Indicates the constraint type. **Valid Values:** • c — check • f — foreign • p — primary • u — unique
IS_ENABLED`	BOOLEAN	Indicates if a constraint for a primary key, unique key, or check constraint is currently enabled. Can be t (True) or f (False).
PREDICATE	VARCHAR	For check constraints, the SQL expression.

See also

ANALYZE_CONSTRAINTS

TABLES

Provides information about all tables in the database.

The TABLE_SCHEMA and TABLE_NAME columns are case-sensitive. To restrict a query based on those columns, use the case-insensitive ILIKE predicate. For example:

```
=> SELECT table_schema, table_name FROM v_catalog.tables
  WHERE table_schema ILIKE 'Store%';
```

Column Name	Data Type	Description
TABLE_SCHEMA_ID	INTEGER	A unique numeric ID that identifies the schema and is assigned by the Vertica catalog.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.
TABLE_ID	INTEGER	A unique numeric ID that identifies the table and is assigned by the Vertica catalog.
TABLE_NAME	VARCHAR	The table name for which information is listed.
OWNER_ID	INTEGER	A unique numeric ID that identifies the owner and is assigned by the Vertica catalog.
OWNER_NAME	VARCHAR	The name of the user who created the table.
IS_TEMP_TABLE	BOOLEAN	Whether this table is a temporary table.
IS_SYSTEM_TABLE	BOOLEAN	Whether this table is a system table.
FORCE_OUTER	INTEGER	Whether this table is joined to another as an inner or outer input. For details, see Controlling join inputs .
IS_FLEXTABLE	BOOLEAN	Whether the table is a Flex table.
IS_SHARED	BOOLEAN	Whether the table is located on shared storage. Not used for temporary tables in Eon Mode.
HAS_AGGREGATE_PROJECTION	BOOLEAN	Whether the table has live aggregate projections.
SYSTEM_TABLE_CREATOR	VARCHAR	The name of the process that created the table, such as Designer.
PARTITION_EXPRESSION	VARCHAR	The table's partition expression .
CREATE_TIME	TIMESTAMP	When the table was created.
TABLE_DEFINITION	VARCHAR	For external tables, the COPY FROM portion of the table definition.
RECOVER_PRIORITY	INTEGER	The priority rank for the table for a Recovery By Table .
STORAGE_MODE	INTEGER	Deprecated, always set to DIRECT.
PARTITION_GROUP_EXPRESSION	VARCHAR	The expression of a GROUP BY clause that qualifies a table's partition clause .
ACTIVE_PARTITION_COUNT	INTEGER	The table's active partition count as set by CREATE TABLE or ALTER TABLE . If null, the table gets its active partition count from the ActivePartitionCount configuration parameter. For details, see Active and inactive partitions .
IS_MERGEOUT_ENABLED	BOOLEAN	Whether mergeout is enabled (t) or disabled (f) on ROS containers that consolidate projection data of this table. By default, mergeout is enabled on all tables. You can disable mergeout on a table with ALTER TABLE . For details, see Disabling mergeout on specific tables .
IMMUTABLE_ROWS_SINCE_TIMESTAMP	TIMESTAMPTZ	Set only for immutable tables , the server system time when immutability was applied to this table. This value can help with long-term timestamp retrieval and efficient comparison.

IMMUTABLE_ROWS_SINCE_EPOCH	INTEGER	Set only for immutable tables , the epoch that was current when immutability was applied. This setting can help protect the table from attempts to pre-insert records with a future timestamp, so that row's epoch is less than the table's immutability epoch.
IS_EXTERNAL_ICEBERG_TABLE	BOOLEAN	Whether this table is an Iceberg table .

Examples

Find when tables were created:

```
=> SELECT table_schema, table_name, create_time FROM tables;
 table_schema |      table_name       |          create_time
--------------+-----------------------+-------------------------------
 public       | customer_dimension    | 2011-08-15 11:18:25.784203-04
 public       | product_dimension     | 2011-08-15 11:18:25.815653-04
 public       | promotion_dimension   | 2011-08-15 11:18:25.850592-04
 public       | date_dimension        | 2011-08-15 11:18:25.892347-04
 public       | vendor_dimension      | 2011-08-15 11:18:25.942805-04
 public       | employee_dimension    | 2011-08-15 11:18:25.966985-04
 public       | shipping_dimension    | 2011-08-15 11:18:25.999394-04
 public       | warehouse_dimension   | 2011-08-15 11:18:26.461297-04
 public       | inventory_fact        | 2011-08-15 11:18:26.513525-04
 store        | store_dimension       | 2011-08-15 11:18:26.657409-04
 store        | store_sales_fact      | 2011-08-15 11:18:26.737535-04
 store        | store_orders_fact     | 2011-08-15 11:18:26.825801-04
 online_sales | online_page_dimension | 2011-08-15 11:18:27.007329-04
 online_sales | call_center_dimension | 2011-08-15 11:18:27.476844-04
 online_sales | online_sales_fact     | 2011-08-15 11:18:27.49749-04
(15 rows)
```

Find out whether certain tables are temporary and flex tables:

```
=> SELECT distinct table_name, table_schema, is_temp_table, is_flextable FROM v_catalog.tables
   WHERE table_name ILIKE 't%';
  table_name  | table_schema | is_temp_table | is_flextable
--------------+--------------+---------------+--------------
 t2_temp      | public       | t             | t
 tt_keys      | public       | f             | f
 t2_temp_keys | public       | f             | f
 t3           | public       | t             | f
 t1           | public       | f             | f
 t9_keys      | public       | f             | f
 t2_keys      | public       | f             | t
 t6           | public       | t             | f
 t5           | public       | f             | f
 t2           | public       | f             | t
 t8           | public       | f             | f
 t7           | public       | t             | f
 tt           | public       | t             | t
 t2_keys_keys | public       | f             | f
 t9           | public       | t             | t
(15 rows)
```

TEXT_INDICES

Provides summary information about the text indices in Vertica.

Column Name	Data Type	Description
INDEX_ID	INTEGER	A unique numeric ID that identifies the index and is assigned by the Vertica catalog.

INDEX_NAME	VARCHAR	The name of the text index.
INDEX_SCHEMA_NAME	VARCHAR	The schema name of the text index.
SOURCE_TABLE_ID	INTEGER	A unique numeric ID that identifies the table and is assigned by the Vertica catalog.
SOURCE_TABLE_NAME	VARCHAR	The name of the source table used to build the index.
SOURCE_TABLE_SCHEMA_NAME	VARCHAR	The schema name of the source table.
TOKENIZER_ID	INTEGER	A unique numeric ID that identifies the tokenizer and is assigned by the Vertica catalog.
TOKENIZER_NAME	VARCHAR	The name of the tokenizer used when building the index.
TOKENIZER_SCHEMA_NAME	VARCHAR	The schema name of the tokenizer.
STEMMER_ID	INTEGER	A unique numeric ID that identifies the stemmer and is assigned by the Vertica catalog.
STEMMER_NAME	VARCHAR	The name of the stemmer used when building the index.
STEMMER_SCHEMA_NAME	VARCHAR	The schema name of the stemmer.
TEXT_COL	VARCHAR	The text column used to build the index.

TYPES

Provides information about supported data types. This table does not include inlined complex types; see COMPLEX_TYPES instead. This table does include arrays and sets of primitive types.

Column Name	Data Type	Description
TYPE_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog, which identifies the specific data type.
ODBC_TYPE	INTEGER	The numerical ODBC type.
ODBC_SUBTYPE	INTEGER	The numerical ODBC subtype, used to differentiate types such as time and interval that have multiple subtypes.
JDBC_TYPE	INTEGER	The numerical JDBC type.
JDBC_SUBTYPE	INTEGER	The numerical JDBC subtype, used to differentiate types such as time and interval that have multiple subtypes.
MIN_SCALE	INTEGER	The minimum number of digits supported to the right of the decimal point for the data type.
MAX_SCALE	INTEGER	The maximum number of digits supported to the right of the decimal point for the data type. A value of 0 is used for types that do not use decimal points.
COLUMN_SIZE	INTEGER	The number of characters required to display the type. See: http://msdn.microsoft.com/en-us/library/windows/desktop/ms711786%28v=VS.85%29.aspx for the details on COLUMN_SIZE for each type.
INTERVAL_MASK	INTEGER	For data types that are intervals, the bitmask to determine the range of the interval from the Vertica TYPE_ID. Details are available in the Vertica SDK.
TYPE_NAME	VARCHAR	The data type name associated with a particular data type ID.

CREATION_PARAMETERS	VARCHAR	A list of keywords, separated by commas, corresponding to each parameter that the application may specify in parentheses when using the name that is returned in the TYPE_NAME field. The keywords in the list can be any of the following: length, precision, or scale. They appear in the order that the syntax requires them to be used.

USER_AUDITS

Lists the results of database and object size audits generated by users calling the AUDIT function. See Monitoring database size for license compliance for more information.

Column Name	Data Type	Description
SIZE_BYTES	INTEGER	The estimated raw data size of the database
USER_ID	INTEGER	The ID of the user who generated the audit
USER_NAME	VARCHAR	The name of the user who generated the audit
OBJECT_ID	INTEGER	The ID of the object being audited
OBJECT_TYPE	VARCHAR	The type of object being audited (table, schema, etc.)
OBJECT_SCHEMA	VARCHAR	The schema containing the object being audited
OBJECT_NAME	VARCHAR	The name of the object being audited
AUDITED_SCHEMA_NAME	VARCHAR	The name of the schema on which you want to query HISTORICAL data. After running audit on a table, you can drop the table. In this case, object_schema becomes NULL.
AUDITED_OBJECT_NAME	VARCHAR	The name of the object on which you want to query HISTORICAL data. After running audit on a table, you can drop the table. In this case, object_name becomes NULL.
LICENSE_NAME	VARCHAR	The name of the license. After running a compliance audit, the value for this column is always vertica .
AUDIT_START_TIMESTAMP	TIMESTAMPTZ	When the audit started
AUDIT_END_TIMESTAMP	TIMESTAMPTZ	When the audit finished
CONFIDENCE_LEVEL_PERCENT	FLOAT	The confidence level of the size estimate
ERROR_TOLERANCE_PERCENT	FLOAT	The error tolerance used for the size estimate
USED_SAMPLING	BOOLEAN	Whether data was randomly sampled (if false, all of the data was analyzed)
CONFIDENCE_INTERVAL_LOWER_BOUND_BYTES	INTEGER	The lower bound of the data size estimate within the confidence level
CONFIDENCE_INTERVAL_UPPER_BOUND_BYTES	INTEGER	The upper bound of the data size estimate within the confidence level
SAMPLE_COUNT	INTEGER	The number of data samples used to generate the estimate
CELL_COUNT	INTEGER	The number of cells in the database

USER_CLIENT_AUTH

Provides information about the client authentication methods that are associated with database users. You associate an authentication method with a user using GRANT (Authentication) .

Column Name	Data Type	Description
USER_OID	INTEGER	A unique identifier for that user.
USER_NAME	VARCHAR	Name of the user.
AUTH_OID	INTEGER	A unique identifier for the authentication method you are using.
AUTH_NAME	VARCHAR	Name that you gave to the authentication method.
GRANTED_TO	BOOLEAN	Name of the user with whom you have associated the authentication method using GRANT (Authentication) .

USER_CONFIGURATION_PARAMETERS

Provides information about user-level configuration parameters that are in effect for database users.

Column Name	Data Type	Description
USER_NAME	VARCHAR	Name of a database user with user-level settings.
PARAMETER_NAME	VARCHAR	The configuration parameter name.
CURRENT_ VALUE	VARCHAR	The parameter's current setting for this user.
DEFAULT_VALUE	VARCHAR	The parameter's default value.

Privileges

Superuser only

Examples

```
=> SELECT * FROM user_configuration_parameters;
 user_name |      parameter_name       | current_value | default_value
-----------+---------------------------+---------------+---------------
 Yvonne    | LoadSourceStatisticsLimit | 512           | 256
(1 row)

=> ALTER USER Ahmed SET DepotOperationsForQuery='FETCHES';
ALTER USER
=> ALTER USER Yvonne SET DepotOperationsForQuery='FETCHES';
ALTER USER
=> SELECT * FROM user_configuration_parameters;
 user_name |      parameter_name       | current_value | default_value
-----------+---------------------------+---------------+---------------
 Ahmed     | DepotOperationsForQuery   | FETCHES       | ALL
 Yvonne    | DepotOperationsForQuery   | FETCHES       | ALL
 Yvonne    | LoadSourceStatisticsLimit | 512           | 256
(3 rows)
```

USER_FUNCTION_PARAMETERS

Provides information about the parameters of a C++ user-defined function (UDx). You can only view parameters that have the Properties.visible parameter set to TRUE .

Column Name	Data Type	Description

SCHEMA_NAME	VARCHAR(128)	The schema to which the function belongs.
FUNCTION_NAME	VARCHAR(128)	The name assigned by the user to the user-defined function.
FUNCTION_TYPE	VARCHAR(128)	The type of user-defined function. For example, 'User Defined Function'.
FUNCTION_ARGUMENT_TYPE	VARCHAR(8192)	The number and data types of input arguments for the function.
PARAMETER_NAME	VARCHAR(128)	The name of the parameter for the user-defined function.
DATA_TYPE	VARCHAR(128)	The data type of the parameter.
DATA_TYPE_ID	INTEGER	A number specifying the ID for the parameter's data type.
DATA_TYPE_LENGTH	INTEGER	The maximum length of the parameter's data type.
IS_REQUIRED	BOOLEAN	Indicates whether the parameter is required or not. If set to TRUE, and you don't provide the parameter, Vertica throws an error.
CAN_BE_NULL	BOOLEAN	Indicates whether the parameter can be passed as a NULL value. If set to FALSE, you pass the parameter with a NULL value, Vertica throws an error.
COMMENT	VARCHAR(128)	A user-supplied description of the parameter.

Privileges

Any user can query the USER_FUNCTION_PARAMETERS table. However, users can only see table information about those UDx functions which the user has permission to use.

See also

- Developing user-defined extensions (UDxs)
- UDx parameters

USER_FUNCTIONS

Returns metadata about user-defined SQL functions (which store commonly used SQL expressions as a function in the Vertica catalog) and user-defined functions.

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	The name of the schema in which this function exists.
FUNCTION_NAME	VARCHAR	The name assigned by the user to the SQL function or user-defined function.
PROCEDURE_TYPE	VARCHAR	The type of user-defined function. For example, 'User Defined Function'.
FUNCTION_RETURN_TYPE	VARCHAR	The data type name that the SQL function returns.
FUNCTION_ARGUMENT_TYPE	VARCHAR	The number and data types of parameters for the function.
FUNCTION_DEFINITION	VARCHAR	The SQL expression that the user defined in the SQL function's function body.
VOLATILITY	VARCHAR	The SQL function's volatility (whether a function returns the same output given the same input). Can be immutable , volatile , or stable .

IS_STRICT	BOOLEAN	Indicates whether the SQL function is strict , where *t* is true and *f* is false.
IS_FENCED	BOOLEAN	Indicates whether the function runs in Fenced and unfenced modes or not.
COMMENT	VARCHAR	A comment about this function provided by the function creator.

Notes

- The volatility and strictness of a SQL function are automatically inferred from the function definition in order that Vertica determine the correctness of usage, such as where an immutable function is expected but a volatile function is provided.
- The volatility and strictness of a UDx is defined by the UDx's developer.

Examples

Create a SQL function called myzeroifnull in the public schema:

```
=> CREATE FUNCTION myzeroifnull(x INT) RETURN INT
  AS BEGIN
    RETURN (CASE WHEN (x IS NOT NULL) THEN x ELSE 0 END);
  END;
```

Now query the USER_FUNCTIONS table. The query returns just the myzeroifnull macro because it is the only one created in this schema:

```
=> SELECT * FROM user_functions;
-[ RECORD 1 ]----------+--------------------------------------------------
schema_name            | public
function_name          | myzeroifnull
procedure_type         | User Defined Function
function_return_type   | Integer
function_argument_type | x Integer
function_definition    | RETURN CASE WHEN (x IS NOT NULL) THEN x ELSE 0 END
volatility             | immutable
is_strict              | f
is_fenced              | f
comment                |
```

See also

- CREATE FUNCTION (SQL)
- ALTER FUNCTION (scalar)
- DROP FUNCTION

USER_PROCEDURES

Provides information about stored procedures and external procedures . Users can only view procedures that they can execute.

Column Name	Data Type	Description
PROCEDURE_NAME	VARCHAR	The name of the procedure.
OWNER	VARCHAR	The owner (definer) of the procedure.
LANGUAGE	VARCHAR	The language in which the procedure is defined. For external procedures, this will be EXTERNAL. For stored procedures, this will be one of the supported languages .

SECURITY	VARCHAR	The privileges to use when executing the procedure, one of the following: • DEFINER : Executes the procedure with the privileges of the owner (definer) of the procedure. • INVOKER : Executes the procedure with the privileges of the invoker. For details, see Executing stored procedures .
PROCEDURE_ARGUMENTS	VARCHAR	The arguments of the procedure.
SCHEMA_NAME	VARCHAR	The schema in which the procedure was defined.

Privileges

Non-superusers can only view information on a procedure if they have:

- USAGE privileges on the procedure's schema.
- Ownership or EXECUTE privileges on the procedure.

Examples

```
=> SELECT * FROM user_procedures;
  procedure_name   |  owner  | language | security |                        procedure_arguments                         | schema_name
-------------------+---------+----------+----------+--------------------------------------------------------------------+-------------
 accurate_auc      | dbadmin | PL/vSQL  | INVOKER  | relation varchar, observation_col varchar, probability_col varchar, epsilon float | public
 conditionalTable  | dbadmin | PL/vSQL  | INVOKER  | b boolean                                                          | public
 update_salary     | dbadmin | PL/vSQL  | INVOKER  | x int, y varchar                                                   | public
(3 rows)
```

USER_SCHEDULES

Contains information about schedules .

Column Name	Data Type	Description
SCHEDULE_NAME	VARCHAR	The name of the schedule.
SCHEMA_NAME	VARCHAR	The schedule's schema.
OWNER	VARCHAR	The owner of the schedule.
ATTACHED_TRIGGER	VARCHAR	The trigger attached to the schedule , if any.
ATTACHED_TRIGGER	BOOLEAN	Whether the schedule is enabled.
DATE_TIME_TYPE	VARCHAR	The format for the scheduled event, one of the following: • CRON • DATE_TIME_LIST
DATE_TIME_STRING	VARCHAR	The string used to schedule the event, one of the following: • A cron expression • A comma-separated list of timestamps

Examples

To view schedules:

```
=> SELECT * FROM user_schedules;
    schedule_name     | schema_name |  owner  |        attached_trigger        | enabled | date_time_type |                date_time_string
----------------------+-------------+---------+--------------------------------+---------+----------------+------------------------------------------------
daily_1am             | management  | dbadmin | log_user_actions               | t       | CRON           | 0 1 * * *
biannual_22_noon_gmt  | public      | dbadmin | refresh_logs                   | t       | DATE_TIME_LIST | 2022-01-01 12:00:00-00, 2022-06-01 12:00:00-00
```

USER_TRANSFORMS

Lists the currently-defined user-defined transform functions (UDTFs).

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR(128)	The name of the schema containing the UDTF.
FUNCTION_NAME	VARCHAR(128)	The SQL function name assigned by the user.
FUNCTION_RETURN_TYPE	VARCHAR(128)	The data types of the columns the UDTF returns.
FUNCTION_ARGUMENT_TYPE	VARCHAR(8192)	The data types of the columns that make up the input row.
FUNCTION_DEFINITION	VARCHAR(128)	A string containing the name of the factory class for the UDTF, and the name of the library that contains it.
IS_FENCED	BOOLEAN	Whether the UDTF runs in fenced mode.

Privileges

No explicit permissions are required; however, users see only UDTFs contained in schemas to which they have read access.

See also

- Transform functions (UDTFs)
- CREATE TRANSFORM FUNCTION

USERS

Provides information about all users in the database.

> **Tip**
> To see if a role has been assigned to a user, call the function HAS_ROLE .

Column Name	Data Type	Description
USER_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog, which identifies the user.
USER_NAME	VARCHAR	The user name for which information is listed.
IS_SUPER_USER	BOOLEAN	A system flag, where *t* (true) identifies the superuser created at the time of installation. All other users are denoted by *f* (false).
PROFILE_NAME	VARCHAR	The name of the profile to which the user is assigned. The profile controls the user's password policy.
IS_LOCKED	BOOLEAN	Whether the user's account is locked. A locked user cannot log into the system.
LOCK_TIME	TIMESTAMPTZ	When the user's account was locked. Used to determine when to automatically unlock the account, if the user's profile has a PASSWORD_LOCK_TIME parameter set.
RESOURCE_POOL	VARCHAR	The resource pool to which the user is assigned.

MANAGED_BY_OAUTH2_AUTH_ID	INTEGER	The ID of the OAuth authentication record used to authenticate and provision the user, if any.

LAST_LOGIN_TIME	TIMESTAMPTZ	The last time the user logged in.
		
		Note
		The LAST_LOGIN_TIME as recorded by the USERS system table is not persistent; if the database is restarted, the LAST_LOGIN_TIME for users created by just-in-time user provisioning is set to the database start time (this appears as an empty value in LAST_LOGIN_TIME).
		
		You can view the database start time by querying the DATABASES system table:
		`=> SELECT database_name, start_time FROM databases;`
		`database_name | start_time`
		`---------------+-------------------------------`
		`VMart | 2023-02-06 14:26:50.630054-05`
		`(1 row)`

MEMORY_CAP_KB	VARCHAR	The maximum amount of memory a query run by the user can consume, in kilobytes.
TEMP_SPACE_CAP_KB	VARCHAR	The maximum amount of temporary disk space a query run by the user can consume, in kilobytes.
RUN_TIME_CAP	VARCHAR	The maximum amount of time any of the user's queries are allowed to run.
MAX_CONNECTIONS	VARCHAR	The maximum number of connections allowed for this user.
CONNECTION_LIMIT_MODE	VARCHAR	Indicates whether the user sets connection limits through the node or in database mode.
IDLE_SESSION_TIMEOUT	VARCHAR	How the user handles idle session timeout limits, one of the following: • unlimited : There is no idle session time limit for the user. • default : The user's idle session time limit is the value of the DefaultIdleSessionTimeout database parameter, or if that parameter is not set or the user is a superuser, there is no timeout limit. To view the value of DefaultIdleSessionTimeout parameter, use the SHOW DATABASE statement: `=> SHOW DATABASE DEFAULT DEFAULTIDLESESSIONTIMEOUT;` • Interval literal : Interval after which the user's idle session is disconnected.
GRACE_PERIOD	VARCHAR	Specifies how long a user query can block on any session socket, while awaiting client input or output. If the socket is blocked for a continuous period that exceeds the grace period setting, the server shuts down the socket and throws a fatal error. The session is then terminated.
ALL_ROLES	VARCHAR	Roles assigned to the user. An asterisk in ALL_ROLES output means role granted WITH ADMIN OPTION. See Database Roles.
DEFAULT_ROLES	VARCHAR	Default roles assigned to the user. An asterisk in DEFAULT_ROLES output means role granted WITH ADMIN OPTION. See Enabling roles automatically .
SEARCH_PATH	VARCHAR	Sets the default schema search path for the user. See Setting search paths .
LDAP_DN	VARCHAR	Indicates whether or not the Vertica Analytic Database user maps to an LDAP Link user. When the column is set to dn , the Vertica user maps to LDAP Link..

LDAP_URI_HASH	INTEGER	The URI hash number for the LDAP user.
IS_ORPHANED_FROM_LDAP	BOOLEAN	Indicates if the user is disconnected (orphaned) from LDAP, set to one of the following: • t : User is orphaned • f : User is not orphaned For more information see Troubleshooting LDAP link issues

See also

- GRANTS
- HAS_ROLE

VIEW_COLUMNS

Provides view attribute information.

> **Note**
> If you drop a table that is referenced by a view, Vertica does not drop the view. However, attempts to access information about it from VIEW_COLUMNS return an error that the view is invalid.

Column Name	Data Type	Description
TABLE_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog that identifies this view.
TABLE_SCHEMA	VARCHAR	The name of this view's schema.
TABLE_NAME	VARCHAR	The view name.
COLUMN_ID	VARCHAR	A unique VARCHAR ID, assigned by the Vertica catalog, that identifies a column in this view.
COLUMN_NAME	VARCHAR	The name of a column in this view.
DATA_TYPE	VARCHAR	The data type of a view column.
DATA_TYPE_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog, which identifies a view column's data type.
DATA_TYPE_LENGTH	INTEGER	The data type's maximum length.
CHARACTER_MAXIMUM_LENGTH	INTEGER	The column's maximum length, valid only for character types.
NUMERIC_PRECISION	INTEGER	The column's number of significant decimal digits.
NUMERIC_SCALE	INTEGER	The column's number of fractional digits.
DATETIME_PRECISION	INTEGER	For TIMESTAMP data type, returns the declared precision; returns null if no precision was declared.
INTERVAL_PRECISION	INTEGER	The number of fractional digits retained in the seconds field.
ORDINAL_POSITION	INTEGER	The position of the column relative to other columns in the view.

See also

VIEWS

VIEW_TABLES

Shows details about view-related dependencies, including the table that reference a view, its schema, and owner.

Column Name	Data Type	Description
TABLE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theview.
TABLE_SCHEMA	VARCHAR	Name of the view schema.
TABLE_NAME	VARCHAR	Name of the view.
REFERENCE_TABLE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theview's source table.
REFERENCE_TABLE_SCHEMA	VARCHAR	Name of the view's source table schema.
REFERENCE_TABLE_NAME	VARCHAR	Name of the view's source table.
REFERENCE_TABLE_OWNER_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theview owner.

VIEWS

Provides information about all views within the system. See Views for more information.

Column Name	Data Type	Description
TABLE_SCHEMA_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theview schema.
TABLE_SCHEMA	VARCHAR	The name of the view schema.
TABLE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theview.
TABLE_NAME	VARCHAR	The view name.
OWNER_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theview owner.
OWNER_NAME	VARCHAR	View owner's user name
VIEW_DEFINITION	VARCHAR	The query that defines the view.
IS_SYSTEM_VIEW	BOOLEAN	Indicates whether the view is a system view.
SYSTEM_VIEW_CREATOR	VARCHAR	View creator's user name.
CREATE_TIME	TIMESTAMP	Specifies when this view was created.
IS_LOCAL_TEMP_VIEW	BOOLEAN	Indicates whether this view is a temporary view stored locally.
INHERIT_PRIVILEGES	BOOLEAN	Indicates whether inherited privileges are enabled for this view. For details, see Setting privilege inheritance on tables and views .

See also

VIEW_COLUMNS

V_MONITOR schema

The system tables in this section reside in the v_monitor schema. These tables provide information about the health of the Vertica database.

In this section

- ACTIVE_EVENTS
- ALLOCATOR_USAGE

- COLUMN_STORAGE
- COMMUNAL_CLEANUP_RECORDS
- COMMUNAL_TRUNCATION_STATUS
- CONFIGURATION_CHANGES
- CONFIGURATION_PARAMETERS
- CPU_USAGE
- CRITICAL_HOSTS
- CRITICAL_NODES
- CRITICAL_SUBCLUSTERS
- CURRENT_SESSION
- DATA_COLLECTOR
- DATA_READS
- DATABASE_BACKUPS
- DATABASE_CONNECTIONS
- DATABASE_MIGRATION_STATUS
- DELETE_VECTORS
- DEPLOY_STATUS
- DEPLOYMENT_PROJECTION_STATEMENTS
- DEPLOYMENT_PROJECTIONS
- DEPOT_EVICTIONS
- DEPOT_FETCH_QUEUE
- DEPOT_FETCHES
- DEPOT_FILES
- DEPOT_PIN_POLICIES
- DEPOT_SIZES
- DEPOT_UPLOADS
- DESIGN_QUERIES
- DESIGN_STATUS
- DESIGN_TABLES
- DESIGNS
- DISK_QUOTA_USAGES
- DISK_RESOURCE_REJECTIONS
- DISK_STORAGE
- DRAINING_STATUS
- ERROR_MESSAGES
- EVENT_CONFIGURATIONS
- EXECUTION_ENGINE_PROFILES
- EXTERNAL_TABLE_DETAILS
- HIVE_CUSTOM_PARTITIONS_ACCESSED
- HOST_RESOURCES
- IO_USAGE
- LDAP_LINK_DRYRUN_EVENTS
- LDAP_LINK_EVENTS
- LOAD_SOURCES
- LOAD_STREAMS
- LOCK_USAGE
- LOCKS
- LOGIN_FAILURES
- MEMORY_EVENTS
- MEMORY_USAGE
- MERGEOUT_PROFILES
- MODEL_STATUS_HISTORY
- MONITORING_EVENTS
- NETWORK_INTERFACES
- NETWORK_USAGE
- NODE_EVICTIONS
- NODE_RESOURCES
- NODE_STATES
- NOTIFIER_ERRORS
- OUTPUT_DEPLOYMENT_STATUS
- OUTPUT_EVENT_HISTORY

- PARTITION_COLUMNS
- PARTITION_REORGANIZE_ERRORS
- PARTITION_STATUS
- PARTITIONS
- PROCESS_SIGNALS
- PROJECTION_RECOVERIES
- PROJECTION_REFRESHES
- PROJECTION_STORAGE
- PROJECTION_USAGE
- QUERY_CONSUMPTION
- QUERY_EVENTS
- QUERY_METRICS
- QUERY_PLAN_PROFILES
- QUERY_PROFILES
- QUERY_REQUESTS
- REBALANCE_OPERATIONS
- REBALANCE_PROJECTION_STATUS
- REBALANCE_TABLE_STATUS
- RECOVERY_STATUS
- REMOTE_REPLICATION_STATUS
- REPARENTED_ON_DROP
- RESHARDING_EVENTS
- RESOURCE_ACQUISITIONS
- RESOURCE_POOL_MOVE
- RESOURCE_POOL_STATUS
- RESOURCE_QUEUES
- RESOURCE_REJECTION_DETAILS
- RESOURCE_REJECTIONS
- RESOURCE_USAGE
- SESSION_MARS_STORE
- SESSION_PARAMETERS
- SESSION_PROFILES
- SESSIONS
- SPREAD_STATE
- STORAGE_BUNDLE_INFO_STATISTICS
- STORAGE_CONTAINERS
- STORAGE_POLICIES
- STORAGE_TIERS
- STORAGE_USAGE
- STRATA
- STRATA_STRUCTURES
- SYSTEM
- SYSTEM_RESOURCE_USAGE
- SYSTEM_SERVICES
- SYSTEM_SESSIONS
- TABLE_RECOVERIES
- TABLE_RECOVERY_STATUS
- TABLE_STATISTICS
- TLS_CONFIGURATIONS
- TRANSACTIONS
- TRUNCATED_SCHEMATA
- TUNING_RECOMMENDATIONS
- TUPLE_MOVER_OPERATIONS
- UDFS_EVENTS
- UDFS_OPS_PER_HOUR
- UDFS_OPS_PER_MINUTE
- UDFS_STATISTICS
- UDX_EVENTS
- UDX_FENCED_PROCESSES
- USER_LIBRARIES
- USER_LIBRARY_MANIFEST

- USER_SESSIONS

ACTIVE_EVENTS

Returns all active events in the cluster. See Monitoring events .

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name where the event occurred.
EVENT_CODE	INTEGER	A numeric ID that indicates the type of event. See Event Types for a list of event type codes.
EVENT_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog, which identifies the specific event.
EVENT_SEVERITY	VARCHAR	The severity of the event from highest to lowest. These events are based on standard syslog severity types. • 0—Emergency • 1—Alert • 2—Critical • 3—Error • 4—Warning • 5—Notice • 6—Informational • 7—Debug
EVENT_POSTED_TIMESTAMP	TIMESTAMP	The year, month, day, and time the event was reported. The time is posted in military time.
EVENT_EXPIRATION	VARCHAR	The year, month, day, and time the event expire. The time is posted in military time. If the cause of the event is still active, the event is posted again.
EVENT_CODE_DESCRIPTION	VARCHAR	A brief description of the event and details pertinent to the specific situation.
EVENT_PROBLEM_DESCRIPTION	VARCHAR	A generic description of the event.
REPORTING_NODE	VARCHAR	The name of the node within the cluster that reported the event.
EVENT_SENT_TO_CHANNELS	VARCHAR	The event logging mechanisms that are configured for Vertica. These can include vertica.log , (configured by default) syslog, and SNMP.
EVENT_POSTED_COUNT	INTEGER	Tracks the number of times an event occurs. Rather than posting the same event multiple times, Vertica posts the event once and then counts the number of additional instances in which the event occurs.

ALLOCATOR_USAGE

Provides real-time information on the allocation and reuse of memory pools for a Vertica node.

There are two memory pools in Vertica, global and SAL. The global memory pool is related to Vertica catalog objects. The SAL memory pool is related to the system storage layer. These memory pools are physical structures from which Vertica allocates and reuses portions of memory.

Within the memory pools, there are two allocation types. Both global and SAL memory pools include chunk and object memory allocation types.

- *Chunk* allocations are from tiered storage, and are grouped into sizes, in bytes, that are powers of 2.
- *Object* allocations are object types, for example, a table or projection. Each object assumes a set size.

The table provides detailed information on these memory pool allocations.

Column Name	Data Type	Description

NODE_NAME	VARCHAR	The name of the node from which Vertica has collected this allocator information.
POOL_NAME	VARCHAR	One of two memory pools: • global: Memory pool is related to Vertica catalog objects. • SAL: Memory pool is related to the system storage layer.
ALLOCATION_TYPE	VARCHAR	One of two memory allocation types: • chunk: Chunk allocations are grouped into sizes that are powers of 2. • object: Object allocations assume a set amount of memory based upon the specific object.
UNIT_SIZE	INTEGER	The size, in bytes, of the memory allocation. For example, if the allocation type is a table (an object type), then Vertica allots 8 bytes.
FREE_COUNT	INTEGER	Indicates the count of blocks of freed memory that Vertica has reserved for future memory needs. For example, if you delete a table, Vertica reserves the 8 bytes originally allotted for the table. The 8 bytes freed become 1 unit of memory that Vertica adds to this column.
FREE_BYTES	INTEGER	Indicates the number of freed memory bytes. For example, with a table deletion, Vertica adds 8 bytes to this column. **Note** Vertica does not release memory after originally allocating it, unless the node or database is restarted.
USED_COUNT	INTEGER	Indicates the count of in-use blocks for this allocation. For example, if your database includes two table objects, Vertica adds 2 to this column.
USED_BYTES	INTEGER	The number of bytes of in-use blocks of memory. For example, if your database includes two table objects, each of which assume 8 bytes, Vertica adds 16 to this column.
TOTAL_SIZE	INTEGER	Indicates the number of bytes that is the sum of all free and used memory.
CAPTURE_TIME	TIMESTAMPTZ	Indicates the current timestamp for when Vertica collected the for this table.
ALLOCATION_NAME	VARCHAR	Provides the name of the allocation type. • If the allocation is an object type, provides the name of the object. For example, CAT::Schema . Object types can also have the name internal , meaning that the object is an internal data structure. Those object types that are not internal are prefaced with either CAT or SAL . Those prefaced with CAT indicate memory from the global memory pool. SAL indicates memory from the system storage memory pool. • If the allocation type is chunk, indicates a power of 2 in this field to represent the number of bytes assumed by the chunk. For example, 2^5 .

Sample: how memory pool memory is allotted, retained, and freed

The following table shows sample column values based upon a hypothetical example. The sample illustrates how column values change based upon addition or deletion of a table object.

- When you add a table object (t1), Vertica assumes a UNIT_SIZE of 8 bytes, with a USED_COUNT of 1.
- When you add a second table object (t2), the USED_COUNT increases to 2. Since each object assumes 8 bytes, USED_BYTES increases to 16.
- When you delete one of the two table objects, Vertica USED_COUNT decreases to 1, and USED_BYTES decreases to 8. Since Vertica retains the memory for future use, FREE_BYTES increases to 8, and FREE_COUNT increases to 1.
- Finally, when you create a new table object (t3), Vertica frees the memory for reuse. FREE_COUNT and FREE_BYTES return to 0.

Column Names	Add One Table Object (t1)	Add a Second Table Object (t2)	Delete a Table Object (t2)	Create a New Table Object (t3)
NODE_NAME	v_vmart_node0001	v_vmart_node0001	v_vmart_node0001	v_vmart_node0001
POOL_NAME	global	global	global	global
ALLOCATION_TYPE	object	object	object	object
UNIT_SIZE	8	8	8	8
FREE_COUNT	0	0	1	0
FREE_BYTES	0	0	8	0
USED_COUNT	1	2	1	2
USED_BYTES	8	16	8	16
TOTAL_SIZE	8	16	16	16
CAPTURE_TIME	2017-05-24 13:28:07.83855-04	2017-05-24 14:16:04.480953-04	2017-05-24 14:16:32.077322-04	2017-05-24 14:17:07.320745-04
ALLOCATION_NAME	CAT::Table	CAT::Table	CAT::Table	CAT::Table

Examples

The following example shows one sample record for a chunk allocation type, and one for an object type.

```
=> \x
Expanded display is on.


=> select * from allocator_usage;
-[ RECORD 1 ]---+----------------------------
node_name       | v_vmart_node0004
pool_name       | global
allocation_type | chunk
unit_size       | 8
free_count      | 1069
free_bytes      | 8552
used_count      | 7327
used_bytes      | 58616
total_size      | 67168
capture_time    | 2017-05-24 13:28:07.83855-04
allocation_name | 2^3
.
.
.
-[ RECORD 105 ]-+----------------------------
node_name       | v_vmart_node0004
pool_name       | SAL
allocation_type | object
unit_size       | 128
free_count      | 0
free_bytes      | 0
used_count      | 2
used_bytes      | 256
total_size      | 256
capture_time    | 2017-05-24 14:44:30.153892-04
allocation_name | SAL::WOSAlloc
.
.
.
```

COLUMN_STORAGE

Returns the amount of disk storage used by each column of each projection on each node.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
COLUMN_ID	INTEGER	Catalog-assigned integer value that uniquely identifies thecolumn.
COLUMN_NAME	VARCHAR	The column name for which information is listed.
ROW_COUNT	INTEGER	The number of rows in the column.
USED_BYTES	INTEGER	The disk storage allocation of the column in bytes.
ENCODINGS	VARCHAR	The encoding type for the column.
COMPRESSION	VARCHAR	The compression type for the column. You can compare ENCODINGS and COMPRESSION columns to see how different encoding types affect column storage when optimizing for compression.
ROS_COUNT	INTEGER	The number of ROS containers.

PROJECTION_ID	INTEGER	Catalog-assigned numeric value that uniquely identifies the projection.
PROJECTION_NAME	VARCHAR	The associated projection name for the column.
PROJECTION_SCHEMA	VARCHAR	The name of the schema associated with the projection.
ANCHOR_TABLE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theanchor table.
ANCHOR_TABLE_NAME	VARCHAR	The associated table name.
ANCHOR_TABLE_SCHEMA	VARCHAR	The associated table's schema name.
ANCHOR_TABLE_COLUMN_ID	VARCHAR	Catalog-assigned VARCHAR value that uniquely identifies a table column.
ANCHOR_TABLE_COLUMN_NAME	VARCHAR	The name of the anchor table.

COMMUNAL_CLEANUP_RECORDS

Eon Mode only

This system table lists files that Vertica considers leaked on an Eon Mode communal storage. Leaked files are files that are detected as needing deletion but were missed by the normal cleanup mechanisms. This information helps you determine how much space on the communal storage you can reclaim or have reclaimed by cleaning up the leaked files.

Column Name	**Data Type**	**Description**
detection_timestamp	TIMESTAMPTZ	Timestamp at which the file was detected as leaked.
location_path	VARCHAR	The path of communal storage location.
file_name	VARCHAR	The name of the leaked file.
size_in_bytes	INTEGER	The size of the leaked file in bytes.
queued_for_delete	BOOLEAN	Specifies whether the file was queued for deletion. Files queued for deletion might not be deleted right away. Also, a subsequent call to clean_communal_storage reports these files as leaked if the files hadn't already been deleted.

Examples

```
=> SELECT clean_communal_storage('true');
                                                
```

USER_NAME	VARCHAR	Name of the user who changed configuration parameters at the time Vertica recorded the session.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
PARAMETER	VARCHAR	Name of the changed parameter. See Configuration parameter management for a detailed list of supported parameters.
VALUE	VARCHAR	New value of the configuration parameter.

Privileges

Superuser

CONFIGURATION_PARAMETERS

Provides information about all configuration parameters that are currently in use by the system.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node names of the database cluster. ALL indicates that all nodes have the same value.
PARAMETER_NAME	VARCHAR	The parameter name.
CURRENT_VALUE	VARCHAR	The parameter's current setting.
RESTART_VALUE	VARCHAR	The parameter's value after the next restart.
DATABASE_VALUE	VARCHAR	The value that is set at the database level. If no database-level value is set, the value reflects the default value.
DEFAULT_VALUE	VARCHAR	The parameter's default value.
CURRENT_LEVEL	VARCHAR	Level at which CURRENT_VALUE is set, one of the following: • NODE • DATABASE • SESSION • DEFAULT
RESTART_LEVEL	VARCHAR	Level at which the parameter will be set after the next restart, one of the following: • NODE • DATABASE • DEFAULT
IS_MISMATCH	BOOLEAN	Whether CURRENT_VALUE and RESTART_VALUE match.
GROUPS	VARCHAR	A group to which the parameter belongs—for example, OptVOptions. **Note** Most configuration parameters do not belong to any group.

ALLOWED_LEVELS	VARCHAR	Levels at which the specified parameter can be set, a comma-delimited list of any of the following values: • DATABASE • NODE • SESSION • USER
SUPERUSER_VISIBLE_ONLY	BOOLEAN	Whether non-superusers can view all parameter settings. If true, the following columns are masked to non-superusers: • CURRENT_VALUE • RESTART_VALUE • DATABASE_VALUE • DEFAULT_VALUE
CHANGE_UNDER_SUPPORT_GUIDANCE	BOOLEAN	Whether the parameter is intended for use only under guidance from Vertica technical support.
CHANGE_REQUIRES_RESTART	BOOLEAN	Whether the configuration change requires a restart.
DESCRIPTION	VARCHAR	Describes the parameter's usage.

Examples

The following example shows a case where the parameter requires a restart for the new setting to take effect:

```
=> SELECT * FROM CONFIGURATION_PARAMETERS WHERE parameter_name = 'RequireFIPS';
-[ RECORD 1 ]----------+---------------
node_name              | ALL
parameter_name         | RequireFIPS
current_value          | 0
restart_value          | 0
database_value         | 0
default_value          | 0
current_level          | DEFAULT
restart_level          | DEFAULT
is_mismatch            | f
groups                 |
allowed_levels         | DATABASE
superuser_visible_only | f
change_under_support_guidance | f
change_requires_restart | t
description            | Execute in FIPS mode
```

The following example shows a case where a non-superuser is viewing a parameter with restricted visibility:

```
=> \c VMart nonSuperuser
You are now connected to database "VMart" as user "nonSuperuser".
=> SELECT * FROM CONFIGURATION_PARAMETERS WHERE superuser_visible_only = 't';
-[ RECORD 1 ]-----------+-----------------------------------------------
node_name               | ALL
parameter_name          | S3BucketCredentials
current_value           | ********
restart_value           | ********
database_value          | ********
default_value           | ********
current_level           | DEFAULT
restart_level           | DEFAULT
is_mismatch             | f
groups                  |
allowed_levels          | SESSION, DATABASE
superuser_visible_only  | t
change_under_support_guidance | f
change_requires_restart | f
description             | JSON list mapping S3 buckets to specific credentials.
```

See also

Configuration parameter management

CPU_USAGE

Records CPU usage history on the system.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
START_TIME	TIMESTAMP	Beginning of history interval.
END_TIME	TIMESTAMP	End of history interval.
AVERAGE_CPU_USAGE_PERCENT	FLOAT	Average CPU usage in percent of total CPU time (0-100) during history interval.

Privileges

Superuser

CRITICAL_HOSTS

Lists the critical hosts whose failure would cause the database to become unsafe and force a shutdown.

Column Name	Data Type	Description
HOST_NAME	VARCHAR	Name of a critical host

Privileges

None

CRITICAL_NODES

Lists the critical nodes whose failure would cause the database to become unsafe and force a shutdown.

Column Name	Data Type	Description
NODE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies thenode.
NODE_NAME	VARCHAR	Name of a critical node.

CRITICAL_SUBCLUSTERS

Lists the primary subclusters whose loss would cause the database to become unsafe and force it to shutdown. Vertica checks this table before stopping a subcluster to ensure it will not trigger a database shutdown. If you attempt to stop or remove a subcluster in this table, Vertica returns an error message. See Starting and stopping subclusters for more information.

This table only has contents when the database is in Eon Mode and when one or more subclusters are critical.

Column Name	Data Type	Description
SUBCLUSTER_OID	INTEGER	Unique identifier for the subcluster.
SUBCLUSTER_NAME	VARCHAR	The name of the subcluster in a critical state.

Examples

```
=> SELECT * FROM critical_subclusters;
 subcluster_oid   |  subcluster_name
------------------+--------------------
 45035996273704996 | default_subcluster
(1 row)
```

See also

- SUBCLUSTERS
- ALTER SUBCLUSTER
- DEMOTE_SUBCLUSTER_TO_SECONDARY
- PROMOTE_SUBCLUSTER_TO_PRIMARY
- SHUTDOWN_SUBCLUSTER

CURRENT_SESSION

Returns information about the current active session. Use this table to find out the current session's sessionID and get the duration of the previously-run query.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node for which information is listed
USER_NAME	VARCHAR	Name used to log into the database, NULL if the session is internal
CLIENT_OS_HOSTNAME	VARCHAR	The hostname of the client as reported by their operating system.
CLIENT_HOSTNAME	VARCHAR	The IP address and port of the TCP socket from which the client connection was made; NULL if the session is internal. Vertica accepts either IPv4 or IPv6 connections from a client machine. If the client machine contains mappings for both IPv4 and IPv6, the server randomly chooses one IP address family to make a connection. This can cause the CLIENT_HOSTNAME column to display either IPv4 or IPv6 values, based on which address family the server chooses.

TYPE	INTEGER	Identifies the session type, one of the following integer values: • 1: Client • 2: DBD • 3: Merge out • 4: Move out • 5: Rebalance cluster • 6: Recovery • 7: Refresh • 8: Shutdown • 9: License audit • 10: Timer service • 11: Connection • 12: VSpread • 13: Sub-session • 14: Repartition table
CLIENT_PID	INTEGER	Process identifier of the client process that issued this connection. This process might be on a different machine than the server.
LOGIN_TIMESTAMP	TIMESTAMP	When the user logged into the database or the internal session was created. This column can help identify open sessions that are idle.
SESSION_ID	VARCHAR	Identifier required to close or interrupt a session. This identifier is unique within the cluster at any point in time, but can be reused when the session closes.
CLIENT_LABEL	VARCHAR	User-specified label for the client connection that can be set when using ODBC. See Label in ODBC DSN connection properties .
TRANSACTION_START	TIMESTAMP	When the current transaction started, NULL if no transaction is running
TRANSACTION_ID	VARCHAR	Hexadecimal identifier of the current transaction, NULL if no transaction is in progress
TRANSACTION_DESCRIPTION	VARCHAR	Description of the current transaction
STATEMENT_START	TIMESTAMP	When the current statement started execution, NULL if no statement is running
STATEMENT_ID	VARCHAR	Unique numeric ID for the currently-running statement, NULL if no statement is being processed. Combined, TRANSACTION_ID and STATEMENT_ID uniquely identify a statement within a session.
LAST_STATEMENT_DURATION_US	INTEGER	Duration in microseconds of the last completed statement
CURRENT_STATEMENT	VARCHAR	The currently-running statement, if any. NULL indicates that no statement is currently being processed.
LAST_STATEMENT	VARCHAR	NULL if the user has just logged in, otherwise the currently running statement or most recently completed statement.
EXECUTION_ENGINE_PROFILING_CONFIGURATION	VARCHAR	See Profiling Settings below.
QUERY_PROFILING_CONFIGURATION	VARCHAR	See Profiling Settings below.
SESSION_PROFILING_CONFIGURATION	VARCHAR	See Profiling Settings below.

CLIENT_TYPE	VARCHAR	Type of client from which the connection was made, one of the following: • ADO.NET Driver • ODBC Driver • JDBC Driver • vsql
CLIENT_VERSION	VARCHAR	Client version
CLIENT_OS	VARCHAR	Client operating system
CLIENT_OS_USER_NAME	VARCHAR	Identifies the user that logged into the database, also set for unsuccessful login attempts.
REQUESTED_PROTOCOL	VARCHAR	Communication protocol version that the ODBC client driver sends to Vertica server, used to support backward compatibility with earlier server versions.
EFFECTIVE_PROTOCOL	VARCHAR	Minimum protocol version supported by client and driver.

Profiling settings

The following columns show settings for different profiling categories:

- EXECUTION_ENGINE_PROFILING_CONFIGURATION
- QUERY_PROFILING_CONFIGURATION
- SESSION_PROFILING_CONFIGURATION

These can have the following values:

- Empty: No profiling is set
- Session : On for current session.
- Global : On by default for all sessions.
- > Session , Global : On by default for all sessions, including current session.

For information about controlling profiling settings, see Enabling profiling .

DATA_COLLECTOR

Shows settings for all Data collector components: their current retention policy properties and other data collection statistics.

Data Collector is on by default. To turn it off, set configuration parameter EnableDataCollector to 0.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node name on which data is stored.
COMPONENT	VARCHAR	Name of the component.
TABLE_NAME	VARCHAR	The data collector table name for which information is listed.
DESCRIPTION	VARCHAR	Short description about the component.
ACCESS_RESTRICTED	BOOLEAN	Indicates whether access to the table is restricted to the DBADMIN, PSEUDOSUPERUSER, or SYSMONITOR roles.
MEMORY_BUFFER_SIZE_KB	INTEGER	Specifies in kilobytes the maximum amount of data that is buffered in memory before moving it to disk. You can modify this value with SET_DATA_COLLECTOR_POLICY .

DISK_SIZE_KB	INTEGER	Specifies in kilobytes the maximum disk space allocated for this component's Data Collector table. If set to 0, the Data Collector retains only as much component data as it can buffer in memory, as specified by MEMORY_BUFFER_SIZE_KB. You can modify this value with SET_DATA_COLLECTOR_POLICY .
INTERVAL_SET	BOOLEAN	Boolean, specifies whether time-based retention is enabled (INTERVAL_TIME is ≥ 0).
INTERVAL_TIME	INTERVAL	INTERVAL data type that specifies how long data of a given component is retained in that component's Data Collector table. You can modify this value with SET_DATA_COLLECTOR_POLICY or SET_DATA_COLLECTOR_TIME_POLICY . For example, if you specify component TupleMoverEvents and set interval-time to an interval of two days ('2 days'::interval), the Data Collector table dc_tuple_mover_events retains records of Tuple Mover activity over the last 48 hours. Older Tuple Mover data are automatically dropped from this table. **Note** Setting a component's policy's interval_time property has no effect on how much data storage the Data Collector retains on disk for that component. Maximum disk storage capacity is determined by the disk_size_kb property. Setting the interval_time property only affects how long data is retained by the component's Data Collector table. For details, see Configuring data retention policies .
RECORD_TOO_BIG_ERRORS	INTEGER	Integer that increments by one each time an error is thrown because data did not fit in memory (based on the data collector retention policy).
LOST_BUFFERS	INTEGER	Number of buffers lost.
LOST_RECORDS	INTEGER	Number of records lost.
RETIRED_FILES	INTEGER	Number of retired files.
RETIRED_RECORDS	INTEGER	Number of retired records.
CURRENT_MEMORY_RECORDS	INTEGER	The current number of rows in memory.
CURRENT_DISK_RECORDS	INTEGER	The current number of rows stored on disk.
CURRENT_MEMORY_BYTES	INTEGER	Total current memory used in kilobytes.
CURRENT_DISK_BYTES	INTEGER	Total current disk space used in kilobytes.
FIRST_TIME	TIMESTAMP	Timestamp of the first record.
LAST_TIME	TIMESTAMP	Timestamp of the last record
KB_PER_DAY	FLOAT	Total kilobytes used per day.

Examples

Get the current status of resource pools:

```
=> SELECT * FROM data_collector WHERE component = 'ResourcePoolStatus' ORDER BY node_name;
-[ RECORD 1 ]----------+---------------------------------
node_name              | v_vmart_node0001
component              | ResourcePoolStatus
table_name             | dc_resource_pool_status
description            | Resource Pool status information
access_restricted      | t
```

```
memory_buffer_size_kb  | 64
disk_size_kb           | 25600
interval_set           | f
interval_time          | 0
record_too_big_errors  | 0
lost_buffers           | 0
lost_records           | 0
retired_files          | 385
retired_records        | 3492335
current_memory_records | 0
current_disk_records   | 30365
current_memory_bytes   | 0
current_disk_bytes     | 21936993
first_time             | 2020-08-14 11:03:28.007894-04
last_time              | 2020-08-14 11:59:41.005675-04
kb_per_day             | 548726.098227313
-[ RECORD 2 ]----------+---------------------------------
node_name              | v_vmart_node0002
component              | ResourcePoolStatus
table_name             | dc_resource_pool_status
description            | Resource Pool status information
access_restricted      | t
memory_buffer_size_kb  | 64
disk_size_kb           | 25600
interval_set           | f
interval_time          | 0
record_too_big_errors  | 0
lost_buffers           | 0
lost_records           | 0
retired_files          | 385
retired_records        | 3492335
current_memory_records | 0
current_disk_records   | 28346
current_memory_bytes   | 0
current_disk_bytes     | 20478345
first_time             | 2020-08-14 11:07:12.006484-04
last_time              | 2020-08-14 11:59:41.004825-04
kb_per_day             | 548675.811828872
-[ RECORD 3 ]----------+---------------------------------
node_name              | v_vmart_node0003
component              | ResourcePoolStatus
table_name             | dc_resource_pool_status
description            | Resource Pool status information
access_restricted      | t
memory_buffer_size_kb  | 64
disk_size_kb           | 25600
interval_set           | f
interval_time          | 0
record_too_big_errors  | 0
lost_buffers           | 0
lost_records           | 0
retired_files          | 385
retired_records        | 3492335
current_memory_records | 0
current_disk_records   | 28337
current_memory_bytes   | 0
current_disk_bytes     | 20471843
first_time             | 2020-08-14 11:07:13.008246-04
last_time              | 2020-08-14 11:59:41.006729-04
kb_per_day             | 548675.63541403
```

See also

- Configuring data retention policies
- Data collector utility

DATA_READS

Eon Mode only

Lists each storage location that a query reads in Eon Mode. If the query fetches data from multiple locations, this table provides a row for each location per node that read data. For example, a query might run on three nodes and fetch data from the depot and communal storage. In this case, the table displays six rows for the query: three rows for each node's depot read, and three for each node's communal storage read.

> **Note**
> This table is only populated in Eon Mode.

Column Name	Column Type	Description
START_TIME	TIMESTAMP	When Vertica started reading data from the location.
NODE_NAME	VARCHAR	Name of the node that fetched the data
SESSION_ID	VARCHAR	Unique numeric ID assigned by the Vertica catalog, which identifies the session for which profiling information is captured. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
USER_ID	INT	Unique numeric ID assigned by the Vertica catalog, which identifies the user.
USER_NAME	VARCHAR	Name of the user running the query.
TRANSACTION_ID	INT	Identifier for the transaction within the session, if any. If a session is active but no transaction has begun, TRANSACTION_ID returns NULL.
STATEMENT_ID	INT	Unique numeric ID for the statement that read the data. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID and STATEMENT_ID uniquely identifies a statement within a session; these columns are useful for creating joins with other system tables.
REQUEST_ID	INT	ID of the data request.
LOCATION_ID	INT	ID of the storage location read.
LOCATION_PATH	VARCHAR	Path of the storage container read by the query.
BYTES_READ	INT	Number of bytes read by the query from this location.

Examples

```
=> SELECT * FROM V_MONITOR.DATA_READS WHERE TRANSACTION_ID = 45035996273707457;
-[ RECORD 1 ]--+-------------------------------------------------
start_time     | 2019-02-13 19:43:43.840836+00
node_name      | v_verticadb_node0001
session_id     | v_verticadb_node0001-11828:0x6f3
user_id        | 45035996273704962
user_name      | dbadmin
transaction_id | 45035996273707457
statement_id   | 1
request_id     | 230
location_id    | 45035996273835000
location_path  | /vertica/data/verticadb/v_verticadb_node0001_depot
bytes_read     | 329460142
-[ RECORD 2 ]--+-------------------------------------------------
start_time     | 2019-02-13 19:43:43.8421+00
node_name      | v_verticadb_node0002
session_id     | v_verticadb_node0001-11828:0x6f3
user_id        | 45035996273704962
user_name      | dbadmin
transaction_id | 45035996273707457
statement_id   | 1
request_id     | 0
location_id    | 45035996273835002
location_path  | /vertica/data/verticadb/v_verticadb_node0002_depot
bytes_read     | 329473033
-[ RECORD 3 ]--+-------------------------------------------------
start_time     | 2019-02-13 19:43:43.841845+00
node_name      | v_verticadb_node0003
session_id     | v_verticadb_node0001-11828:0x6f3
user_id        | 45035996273704962
user_name      | dbadmin
transaction_id | 45035996273707457
statement_id   | 1
request_id     | 0
location_id    | 45035996273835004
location_path  | /vertica/data/verticadb/v_verticadb_node0003_depot
bytes_read     | 329677294
```

DATABASE_BACKUPS

Lists historical information for each backup that successfully completed after running the vbr utility. This information is useful for determining whether to create a new backup before you advance the AHM . Because this system table displays historical information, its contents do not always reflect the current state of a backup repository. For example, if you delete a backup from a repository, the DATABASE_BACKUPS system table continues to display information about it.

To list existing backups, run vbr as described in Viewing backups .

Column Name	Data Type	Description
BACKUP_TIMESTAMP	TIMESTAMP	The timestamp of the backup.
NODE_NAME	VARCHAR	The name of the initiator node that performed the backup.
SNAPSHOT_NAME	VARCHAR	The name of the backup, as specified in the snapshotName parameter of the vbr configuration file.
BACKUP_EPOCH	INTEGER	The database epoch at which the backup was saved.
NODE_COUNT	INTEGER	The number of nodes backed up in the completed backup, and as listed in the [Mapping *n*] sections of the configuration file.

OBJECTS	VARCHAR	The name of the object(s) contained in an object-level backup. This column is empty if the record is for a full cluster backup.
FILE_SYSTEM_TYPE	VARCHAR	The type of file system, such as Linux.

Privileges

Superuser

DATABASE_CONNECTIONS

Lists the connections to other databases for importing and exporting data. See Moving Data Between Vertica Databases .

Column Name	Data Type	Description
DATABASE	VARCHAR	The name of the connected database
USERNAME	VARCHAR	The username used to create the connection
HOST	VARCHAR	The host name used to create the connection
PORT	VARCHAR	The port number used to create the connection
ISVALID	BOOLEAN	Whether the connection is still open and usable or not

Examples

```
=> CONNECT TO VERTICA vmart USER dbadmin PASSWORD '' ON '10.10.20.150',5433;
CONNECT
=> SELECT * FROM DATABASE_CONNECTIONS;
 database | username |     host     | port | isvalid
----------+----------+--------------+------+---------
 vmart    | dbadmin  | 10.10.20.150 | 5433 | t
(1 row)
```

DATABASE_MIGRATION_STATUS

Provides real-time and historical data on Enterprise-to-Eon database migration attempts.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of a node in the source Enterprise database.
TRANSACTION_ID	VARCHAR	Hexadecimal identifier of the migration process transaction.
PHASE	VARCHAR	A stage of database migration on a given node, one of the following, listed in order of execution: • Catalog Conversion: Conversion of enterprise-mode catalog to Eon-compatible catalog. **Note** No data is transferred during this phase, so BYTES_TO_TRANSFER and BYTES_TRANSFERED are always set to 0. • Data Transfer: Transfer of data files and library files to communal storage • Catalog Transfer: Includes transfer of checkpoint and transaction log files.

STATUS	VARCHAR	Specifies status of a given phase, one of the following: • RUNNING • COMPLETED • ABORT ABORT indicates a given migration phase was unable to complete—for example, the client disconnected, or a network outage occurred—and the migration returned with an error. In this case, call MIGRATE_ENTERPRISE_TO_EON again to restart migration. For details, see Handling Interrupted Migration .
BYTES_TO_TRANSFER	INTEGER	For each migration phase, the size of data to transfer to communal storage, set when phase status is RUNNING: • Catalog Conversion: 0 • Data Transfer: Size of data files and library files • Catalog Transfer: Size of transaction logs
BYTES_TRANSFERRED	INTEGER	For each migration phase, the size of data transfered to communal storage. This value is updated while phase status is RUNNING, and set to the total number of bytes transferred when status is COMPLETED: • Catalog Conversion: 0 • Data Transfer: Size of data files and library files • Catalog Transfer: Size of transaction logs
COMMUNAL_STORAGE_LOCATION	VARCHAR	URL of targeted communal storage location
START_TIME	TIMESTAMP	Demarcate the start and end of each PHASE-specified migration operation.
END_TIME		

Privileges

Superuser

Examples

The following example shows data of a migration that is in progress:

```
=> SELECT node_name, phase, status, bytes_to_transfer, bytes_transferred, communal_storage_location FROM database_migration_status ORDER BY
node_name, start_time;
    node_name    |       phase        |  status   | bytes_to_transfer | bytes_transferred | communal_storage_location
-----------------+--------------------+-----------+-------------------+-------------------+--------------------------
 v_vmart_node0001 | Catalog Conversion | COMPLETED |                 0 |                 0 | s3://verticadbbucket/
 v_vmart_node0001 | Data Transfer      | COMPLETED |              1134 |              1134 | s3://verticadbbucket/
 v_vmart_node0001 | Catalog Transfer   | COMPLETED |              3765 |              3765 | s3://verticadbbucket/
 v_vmart_node0002 | Catalog Conversion | COMPLETED |                 0 |                 0 | s3://verticadbbucket/
 v_vmart_node0002 | Data Transfer      | COMPLETED |              1140 |              1140 | s3://verticadbbucket/
 v_vmart_node0002 | Catalog Transfer   | COMPLETED |              3766 |              3766 | s3://verticadbbucket/
 v_vmart_node0003 | Catalog Conversion | COMPLETED |                 0 |                 0 | s3://verticadbbucket/
 v_vmart_node0003 | Data Transfer      | RUNNING   |           5272616 |            183955 | s3://verticadbbucket/
```

DELETE_VECTORS

Holds information on deleted rows to speed up the delete process.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The name of the node storing the deleted rows.

SCHEMA_NAME	VARCHAR	The name of the schema where the deleted rows are located.
PROJECTION_NAME	VARCHAR	The name of the projection where the deleted rows are located.
DV_OID	INTEGER	The unique numeric ID (OID) that identifies this delete vector.
STORAGE_OID	INTEGER	The unique numeric ID (OID) that identifies the storage container that holds the delete vector.
SAL_STORAGE_ID	VARCHAR	Unique hexadecimal numeric ID assigned by the Vertica catalog, which identifies the storage.
DELETED_ROW_COUNT	INTEGER	The number of rows deleted.
USED_BYTES	INTEGER	The number of bytes used to store the deletion.
START_EPOCH	INTEGER	The start epoch of the data in the delete vector.
END_EPOCH	INTEGER	The end epoch of the data in the delete vector.

Examples

After you delete data from a Vertica table, that data is marked for deletion. To see the data that is marked for deletion, query the DELETE_VECTORS system table.

Run PURGE to remove the delete vectors from ROS containers.

```
=> SELECT * FROM test1;
 number
--------
     3
    12
    33
    87
    43
    99
(6 rows)
=> DELETE FROM test1 WHERE number > 50;
 OUTPUT
--------
     2
(1 row)
=> SELECT * FROM test1;
 number
--------
    43
     3
    12
    33
(4 rows)
=> SELECT node_name, projection_name, deleted_row_count FROM DELETE_VECTORS;
    node_name     | projection_name | deleted_row_count
------------------+-----------------+-------------------
 v_vmart_node0002 | test1_b1        |                 1
 v_vmart_node0001 | test1_b1        |                 1
 v_vmart_node0001 | test1_b0        |                 1
 v_vmart_node0003 | test1_b0        |                 1
(4 rows)
=> SELECT PURGE();
...
(Table: public.test1) (Projection: public.test1_b0)
(Table: public.test1) (Projection: public.test1_b1)
...
(4 rows)
```

After the ancient history mark (AHM) advances:

```
=> SELECT * FROM DELETE_VECTORS;
 (No rows)
```

See also

- PURGE_PARTITION
- PURGE_PROJECTION
- PURGE_TABLE
- Purging deleted data

DEPLOY_STATUS

Records the history of deployed Database Designer designs and their deployment steps.

Column Name	Data Type	Description
EVENT_TIME	TIMESTAMP	Time when the row recorded the event.
USER_NAME	VARCHAR	Name of the user who deployed a design at the time Vertica recorded the session.
DEPLOY_NAME	VARCHAR	Name the deployment, same as the user-specified design name.

DEPLOY_STEP	VARCHAR	Steps in the design deployment.
DEPLOY_STEP_STATUS	VARCHAR	Textual status description of the current step in the deploy process.
DEPLOY_STEP_COMPLETE_PERCENT	FLOAT	Progress of current step in percentage (0–100).
DEPLOY_COMPLETE_PERCENT	FLOAT	Progress of overall deployment in percentage (0–100).
ERROR_MESSAGE	VARCHAR	Error or warning message during deployment.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

DEPLOYMENT_PROJECTION_STATEMENTS

Contains information about CREATE PROJECTION statements used to deploy a database design. Each row contains information about a different CREATE PROJECTION statement. The function DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY populates this table.

Column Name	Column Type	Description
DEPLOYMENT_ID	INTEGER	Unique ID that Database Designer assigned to the deployment.
DESIGN_NAME	VARCHAR	Unique name that the user assigned to the design.
DEPLOYMENT_PROJECTION_ID	INTEGER	Unique ID assigned to the output projection by Database Designer.
STATEMENT_ID	INTEGER	Unique ID assigned to the statement type that creates the projection.
STATEMENT	VARCHAR	Text for the statement that creates the projection.

DEPLOYMENT_PROJECTIONS

Contains information about projections created and dropped during the design. Each row contains information about a different projection. Database Designer populates this table after the design is deployed.

Column Name	Column Type	Description
deployment_id	INTEGER	Unique ID that Database Designer assigned to the deployment.
deployment_projection_id	INTEGER	Unique ID that Database Designer assigned to the output projection.
design_name	VARCHAR	Name of the design being deployed.
deployment_projection_name	VARCHAR	Name that Database Designer assigned to the projection.
anchor_table_schema	VARCHAR	Name of the schema that contains the table the projection is based on.
anchor_table_name	VARCHAR	Name of the table the projection is based on.
deployment_operation	VARCHAR	Action being taken on the projection, for example, add or drop.

deployment_projection_type	VARCHAR	Indicates whether Database Designer has proposed new projections for this design (DBD) or is using the existing catalog design (CATALOG). The REENCODED suffix indicates that the projection sort order and segmentation are the same, but the projection columns have new encodings: • DBD • CATALOG • DBD_REENCODED • CATALOG_REENCODED
deploy_weight	INTEGER	Weight of this projection in creating the design. This field is always 0 for projections that have been dropped.
estimated_size_on_disk	INTEGER	Approximate size of the projection on disk, in MB.

DEPOT_EVICTIONS

Eon Mode only

Records data on eviction of objects from the depot.

Column Name	Data Type	Description
START_TIME	TIMESTAMP	Demarcate the start and end of each depot eviction operation.
END_TIME		
NODE_NAME	VARCHAR	Name of a node where the eviction occurred.
SESSION_ID	VARCHAR	Unique numeric ID assigned by the Vertica catalog, which identifies the session for which profiling information is captured. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
USER_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the user.
USER_NAME	VARCHAR	The user who made changes to the depot.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any. If a session is active but no transaction has begun, TRANSACTION_ID returns NULL.
STATEMENT_ID	INTEGER	Unique numeric ID for the statement that caused the eviction. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID and STATEMENT_ID uniquely identifies a statement within a session; these columns are useful for creating joins with other system tables.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session.
STORAGE_ID	VARCHAR	Unique hexadecimal numeric ID assigned by the Vertica catalog, which identifies the storage.
STORAGE_OID	INTEGER	Numeric ID assigned by the Vertica catalog, which identifies the storage.
FILE_SIZE_BYTES	INTEGER	The size of the file in bytes that was evicted.
NUMBER_HITS	INTEGER	The number of times the file was accessed.
LAST_ACCESS_TIME	TIMESTAMP	The last time the file was read.

REASON	VARCHAR	The reason the file was evicted, one of the following: • DROP SUBSCRIPTION • CLEAR DEPOT • EVICTION DUE TO NEW • DROP OBJECT • LOAD • QUERY • PEER TO PEER FILL • DEPOT FILL AT STARTUP • DEPOT SIZE CHANGE
IS_PINNED	BOOLEAN	Specifies whether the file is pinned to this depot.

Examples

```
=> SELECT * FROM V_MONITOR.DEPOT_EVICTIONS LIMIT 2;
-[ RECORD 1 ]----+--------------------------------------------------
start_time       | 2019-02-20 15:32:26.765937+00
end_time         | 2019-02-20 15:32:26.766019+00
node_name        | v_verticadb_node0001
session_id       | v_verticadb_node0001-8997:0x3e
user_id          | 45035996273704962
user_name        | dbadmin
transaction_id   | 45035996273705450
statement_id     | 1
request_id       | 406
storage_id       | 0000000000000000000000000000000000a000000001fbf6
storage_oid      | 45035996273842065
file_size_bytes  | 61
number_hits      | 1
last_access_time | 2019-02-20 15:32:26.668094+00
reason           | DROP OBJECT
is_pinned        | f
-[ RECORD 2 ]----+--------------------------------------------------
start_time       | 2019-02-20 15:32:26.812803+00
end_time         | 2019-02-20 15:32:26.812866+00
node_name        | v_verticadb_node0001
session_id       | v_verticadb_node0001-8997:0x3e
user_id          | 45035996273704962
user_name        | dbadmin
transaction_id   | 45035996273705453
statement_id     | 1
request_id       | 409
storage_id       | 0000000000000000000000000000000000a000000001fbf6
storage_oid      | 45035996273842079
file_size_bytes  | 91
number_hits      | 1
last_access_time | 2019-02-20 15:32:26.770807+00
reason           | DROP OBJECT
is_pinned        | f
```

DEPOT_FETCH_QUEUE

Eon Mode only

Lists all pending depot requests for queried file data to fetch from communal storage.

Column Name	Data Type	Description

NODE_NAME	VARCHAR	Name of the node that requested the fetch.
SAL_STORAGE_ID	VARCHAR	Unique hexadecimal numeric ID assigned by the Vertica catalog, which identifies the storage.
TRANSACTION_ID	INTEGER	Identifies the transaction that contains the fetch-triggering query.
SOURCE_FILE_NAME	VARCHAR	Full path in communal storage of the file to fetch.
DESTINATION_FILE_NAME	VARCHAR	Destination path of the file to fetch.

Examples

```
=> \x
Expanded display is on.
=> SELECT * FROM depot_fetch_queue;
-[ RECORD 1 ]----------+-----------------------------------------------------------------
node_name              | v_example_db_node0002
sal_storage_id         | 029b6fac864e1982531dcde47d00edc500d000000001d5e7
transaction_id         | 45035996273705983
source_file_name       | s3://mydata/mydb/14a/029b6fac864e1982531dcde47d00edc500d000
                       000001d5e7_0.gt
destination_file_name  | /vertica/example_db/v_example_db_node0002_depot/747/029b6fac
                       864e1982531dcde47d00edc500d000000001d5eb_0.gt
-[ RECORD 2 ]----------+-----------------------------------------------------------------
node_name              | v_example_db_node0003
sal_storage_id         | 026635d69719c45e8d2d86f5a4d62c7b00b000000001d5e7
transaction_id         | 45035996273705983
source_file_name       | s3://mydata/mydb/4a5/029b6fac864e1982531dcde47d00edc500d0000
                       00001d5eb_0.gt
destination_file_name  | /vertica/example_db/v_example_db_node0002_depot/751/026635d6
                       9719c45e8d2d86f5a4d62c7b00b000000001d5e7_0.gt
```

DEPOT_FETCHES

Eon Mode only

Records data of depot fetch requests.

> **Note**
> Vertica reports all fetches to this table, including failed fetch attempts and their causes.

Column Name	Data Type	Description
START_TIME	TIMESTAMP	Demarcate the start and end of each depot fetch operation.
END_TIME		
NODE_NAME	VARCHAR	Identifies the node that initiated fetch request.
TRANSACTION_ID	INTEGER	Uniquely identifies the transaction of the query that required the fetched file.
STORAGE_ID	VARCHAR	Unique hexadecimal numeric ID assigned by the Vertica catalog, to identify the storage.
STORAGE_OID	INTEGER	Numeric ID assigned by the Vertica catalog, which identifies the storage.
FILE_SIZE_BYTES	INTEGER	Fetch size in bytes.
SOURCE_FILE	VARCHAR	Source file path used, set to null if the file was fetched from a peer.

DESTINATION_FILE	VARCHAR	Destination file path
SOURCE_NODE	VARCHAR	Source node from which the file was fetched, set to one of the following: • Name of the source node if file was fetched from a peer • Null if file was fetched from communal storage
IS_SUCCESSFUL	BOOLEAN	Boolean, specifies whether this fetch succeeded.
REASON	VARCHAR	Reason why the fetch failed, null if IS_SUCCESSFUL is true.

Examples

```
=> \x
Expanded display is on.

=> SELECT * FROM DEPOT_FETCHES LIMIT 2;
-[ RECORD 1 ]----+-----------------------------------------------------------------
start_time       | 2019-08-30 15:16:15.125962+00
end_time         | 2019-08-30 15:16:15.126791+00
node_name        | v_verticadb_node0001
transaction_id   | 45035996273706225
storage_id       | 0239ef74126e70db410b301610f1e5b500b0000000020d65
storage_oid      | 45035996273842065
file_size_bytes  | 53033
source_file      |
destination_file | /vertica/data/verticadb/v_verticadb_node0001_depot/957/0239e
                   f74126e70db410b301610f1e5b500b0000000020d65_0.gt
source_node      | v_verticadb_node0002
is_successful    | t
reason           |
-[ RECORD 2 ]----+-----------------------------------------------------------------
start_time       | 2019-08-30 15:16:15.285208+00
end_time         | 2019-08-30 15:16:15.285949+00
node_name        | v_verticadb_node0001
transaction_id   | 45035996273706234
storage_id       | 0239ef74126e70db410b301610f1e5b500b0000000020dc7
storage_oid      | 45035996273842075
file_size_bytes  | 69640
source_file      |
destination_file | /vertica/data/verticadb/v_verticadb_node0001_depot/055/0239e
                   f74126e70db410b301610f1e5b500b0000000020dc7_0.gt
source_node      | v_verticadb_node0002
is_successful    | t
reason           |

=> select node_name,transaction_id,storage_id,is_successful,reason FROM
   depot_fetches WHERE is_successful = 'f' LIMIT 3;
-[ RECORD 1 ]--+---------------------------------------------------
node_name      | v_verticadb_node0001
transaction_id | 45035996273721070
storage_id     | 0289281ac4c1f6580b95096fab25290800b0000000027d09
is_successful  | f
reason         | Could not create space in the depot
-[ RECORD 2 ]--+---------------------------------------------------
node_name      | v_verticadb_node0001
transaction_id | 45035996273721070
storage_id     | 0289281ac4c1f6580b95096fab25290800b0000000027d15
is_successful  | f
reason         | Could not create space in the depot
-[ RECORD 3 ]--+---------------------------------------------------
node_name      | v_verticadb_node0002
transaction_id | 45035996273721070
storage_id     | 02693f1c68266e38461084a840ee42aa00c0000000027d09
is_successful  | f
reason         | Could not create space in the depot
```

DEPOT_FILES

Eon Mode only

Lists all objects contained in all database depots.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The name of the node this file is on.
SAL_STORAGE_ID	VARCHAR	Unique hexadecimal numeric ID assigned by the Vertica catalog, which identifies the storage.
STORAGE_OID	INTEGER	Numeric ID assigned by the Vertica catalog, which identifies the storage.
COMMUNAL_FILE_PATH	VARCHAR	The path to the original file in communal storage. On AWS, this is an S3 URI.
DEPOT_FILE_PATH	VARCHAR	The path to the file in the depot.
SHARD_NAME	VARCHAR	The name of the shard this file is a part of.
STORAGE_TYPE	VARCHAR	The type of system storing this file.
NUMBER_OF_ACCESSES	INTEGER	The number of times this file has been accessed.
FILE_SIZE_BYTES	INTEGER	How large the file is in bytes.
LAST_ACCESS_TIME	TIMESTAMPTZ	A timestamp of when the file was last accessed.
ARRIVAL_TIME	TIMESTAMPTZ	When Vertica loaded the file into the depot.
SOURCE	VARCHAR	Where the file came from. One of the following: • LOAD • QUERY • PEER TO PEER FILL • DEPOT FILL AT STARTUP
IS_PINNED	BOOLEAN	Specifies whether the file is pinned to this depot.

Examples

```
=> \x
Expanded display is on.
=> SELECT * FROM depot_files LIMIT 2;

-[ RECORD 1 ]------+----------------------------------------------------------------
node_name          | v_verticadb_node0001
sal_storage_id     | 0275d4a7c99795d22948605e5940758900a000000001d1b1
storage_oid        | 45035996273842075
communal_file_path | s3://mybucket/myfolder/mydb/475/0275d4a7c99795d22948605e5940758900a000000001d1
b1/0275d4a7c99795d22948605e5940758900a000000001d1b1_
depot_file_path    | /vertica/data/verticadb/v_verticadb_node0001_depot/177/0275d4a7c99795d229486
05e5940758900a000000001d1b1/0275d4a7c99795d22948605e
shard_name         | replica
storage_type       | DFS
number_of_accesses | 0
file_size_bytes    | 456465645
last_access_time   | 2018-09-05 17:34:30.417274+00
arrival_time       | 2018-09-05 17:34:30.417274+00
source             | DEPOT FILL AT STARTUP
is_pinned          | f
-[ RECORD 2 ]------+----------------------------------------------------------------
node_name          | v_verticadb_node0001
sal_storage_id     | 0275d4a7c99795d22948605e5940758900a000000001d187
storage_oid        | 45035996273842079
communal_file_path | s3://mybucket/myfolder/mydb/664/0275d4a7c99795d22948605e5940758900a000000001d1
87/0275d4a7c99795d22948605e5940758900a000000001d187_
depot_file_path    | /vertica/data/verticadb/v_verticadb_node0001_depot/135/0275d4a7c99795d229486
05e5940758900a000000001d187/0275d4a7c99795d22948605e
shard_name         | replica
storage_type       | DFS
number_of_accesses | 0
file_size_bytes    | 40
last_access_time   | 2018-09-05 17:34:30.417244+00
arrival_time       | 2018-09-05 17:34:30.417244+00
source             | DEPOT FILL AT STARTUP
is_pinned          | f
```

DEPOT_PIN_POLICIES

Eon Mode only

Lists all objects —tables, projections, and table partitions—that are pinned to database depots.

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	Schema of the pinned object.
OBJECT_NAME	VARCHAR	Name of the pinned object.
POLICY_DETAILS	VARCHAR	Specifies the object type, one of the following: • Table • Projection • Partition
MIN_VAL	VARCHAR	If the pinned object is one or more contiguous table partitions, specifies the range of partition keys.
MAX_VAL		
LOCATION_LABEL		The depot's storage location label.

See also

- SET_DEPOT_PIN_POLICY_PARTITION
- SET_DEPOT_PIN_POLICY_PROJECTION
- SET_DEPOT_PIN_POLICY_TABLE
- Managing depot caching

DEPOT_SIZES

Eon Mode only

Reports depot caching capacity on Vertica nodes.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The name of the node containing the depot.
LOCATION_OID	INTEGER	Catalog-assigned integer value that uniquely identifies the storage location storing the depot.
LOCATION_PATH	VARCHAR	The path where the depot is stored.
LOCATION_LABEL	VARCHAR	The label associated with the depot's storage location.
MAX_SIZE_BYTES	INTEGER	The maximum size the depot can contain, in bytes.
CURRENT_USAGE_BYTES	INTEGER	The current size of the depot, in bytes.

Examples

```
=> \x
Expanded display is on.
=> SELECT * FROM Depot_Sizes;

-[ RECORD 1 ]-------+----------------------------------------------------

node_name           | v_verticadb_node0003
location_oid        | 45035996273823200
location_path       | /vertica/data/verticadb/v_verticadb_node0003_depot
location_label      | auto-data-depot
max_size_bytes      | 0
current_usage_bytes | 0

-[ RECORD 2 ]-------+----------------------------------------------------
node_name           | v_verticadb_node0001
location_oid        | 45035996273823196
location_path       | /vertica/data/verticadb/v_verticadb_node0001_depot
location_label      | auto-data-depot
max_size_bytes      | 33686316032
current_usage_bytes | 206801871

-[ RECORD 3 ]-------+----------------------------------------------------
node_name           | v_verticadb_node0002
location_oid        | 45035996273823198
location_path       | /vertica/data/verticadb/v_verticadb_node0002_depot
location_label      | auto-data-depot
max_size_bytes      | 0
current_usage_bytes | 0
```

DEPOT_UPLOADS

Eon Mode only

Lists details about depot uploads to communal storage.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The name of the node on which the depot resides.
PLAN_ID	VARCHAR	A unique node-specific numeric ID for each plan run by the Optimizer.
SUBMIT_TIME	TIMESTAMP	The time the task was submitted to the uploader.
START_TIME	TIMESTAMP	The time the upload started.
END_TIME	TIMESTAMP	The time the upload ended.
SOURCE_FILE	VARCHAR	The source file path used.
DESTINATION_FILE	VARCHAR	The destination file path.
FILE_SIZE_BYTES	INTEGER	The size of the uploaded file, in bytes.
MEMORY_USED_KB	INTEGER	The size of the uploader file buffer for the task. Valid for a task with a RUNNING or COMPLETED status. For a RUNNING status, this shows the current file buffer size, (whatever the uploader is using, which may grow over time for large uploads). For a COMPLETED status, this shows the largest size used in case the buffer grew during the upload.
STATUS	VARCHAR	The status of the task, valid values are: COMPLETED - the task has completed QUEUED - the task is still in the queue, but haven't been picked up by the uploader. RUNNING - the task is currently running and the corresponding file is uploading.

DESIGN_QUERIES

Contains info about design queries for a given design. The following functions populate this table:

- DESIGNER_ADD_DESIGN_QUERIES
- DESIGNER_ADD_DESIGN_QUERIES_FROM_RESULTS
- DESIGNER_ADD_DESIGN_QUERY

Column Name	Column Type	Description
DESIGN_ID	INTEGER	Unique id that Database Designer assigned to the design.
DESIGN_NAME	VARCHAR	Name that you specified for the design.
DESIGN_QUERY_ID	INTEGER	Unique id that Database Designer assigned to the design query.
DESIGN_QUERY_ID_INDEX	INTEGER	Database Designer chunks the query text if it exceeds the maximum attribute size before storing it in this table. Database Designer stored all chunks stored under the same value of DESIGN_QUERY_ID. DESIGN_QUERY_ID_INDEX keeps track of the order of the chunks, starting with 0 and ending in n, the index of the final chunk.
QUERY_TEXT	VARCHAR	Text of the query chunk, or the entire query text if it does not exceed the maximum attribute size.

WEIGHT	FLOAT	A value from 0 to 1 that indicates the importance of that query in creating the design. Assign a higher weight to queries that you run frequently so that Database Designer prioritizes those queries in creating the design. Default: 1.
DESIGN_QUERY_SEARCH_PATH	VARCHAR	The search path with which the query is to be parsed.
DESIGN_QUERY_SIGNATURE	INTEGER	Categorizes queries that affect the design that Database Designer creates in the same way. Database Designer assigns a signature to each query, weights one query for each signature group, depending on how many queries there are with that signature, and Database Designer considers that query when creating the design.

Example

Add queries to VMART_DESIGN and query the DESIGN_QUERIES table:

```
=> SELECT DESIGNER_ADD_DESIGN_QUERIES('VMART_DESIGN', '/tmp/examples/vmart_queries.sql','true');
 DESIGNER_ADD_DESIGN_QUERIES
-----------------------------
Number of accepted queries                    =9
Number of queries referencing non-design tables =0
Number of unsupported queries                 =0
Number of illegal queries                     =0
=> \x
Expanded display is on.
=> SELECT * FROM V_MONITOR.DESIGN.QUERIES
-[ RECORD 1 ]------------+-------------------
design_id                | 45035996273705090
design_name              | vmart_design
design_query_id          | 1
design_query_id_index    | 0
query_text               | SELECT fat_content
FROM (
SELECT DISTINCT fat_content
  FROM product_dimension
  WHERE department_description
  IN ('Dairy') ) AS food
  ORDER BY fat_content
  LIMIT 5;
weight                   | 1
design_query_search_path | v_dbd_vmart_design_vmart_design_ltt, "$user", public, v_catalog, v_monitor, v_internal
design_query_signature   | 45035996273724651

-[ RECORD 2]-------------+-------------------
design_query_id          | 2
design_query_id_index    | 0
query_text               | SELECT order_number, date_ordered
        FROM store.store_orders_fact orders
        WHERE orders.store_key IN (
        SELECT store_key
        FROM store.store_dimension
        WHERE store_state = 'MA')
        AND orders.vendor_key NOT IN (
        SELECT vendor_key
        FROM public.vendor_dimension
        WHERE vendor_state = 'MA')
        AND date_ordered < '2012-03-01';

weight                   | 1
design_query_search_path | v_dbd_vmart_design_vmart_design_ltt, "$user", public, v_catalog, v_monitor, v_internal
design_query_signature   | 45035996273724508
-[ RECORD 3]-------------+-------------------
...
```

DESIGN_STATUS

Records the progress of a running Database Designer design or history of the last Database Designer design executed by the current user.

Column Name	Data Type	Description
EVENT_TIME	TIMESTAMP	Time when the row recorded the event.
USER_NAME	VARCHAR	Name of the user who ran a design at the time Vertica recorded the session.
DESIGN_NAME	VARCHAR	Name of the user-specified design.

DESIGN_PHASE	VARCHAR	Phase of the design.
PHASE_STEP	VARCHAR	Substep in each design phase
PHASE_STEP_COMPLETE_PERCENT	FLOAT	Progress of current substep in percentage (0–100).
PHASE_COMPLETE_PERCENT	FLOAT	Progress of current design phase in percentage (0–100).

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

Examples

The following example shows the content of the DESIGN_STATUS table of a complete Database Designer run:

```
=> SELECT event_time, design_name, design_phase, phase_complete_percent
   FROM v_monitor.design_status;
event_time          | design_name | design_phase                                | phase_complete_percent
--------------------+-------------+---------------------------------------------+-----------------------
2012-02-14 10:31:20 | design1     | Design started                              |
2012-02-14 10:31:21 | design1     | Design in progress: Analyze statistics phase   |
2012-02-14 10:31:21 | design1     | Analyzing data statistics                   | 33.33
2012-02-14 10:31:22 | design1     | Analyzing data statistics                   | 66.67
2012-02-14 10:31:24 | design1     | Analyzing data statistics                   | 100
2012-02-14 10:31:25 | design1     | Design in progress: Query optimization phase   |
2012-02-14 10:31:25 | design1     | Optimizing query performance                | 37.5
2012-02-14 10:31:31 | design1     | Optimizing query performance                | 62.5
2012-02-14 10:31:36 | design1     | Optimizing query performance                | 75
2012-02-14 10:31:39 | design1     | Optimizing query performance                | 87.5
2012-02-14 10:31:41 | design1     | Optimizing query performance                | 87.5
2012-02-14 10:31:42 | design1     | Design in progress: Storage optimization phase |
2012-02-14 10:31:44 | design1     | Optimizing storage footprint                | 4.17
2012-02-14 10:31:44 | design1     | Optimizing storage footprint                | 16.67
2012-02-14 10:32:04 | design1     | Optimizing storage footprint                | 29.17
2012-02-14 10:32:04 | design1     | Optimizing storage footprint                | 31.25
2012-02-14 10:32:05 | design1     | Optimizing storage footprint                | 33.33
2012-02-14 10:32:05 | design1     | Optimizing storage footprint                | 35.42
2012-02-14 10:32:05 | design1     | Optimizing storage footprint                | 37.5
2012-02-14 10:32:05 | design1     | Optimizing storage footprint                | 62.5
2012-02-14 10:32:39 | design1     | Optimizing storage footprint                | 87.5
2012-02-14 10:32:39 | design1     | Optimizing storage footprint                | 87.5
2012-02-14 10:32:41 | design1     | Optimizing storage footprint                | 100
2012-02-14 10:33:12 | design1     | Design completed successfully               |
(24 rows)
```

DESIGN_TABLES

Contains information about all the design tables for all the designs for which you are the owner. Each row contains information about a different design table. Vertica creates this table when you run DESIGNER_CREATE_DESIGN .

Column Name	Column Type	Description
DESIGN_NAME	VARCHAR	Unique name that the user specified for the design.
DESIGN_TABLE_ID	INTEGER	Unique ID that Database Designer assigned to the design table.
TABLE_SCHEMA	VARCHAR	Name of the schema that contains the design table.
TABLE_ID	INTEGER	System object identifier (OID) assigned to the design table.

TABLE_NAME	VARCHAR	Name of the design table.

Example

Add all the tables from the VMart database to the design VMART_DESIGN. This operation populates the DESIGN_TABLES table:

```
=> SELECT DESIGNER_ADD_DESIGN_TABLES('VMART_DESIGN','online_sales.*');
DESIGNER_ADD_DESIGN_TABLES
----------------------------
 3
(1 row)
=> SELECT DESIGNER_ADD_DESIGN_TABLES('VMART_DESIGN','public.*');
DESIGNER_ADD_DESIGN_TABLES
----------------------------
 9
(1 row)
=> SELECT DESIGNER_ADD_DESIGN_TABLES('VMART_DESIGN','store.*');
DESIGNER_ADD_DESIGN_TABLES
----------------------------
 3
(1 row)
=> SELECT * FROM DESIGN_TABLES;
design_name  | design_table_id | table_schema |     table_id      |      table_name
-------------+-----------------+--------------+-------------------+----------------------
VMART_DESIGN |               1 | online_sales | 45035996373718754 | online_page_dimension
VMART_DESIGN |               2 | online_sales | 45035996373718758 | call_center_dimension
VMART_DESIGN |               3 | online_sales | 45035996373718762 | online_sales_fact
VMART_DESIGN |               4 | public       | 45035996373718766 | customer_dimension
VMART_DESIGN |               5 | public       | 45035996373718770 | product_dimension
VMART_DESIGN |               6 | public       | 45035996373718774 | promotion_dimension
VMART_DESIGN |               7 | public       | 45035996373718778 | date_dimension
VMART_DESIGN |               8 | public       | 45035996373718782 | vendor_dimension
VMART_DESIGN |               9 | public       | 45035996373718786 | employee_dimension
VMART_DESIGN |              10 | public       | 45035996373718822 | shipping_dimension
VMART_DESIGN |              11 | public       | 45035996373718826 | warehouse_dimension
VMART_DESIGN |              12 | public       | 45035996373718830 | inventory_face
VMART_DESIGN |              13 | store        | 45035996373718794 | store_dimension
VMART_DESIGN |              14 | store        | 45035996373718798 | store_sales_fact
VMART_DESIGN |              15 | store        | 45035996373718812 | store_orders_fact
      (15 rows)
```

DESIGNS

Contains information about a Database Designer design. After you create a design and specify certain parameters for Database Designer, DESIGNER_CREATE_DESIGN creates this table in the V_MONITOR schema.

Column Name	Column Type	Description
DESIGN_ID	INTEGER	Unique ID that Database Designer assigns to this design.
DESIGN_NAME	VARCHAR	Name that the user specifies for the design.
KSAFETY_LEVEL	INTEGER	K-safety level for the design. Database Designer assigns a K-safety value of 0 for clusters with 1 or 2 nodes, and assigns a value of 1 for clusters with 3 or more nodes.

OPTIMIZATION_OBJECTIVE	VARCHAR	Name of the optimization objective for the design. Valid values are: • QUERY • LOAD • BALANCED (default)
DESIGN_TYPE	VARCHAR	Name of the design type. Valid values are: • COMPREHENSIVE (default) • INCREMENTAL
PROPOSE_SUPER_FIRST	BOOLEAN	Specifies to propose superprojections before projections, by default f . If DESIGN_MODE is COMPREHENSIVE , this field has no impact.
DESIGN_AVAILABLE	BOOLEAN	t if the design is currently available, otherwise, f (default).
COLLECTED_STATISTICS	BOOLEAN	t if statistics are to be collected when creating the design, otherwise, f (default).
POPULATE_DESIGN_TABLES_FROM_QUERIES	BOOLEAN	t if you want to populate the design tables from the design queries, otherwise, f (default).
ENCODING_DESIGN	BOOLEAN	t if the design is an encoding optimization design on pre-existing projections, otherwise, f (default).
DEPLOYMENT_PARALLELISM	INTEGER	Number of tables to be deployed in parallel when the design is complete. Default: 0
UNSEGMENTED_PROJECTIONS	BOOLEAN	t if you specify unsegmented projections, otherwise, f (default).
ANALYZE_CORRELATIONS_MODE	INTEGER	Specifies how Database Designer should handle existing column correlations in a design and whether or not Database Designer should reanalyze existing column correlations. • 0: (default) Ignore column correlations when creating the design. • 1: Consider the existing correlations in the tables when creating the design. • 2: Analyze column correlations if not previously performed, and consider the column correlations when creating the design. • 3: Analyze all column correlations in the tables and consider them when creating the design, even if they have been analyzed previously.

DISK_QUOTA_USAGES

Provides information about schemas and tables that have disk quotas. Schemas and tables without quotas are not included.

Column Name	Data Type	Description
OBJECT_OID	INTEGER	Unique identifier for a schema or table.
OBJECT_NAME	VARCHAR	Name of the schema or table. Table names include the schema prefix.
IS_SCHEMA	BOOLEAN	Whether the object is a schema. If false, the object is a table.
TOTAL_DISK_USAGE_IN_BYTES	INTEGER	Current usage of the object. For information about what is counted, see Disk quotas .
DISK_QUOTA_IN_BYTES	INTEGER	Current quota for the object.

Examples

```
=> SELECT * FROM DISK_QUOTA_USAGES;
   object_oid     | object_name | is_schema | total_disk_usage_in_bytes | disk_quota_in_bytes
------------------+-------------+-----------+---------------------------+---------------------
 45035996273705100 | s           | t         |                       307 |               10240
 45035996273705104 | public.t    | f         |                       614 |                1024
 45035996273705108 | s.t         | f         |                       307 |                2048
(3 rows)
```

DISK_RESOURCE_REJECTIONS

Returns requests for resources that are rejected due to disk space shortages. Output is aggregated by both RESOURCE_TYPE and REJECTED_REASON to provide more comprehensive information.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
RESOURCE_TYPE	VARCHAR	The resource request requester (example: Temp files).
REJECTED_REASON	VARCHAR	One of the following: • Insufficient disk space • Failed volume
REJECTED_COUNT	INTEGER	Number of times this REJECTED_REASON has been given for this RESOURCE_TYPE .
FIRST_REJECTED_TIMESTAMP	TIMESTAMP	The time of the first rejection for this REJECTED_REASON and RESOURCE_TYPE.
LAST_REJECTED_TIMESTAMP	TIMESTAMP	The time of the most recent rejection for this REJECTED_REASON and RESOURCE_TYPE .
LAST_REJECTED_VALUE	INTEGER	The value of the most recent rejection for this REJECTED_REASON and RESOURCE_TYPE .

See also

- RESOURCE_REJECTIONS
- CLEAR_RESOURCE_REJECTIONS

DISK_STORAGE

Returns the amount of disk storage used by the database on each node. Each node can have one or more storage locations, and the locations can be on different disks with separate properties, such as free space, used space, and block size. The information in this system table is useful in determining where data files reside.

All returned values for this system table are in the context of the file system of the host operating system, and are not specific to Vertica-specific space.

The storage usage annotation called CATALOG indicates that the location is used to store the catalog. Each CATALOG location is specified only when creating a new database. You cannot add a CATALOG location annotation using CREATE LOCATION , nor remove an existing CATALOG annotation.

Storage location performance

The performance of a storage location is measured with two values:

- Throughput in MB/sec
- Latency in seeks/sec

These two values are converted to a single number (Speed) with the following formula:

$$read\text{-}time = (1/throughput) + (1/latency)$$

- *read-time* : Time to read 1MB of data
- 1/ *throughput* : Time to read 1MB of data
- 1/ *latency* : Time to seek to the data.

A disk is faster than another disk if its *read-time* is less.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node name for which information is listed.
STORAGE_PATH	VARCHAR	Path where the storage location is mounted.
STORAGE_USAGE	VARCHAR	Type of information stored in the location, one of the following: • DATA : Only data is stored in the location. • TEMP : Only temporary files that are created during loads or queries are stored in the location. • DATA,TEMP : Both types of files are stored in the location. • USER : The storage location can be used by non-dbadmin users, who are granted access to the storage location • CATALOG : The area is used for the Vertica catalog. This usage is set internally and cannot be removed or changed.
RANK	INTEGER	Integer rank assigned to the storage location based on its performance. Ranks are used to create a storage locations on which projections, columns, and partitions are stored on different disks based on predicted or measured access patterns. See Managing storage locations .
THROUGHPUT	INTEGER	Integer that measures a storage location's performance in MB/sec. 1/ *throughput* is the time taken to read 1MB of data.
LATENCY	INTEGER	Integer that measures a storage location's performance in seeks/sec. 1/ *latency* is the time taken to seek to the data.
STORAGE_STATUS	VARCHAR	Status of the storage location, one of the following: • active • retired
DISK_BLOCK_SIZE_BYTES	INTEGER	Block size of the disk in bytes
DISK_SPACE_USED_BLOCKS	INTEGER	Number of disk blocks in use
DISK_SPACE_USED_MB	INTEGER	Number of megabytes of disk storage in use
DISK_SPACE_FREE_BLOCKS	INTEGER	Number of free disk blocks available
DISK_SPACE_FREE_MB	INTEGER	Number of megabytes of free storage available
DISK_SPACE_FREE_PERCENT	VARCHAR	Percentage of free disk space remaining

DRAINING_STATUS

Returns the draining status of each node in a database. The table also provides aggregate user session counts and information about the oldest user session connected to each node. For more information about the user sessions connected to a database, see SESSIONS .

> **Note**
> If you aren't a superuser, the OLDEST_SESSION columns contain only information about sessions for which you have privileges. Unprivileged users do not see session details, but they do see the node draining status and the user session aggregate count.

Column Name	Data Type	Description

NODE_NAME	VARCHAR	Name of the node for which information is listed.
SUBCLUSTER_NAME	VARCHAR	Name of the subcluster that contains the node.
IS_DRAINING	BOOLEAN	True if the node is draining; otherwise, false.
COUNT_CLIENT_USER_SESSIONS	INTEGER	Number of user client sessions connected to the node.
OLDEST_SESSION_USER	VARCHAR	Name of the user with the oldest live session connected to the node. NULL if no users are connected.
OLDEST_SESSION_ID	VARCHAR	Identifier associated with OLDEST_SESSION_USER . This is required to close or interrupt a session. NULL if no users are connected.
OLDEST_SESSION_LOGIN_TIMESTAMP	TIMESTAMP	Date and time the OLDEST_SESSION_USER logged into the database. NULL if no users are connected.

ERROR_MESSAGES

Lists system error messages and warnings Vertica encounters while processing queries. Some errors occur when no transaction is in progress, so the transaction identifier or statement identifier columns might return NULL.

Column Name	Data Type	Description
EVENT_TIMESTAMP	TIMESTAMPTZ	Time when the row recorded the event
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information
USER_ID	INTEGER	Identifier of the user who received the error message
USER_NAME	VARCHAR	Name of the user who received the error message when Vertica recorded the session
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID, and REQUEST_ID uniquely identifies a statement within a session.
ERROR_LEVEL	VARCHAR	Severity of the error, one of the following: • LOG • INFO • NOTICE • WARNING • ERROR • ROLLBACK • INTERNAL • FATAL • PANIC
ERROR_CODE	INTEGER	Error code that Vertica reports
MESSAGE	VARCHAR	Textual output of the error message

DETAIL	VARCHAR	Additional information about the error message, in greater detail
HINT	VARCHAR	Actionable hint about the error. For example: HINT: Set the locale in this session to en_US@collation=binary using the command "\locale en_US@collation=binary"

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

EVENT_CONFIGURATIONS

Monitors the configuration of events.

Column Name	Data Type	Description
EVENT_ID	VARCHAR	The name of the event.
EVENT_DELIVERY_CHANNELS	VARCHAR	The delivery channel on which the event occurred.

EXECUTION_ENGINE_PROFILES

Provides profiling information about runtime query execution. The hierarchy of IDs, from highest level to actual execution, is:

- PATH_ID
- BASEPLAN_ID
- LOCALPLAN_ID
- OPERATOR_ID

Counters (output from the COUNTER_NAME column) are collected for each actual Execution Engine (EE) operator instance.

The following columns combine to form a unique key:

- TRANSACTION_ID
- STATEMENT_ID
- NODE_NAME
- OPERATOR_ID
- COUNTER_NAME
- COUNTER_TAG

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node name for which information is listed.
USER_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the user.
USER_NAME	VARCHAR	User name for which query profile information is listed.
SESSION_ID	VARCHAR	Identifier of the session for which profiling information is captured. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session if any; otherwise NULL .
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed.

OPERATOR_NAME	VARCHAR	Name of the Execution Engine (EE) component; for example, NetworkSend .
OPERATOR_ID	INTEGER	Identifier assigned by the EE operator instance that performs the work. OPERATOR_ID is different from LOCALPLAN_ID because each logical operator, such as Scan , may be executed by multiple threads concurrently. Each thread operates on a different operator instance, which has its own ID.
BASEPLAN_ID	INTEGER	Assigned by the optimizer on the initiator to EE operators in the original base (EXPLAIN) plan. Each EE operator in the base plan gets a unique ID.
PATH_ID	INTEGER	Identifier that Vertica assigns to a query operation or *path* ; for example to a logical grouping operation that might be performed by multiple execution engine operators. For each path, the same PATH ID is shared between the query plan (using EXPLAIN output) and in error messages that refer to joins.
LOCALPLAN_ID	INTEGER	Identifier assigned by each local executor while preparing for plan execution (local planning). Some operators in the base plan, such as the Root operator, which is connected to the client, do not run on all nodes. Similarly, certain operators, such as ExprEval , are added and removed during local planning due to implementation details.
ACTIVITY_ID	INTEGER	Identifier of the plan activity.
RESOURCE_ID	INTEGER	Identifier of the plan resource.
COUNTER_NAME	VARCHAR	Name of the counter (see Counter Names below). The counter counts events for one statement.
COUNTER_TAG	VARCHAR	String that uniquely identifies the counter for operators that might need to distinguish between different instances. For example, COUNTER_TAG is used to identify to which of the node bytes are being sent to or received from the NetworkSend operator.
COUNTER_VALUE	INTEGER	Value of the counter.
IS_EXECUTING	BOOLEAN	Indicates whether the profile is active or completed.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

Counter names

The value of COUNTER_NAME can be any of the following:

COUNTER_NAME	Description
active threads	A counter of the LoadUnion operator, indicates the number of input threads (Load operators) that are currently processing input.
blocks analyzed by SIPs expression	Number of data blocks analyzed by SIPS expression from the Scan operator.
blocks filtered by SIPs expression	Number of data blocks filtered by SIPS expression from the Scan operator.
blocks filtered by SIPs value lists	Number of data blocks filtered by SIPS sorted value lists from the Scan operator.
buffers spilled	[NetworkSend] Buffers spilled to disk by NetworkSend .

bytes read from disk	[Scan] Amount of data read (locally or remotely) from ROS containers on disk.
bytes read from disk cache	[Scan] Amount of data read from cache .
bytes received	Number of bytes received over the network for query execution.
bytes sent	[NetworkSend Size of data after encoding and compression sent over the network (actual network bytes).
bytes spilled	[NetworkSend] Bytes spilled to disk by NetworkSend.
bytes total	[SendFiles] (recover-by-container plan): Total number of bytes to send/receive.
cached storages cumulative size (bytes)	[StorageMerge] Total amount of temp space used by operator for caching .
cached storages current size (bytes)	[StorageMerge] Current amount of temp space used for caching .
cached storages peak size (bytes)	[StorageMerge] Peak amount of temp space an operator used for caching .
clock time (µs)	Real-time clock time spent processing the query, in microseconds.
clock time (µs) of UDChunker	Real-time clock time spent in the UDChunker phase of a load operation, in microseconds. Use the COUNTER_TAG column to distinguish among load sources.
clock time (µs) of UDFilter(s)	Real-time clock time spent in all UDFilter phases of a load operation, in microseconds. Use the COUNTER_TAG column to distinguish among load sources.
clock time (µs) of UDParser	Real-time clock time spent in the UDParser phase of a load operation, in microseconds. Use the COUNTER_TAG column to distinguish among load sources.
clock time (µs) of UDSource	Real-time clock time spent in the UDSource phase of a load operation, in microseconds. Use the COUNTER_TAG column to distinguish among load sources.
completed merge phases	Number of merge phases already completed by an LSort or DataTarget operator. Compare to the total merge phases . Variants on this value include join inner completed merge phases .
cumulative size of raw temp data (bytes)	Total amount of temporary data the operator has written to files. Compare to cumulative size of temp files (bytes) to understand impact of encoding and compression in an externalizing operator. Variants on this value include join inner cumulative size of raw temp files (bytes) .
cumulative size of temp files (bytes)	For externalizing operators only, the total number of encoded and compressed temp data the operator has written to files. A sort operator might go through multiple merge phases, where at each pass sorted chunks of data are merged into fewer chunks. This counter remembers the cumulative size of all temp files past and present. Variants on this value include join inner cumulative size of temp files (bytes) .
current allocated rid memory (bytes)	Per-rid memory tracking: current allocation amount under this rid.
current file handles	Number of files open.

current memory allocations (count)	Number of actual allocator calls made.
current memory capacity (bytes)	Amount of system memory held, which includes chunks that are only partially consumed.
current memory overhead (bytes)	Memory consumed, for example, by debug headers. (Normally no overhead.)
current memory padding (bytes)	Memory padding for free list tiers (2^n bytes).
current memory requested (bytes)	Memory actually requested by the caller.
current size of temp files (bytes)	For externalizing operators only, the current size of the encoded and compressed temp data that the operator has written to files. Variants on this value include join inner current size of temp files (bytes) .
current threads	Unused.
current unbalanced memory allocations (count)	Pooled version of "current memory XXX" counters.
current unbalanced memory capacity (bytes)	
current unbalanced memory overhead (bytes)	
current unbalanced memory requested (bytes)	
distinct value estimation time (µs)	[Analyze Statistics] Time (in microcseconds) spent to estimate number of distinct values from the sample after data is read off disk and into the statistical sample.
encoded bytes received	[NetworkRecv] Size of received data after decompressed (but still encoded) received over the network.
encoded bytes sent	[NetworkSend] Size of data sent over the network after encoding.
end time	Time (timestamp) when Vertica stopped processing the operation
estimated rows produced	Number of rows that the optimizer estimated would be produced. See rows produced for the actual number of rows that are produced.
exceptions cumulative size of raw temp data (bytes)	Counters that store the total or current size of exception data.

exceptions rows cumulative size of temp files (bytes)	
exceptions rows current size of temp files (bytes)	
execution time (µs)	CPU clock time spent processing the query, in microseconds.
fast aggregated rows	Number of rows being processed by fast aggregations in the hash groupby operator (no group/aggregation).
files completed	Relevant only to SendFiles/RecvFiles operators (that is, recover-by-container plan) number of files sent/received.
files total	Relevant only to SendFiles/RecvFiles operators (that is, recover-by-container plan) total number of files to send/receive.
Hadoop FS bytes read through native libhdfs++ client	[Scan, Load] Number of bytes read from an hdfs source (using libhdfs++).
Hadoop FS bytes read through webhdfs	[Scan, Load] Number of bytes read from a webhdfs source.
Hadoop FS bytes written through webhdfs	[DataTarget] Number of bytes written to webhdfs storage.
Hadoop FS hdfs:// operations that used native libhdfs++ calls	[Scan, Load, DataTarget] Number of times Vertica opened a file with an hdfs:// URL and used the native hdfs protocol
Hadoop FS hdfs:// operations that used webhdfs calls	[Scan, Load, DataTarget] Number of times Vertica opened a file with an hdfs:// URL and used the webhdfs protocol
Hadoop FS read operations through native libhdfs++ client failure count	[Scan, Load] Number of times a native libhdfs++ source encountered an error and gave up
Hadoop FS read operations through native libhdfs++ client retry count	[Scan, Load] Number of times a native libhdfs++ source encountered an error and retried
Hadoop FS read operations through webhdfs failure count	[Scan, Load] Number of times a webhdfs source encountered an error and gave up
Hadoop FS read operations through webhdfs retry count	[Scan, Load] Number of times a webhdfs source encountered an error and retried

Hadoop FS write operations through webhdfs failure count	[DataTarget] Number of times a webhdfs write encountered an error and gave up
Hadoop FS write operations through webhdfs retry count	[DataTarget] Number of times a webhdfs write encountered an error and retried
histogram creation time(µs)	[Analyze Statistics] Time spent estimating the number of distinct values from the sample after data is read off disk and into the statistical sample.
initialization time (µs)	Time in microseconds spent initializing an operator during the CompilePlan step of query processing. For example, initialization time could include the time spent compiling expressions and gathering resources.
input queue wait (µs)	Time in microseconds that an operator spends waiting for upstream operators.
input rows	Actual number of rows that were read into the operator.
input size (bytes)	Total number of bytes of the Load operator's input source, where NULL is unknown (read from FIFO).
inputs processed	Number of sources processed by a Load operator.
intermediate rows to process	Number of rows to process in a phase as determined by a sort or GROUP BY (HASH) .
join inner clock time (µs)	Real clock time spent on processing the inner input of the join operator.
join inner completed mergephases	See the completed merge phases counter.
join inner cumulative size of raw temp data (bytes)	
join inner cumulative size of temp files (bytes)	
join inner current size of temp files (bytes)	
join inner execution time (µs)	The CPU clock time spent on processing the inner input of the join operator.
join inner hash table building time (µs)	Time spent for building the hash table for the inner input of the join operator.
join inner hash table collisions	Number of hash table collisions that occurred when building the hash table for the inner input of the join operator.

join inner hash table entries	Number of hash table entries for the inner input of the join operator.
join inner total merge phases	See the completed merge phases counter.
join outer clock time (µs)	Real clock time spent on processing the outer input of the join operator (including doing the join).
join outer execution time (µs)	CPU clock time spent on processing the outer input of the join operator (including doing the join).
max sample size (rows)	[Analyze Statistics] Maximum number of rows that will be stored in the statistical sample.
memory reserved (bytes)	Memory reserved by this operator. Deprecated.
network wait (µs)	[NetworkSend, NetworkRecv] Time in microseconds spent waiting on the network.
number of bytes read from persistent storage	Estimated number of bytes read from persistent storage to process this query.
number of bytes read from depot storage	Estimated number of bytes read from the depot to process this query.
number of cancel requests received	Number of cancel requests received (per operator) when cancelling a call to the execution engine.
number of invocations	Number of times a UDSF function was invoked.
number of storage containers opened	[Scan] Number of containers opened by the operator, at least 1. If the scan operator switches containers, this counter increases accordingly. See Local caching of storage containers for details.
output queue wait (µs)	Time in microseconds that an operator spends waiting for the output buffer to be consumed by a downstream operator.
peak allocated rid memory (bytes)	Per-rid memory tracking: peak allocation amount under this rid.
peak cooperating threads	Peak number of threads which parsed (in parallel) a single load source, using "cooperative parse." counter_tag indicates the source when joining with dc_load_events .
peak file handles	Peak value of the corresponding "current XXX" counters.
peak memory allocations (count)	
peak memory capacity (bytes)	
peak memory overhead (bytes)	

peak memory padding (bytes)	
peak memory requested (bytes)	
peak temp space	
peak threads	
peak unbalanced memory allocations (count)	
peak unbalanced memory capacity (bytes)	
peak unbalanced memory overhead (bytes)	
peak unbalanced memory padding (bytes)	
peak unbalanced memory requested (bytes)	
portion offset	Offset value of a portion descriptor in an apportioned load. counter_tag indicates the source when joining with dc_load_events .
portion size	Size value of a portion descriptor in an apportioned load. counter_tag indicates the source when joining with dc_load_events .
producer stall (µs)	[NetworkSend] Time in microseconds spent by NetworkSend when stalled waiting for network buffers to clear.
producer wait (µs)	[NetworkSend] Time in microseconds spent by the input operator making rows to send.
read (bytes)	Number of bytes read from the input source by the Load operator.
receive time (µs)	Time in microseconds that a Recv operator spends reading data from its socket.
rejected data cumulative size of raw temp data (bytes)	Counters that store total or current size of rejected row numbers. Are variants of: • cumulative size of raw temp data (bytes) • cumulative size of temp files (bytes) • current size of temp files (bytes)
rejected data cumulative size of temp files (bytes)	
rejected data current sizeof temp files (bytes)	

rejected rows cumulative size of raw temp data (bytes)	
rejected rows cumulative size of temp files (bytes)	
rejected rows current size of temp files (bytes)	
reserved rid memory (bytes)	Per-rid memory tracking: total memory reservation under this rid.
rle rows produced	Number of physical tuples produced by an operator. Complements the rows produced counter, which shows the number of logical rows produced by an operator. For example, if a value occurs 1000 rows consecutively and is RLE encoded, it counts as 1000 rows produced not only 1 rle rows produced .
ROS blocks bounded	[DataTarget] Number of ROS blocks created, due to boundary alignment with RLE prefix columns, when an EE DataTarget operator is writing to ROS containers.
ROS blocks encoded	[DataTarget] Number of ROS blocks created when an EE DataTarget operator is writing to ROS containers.
ROS bytes written	[DataTarget] Number of bytes written to disk when an EE DataTarget operator is writing to ROS containers.
rows added by predicate analysis	Number of rows in the query results that were added without individual evaluation, based on the predicate and range of possible results in a block.
rows filtered by SIPs expression	Number of rows filtered by the SIPS expression from the Scan operator.
rows filtered by query predicate	Number of rows excluded from query results because they failed a condition (predicate), for example in a WHERE clause.
rows in sample	[Analyze Statistics] Actual number of rows that will be stored in the statistical sample.
rows output by sort	[DataTarget] Number of rows sorted when an EE DataTarget operator is writing to ROS containers.
rows processed	[DataSource] Number of rows processed when an EE DataSource operator is reading from ROS containers.
rows processed by SIPs expression	Number of rows processed by the SIPS expression in the Scan operator.
rows produced	Number of logical rows produced by an operator. See also the rle rows produced counter.
rows pruned by query predicates	Number of rows discarded from query results because, based on predicates and value ranges, no row in the block could satisfy the predicate.
rows pruned by valindex	[DataSource] Number of rows it skips direct scanning with help of valindex when an EE DataSource operator is writing to ROS containers. This counter's value is not greater than "rows processed" counter.
rows read in sort	See the counter total rows read in sort .

rows received	[NetworkRecv] Number of received sent over the network.
rows rejected	Number of rows rejected by the Load operator.
rows sent	[NetworkSend] Number of rows sent over the network.
rows to process	Total number of rows to be processed in a phase, based upon the number of table accesses. Compare to the counter, rows processed . Divide the rows processed value by the rows to process value for percent completion.
rows written in join sort	Total number of rows being read out of the sort facility in Join.
rows written in sort	Number of rows read out of the sort by the SortManager. This counter and the counter total rows read from sort are typically equal.
send time (µs)	Time in microseconds that a Send operator spends writing data to its socket.
start time	Time (timestamp) when Vertica started to process the operation.
total merge phases	Number of merge phases an LSort or DataTarget operator must complete to finish sorting its data. NULL until the operator can compute this value (all data must first be ingested by the operator). Variants on this value include join inner total merge phases .
total rows read in join sort	Total number of rows being put into the sort facility in Join.
total rows read in sort total	Total number of rows ingested into the sort by the SortManager. This counter and the counter rows written in sort are typically equal.
total rows written in sort	See the counter, rows written in sort .
total sources	Total number of distinct input sources processed in a load.
unpacked (bytes)	Number of bytes produced by a compressed source in a load (for example, for a gzip file, the size of the file when decompressed).
wait clock time (µs)	StorageUnion wait time in microseconds.
written rows	[DataTarget] Number of rows written when an EE DataTarget operator writes to ROS containers

Examples

The two queries below show the contents of the EXECUTION_ENGINE_PROFILES table:

```
=> SELECT operator_name, operator_id, counter_name, counter_value
   FROM EXECUTION_ENGINE_PROFILES WHERE operator_name = 'Scan'
   ORDER BY counter_value DESC;
 operator_name | operator_id | counter_name |  counter_value
---------------+-------------+--------------+------------------
 Scan          |          20 | end time     | 1559929719983785
 Scan          |          20 | start time   | 1559929719983737
 Scan          |          18 | end time     | 1559929719983358
 Scan          |          18 | start time   | 1559929718069860
 Scan          |          16 | end time     | 1559929718069319
 Scan          |          16 | start time   | 1559929718069188
 Scan          |          14 | end time     | 1559929718068611
 Scan          |          18 | end time     | 1559929717579145
 Scan          |          18 | start time   | 1559929717579083
 Scan          |          16 | end time     | 1559929717578509
 Scan          |          18 | end time     | 1559929717379346
 Scan          |          18 | start time   | 1559929717379307
 Scan          |          16 | end time     | 1559929717378879
 Scan          |          16 | start time   | 1559929716894312
 Scan          |          14 | end time     | 1559929716893599
 Scan          |          14 | start time   | 1559929716893501
 Scan          |          12 | end time     | 1559929716892721
 Scan          |          16 | start time   | 1559929716666110
...

=> SELECT DISTINCT counter_name FROM execution_engine_profiles;
                counter_name
-------------------------------------------------------
 reserved rid memory (bytes)
 rows filtered by SIPs expression
 rows output by sort
 chunk rows scanned squared
 join inner execution time (us)
 current unbalanced memory requested (bytes)
 clock time (us)
 join outer clock time (us)
 exception handling execution time (us)
 peak memory capacity (bytes)
 bytes received
 peak memory requested (bytes)
 send time (us)
 ROS blocks encoded
 current size of temp files (bytes)
 peak memory allocations (count)
 current unbalanced memory overhead (bytes)
 rows segmented
...
```

The following query includes the path_id column, which links the path that the query optimizer takes (via the EXPLAIN command's textual output) with join error messages.

```
=> SELECT operator_name, path_id, counter_name, counter_value FROM execution_engine_profiles where operator_name = 'Join';
 operator_name | path_id |                  counter_name                  | counter_value
---------------+---------+------------------------------------------------+---------------
Join           |      64 | current memory allocations (count)             |             0
Join           |      64 | peak memory allocations (count)                |            57
Join           |      64 | current memory requested (bytes)               |             0
Join           |      64 | peak memory requested (bytes)                  |       1698240
Join           |      64 | current memory overhead (bytes)                |             0
Join           |      64 | peak memory overhead (bytes)                   |             0
Join           |      64 | current memory padding (bytes)                 |             0
Join           |      64 | peak memory padding (bytes)                    |        249840
Join           |      64 | current memory capacity (bytes)                |             0
Join           |      64 | peak memory capacity (bytes)                   |        294912
Join           |      64 | current unbalanced memory allocations (count)  |           145
Join           |      64 | peak unbalanced memory allocations (count)     |           146
Join           |      64 | current unbalanced memory requested (bytes)    |        116506
Join           |      64 | peak unbalanced memory requested (bytes)       |       1059111
Join           |      64 | current unbalanced memory overhead (bytes)     |          3120
Join           |      64 | peak unbalanced memory overhead (bytes)        |          3120
...
```

See also

- Profiling database performance
- QUERY_CONSUMPTION

EXTERNAL_TABLE_DETAILS

Returns the amount of disk storage used by the source files backing external tables in the database. The information in this system table is useful in determining Hadoop license compliance.

When computing the size of an external table, Vertica counts all data found in the location specified by the COPY FROM clause. If you have a directory that contains ORC and delimited files, for example, and you define your external table with "COPY FROM *" instead of "COPY FROM *.orc", this table includes the size of the delimited files. (You would probably also encounter errors when querying that external table.) When you query this system table Vertica does not validate your table definition; it just uses the path to find files to report.

Restrict your queries to filter by schema, table, or format to avoid expensive queries. Vertica calculates the values in this table at query time, so "SELECT *" accesses every input file contributing to every external table.

Predicates in queries may use only the TABLE_SCHEMA, TABLE_NAME, and SOURCE_FORMAT columns. Values are case-sensitive.

This table includes TEMP external tables.

This table reports only data that the current user can read. To include all the data backing external tables, either query this table as a user that has access to all HDFS data or use a session delegation token that grants this access. For more information about using delegation tokens, see Accessing kerberized HDFS data .

Column Name	Data Type	Description
SCHEMA_OID	INTEGER	The unique identification number of the schema in which the external table resides.
TABLE_SCHEMA	VARCHAR	The name of the schema in which the external table resides.
TABLE_OID	INTEGER	A unique numeric ID assigned by the Vertica catalog that identifies the table.
TABLE_NAME	VARCHAR	The table name.
SOURCE_FORMAT	VARCHAR	The data format the source file used, one of ORC, PARQUET, DELIMITED, USER DEFINED, or NULL if another format.
TOTAL_FILE_COUNT	INTEGER	The number of files used to store this table's data, expanding globs and partitions.

TOTAL_FILE_SIZE_BYTES	INTEGER	Total number of bytes used by all of this table's data files.
SOURCE_STATEMENT	VARCHAR	The load statement used to copy data from the source files.
FILE_ACCESS_ERROR	VARCHAR	The access error returned during the source statement. NULL, if there was no access error during the source statement.

HIVE_CUSTOM_PARTITIONS_ACCESSED

This table provides information about all custom locations for Hive partition data that Vertica has accessed. It applies when Hive uses a non-default location for partition data, the HCatalog Connector is used to access that data, and the CREATE HCATALOG SCHEMA statement for the schema sets the CUSTOM_PARTITIONS parameter.

Column Name	Data Type	Description
ACCESS_TIME	TIMESTAMPTZ	Time when Vertica accessed the partition data.
ACCESS_NODE	VARCHAR(128)	Name of the node that performed the access.
TRANSACTION_ID	INTEGER	Identifier for the query that produced the access.
FILESYSTEM	VARCHAR(128)	File system of the partition data. This value is the scheme portion of the URL.
AUTHORITY	VARCHAR(128)	If the file system is HDFS, this value is the nameservice. If the file system is S3, it is the name of the bucket.
URL	VARCHAR(6400)	Full path to the partition.

Privileges

No explicit permissions are required; however, users see only the records that correspond to schemas they have permissions to access.

HOST_RESOURCES

Provides a snapshot of the node. This is useful for regularly polling the node with automated tools or scripts.

Column Name	Data Type	Description
HOST_NAME	VARCHAR	The host name for which information is listed.
OPEN_FILES_LIMIT	INTEGER	The maximum number of files that can be open at one time on the node.
THREADS_LIMIT	INTEGER	The maximum number of threads that can coexist on the node.
CORE_FILE_LIMIT_MAX_SIZE_BYTES	INTEGER	The maximum core file size allowed on the node.
PROCESSOR_COUNT	INTEGER	The number of system processors.
PROCESSOR_CORE_COUNT	INTEGER	The number of processor cores in the system.
PROCESSOR_DESCRIPTION	VARCHAR	A description of the processor. For example: Inter(R) Core(TM)2 Duo CPU T8100 @2.10GHz (1 row)
OPENED_FILE_COUNT	INTEGER	The total number of open files on the node.
OPENED_SOCKET_COUNT	INTEGER	The total number of open sockets on the node.

OPENED_NONFILE_NONSOCKET_COUNT	INTEGER	The total number of *other* file descriptions open in which 'other' could be a directory or FIFO. It is not an open file or socket.
TOTAL_MEMORY_BYTES	INTEGER	The total amount of physical RAM, in bytes, available on the system.
TOTAL_MEMORY_FREE_BYTES	INTEGER	The amount of physical RAM, in bytes, left unused by the system.
TOTAL_BUFFER_MEMORY_BYTES	INTEGER	The amount of physical RAM, in bytes, used for file buffers on the system
TOTAL_MEMORY_CACHE_BYTES	INTEGER	The amount of physical RAM, in bytes, used as cache memory on the system.
TOTAL_SWAP_MEMORY_BYTES	INTEGER	The total amount of swap memory available, in bytes, on the system.
TOTAL_SWAP_MEMORY_FREE_BYTES	INTEGER	The total amount of swap memory free, in bytes, on the system.
DISK_SPACE_FREE_MB	INTEGER	The free disk space available, in megabytes, for all storage location file systems (data directories).
DISK_SPACE_USED_MB	INTEGER	The disk space used, in megabytes, for all storage location file systems.
DISK_SPACE_TOTAL_MB	INTEGER	The total free disk space available, in megabytes, for all storage location file systems.

Examples

```
=> SELECT * FROM HOST_RESOURCES;
-[ RECORD 1 ]------------------+-----------------------------------------
host_name                     | 10.20.100.247
open_files_limit              | 65536
threads_limit                 | 3833
core_file_limit_max_size_bytes | 0
processor_count               | 2
processor_core_count          | 2
processor_description         | Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
opened_file_count             | 8
opened_socket_count           | 15
opened_nonfile_nonsocket_count | 7
total_memory_bytes            | 4018823168
total_memory_free_bytes       | 126550016
total_buffer_memory_bytes     | 191803392
total_memory_cache_bytes      | 2418753536
total_swap_memory_bytes       | 2147479552
total_swap_memory_free_bytes  | 2145771520
disk_space_free_mb            | 18238
disk_space_used_mb            | 30013
disk_space_total_mb           | 48251
-[ RECORD 2 ]------------------+-----------------------------------------
host_name                     | 10.20.100.248
open_files_limit              | 65536
threads_limit                 | 3833
core_file_limit_max_size_bytes | 0
processor_count               | 2
processor_core_count          | 2
processor_description         | Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
opened_file_count             | 8
opened_socket_count           | 9
opened_nonfile_nonsocket_count | 7
total_memory_bytes            | 4018823168
total_memory_free_bytes       | 356466688
total_buffer_memory_bytes     | 327278592
total_memory_cache_bytes      | 2744279040
total_swap_memory_bytes       | 2147479552
total_swap_memory_free_bytes  | 2147479552
disk_space_free_mb            | 37102
disk_space_used_mb            | 11149
disk_space_total_mb           | 48251
-[ RECORD 3 ]------------------+-----------------------------------------
host_name                     | 10.20.100.249
open_files_limit              | 65536
threads_limit                 | 3833
core_file_limit_max_size_bytes | 0
processor_count               | 2
processor_core_count          | 2
processor_description         | Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
opened_file_count             | 8
opened_socket_count           | 9
opened_nonfile_nonsocket_count | 7
total_memory_bytes            | 4018823168
total_memory_free_bytes       | 241610752
total_buffer_memory_bytes     | 309395456
total_memory_cache_bytes      | 2881675264
total_swap_memory_bytes       | 2147479552
total_swap_memory_free_bytes  | 2147479552
disk_space_free_mb            | 37430
disk_space_used_mb            | 10821
disk_space_total_mb           | 48251
```

IO_USAGE

Provides disk I/O bandwidth usage history for the system.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
START_TIME	TIMESTAMP	Beginning of history interval.
END_TIME	TIMESTAMP	End of history interval.
READ_KBYTES_PER_SEC	FLOAT	Counter history of the number of bytes read measured in kilobytes per second.
WRITTEN_KBYTES_PER_SEC	FLOAT	Counter history of the number of bytes written measured in kilobytes per second.

Privileges

Superuser

LDAP_LINK_DRYRUN_EVENTS

Collects the results from LDAP dry run meta-functions:

- LDAP_LINK_DRYRUN_CONNECT
- LDAP_LINK_DRYRUN_SEARCH
- LDAP_LINK_DRYRUN_SYNC

For detailed instructions on using these meta-functions, see Configuring LDAP link with dry runs .

Column Name	Data Type	Description
EVENT_TIMESTAMP	TIMESTAMP	The date and time of an LDAP server and Vertica LDAP Link interaction.
NODE_NAME	VARCHAR	The clerk node.
SESSION_ID	VARCHAR	The identification number of the LDAP Link session.
USER_ID	INTEGER	The unique, system-generated user identification number.
USER_NAME	VARCHAR	The name of the user for which the information is listed.
TRANSACTION_ID	INTEGER	The system-generated transaction identification number. Is NULL if a transaction id does not exist.
EVENT_TYPE	VARCHAR	The result of a dry run.
ENTRY_NAME	VARCHAR	The name of the object on which the event occurred, if applicable. For example, the event SYNC-STARTED does not use an object.
ROLE_NAME	VARCHAR	The name of a role.
LDAPURIHASH	INTEGER	The URI hash number for the LDAP user.
LDAP_URI	VARCHAR	The URI for the LDAP server.
BIND_DN	VARCHAR	The Distinguished Name used for the dry run bind.
FILTER_GROUP	VARCHAR	The group attribute passed to the dry run meta-functions as LDAPLinkFilterGroup.
FILTER_USER	VARCHAR	The user attribute passed to the dry run meta-functions as LDAPLinkFilterUser.

LINK_SCOPE	VARCHAR	The DN level to replicate, passed to the dry run meta-functions as LDAPLinkScope.
SEARCH_BASE	VARCHAR	The DN level from which LDAP Link begins the search, passed to the dry run meta-functions as LDAPLinkSearchBase.
GROUP_MEMBER	VARCHAR	Identifies the members of an LDAP group, passed to the dry run meta-functions as LDAPLinkGroupMembers.
GROUP_NAME	VARCHAR	The LDAP field to use when creating a role name in Vertica, passed to the dry run meta-functions as LDAPLinkGroupName.
LDAP_USER_NAME	VARCHAR	The attribute that identifies individual users, passed to the dry run meta-functions as LDAPLinkUserName.
TLS_REC_CERT	VARCHAR	The connection policy used for the dry run connection for certificate management. This connection policy is set through the LDAPLink TLS Configuration .
TLS_CA_CERT	VARCHAR	The CA certificate used for the dry run connection specified by the LDAPLink TLS Configuration .

LDAP_LINK_EVENTS

Monitors events that occurred during an LDAP Link synchronization.

Column Name	Data Type	Description
EVENT_TIMESTAMP	TIMESTAMP	The time the event occurred.
NODE_NAME	VARCHAR	The name of the node or nodes for which the information is listed.
SESSION_ID	VARCHAR	The identification number of the LDAP Link session.
USER_ID	INTEGER	The unique, system-generated user identification number.
USER_NAME	VARCHAR	The name of the user for which the information is listed.
TRANSACTION_ID	INTEGER	The system-generated transaction identification number. Is NULL if a transaction id does not exist.
EVENT_TYPE	VARCHAR	The type of event being logged, for example USER_CREATED and PROCESSING_STARTED.
ENTRY_NAME	VARCHAR	The name of the object on which the event occurred, if applicable. For example, the event SYNC-STARTED does not use an object.
ENTRY_OID	INTEGER	The unique identification number for the object on which the event occurred, if applicable.
LDAPURIHASH	INTEGER	The URI hash number for the LDAP user.

LOAD_SOURCES

Like LOAD_STREAMS , monitors active and historical load metrics on each node. The LOAD_SOURCES table breaks information down by source and portion. Rows appear in this table only for COPY operations that are profiled or run for more than one second. LOAD_SOURCES does not record information about loads from ORC or Parquet files or COPY LOCAL.

A row is added to this table when the loading of a source or portion begins. Column values related to the progress of the load are updated during the load operation.

Columns that uniquely identify the load source (the various ID and name columns) and column IS_EXECUTING always have non-NULL values.

Column Name	Data Type	Description

SESSION_ID	VARCHAR	Identifier of the session for which Vertica captures load stream information. This identifier is unique within the cluster for the current session but can be reused in a subsequent session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within a session. If a session is active, but no transaction has begun, this value is NULL.
STATEMENT_ID	INTEGER	Unique numeric ID for the currently running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID uniquely identifies a statement within a session.
STREAM_NAME	VARCHAR	Load stream identifier. If the user does not supply a specific name, the STREAM_NAME default value is *tablename* - *ID* , where: • *tablename* is the table into which data is being loaded. • *ID* is an integer value. *ID* is guaranteed to be unique within the current session on a node. This system table includes stream names for every COPY statement that takes more than 1 second to run. The 1-second duration includes the time to plan and execute the statement.
SCHEMA_NAME	VARCHAR	Schema name for which load information is listed. Lets you identify two streams that are targeted at tables with the same name in different schemas. NULL, if selecting from an external table.
TABLE_OID	INTEGER	A unique numeric ID assigned by the Vertica catalog that identifies the table. NULL, if selecting from an external table.
TABLE_NAME	VARCHAR	Name of the table being loaded. NULL, if selecting from an external table.
NODE_NAME	VARCHAR	Name of the node loading the source.
SOURCE_NAME	VARCHAR	• Full file path if copying from a file. • Value returned by getUri() if the source is a user-defined source. • STDIN if loading from standard input.
PORTION_OFFSET	INTEGER	Offset of the source portion, or NULL if not apportioned.
PORTION_SIZE	INTEGER	Size of the source portion, or NULL if not apportioned.
IS_EXECUTING	BOOLEAN	Whether this source is currently being parsed, where *t* is true and *f* is false.
READ_BYTES	INTEGER	Number of bytes read from the input file.
ROWS_PRODUCED	INTEGER	Number of rows produced from parsing the source.
ROWS_REJECTED	INTEGER	Number of rows rejected from parsing the source. If CopyFaultTolerantExpressions is true, also includes rows rejected during expression evaluation.
INPUT_SIZE	INTEGER	Size of the input source in bytes, or NULL for unsized sources. For UDSources, this value is the value returned by getSize() .
PARSE_COMPLETE_PERCENT	INTEGER	Percent of rows from the input file that have been parsed.
FAILURE_REASON	VARCHAR	Indicates cause for failure, one of the following: • Load source aborted, error message indicates cause. For example: COPY: Could not open file [*filename*] for reading; Permission denied • Load canceled, displays error message: Statement interrupted In all other cases, set to NULL.

PEAK_COOPERATING_THREADS	INTEGER	The peak number of threads parsing this source in parallel.
CLOCK_TIME_SOURCE	INTEGER	Displays in real-time how many microseconds (µs) have been consumed by the UDSource phase of a load operation.
CLOCK_TIME_FILTERS	INTEGER	Displays in real-time how many microseconds (µs) have been consumed by all UDFilter phases of a load operation.
CLOCK_TIME_CHUNKER	INTEGER	Displays in real-time how many microseconds (µs) have been consumed by the UDChunker phase of a load operation.
CLOCK_TIME_PARSER	INTEGER	Displays in real-time how many microseconds (µs) have been consumed by the UDParser phase of a load operation.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

LOAD_STREAMS

Monitors active and historical load metrics for load streams. This is useful for obtaining statistics about how many records got loaded and rejected from the previous load. Vertica maintains system table metrics until they reach a designated size quota (in kilobytes). This quota is set through internal processes, which you cannot set or view directly.

Column Name	Data Type	Description
SESSION_ID	VARCHAR	Identifier of the session for which Vertica captures load stream information. This identifier is unique within the cluster for the current session, but can be reused in a subsequent session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within a session. If a session is active but no transaction has begun, this is NULL.
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID uniquely identifies a statement within a session.
STREAM_NAME	VARCHAR	Load stream identifier. If the user does not supply a specific name, the STREAM_NAME default value is: *tablename-ID* where *tablename* is the table into which data is being loaded, and *ID* is an integer value, guaranteed to be unique with the current session on a node. This system table includes stream names for every COPY statement that takes more than 1-second to run. The 1-second duration includes the time to plan and execute the statement.
SCHEMA_NAME	VARCHAR	Schema name for which load stream information is listed. Lets you identify two streams that are targeted at tables with the same name in different schemas
TABLE_ID	INTEGER	Catalog-assigned numeric value that uniquely identifies the table.
TABLE_NAME	VARCHAR	Name of the table being loaded.
LOAD_START	VARCHAR	Linux system time when the load started.
LOAD_DURATION_MS	NUMERIC(54,0)	Duration of the load stream in milliseconds.
IS_EXECUTING	BOOLEAN	Indicates whether the load is executing, where *t* is true and *f* is false.
ACCEPTED_ROW_COUNT	INTEGER	Number of rows loaded.

REJECTED_ROW_COUNT	INTEGER	Number of rows rejected.
READ_BYTES	INTEGER	Number of bytes read from the input file.
INPUT_FILE_SIZE_BYTES	INTEGER	Size of the input file in bytes. **Note:** When using STDIN as input, the input file size is zero (0).
PARSE_COMPLETE_PERCENT	INTEGER	Percent of rows from the input file that have been parsed.
UNSORTED_ROW_COUNT	INTEGER	Cumulative number rows not sorted across all projections. **Note:** UNSORTED_ROW_COUNT could be greater than ACCEPTED_ROW_COUNT because data is copied and sorted for every projection in the target table.
SORTED_ROW_COUNT	INTEGER	Cumulative number of rows sorted across all projections.
SORT_COMPLETE_PERCENT	INTEGER	Percent of rows from the input file that have been sorted.

Privileges

If you have the SYSMONITOR role or are the dbadmin user, this table shows all loads. Otherwise it shows only your loads.

LOCK_USAGE

Provides aggregate information about lock requests, releases, and attempts, such as wait time/count and hold time/count. Vertica records:

- Lock attempts at the end of the locking process
- Lock releases after lock attempts are released

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information on which lock interaction occurs.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
OBJECT_NAME	VARCHAR	Name of object being locked; can be a table or an internal structure (projection, global catalog, or local catalog).
MODE	VARCHAR	Intended operations of the transaction. Otherwise, this value is NONE. For a list of lock modes and compatibility, see Lock modes .
AVG_HOLD_TIME	INTERVAL	Average time (measured in intervals) that Vertica holds a lock.
MAX_HOLD_TIME	INTERVAL	Maximum time (measured in intervals) that Vertica holds a lock.
HOLD_COUNT	INTEGER	Total number of times the lock was granted in the given mode.
AVG_WAIT_TIME	INTERVAL	Average time (measured in intervals) that Vertica waits on the lock.
MAX_WAIT_TIME	INTERVAL	Maximum time (measured in intervals) that Vertica waits on a lock.
WAIT_COUNT	INTEGER	Total number of times lock was unavailable at the time it was first requested.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

See also

- DUMP_LOCKTABLE
- LOCKS
- PROJECTION_REFRESHES
- SELECT
- SESSION_PROFILES

LOCKS

Monitors lock grants and requests for all nodes. If no locks are active, a query on LOCKS returns no rows.

Column Name	Data Type	Description
NODE_NAMES	VARCHAR	Comma-separated list of nodes where lock interaction occurs. A transaction can have the same lock in the same mode in the same scope on multiple nodes. However, the transaction gets only one line in the table.
OBJECT_NAME	VARCHAR	Name of object to lock, either a table or an internal structure: projection, global catalog, or local catalog.
OBJECT_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog that identifies the object to lock.
TRANSACTION_ID	VARCHAR	Identifier of transaction within the session, if any; otherwise NULL . Transaction IDs can be used to join other system tables.
TRANSACTION_DESCRIPTION	VARCHAR	Identifier of transaction and associated description. Typically, this query caused the transaction's creation.
LOCK_MODE	VARCHAR	Transaction's lock type .
LOCK_SCOPE	VARCHAR	Expected duration of the lock after it is granted. Before the lock is granted, Vertica lists the scope as REQUESTED . After a lock is granted, its scope is set to one of the following: • STATEMENT_LOCALPLAN • STATEMENT_COMPILE • STATEMENT_EXECUTE • TRANSACTION_POSTCOMMIT • TRANSACTION All scopes other than TRANSACTION are transient and used only as part of normal query processing.
REQUEST_TIMESTAMP	TIMESTAMP	Time when the transaction began waiting on the lock.
GRANT_TIMESTAMP	TIMESTAMP	Time the transaction acquired or upgraded the lock: • Return values are NULL until the grant occurs. • If the grant occurs immediately, values might be the same as REQUEST_TIMESTAMP .

See also

- Vertica database locks
- DUMP_LOCKTABLE
- LOCK_USAGE
- PROJECTION_REFRESHES
- SELECT
- SESSION_PROFILES
- TRANSACTIONS

LOGIN_FAILURES

This system table lists failures for each failed login attempt. This information helps you determine if a user is having difficulty getting into the database or identify a possible intrusion attempt.

Column Name	Data Type	Description
LOGIN_TIMESTAMP	TIMESTAMPTZ	Time when Vertica recorded the login.
DATABASE_NAME	VARCHAR	The name of the database for the login attempt.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_NAME	VARCHAR	Name of the user whose login failed at the time Vertica recorded the session.
CLIENT_HOSTNAME	VARCHAR	Host name and port of the TCP socket from which the client connection was made. NULL if the session is internal.
CLIENT_PID	INTEGER	Identifier of the client process that issued this connection. In some cases, the client process is on a different machine from the server.
CLIENT_VERSION	VARCHAR	Unused.
CLIENT_OS_USER_NAME	VARCHAR	The name of the user that logged into, or attempted to log into, the database. This is logged even when the login attempt is unsuccessful.
AUTHENTICATION_METHOD	VARCHAR	Name of the authentication method used to validate the client application or user who is trying to connect to the server using the database user name provided Valid values: • Trust • Reject • GSS • LDAP • Ident • Hash • TLS See Configuring client authentication for further information.
CLIENT_AUTHENTICATION_NAME	VARCHAR	Locally created name of the client authentication method.
REASON	**VARCHAR**	**Description of login failure reason.** **Valid values:** • **INVALID USER** • **ACCOUNT LOCKED** • **REJECT** • **FAILED** • **INVALID AUTH METHOD** • **INVALID DATABASE**

Privileges

Superuser

MEMORY_EVENTS

Records events related to Vertica memory usage.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node where the event occurred
EVENT_TIME	TIMESTAMPTZ	Event start time
EVENT_TYPE	VARCHAR	Type of event, one of the following: • MEMORY_REPORT : The Vertica memory poller created a report on memory usage, for the reason specified in EVENT_REASON . For details, see Memory usage reporting . • MALLOC_TRIM : Vertica ran the glibc function malloc_trim() to reclaim glibc-allocated memory. For details, see Memory trimming .
EVENT_REASON	VARCHAR	Reason for the event—for example, trim threshold was greater than *RSS* / *available-memory* .
EVENT_DETAILS	VARCHAR	Additional information about the event—for example, how much memory malloc_trim() reclaimed.
DURATION_US	INTEGER	Duration of the event in microseconds (µs).

Privileges

None

Examples

```
=> SELECT * FROM MEMORY_EVENTS;
-[ RECORD 1 ]-+-----------------------------------------------------------------
event_time    | 2019-05-02 13:17:20.700892-04
node_name     | v_vmart_node0001
event_type    | MALLOC_TRIM
event_reason  | memory_trim()
event_details | pre-trim RSS 378822656 post-trim RSS 372129792 benefit 0.0176675
duration_us   | 7724
```

MEMORY_USAGE

Records system resource history for memory usage. This is useful for comparing memory that Vertica uses versus memory in use by the entire system.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
START_TIME	TIMESTAMP	Beginning of history interval.
END_TIME	TIMESTAMP	End of history interval.
AVERAGE_MEMORY_USAGE_PERCENT	FLOAT	Records the average memory usage in percent of total memory (0-100) during the history interval.

Privileges

Superuser

MERGEOUT_PROFILES

Returns information about and status of automatic mergeout operations.

This table excludes operations with a REQUEST_TYPE of NO_WORK. It also excludes the operations of user-invoked mergeout functions, such as DO_TM_TASK.

Column Name	Data Type	Description
START_TIME	TIMESTAMP	When the Tuple Mover began processing storage location mergeout requests.
END_TIME	TIMESTAMP	When the mergeout finished.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session.
NODE_NAME	VARCHAR	Node name for which information is listed.
SCHEMA_NAME	VARCHAR	The schema for which information is listed.
TABLE_NAME	VARCHAR	The table for which information is listed.
PROJECTION_NAME	VARCHAR	The projection for which information is listed.
PROJECTION_OID	INTEGER	Projection's unique catalog identifier.
REQUEST_TYPE	VARCHAR	Identifies the type of operation performed by the tuple mover. Possible values: • PURGE • MERGEOUT • DVMERGEOUT
EVENT_TYPE	VARCHAR	Displays the status of the mergeout operation. Possible values: • ERROR • RETRY • REQUEST_QUEUED • REQUEST_COMPLETED
THREAD_ID	INTEGER	The ID of the thread that performed the mergeout.
STRATA_NO	INTEGER	The ID of the strata the ROS container belongs to.
PARTITION_KEY	INTEGER	The key of the partition.
CONTAINER_COUNT	INTEGER	The number of ROS containers in the mergeout operation.
TOTAL_SIZE_IN_BYTES	INTEGER	Size in bytes of all ROS containers in the mergeout operation.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

Examples

To following statement returns failed mergeout operations for table public.store_orders.

```
=> SELECT node_name, schema_name, table_name, request_type, event_type FROM mergeout_profiles WHERE event_type='ERROR';
   node_name     | schema_name |  table_name  | request_type | event_type
-----------------+-------------+--------------+--------------+------------
v_vmart_node0002 | public      | store_orders | MERGEOUT     | ERROR
v_vmart_node0002 | public      | store_orders | MERGEOUT     | ERROR
v_vmart_node0001 | public      | store_orders | MERGEOUT     | ERROR
v_vmart_node0001 | public      | store_orders | MERGEOUT     | ERROR
v_vmart_node0003 | public      | store_orders | MERGEOUT     | ERROR
v_vmart_node0003 | public      | store_orders | MERGEOUT     | ERROR
v_vmart_node0003 | public      | store_orders | MERGEOUT     | ERROR
v_vmart_node0002 | public      | store_orders | MERGEOUT     | ERROR
v_vmart_node0001 | public      | store_orders | MERGEOUT     | ERROR
(9 rows)
```

See also

- Mergeout
- Partition mergeout

MODEL_STATUS_HISTORY

Lists the status history of registered machine learning models in the database, including models that have been unregistered or dropped. Only superusers or users to whom superusers have granted sufficient privileges can query the table. Vertica recommends granting access to the MLSUPERVISOR role.

Column Name	Data Type	Description
REGISTERED_NAME	VARCHAR	Abstract name to which the model, identified by MODEL_ID , was registered at the time of the status change. This REGISTERED_NAME can represent a group of models for a higher-level application, where each model in the group has a unique version number.
REGISTERED_VERSION	INTEGER	Unique version number of the registered model under its specified REGISTERED_NAME .
NEW_STATUS	VARCHAR	New status of the registered model.
OLD_STATUS	VARCHAR	Old status of the registered model.
STATUS_CHANGE_TIME	TIMESTAMPTZ	Time at which the model status was changed.
OPERATOR_ID	INTEGER	Internal ID of the user who performed the status change.
OPERATOR_NAME	VARCHAR	Name of the user who performed the status change.
MODEL_ID	INTEGER	Internal ID of the model for which information is listed.
SCHEMA_NAME	VARCHAR	Name of the schema that contains the model. This value is NULL if the model has been dropped.
MODEL_NAME	VARCHAR	Name of the model. This value is NULL if the model has been dropped. Each existing model can be uniquely identified by either its [*schema_name.*] *model_name* or the combination of its REGISTERED_NAME and REGISTERED_VERSION .

Example

If a superuser grants SELECT access of the table to the MLSUPERVISOR role, users with that role can then query the MODEL_STATUS_HISTORY table:

```
-- as superuser
=> GRANT SELECT ON TABLE v_monitor.model_status_history TO MLSUPERVISOR;
WARNING 8555:  You are granting privilege on a system table used by superuser only. Revoke the grant if you are unsure
GRANT PRIVILEGE

-- as user with MLSUPERVISOR role
=> SELECT * FROM MODEL_STATUS_HISTORY;
 registered_name | registered_version |  new_status  |  old_status  |       status_change_time       | operator_id | operator_name |  model_id  | schema_name |
model_name
-----------------+--------------------+--------------+------------- +--------------------------------+-------------+---------------+------------+-------------+-------------------
 app1            |                  1 | UNDER_REVIEW | UNREGISTERED | 2023-01-29 09:09:00.082166-05 | 1224567790  | u1            | 0113756739 | public      |
native_linear_reg
 app1            |                  1 | STAGING      | UNDER_REVIEW | 2023-01-29 11:33:02.052464-05 | 2341679901  | supervisor1   | 0113756739 | public      |
native_linear_reg
 app1            |                  1 | PRODUCTION   | STAGING      | 2023-01-30 04:12:30.481136-05 | 2341679901  | supervisor1   | 0113756739 | public      |
native_linear_reg
(3 rows)
```

See also

- REGISTER_MODEL
- CHANGE_MODEL_STATUS
- REGISTERED_MODELS
- Model versioning

MONITORING_EVENTS

Reports significant events that can affect database performance and functionality if you do not address their root causes.

See Monitoring events for details.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
EVENT_CODE	INTEGER	Numeric identifier that indicates the type of event. See Event Types in Monitoring events for a list of event type codes.
EVENT_ID	INTEGER	Unique numeric ID that identifies the specific event.
EVENT_SEVERITY	VARCHAR	Severity of the event from highest to lowest. These events are based on standard syslog severity types: 0 – Emergency 1 – Alert 2 – Critical 3 – Error 4 – Warning 5 – Notice 6 – Info 7 – Debug
EVENT_POSTED_TIMESTAMP	TIMESTAMPTZ	When this event was posted.

EVENT_CLEARED_TIMESTAMP	TIMESTAMPTZ	When this event was cleared. **Note:** You can also query the ACTIVE_EVENTS system table to see events that have not been cleared.
EVENT_EXPIRATION	TIMESTAMPTZ	Time at which this event expires. If the same event is posted again prior to its expiration time, this field gets updated to a new expiration time.
EVENT_CODE_DESCRIPTION	VARCHAR	Brief description of the event and details pertinent to the specific situation.
EVENT_PROBLEM_DESCRIPTION	VARCHAR	Generic description of the event.

Privileges

Superuser

See also

ACTIVE_EVENTS

NETWORK_INTERFACES

Provides information about network interfaces on all Vertica nodes.

Column Name	Data Type	Description
NODE_ID	INTEGER	Unique identifier for the node that recorded the row.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
INTERFACE	VARCHAR	Network interface name.
IP_ADDRESS_FAMILY	VARCHAR	Network address family (either 'ipv4' or 'ipv6').
IP_ADDRESS	VARCHAR	IP address for this interface.
SUBNET	VARCHAR	IP subnet for this interface.
MASK	VARCHAR	IP network mask for this interface.
BROADCAST_ADDRESS	VARCHAR	IP broadcast address for this interface.

Privileges

None

Examples

```
=> \x
Expanded display is on.
=> SELECT * FROM network_interfaces ORDER BY node_name ASC LIMIT 14;
-[ RECORD 1 ]-----+-----------------------------------------
node_id           | 45035996273704982
node_name         | v_verticadb_node0001
interface         | lo
ip_address_family | ipv6
ip_address        | ::1
subnet            | ::1
mask              | ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
broadcast_address |
-[ RECORD 2 ]-----+-----------------------------------------
node_id           | 45035996273704982
node_name         | v_verticadb_node0001
interface         | ens192
ip_address_family | ipv6
ip_address        | fd9b:1fcc:1dc4:78d3::31
```

ip_address | fd9b:1fcc:1dc4:78d3:
subnet | fd9b:1fcc:1dc4:78d3::
mask | ffff:ffff:ffff:ffff::
broadcast_address |
-[RECORD 3]-----+-----------------------------------
node_id | 45035996273704982
node_name | v_verticadb_node0001
interface | lo
ip_address_family | ipv4
ip_address | 127.0.0.1
subnet | 127.0.0.0
mask | 255.0.0.0
broadcast_address | 127.0.0.1
-[RECORD 4]-----+-----------------------------------
node_id | 45035996273704982
node_name | v_verticadb_node0001
interface | ens192
ip_address_family | ipv4
ip_address | 192.168.111.31
subnet | 192.168.111.0
mask | 255.255.255.0
broadcast_address | 192.168.111.255
-[RECORD 5]-----+-----------------------------------
node_id | 45035996273704982
node_name | v_verticadb_node0001
interface | ens32
ip_address_family | ipv4
ip_address | 10.20.110.21
subnet | 10.20.110.0
mask | 255.255.255.0
broadcast_address | 10.20.110.255
-[RECORD 6]-----+-----------------------------------
node_id | 45035996273704982
node_name | v_verticadb_node0001
interface | ens32
ip_address_family | ipv6
ip_address | fe80::250:56ff:fe8e:61d3
subnet | fe80::
mask | ffff:ffff:ffff:ffff::
broadcast_address |
-[RECORD 7]-----+-----------------------------------
node_id | 45035996273704982
node_name | v_verticadb_node0001
interface | ens192
ip_address_family | ipv6
ip_address | fe80::250:56ff:fe8e:2721
subnet | fe80::
mask | ffff:ffff:ffff:ffff::
broadcast_address |
-[RECORD 8]-----+-----------------------------------
node_id | 45035996273841968
node_name | v_verticadb_node0002
interface | lo
ip_address_family | ipv6
ip_address | ::1
subnet | ::1
mask | ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
broadcast_address |
-[RECORD 9]-----+-----------------------------------
node_id | 45035996273841968
node_name | v_verticadb_node0002
interface | ens192

```
ip_address_family | ipv6
ip_address        | fd9b:1fcc:1dc4:78d3::32
subnet            | fd9b:1fcc:1dc4:78d3::
mask              | ffff:ffff:ffff:ffff::
broadcast_address |
-[ RECORD 10 ]----+-----------------------------------------
node_id           | 45035996273841968
node_name         | v_verticadb_node0002
interface         | lo
ip_address_family | ipv4
ip_address        | 127.0.0.1
subnet            | 127.0.0.0
mask              | 255.0.0.0
broadcast_address | 127.0.0.1
-[ RECORD 11 ]----+-----------------------------------------
node_id           | 45035996273841968
node_name         | v_verticadb_node0002
interface         | ens192
ip_address_family | ipv4
ip_address        | 192.168.111.32
subnet            | 192.168.111.0
mask              | 255.255.255.0
broadcast_address | 192.168.111.255
-[ RECORD 12 ]----+-----------------------------------------
node_id           | 45035996273841968
node_name         | v_verticadb_node0002
interface         | ens32
ip_address_family | ipv4
ip_address        | 10.20.110.22
subnet            | 10.20.110.0
mask              | 255.255.255.0
broadcast_address | 10.20.110.255
-[ RECORD 13 ]----+-----------------------------------------
node_id           | 45035996273841968
node_name         | v_verticadb_node0002
interface         | ens32
ip_address_family | ipv6
ip_address        | fe80::250:56ff:fe8e:1787
subnet            | fe80::
mask              | ffff:ffff:ffff:ffff::
broadcast_address |
-[ RECORD 14 ]----+-----------------------------------------
node_id           | 45035996273841968
node_name         | v_verticadb_node0002
interface         | ens192
ip_address_family | ipv6
ip_address        | fe80::250:56ff:fe8e:2c9c
subnet            | fe80::
mask              | ffff:ffff:ffff:ffff::
broadcast_address |
```

NETWORK_USAGE

Provides network bandwidth usage history on the system. This is useful for determining if Vertica is using a large percentage of its available network bandwidth.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.

START_TIME	TIMESTAMP	Beginning of history interval.
END_TIME	TIMESTAMP	End of history interval.
TX_KBYTES_PER_SEC	FLOAT	Counter history of outgoing (transmitting) usage in kilobytes per second.
RX_KBYTES_PER_SEC	FLOAT	Counter history of incoming (receiving) usage in kilobytes per second.

Privileges

Superuser

NODE_EVICTIONS

Monitors node evictions on the system.

Column Name	Data Type	Description
EVICTION_TIMESTAMP	TIMESTAMPTZ	Timestamp when the eviction request was made.
NODE_NAME	VARCHAR	The node name logging the information.
EVICTED_NODE_NAME	VARCHAR	The node name of the evicted node.
EVICTED_NODE_ID	INTEGER	The evicted node ID.
NODE_STATE_BEFORE_EVICTION	VARCHAR	The previous node state at the time of eviction.

NODE_RESOURCES

Provides a snapshot of the node. This is useful for regularly polling the node with automated tools or scripts.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
HOST_NAME	VARCHAR	The hostname associated with a particular node.
NODE_IDENTIFIER	VARCHAR	A unique identifier for the node.
PROCESS_SIZE_BYTES	INTEGER	The total size of the program.
PROCESS_RESIDENT_SET_SIZE_BYTES	INTEGER	The total number of bytes that the process has in memory.
PROCESS_SHARED_MEMORY_SIZE_BYTES	INTEGER	The amount of shared memory used.
PROCESS_TEXT_MEMORY_SIZE_BYTES	INTEGER	The total number of text bytes that the process has in physical memory. This does not include any shared libraries.
PROCESS_DATA_MEMORY_SIZE_BYTES	INTEGER	The amount of physical memory, in bytes, used for performing processes. This does not include the executable code.
PROCESS_LIBRARY_MEMORY_SIZE_BYTES	INTEGER	The total number of library bytes that the process has in physical memory.
PROCESS_DIRTY_MEMORY_SIZE_BYTES	INTEGER	The number of bytes that have been modified since they were last written to disk.
SPREAD_HOST	VARCHAR	The node name of the spread host.

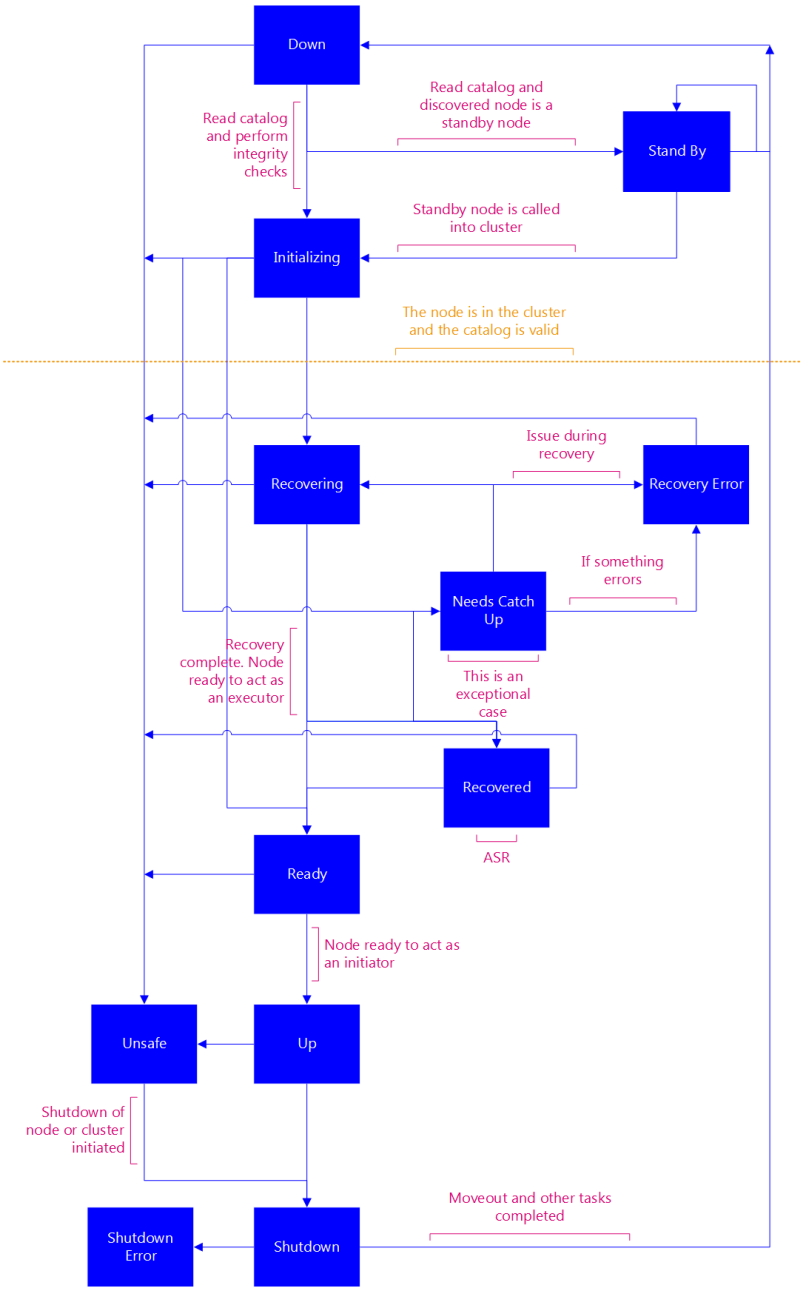
NODE_PORT	VARCHAR	The port used for intra-cluster communication.
DATA_PORT	VARCHAR	The port used by the Vertica client.
DBCLERK	BOOLEAN	Whether this node is the DB clerk. The DB clerk is responsible for coordinating some administrative tasks in the database.

NODE_STATES

Monitors node recovery state-change history on the system. Vertica returns information only on nodes whose state is currently UP. To determine which nodes are not up, query the NODES table.

Column Name	Data Type	Description
EVENT_TIMESTAMP	TIMESTAMPTZ	Time when Vertica recorded the event.
NODE_ID	INTEGER	Catalog-assigned integer value that uniquely identifies thenode.
NODE_NAME	VARCHAR	Name of the node.
NODE_STATE	VARCHAR	The node's state, one of the following: • UP • DOWN • READY • UNSAFE • SHUTDOWN • SHUTDOWN_ERROR • RECOVERING • RECOVER_ERROR • RECOVERED • INITIALIZING • STANDBY • NEEDS_CATCHUP

The following flow chart details different node states:



Privileges

None

NOTIFIER_ERRORS

Reports errors encountered by notifiers .

Column Name	Data Type	Description
ERROR_TIME	TIMESTAMPTZ	The time that the error occurred.
NODE_NAME	VARCHAR	Name of the node that encountered the error.
NOTIFIER_NAME	VARCHAR	Name of the notifier that triggered the error.
DESCRIPTION	VARCHAR	A description of the error.

Privileges

Superuser

OUTPUT_DEPLOYMENT_STATUS

Contains information about the deployment status of all the projections in your design. Each row contains information about a different projection. Vertica populates this table when you deploy the database design by running the function DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY .

Column Name	Column Type	Description
deployment_id	INTEGER	Unique ID that Database Designer assigned to the deployment.
design_name	VARCHAR	Unique name that the user assigned to the design.
deployment_projection_id	INTEGER	Unique ID that Database Designer assigned to the output projection.
deployment_projection_name	VARCHAR	Name that Database Designer assigned to the output projection or the name of the projection to be dropped.
deployment_status	VARCHAR	Status of the deployment: • pending • complete • needs_refresh • in_progress • error
error_message	VARCHAR	Text of any error that occurred when creating or refreshing the specified projection.

OUTPUT_EVENT_HISTORY

Contains information about each stage that Database Designer performs to design and optimize your database design.

Column Name	Data Type	Description
TIME_STAMP	TIMESTAMP	Date and time of the specified stage.
DESIGN_ID	INTEGER	Unique id that Database Designer assigned to the design.
DESIGN_NAME	VARCHAR	Unique name that the user assigned to the design.
STAGE_TYPE	VARCHAR	Design stage that Database Designer was working on at the time indicated by the TIME_STAMP field. Possible values include: • Design in progress • Analyzing data statistics • Optimizing query performance • Optimizing storage footprint • All done • Deployment in progress
ITERATION_NUMBER	INTEGER	Iteration number for the Optimizing query performance stage.
TOTAL_QUERY_COUNT	INTEGER	Total number of design queries in the design.
REMAINING_QUERY_COUNT	INTEGER	Number of design queries remaining for Database Designer to process.
MAX_STEP_NUMBER	INTEGER	Number of steps in the current stage.
CURRENT_STEP_NUMBER	INTEGER	Step in the current stage being processed at the time indicated by the TIME_STAMP field.

CURRENT_STEP_DESCRIPTION	VARCHAR	Name of the step that Database Designer is performing at that time indicated in the TIME_STAMP field. Possible values include: • Design with deployment started • Design in progress: Analyze statistics phase • design_table_name • projection_name • Design in progress: Query optimization phase • Extracting interesting columns • Enumerating sort orders • Setting up projection candidates • Assessing projection candidates • Choosing best projections • Calculating estimated benefit of best projections • Complete • Design in progress: Storage optimization phase • Design completed successfully • Setting up deployment metadata • Identifying projections to be dropped • Running deployment • Deployment completed successfully
TABLE_ID	INTEGER	Unique id that Database Designer assigned to the design table.

Examples

The following example shows the steps that Database Designer performs while optimizing the VMart example database:

```
=> SELECT DESIGNER_CREATE_DESIGN('VMART_DESIGN');
=> SELECT DESIGNER_ADD_DESIGN_TABLES('VMART_DESIGN','public.*');
=> SELECT DESIGNER_ADD_DESIGN_QUERIES('VMART_DESIGN','/tmp/examples/vmart_queries.sql',);
...
=> \x
Expanded display is on.
=> SELECT * FROM OUTPUT_EVENT_HISTORY;
-[ RECORD 1 ] -----------+---------------------------
time_stamp               | 2013-06-05 11:44:41.588
design_id                | 45035996273705090
design_name              | VMART_DESIGN
stage_type               | Design in progress
iteration_number         |
total_query_count        |
remaining_query_count    |
max_step_number          |
current_step_number      |
current_step_description | Design with deployment started
table id                 |
-[ RECORD 2 ] -----------+---------------------------
time_stamp               | 2013-06-05 11:44:41.611
design_id                | 45035996273705090
design_name              | VMART_DESIGN
stage_type               | Design in progress
iteration_number         |
total_query_count        |
remaining_query_count    |
max_step_number          |
current_step_number      |
current_step_description | Design in progress: Analyze statistics phase
table id                 |
-[ RECORD 3 ] -----------+---------------------------
```

```
-[ RECORD 5 ] ----------+----------------------------
time_stamp              | 2013-06-05 11:44:42.011
design_id               | 45035996273705090
design_name             | VMART_DESIGN
stage_type              | Analyzing statistics
iteration_number        |
total_query_count       |
remaining_query_count   |
max_step_number         | 15
current_step_number     | 1
current_step_description | public.customer_dimension
table id                |
...
-[ RECORD 20 ] ----------+----------------------------
time_stamp              | 2013-06-05 11:44:49.324
design_id               | 45035996273705090
design_name             | VMART_DESIGN
stage_type              | Optimizing query performance
iteration_number        | 1
total_query_count       | 9
remaining_query_count   | 9
max_step_number         | 7
current_step_number     | 1
current_step_description | Extracting interesting columns
table id                |
...
-[ RECORD 62 ] ----------+----------------------------
time_stamp              | 2013-06-05 11:51:23.790
design_id               | 45035996273705090
design_name             | VMART_DESIGN
stage_type              | Deployment in progress
iteration_number        |
total_query_count       |
remaining_query_count   |
max_step_number         |
current_step_number     |
current_step_description | Deployment completed successfully
table id                |
```

PARTITION_COLUMNS

For each projection of a partitioned table, shows the following information:

- Disk space used by each column per node.
- Statistics that were collected on partition columns

Disk usage

The column DISK_SPACE_BYTES shows how much disk space the partitioned data uses, including deleted data. So, if you delete rows but do not purge them, the DELETED_ROW_COUNT column changes to show the number of deleted rows in each column; however, DISK_SPACE_BYTES remains unchanged. After deleted rows are purged, Vertica,reclaims the disk space: DISK_SPACE_BYTES changes accordingly, and DELETED_ROW_COUNT is reset to 0.

For grouped partitions, PARTITION_COLUMNS shows the cumulative disk space used for each column per grouped partition. The column GROUPED_PARTITION_KEY, if not null, identifies the partition in which a given column is grouped.

Statistics

STATISTICS_TYPE always shows the most complete type of statistics that are available on a given column, irrespective of timestamp. For example, if you collect statistics for a table on all levels— table , partition , and row , STATISTICS_TYPE is set to FULL (table-level), even if partition- and row-level statistics were collected more recently.

Column Name	Data Type	Description

COLUMN_NAME	VARCHAR	Identifies a named column within the partitioned table.
COLUMN_ID	INTEGER	Unique numeric ID assigned by the Vertica, which identifies the column.
TABLE_NAME	VARCHAR	Name of the partitioned table.
PROJECTION_NAME	VARCHAR	Projection name for which information is listed.
PROJECTION_ID	INTEGER	Unique numeric ID assigned by Vertica, which identifies the projection.
NODE_NAME	VARCHAR	Node that hosts partitioned data.
PARTITION_KEY	VARCHAR	Identifies the table partition.
GROUPED_PARTITION_KEY	VARCHAR	Identifies the grouped partition to which a given column belongs.
ROW_COUNT	INTEGER	The total number of partitioned data rows for each column, including deleted rows.
DELETED_ROW_COUNT	INTEGER	Number of deleted partitioned data rows in each column.
DISK_SPACE_BYTES	INTEGER	Amount of space used by partitioned data.
STATISTICS_TYPE	VARCHAR	Specifies what sort of statistics are used for this column, one of the following listed in order of precedence: 1. FULL: Table-level statistics 2. PARTITION: Partition-level statistics 3. ROWCOUNT: Minimal set of statistics and aggregate row counts
STATISTICS_UPDATED_TIMESTAMP	TIMESTAMPTZ	Specifies when statistics of the type specified in STATISTICS_TYPE were collected for this column.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

Examples

Given the following table definition:

```
=> CREATE TABLE messages
(
   time_interval timestamp NOT NULL,
   thread_id varchar(32) NOT NULL,
   unique_id varchar(53) NOT NULL,
   msg_id varchar(65),
   ...
   )
PARTITION BY ((messages.time_interval)::date);
```

a query on partition_columns might return the following (truncated) results:

```
=> SELECT * FROM partition_columns order by table_name, column_name;
column_name | column_id | table_name | projection_name | projection_id     | node_name        | partition_key | grouped_partition_key  | row_count |
deleted_row_count | disk_space_bytes
--------------------------+----------+---------------+------------------+-----------------+--------------+-----------------------+-----------+-------------------+-----------------
msg_id | 45035996273743190 | messages | messages_super | 45035996273743182 | v_vmart_node0002 | 2010-07-03    |                       | 6147      | 0
| 41145
msg_id | 45035996273743190 | messages | messages_super | 45035996273743182 | v_vmart_node0002 | 2010-07-15    |                       | 178       | 0
| 65
msg_id | 45035996273743190 | messages | messages_super | 45035996273743182 | v_vmart_node0003 | 2010-07-03    |                       | 6782      | 0
| 45107
msg_id | 45035996273743190 | messages | messages_super | 45035996273743182 | v_vmart_node0003 | 2010-07-04    |                       | 866       | 0
| 5883
...

thread_id | 45035996273743186 | messages | messages_super | 45035996273743182 | v_vmart_node0002 | 2010-07-03 |                       | 6147      | 0
| 70565
thread_id | 45035996273743186 | messages | messages_super | 45035996273743182 | v_vmart_node0002 | 2010-07-15 |                       | 178       | 0
| 2429
thread_id | 45035996273743186 | messages | messages_super | 45035996273743182 | v_vmart_node0003 | 2010-07-03 |                       | 6782      | 0
| 77730
thread_id | 45035996273743186 | messages | messages_super | 45035996273743182 | v_vmart_node0003 | 2010-07-04 |                       | 866       | 0
| 10317
...

time_interval | 45035996273743184 | messages | messages_super | 45035996273743182 | v_vmart_node0002 | 2010-07-03 |                       | 6147      | 0
| 6320
time_interval | 45035996273743184 | messages | messages_super | 45035996273743182 | v_vmart_node0002 | 2010-07-15 |                       | 178       | 0
| 265
time_interval | 45035996273743184 | messages | messages_super | 45035996273743182 | v_vmart_node0003 | 2010-07-03 |                       | 6782      | 0
| 6967
time_interval | 45035996273743184 | messages | messages_super | 45035996273743182 | v_vmart_node0003 | 2010-07-04 |                       | 866       | 0
| 892
...

unique_id | 45035996273743188 | messages | messages_super | 45035996273743182 | v_vmart_node0002 | 2010-07-03 |                       | 6147      | 0
| 70747
unique_id | 45035996273743188 | messages | messages_super | 45035996273743182 | v_vmart_node0002 | 2010-07-15 |                       | 178       | 0
| 2460
unique_id | 45035996273743188 | messages | messages_super | 45035996273743182 | v_vmart_node0003 | 2010-07-03 |                       | 6782      | 0
| 77959
unique_id | 45035996273743188 | messages | messages_super | 45035996273743182 | v_vmart_node0003 | 2010-07-04 |                       | 866       | 0
| 10332
unique_id | 45035996273743188 | messages | messages_super | 45035996273743182 | v_vmart_node0003 | 2010-07-15 |                       | 184       | 0
| 2549
...

(11747 rows)
```

PARTITION_REORGANIZE_ERRORS

Monitors all background partitioning tasks, and if Vertica encounters an error, creates an entry in this table with the appropriate information. Does not log repartitioning tasks that complete successfully.

Column Name	Data Type	Description
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
USER_NAME	VARCHAR	Name of the user who received the error at the time Vertica recorded the session.

NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
TABLE_NAME	VARCHAR	Name of the partitioned table.
PROJECTION_ID	INTEGER	Catalog-assigned numeric value that uniquely identifies the projection.
PROJECTION_NAME	VARCHAR	Projection name for which information is listed.
MESSAGE	VARCHAR	Textual output of the error message.
HINT	VARCHAR	Actionable hint about the error.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

PARTITION_STATUS

For each projection of each partitioned table, shows the fraction of its data that is actually partitioned according to the current partition expression. When the partitioning of a table is altered, the value in PARTITION_REORGANIZE_PERCENT for each of its projections drops to zero and goes back up to 100 when all the data is repartitioned.

Column Name	Data Type	Description
PROJECTION_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the projection.
TABLE_SCHEMA	VARCHAR	Name of the schema that contains the partitioned table.
TABLE_NAME	VARCHAR	Table name that is partitioned.
TABLE_ID	INTEGER	Unique numeric ID assigned by the Vertica, which identifies the table.
PROJECTION_SCHEMA	VARCHAR	Schema containing the projection.
PROJECTION_NAME	VARCHAR	Projection name for which information is listed.
PARTITION_REORGANIZE_PERCENT	INTEGER	For each projection, drops to zero and goes back up to 100 when all the data is repartitioned after the partitioning of a table has been altered. Ideally all rows will show 100 (%).

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

PARTITIONS

Displays partition metadata, one row per partition key, per ROS container.

Column Name	Data Type	Description
PARTITION_KEY	VARCHAR	The partition value(s).
PROJECTION_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the projection.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.
PROJECTION_NAME	VARCHAR	The projection name for which information is listed.
ROS_ID	VARCHAR	A unique numeric ID assigned by the Vertica catalog, which identifies the ROS container.

ROS_SIZE_BYTES	INTEGER	The ROS container size in bytes.
ROS_ROW_COUNT	INTEGER	Number of rows in the ROS container.
NODE_NAME	VARCHAR	Node where the ROS container resides.
DELETED_ROW_COUNT	INTEGER	The number of deleted rows in the partition.
LOCATION_LABEL	VARCHAR	The location label of the default storage location.

Notes

- A many-to-many relationship exists between partitions and ROS containers. PARTITIONS displays information in a denormalized fashion.
- To find the number of ROS containers having data of a specific partition, aggregate PARTITIONS over the partition_key column.
- To find the number of partitions stored in a ROS container, aggregate PARTITIONS over the ros_id column.

Examples

See Viewing partition storage data .

PROCESS_SIGNALS

Returns a history of signals that were received and handled by the Vertica process. For details about signals, see the Linux documentation .

Column Name	Data Type	Description
SIGNAL_TIMESTAMP	TIMESTAMPTZ	Time when Vertica recorded the signal.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
SIGNAL_NUMBER	INTEGER	Signal number, refers to POSIX SIGNAL_NUMBER
SIGNAL_CODE	INTEGER	Signal code.
SIGNAL_PID	INTEGER	Linux process identifier of the signal.
SIGNAL_UID	INTEGER	Process ID of sending process.
SIGNAL_ADDRESS	INTEGER	Address at which fault occurred.

Privileges

Superuser

PROJECTION_RECOVERIES

Retains history about projection recoveries. Because Vertica adds an entry per recovery plan, a projection/node pair might appear multiple times in the output.

> **Note**
> You cannot query this or other system tables during cluster recovery; the cluster must be UP to accept connections.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is recovering or has recovered the corresponding projection.
PROJECTION_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	Name of the projection that is being or has been recovered on the corresponding node.

TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any. TRANSACTION_ID initializes as NO_TRANSACTION with a value of 0. Vertica will ignore the recovery query and keep (0) if there's no action to take (no data in the table, etc). When no recovery transaction starts, ignored value appears in this table's STATUS column.
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID uniquely identifies a statement within a session.
METHOD	VARCHAR	Recovery method that Vertica chooses. Possible values are: • incremental • incremental-replay-delete • split • recovery-by-container
STATUS	VARCHAR	Current projection-recovery status on the corresponding node. STATUS can be "queued," which indicates a brief period between the time the query is prepared and when it runs. Possible values are: • queued • running • finished • ignored • error-retry • error-fatal
PROGRESS	INTEGER	An estimate (value in the range [0,100]) of percent complete for the recovery task described by this information. **Note:** The actual amount of time it takes to complete a recovery task depends on a number of factors, including concurrent workloads and characteristics of the data; therefore, accuracy of this estimate can vary. The PROGRESS column value is NULL after the task completes.
DETAIL	VARCHAR	More detailed information about PROGRESS. The values returned for this column depend on the type of recovery plan: • General recovery plans – value displays the estimated progress, as a percent, of the three primary parts of the plan: Scan , Sort, and Write . • Recovery-by-container plans – value begins with CopyStorage: and is followed by the number of bytes copied over the total number of bytes to copy. • Replay delete plans – value begins with Delete: and is followed by the number of deletes replayed over an estimate of the total number of deletes to replay. The DETAIL column value becomes NULL after the recovery plan completes.
START_TIME	TIMESTAMPTZ	Time the recovery task described by this information started.
END_TIME	TIMESTAMPTZ	Time the recovery task described by this information ended.
RUNTIME_PRIORITY	VARCHAR	Determines the amount of runtime resources (CPU, I/O bandwidth) the Resource Manager should dedicate to running queries in the resource pool. Valid values are: • HIGH • MEDIUM • LOW

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

See also

RECOVERY_STATUS

PROJECTION_REFRESHES

System table PROJECTION_REFRESHES records information about refresh operations , successful and unsuccessful. PROJECTION_REFRESHES retains projection refresh data until one of the following events occurs:

- Another refresh operation starts on a given projection.
- CLEAR_PROJECTION_REFRESHES is called and clears data on all projections.
- The table's storage quota is exceeded.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node where the refresh was initiated.
PROJECTION_SCHEMA	VARCHAR	Name of the projection schema.
PROJECTION_ID	INTEGER	Catalog-assigned numeric value that uniquely identifies the projection.
PROJECTION_NAME	VARCHAR	Name of the refreshed projection.
ANCHOR_TABLE_NAME	VARCHAR	Name of the projection's anchor table.
REFRESH_STATUS	VARCHAR	Status of refresh operations for this projection, one of the following: • Queued : Projection is queued for refresh. • Refreshing: Projection refresh is in progress. • Refreshed: Projection refresh is complete. • Failed: Projection refresh failed.
PERCENT_COMPLETE	VARCHAR	Shows the current percentage of completion for the refresh operation. When the refresh is complete, the column is set to NULL.
REFRESH_PHASE	VARCHAR	Indicates how far the refresh has progressed: • Historical: Refresh reached the first phase and is refreshing data from historical data. This refresh phase requires the most amount of time. • Current: Refresh reached the final phase and is attempting to refresh data from the current epoch. To complete this phase, refresh must obtain a lock on the table. If the table is locked by another transaction, refresh is blocked until that transaction completes. The LOCKS system table is useful for determining if a refresh is blocked on a table lock. To determine if a refresh has been blocked, locate the term "refresh" in the transaction description. A refresh has been blocked when the scope for the refresh is REQUESTED and other transactions acquired a lock on the table. This field is NULL until the projection starts to refresh and is NULL after the refresh completes.
REFRESH_METHOD	VARCHAR	Method used to refresh the projection: • Buddy: Projection refreshed from the contents of a buddy projection. This method maintains historical data, so the projection can used for historical queries. • Scratch: Projection refreshed without using a buddy projection. This method does not generate historical data, so the projection cannot participate in historical queries on data that precedes the refresh. • Rebalance: If the projection is segmented, it is refreshed from scratch; if unsegmented, it is refreshed from a buddy projection.

REFRESH_FAILURE_COUNT	INTEGER	Number of times a refresh failed for the projection. REFRESH_FAILURE_COUNT does not indicate whether the projection was eventually refreshed. See REFRESH_STATUS to determine whether the refresh operation is progressing.
SESSION_ID	VARCHAR	Unique numeric ID assigned by the Vertica catalog, which identifies the refresh session.
REFRESH_START	TIMESTAMPTZ	Time the projection refresh started.
REFRESH_DURATION_SEC	INTERVAL SECOND (0)	How many seconds the projection refresh ran.
IS_EXECUTING	BOOLEAN	Differentiates active and completed refresh operations.
RUNTIME_PRIORITY	VARCHAR	Determines how many run-time resources (CPU, I/O bandwidth) the Resource Manager should dedicate to running queries in the resource pool, one of the following: • HIGH • MEDIUM • LOW
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL. **Note** The transaction_id is correlated with the execution plan only when refreshing from scratch. When refreshing from a buddy, multiple sub-transactions are created

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

PROJECTION_STORAGE

Monitors the amount of disk storage used by each projection on each node.

> **Note**
> Projections that have no data never have full statistics. Querying this system table lets you see if your projection contains data.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
PROJECTION_ID	VARCHAR	Catalog-assigned numeric value that uniquely identifies the projection.
PROJECTION_NAME	VARCHAR	The projection name for which information is listed.
PROJECTION_SCHEMA	VARCHAR	The name of the schema associated with the projection.
PROJECTION_COLUMN_COUNT	INTEGER	The number of columns in the projection.
ROW_COUNT	INTEGER	The number of rows in the table's projections, including any rows marked for deletion.
USED_BYTES	INTEGER	Number of bytes in disk storage used to store the compressed projection data. This value should not be compared to the output of the AUDIT function, which returns the raw data size of database objects.

ROS_COUNT	INTEGER	The number of ROS containers in the projection.
ANCHOR_TABLE_NAME	VARCHAR	The associated table name for which information is listed.
ANCHOR_TABLE_SCHEMA	VARCHAR	The associated table schema for which information is listed.
ANCHOR_TABLE_ID	INTEGER	A unique numeric ID, assigned by the Vertica catalog, which identifies the anchor table.

See also

- PROJECTIONS
- ANALYZE_STATISTICS

PROJECTION_USAGE

Records information about projections Vertica used in each processed query.

Column Name	Data Type	Description
QUERY_START_TIMESTAMP	TIMESTAMPTZ	Value of query at beginning of history interval.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_NAME	VARCHAR	Name of the user at the time Vertica recorded the session.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL .
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID, and REQUEST_ID uniquely identifies a statement within a session.
IO_TYPE	VARCHAR	Input/output.
PROJECTION_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	Projection name for which information is listed.
ANCHOR_TABLE_ID	INTEGER	Unique numeric ID assigned by the Vertica, which identifies the anchor table.
ANCHOR_TABLE_SCHEMA	VARCHAR	Name of the schema that contains the anchor table.
ANCHOR_TABLE_NAME	VARCHAR	Name of the projection's associated anchor table.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

QUERY_CONSUMPTION

Summarizes execution of individual queries. Columns STATEMENT_ID and TRANSACTION_ID combine as unique keys to these queries. One exception applies: a query with multiple plans has as many records.

Column Name	Data Type	Description
START_TIME END_TIME	TIMESTAMP	Demarcate the start and end of query execution, whether successful or not.

SESSION_ID	VARCHAR	Identifies the session where profiling information was captured. This identifier is unique within the cluster at any point in time, but can be reused when the session closes.
USER_ID	INTEGER	Unique numeric user identifier assigned by the Vertica catalog.
USER_NAME	VARCHAR	User name specified by this query profile.
TRANSACTION_ID	INTEGER	Identifies the transaction in which the query ran.
STATEMENT_ID	INTEGER	Numeric identifier of this query, unique within the query transaction.
CPU_CYCLES_US	INTEGER	Sum, in microseconds, of CPU cycles spent by all threads to process this query.
NETWORK_BYTES_SENT NETWORK_BYTES_RECEIVED	INTEGER	Total amount of data sent/received over the network by execution engine operators.
DATA_BYTES_READ DATA_BYTES_WRITTEN	INTEGER	Total amount of data read/written by storage operators from and to disk, includes all locations: local, HDFS, S3.
DATA_BYTES_LOADED	INTEGER	Total amount of data loaded from external sources: COPY, external tables, and data load.
BYTES_SPILLED	INTEGER	Total amount of data spilled to disk—for example, by SortManager, Join, and NetworkSend operators.
INPUT_ROWS	INTEGER	Number of unfiltered input rows from DataSource and Load operators. INPUT_ROWS shows the number of input rows that the query plan worked with, but excludes intermediate processing. For example, INPUT_ROWS excludes how many times SortManager spilled and read the same row.
INPUT_ROWS_PROCESSED	INTEGER	Value of INPUT_ROWS minus what was filtered by applying query predicates (valindex) and SIPs, and rows rejected by COPY.
PEAK_MEMORY_KB	INTEGER	Peak memory reserved by the resource manager for this query.
THREAD_COUNT	INTEGER	Maximum number of threads opened to process this query.
DURATION_MS	INTEGER	Total wall clock time, in milliseconds, spent to process this query.
RESOURCE_POOL	VARCHAR	Name of the resource pool where the query was executed.
OUTPUT_ROWS	INTEGER	Number of rows output to the client.
REQUEST_TYPE	VARCHAR	Type of query—for example, QUERY or DDL.
LABEL	VARCHAR	Label included as a LABEL hint in this query.
IS_RETRY	BOOLEAN	This query was tried earlier.
SUCCESS	BOOLEAN	This query executed successfully.

QUERY_EVENTS

Returns information about query planning, optimization, and execution events.

Column Name	Data Type	Description
EVENT_TIMESTAMP	TIMESTAMPTZ	Time when Vertica recorded the event.

NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_ID	INTEGER	Identifier of the user for the query event.
USER_NAME	VARCHAR	Name of the user for which Vertica lists query information at the time it recorded the session.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
REQUEST_ID*	INTEGER	Unique identifier of the query request in the user session.
TRANSACTION_ID*	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL .
STATEMENT_ID*	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed.
EVENT_CATEGORY	VARCHAR	Category of event: OPTIMIZATION or EXECUTION.
EVENT_TYPE	VARCHAR	Type of event. For details on each type, see the following sections: • Informational Event Types • Event Types for Corrective Action • Critical Event Types
EVENT_DESCRIPTION	VARCHAR	Generic description of the event.
OPERATOR_NAME	VARCHAR	Name of the Execution Engine component that generated the event, if applicable; for example, NetworkSend. Values from the OPERATOR_NAME and PATH_ID columns let you tie a query event back to a particular operator in the query plan. If the event did not come from a specific operator, the OPERATOR_NAME column is NULL.
PATH_ID	INTEGER	Unique identifier that Vertica assigns to a query operation or path in a query plan, NULL if the event did not come from a specific operaton. For more information, see EXECUTION_ENGINE_PROFILES .
OBJECT_ID	INTEGER	Object identifier such as projection or table to which the event refers.
EVENT_DETAILS	VARCHAR	Free-form text describing the specific event.
EVENT_SEVERITY	VARCHAR	Indicates severity of the event with one of the following values: • Informational : No action required • Warning : Remedial action recommended as specified in SUGGESTED_ACTION • Critical : Remedial action required, as specified by SUGGESTED_ACTION
SUGGESTED_ACTION	VARCHAR	Specifies remedial action, recommended or required as indicated by EVENT_SEVERITY .

Informational event types

Event Type	**Description**
CSE ANALYSIS	The optimizer performed Common subexpressions analysis
CSE ANALYSIS STATS	Time spent on Common subexpressions analysis (msec)
EXPRESSION_EVAL_ERROR	An exception occurred during evaluation of an expression

EXTERNAL_PREDICATE_PUSHDOWN_NOT_SUPPORTED	Predicate pushdown for older Hive versions may not be supported. For more information, see Querying external tables .
FLATTENED SUBQUERIES	Subqueries flattened in FROM clause
GROUP_BY_PREPASS_FALLBACK	Vertica could not run an optimization. In-memory prepass is disabled. The projection may not be optimal.
GROUPBY PUSHDOWN	Internal to Vertica
LibHDFS++ FAILOVER RETRY	Vertica attempted to contact a NameNode on an HDFS cluster that uses High Availability NameNode and did not receive a response. Vertica retried with a different NameNode.
LibHDFS++ MANUAL FALLBACK	Vertica accessed HDFS using the hdfs URL scheme but HDFSUseWebHDFS is set. Vertica fell back to WebHDFS.
LibHDFS++ UNSUPPORTED OPERATION	Vertica accessed HDFS using the hdfs URL scheme, but the HDFS cluster uses an unsupported feature such as wire encryption or HTTPS_ONLY or the Vertica session uses delegation tokens. Vertica fell back to WebHDFS.
MERGE_CONVERTED_TO_UNION	Vertica has converted a merge operator to a union operator due to the sort order of the multi-threaded storage access stream.
NO GROUPBY PUSHDOWN	Internal to Vertica
NODE PRUNING	Vertica performed node pruning, which is similar to partition pruning, but at the node level.
ORC_FILE_INFO	A query of ORC files encountered missing information (such as time zone) or an unrecognized ORC version. For missing information, Vertica uses a default value (such as the local time zone).
ORC_SOURCE_PRUNED	An entire ORC file was pruned during predicate pushdown.
ORC_STRIPES_PRUNED	The identified stripes were pruned during predicate pushdown. If an entire ORC file was pruned, it is instead recorded with an ORC_SOURCE_PRUNED event.
OUTER OVERRIDE NOT USED	Vertica found swapping inner/outer tables in a join unnecessary because the inner/outer tables were in good order. (For example, a smaller table was used in an inner join.)
OUTER OVERRIDE USED	For efficiency and optimization, Vertica has swapped the inner/outer tables in a join. Vertica used the smaller table as the inner table.
PARQUET_ROWGROUPS_PRUNED	The identified row groups were pruned during predicate pushdown.
PARTITION_PATH_PRUNED	A path (reported in event details) was pruned.
PARTITION_PATH_REJECTED	Could not evaluate partition column predicate on a path from source list. Path will be rejected.
PARTITION_PRUNING	COPY pruned partitions. The event reports how many paths were pruned, and PARTITION_PATH_PRUNED events record more details.
PREDICATES_DISCARDED_FROM_SCAN	Some predicates have been discarded from this scan because expression analysis shows they are not needed.
REJECT_ROWNUMS_HIT_BUFFER_LIMIT	Buffering row numbers during rejection hit buffer limit
SEQUENCE CACHE REFILLED	Vertica has refilled sequence cache.

SIP_FALLBACK	This optimization did not apply to this query type.
SMALL_MERGE_REPLACED	Vertica has chosen a more efficient way to access the data by replacing a merge.
STORAGE_CONTAINERS_ELIMINATED	Vertica has performed partition pruning for the purpose of optimization.
TRANSITIVE PREDICATE	Vertica has optimized by adding predicates to joins where it makes logical sense to do so. For example, for the statement, SELECT * FROM A, B WHERE A.a = B.a AND A.a = 1; Vertica may add a predicate B a = 1 as a filter for better storage access of table B .
TYPE_MISMATCH_COLUMNS_PARQUETPARSER	The Parquet parser used loose schema matching to load data, and could not coerce values in the Parquet data to the types defined for the table. By default the parser rejects the row. For more information, see PARQUET .
UNMATCHED_TABLE_COLUMNS_PARQUETPARSER	The Parquet parser used loose schema matching to load data, and columns in the table had no corresponding columns in the data. The columns were given values of NULL.
VALUE_TRUNCATED	A character value is too long.
WEBHDFS FAILOVER RETRY	Vertica attempted to contact a NameNode on an HDFS cluster that uses High Availability NameNode and did not receive a response. Vertica retried with a different NameNode.

Warning event types

Review the following event types and recommended actions:

Event Type	**Description**	**Recommended Action**
AUTO_PROJECTION_USED	The optimizer used an auto-projection to process this query.	Create a projection that is appropriate for this query and others like it; consider using Database Designer to generate query-specific projections.
GROUP_BY_SPILLED	This event type is typically related to a specific type of query, which you might need to adjust.	Identify the type of query and make adjustments accordingly. You might need to adjust resource pools, projections, or the amount of RAM available. Try running the query on a cluster with no additional workload.
INVALID COST	When creating a query plan, the optimizer calculated an invalid cost for a path: not-a-number (NaN) value, infinity value, or negative value. The path cost was set to its default value.	No action available to users.
PATTERN_MATCH_NMEE	More than one pattern event is true for a single row	Modify event expressions to ensure that only one event can be true for any row. Alternatively, modify the query using a MATCH clause with ROWS MATCH FIRST EVENT .
PREDICATE OUTSIDE HISTOGRAM	A predicate value you are trying to match does not exist in a set of possible values for a specific column. For example, you try to match a VARCHAR value WHERE mystring = "ABC<newline>". In this case, the newline character throws off the predicate matching optimizations.	Run ANALYZE_STATISTICS on the column.

RESEGMENTED_MANY_ROWS	This event type is typically related to a specific type of query, which you might need to adjust.	Do projections need to be segmented in a different way to allow for join locality? Can you rewrite the query to filter out more rows at storage access time? (Typically, Vertica does so automatically through predicate pushdown.) Review your explain plan.
RLE_OVERRIDDEN	The average run counts are not large enough for Run Length Encoding (RLE). This event occurs with queries where the filtered results for certain columns do not work with RLE because cardinality is less than 10.	Review and rewrite your query, if necessary.

Critical event types

Review the following event types, and resolve issues as recommended:

Event Type	Description	Required Action
DELETE WITH NON OPTIMIZED PROJECTION	One or more projections do not have your delete filter column in their sort order, causing Vertica difficulty identifying ros to mark as deleted.	Add the delete filter column to the end of every projection sort order for your target delete table.
JOIN_SPILLED	Vertica has spilled a join to disk. A join spill event slows down the subject query and all other queries as it consumes resources while using disk as virtual memory.	Try the following: 1. Review the explain plan. The query might be too ambitious, for example, cross joining two large tables. 2. Consider adding the query to a lower priority pool to reduce impact on other queries. 3. Create projections that allow for a merge join instead of a hash join. 4. Adjust the PLANNEDCONCURRENCYresource pool so that queries have more memory to execute.
MEMORY LIMIT HIT	Indicates query complexity or, possibly, lack of available system memory.	Consider adjusting the MAXMEMORYSIZE and PLANNEDCONCURRENCY resource pools so that the optimizer has sufficient memory. On a heavily used system, this event may occur more frequently.
NO HISTOGRAM	Indicates a table does not have an updated column histogram.	Running the function ANALYZE_STATISTICS most often corrects this issue.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

See also

- EXECUTION_ENGINE_PROFILES
- QUERY_CONSUMPTION
- QUERY_PLAN_PROFILES

QUERY_METRICS

Monitors the sessions and queries running on each node.

> **Note**
> Totals in this table are reset each time the database restarts.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
ACTIVE_USER_SESSION_COUNT	INTEGER	The number of active user sessions (connections).
ACTIVE_SYSTEM_SESSION_COUNT	INTEGER	The number of active system sessions.
TOTAL_USER_SESSION_COUNT	INTEGER	The total number of user sessions.
TOTAL_SYSTEM_SESSION_COUNT	INTEGER	The total number of system sessions.
TOTAL_ACTIVE_SESSION_COUNT	INTEGER	The total number of active user and system sessions.
TOTAL_SESSION_COUNT	INTEGER	The total number of user and system sessions.
RUNNING_QUERY_COUNT	INTEGER	The number of queries currently running.
EXECUTED_QUERY_COUNT	INTEGER	The total number of queries that ran.

QUERY_PLAN_PROFILES

Provides detailed execution status for queries that are currently running in the system. Output from the table shows the real-time flow of data and the time and resources consumed for each path in each query plan.

Column Name	Data Type	Description
TRANSACTION_ID	INTEGER	An identifier for the transaction within the session if any; otherwise NULL .
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID and STATEMENT_ID uniquely identifies a statement within a session; these columns are useful for creating joins with other system tables.
PATH_ID	INTEGER	Unique identifier that Vertica assigns to a query operation or path in a query plan. Textual representation for this path is output in the PATH_LINE column.
PATH_LINE_INDEX	INTEGER	Each plan path in QUERY_PLAN_PROFILES could be represented with multiple rows. PATH_LINE_INDEX returns the relative line order. You should include the PATH_LINE_INDEX column in the QUERY_PLAN_PROFILES ... ORDER BY clause so rows in the result set appear as they do in EXPLAIN-generated query plans.
PATH_IS_EXECUTING	BOOLEAN	Status of a path in the query plan. True (*t*) if the path has started running, otherwise false.
PATH_IS_COMPLETE	BOOLEAN	Status of a path in the query plan. True (*t*) if the path has finished running, otherwise false.
IS_EXECUTING	BOOLEAN	Status of a running query. True if the query is currently active (*t*), otherwise false (*f*).
RUNNING_TIME	INTERVAL	The amount of elapsed time the query path took to execute.
MEMORY_ALLOCATED_BYTES	INTEGER	The amount of memory the path used, in bytes.
READ_FROM_DISK_BYTES	INTEGER	The number of bytes the path read from disk (or the disk cache).
RECEIVED_BYTES	INTEGER	The number of bytes received over the network.

SENT_BYTES	INTEGER	Size of data sent over the network by the path.
PATH_LINE	VARCHAR	The query plan text string for the path, associated with the PATH ID and PATH_LINE_INDEX columns.

Privileges

Non-superusers see only the records of tables they have permissions to view.

Best practices

Table results can be very wide. For best results when you query QUERY_PLAN_PROFILES , sort on these columns:

- TRANSACTION_ID
- STATEMENT_ID
- PATH_ID
- PATH_LINE_INDEX

For example:

```
=> SELECT ... FROM query_plan_profiles
    WHERE ...
   ORDER BY transaction_id, statement_id, path_id, path_line_index;
```

Examples

See Profiling query plans

See also

- EXECUTION_ENGINE_PROFILES
- EXPLAIN
- PROFILE
- QUERY_CONSUMPTION
- QUERY_EVENTS

QUERY_PROFILES

Provides information about executed queries.

Column Name	Data Type	Description
SESSION_ID	VARCHAR	The identification of the session for which profiling information is captured. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
TRANSACTION_ID	INTEGER	An identifier for the transaction within the session if any; otherwise NULL .
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID , STATEMENT_ID uniquely identifies a statement within a session.
IDENTIFIER	VARCHAR	A string to identify the query in system tables. **Note:** You can query the IDENTIFIER column to quickly identify queries you have labeled for profiling and debugging. See Labeling statements for details.
NODE_NAME	VARCHAR	The node name for which information is listed.
QUERY	VARCHAR	The query string used for the query.
QUERY_SEARCH_PATH	VARCHAR	A list of schemas in which to look for tables.
SCHEMA_NAME	VARCHAR	The schema name in which the query is being profiled, set only for load operations.

TABLE_NAME	VARCHAR	The table name in the query being profiled, set only for load operations.
QUERY_DURATION_US	NUMERIC(18,0)	The duration of the query in microseconds.
QUERY_START_EPOCH	INTEGER	The epoch number at the start of the given query.
QUERY_START	VARCHAR	The Linux system time of query execution in a format that can be used as a DATE/TIME expression.
QUERY_TYPE	VARCHAR	Is one of INSERT , SELECT , UPDATE , DELETE , UTILITY , or UNKNOWN .
ERROR_CODE	INTEGER	The return error code for the query.
USER_NAME	VARCHAR	The name of the user who ran the query.
PROCESSED_ROW_COUNT	INTEGER	The number of rows returned by the query.
RESERVED_EXTRA_MEMORY_B	INTEGER	Shows how much unused memory (in bytes) remains that is reserved for a given query but is unassigned to a specific operator. This is the memory from which unbounded operators pull first. The MEMORY_INUSE_KB column in system table RESOURCE_ACQUISITIONS shows how much total memory was acquired for each query. If operators acquire all memory acquired for the query, the plan must request more memory from the Vertica resource manager .
IS_EXECUTING	BOOLEAN	Displays information about actively running queries, regardless of whether profiling is enabled.

QUERY_REQUESTS

Returns information about user-issued query requests.

Column Name	**Data Type**	**Description**
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_NAME	VARCHAR	Name of the user who issued the query at the time Vertica recorded the session.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL .
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID, and REQUEST_ID uniquely identifies a statement within a session.

REQUEST_TYPE	VARCHAR	Type of the query request. Examples include, but are not limited to: • QUERY • DDL • LOAD • UTILITY • TRANSACTION • PREPARE • EXECUTE • SET • SHOW
REQUEST	VARCHAR	Query statement.
REQUEST_LABEL	VARCHAR	Label of the query, if available.
SEARCH_PATH	VARCHAR	Contents of the search path.
MEMORY_ACQUIRED_MB	FLOAT	Memory acquired by this query request in megabytes.
SUCCESS	BOOLEAN	Value returned if the query successfully executed.
ERROR_COUNT	INTEGER	Number of errors encountered in this query request (logged in ERROR_MESSAGES table).
START_TIMESTAMP	TIMESTAMPTZ	Beginning of history interval.
END_TIMESTAMP	TIMESTAMPTZ	End of history interval.
REQUEST_DURATION	TIMESTAMPTZ	Length of time in days, hours, minutes, seconds, and milliseconds.
REQUEST_DURATION_MS	INTEGER	Length of time the query ran in milliseconds.
IS_EXECUTING	BOOLEAN	Distinguishes between actively-running (t) and completed (f) queries.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

See also

QUERY_PROFILES

REBALANCE_OPERATIONS

Contains information on historic and ongoing rebalance operations.

Column Name	Data Type	Description
OBJECT_TYPE	VARCHAR	The type of the rebalanced object: • Projection • DFSfile
OBJECT_ID	INTEGER	The ID of the rebalanced object.
OBJECT_NAME	VARCHAR	The name of the rebalanced object. Objects can be tables, projections, or other Vertica objects.
PATH_NAME	VARCHAR	The DFS path for unstructured data being rebalanced.
TABLE_NAME	VARCHAR	The name of the rebalanced table. This value is NULL for DFS files.
TABLE_SCHEMA	VARCHAR	The schema of the rebalanced table. This value is NULL for DFS files.

TRANSACTION_ID	INTEGER	The identifier for the transaction within the session.
STATEMENT_ID	INTEGER	The unique numeric ID for the currently-running statement.
NODE_NAME	VARCHAR	Name of the rebalancing node.
OPERATION_NAME	VARCHAR	Identifies the specific rebalance operation being performed, one of: • Refresh projection, update temporary projection name and ID to *master projection name* • Drop unsegmented replicas • Replicate DFS File • Refresh projection • Drop replaced or replacement projection, rename *temporary projection name* to *original projection name* • Update temp table segments • Prepare : separate • Move storage containers
OPERATION_STATUS	VARCHAR	Specifies status of the rebalance operation, one of the followin: • START • COMPLETE • ABORT
IS_EXECUTING	BOOLEAN	TRUE: the operation is currently running.
REBALANCE_METHOD	VARCHAR	The method that Vertica is using to perform the rebalance, one of the following: • REFRESH: New projections are created according to the new segmentation definition. Data is copied via a refresh plan from projections with the previous segmentation to the new segments. This method is used only if START_REFRESH is called, a configuration parameter is set, or K-safety changes. • REPLICATE: Unsegmented projection data is copied to new nodes and removed from ephemeral nodes. • ELASTIC_CLUSTER: The segmentation of existing segmented projections is altered to adjust to a new cluster topology and data is redistributed accordingly.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
OPERATION_START_TIMESTAMP	TIMESTAMPTZ	The time that the rebalance began.
OPERATION_END_TIMESTAMP	TIMESTAMPTZ	The time that the rebalance ended. If the rebalance is ongoing, this value is NULL.
ELASTIC_CLUSTER_VERSION	INTEGER	The Elastic Cluster has a version. Each time the cluster topology changes, this version increments.
IS_LATEST	BOOLEAN	True if this row pertains to the most recent rebalance activity.

Privileges

Superuser

REBALANCE_PROJECTION_STATUS

Maintain history on rebalance progress for relevant projections.

Column Name	**Data Type**	**Description**

PROJECTION_ID	INTEGER	Identifier of the projection to rebalance.
PROJECTION_SCHEMA	VARCHAR	Schema of the projection to rebalance.
PROJECTION_NAME	VARCHAR	Name of the projection to rebalance.
ANCHOR_TABLE_ID	INTEGER	Anchor table identifier of the projection to rebalance.
ANCHOR_TABLE_NAME	VARCHAR	Anchor table name of the projection to rebalance.
REBALANCE_METHOD	VARCHAR	Method used to rebalance the projection, one of the following: • REFRESH : New projections are created according to the new segmentation definition. Data is copied via a refresh plan from projections with the previous segmentation to the new segments. This method is used only if START_REFRESH is called, a configuration parameter is set, or K-safety changes. • REPLICATE : Unsegmented projection data is copied to new nodes and removed from ephemeral nodes. • ELASTIC_CLUSTER : The segmentation of existing segmented projections is altered to adjust to a new cluster topology and data is redistributed accordingly.
DURATION_SEC	INTERVAL SEC	Deprecated, set to NULL.
SEPARATED_PERCENT	NUMERIC(5,2)	Percent of storage that has been separated for this projection.
TRANSFERRED_PERCENT	NUMERIC(5,2)	Percent of storage that has been transferred, for this projection.
SEPARATED_BYTES	INTEGER	Number of bytes, separated by the corresponding rebalance operation, for this projection.
TO_SEPARATE_BYTES	INTEGER	Number of bytes that remain to be separated by the corresponding rebalance operation for this projection.
TRANSFERRED_BYTES	INTEGER	Number of bytes transferred by the corresponding rebalance operation for this projection.
TO_TRANSFER_BYTES	INTEGER	Number of bytes that remain to be transferred by the corresponding rebalance operation for this projection.
IS_LATEST	BOOLEAN	True if this row pertains to the most recent rebalance activity, where elastic_cluster_version = (SELECT version FROM v_catalog.elastic_cluster);
ELASTIC_CLUSTER_VERSION	INTEGER	The elastic cluster has a version, and each time the cluster topology changes, this version is incremented. This column reflects the version to which this row of information pertains. The TO_* fields (TO_SEPARATE_* and TO_TRANSFER_*) are only valid for the current version. To view only rows from the current, latest or upcoming rebalance operation, use: WHERE elastic_cluster_version = (SELECT version FROM v_catalog.elastic_cluster);

Privileges

Superuser

See also

- ELASTIC_CLUSTER
- REBALANCE_TABLE_STATUS

REBALANCE_TABLE_STATUS

Maintain history on rebalance progress for relevant tables.

Column Name	Data Type	Description
TABLE_ID	INTEGER	Identifier of the table that will be, was, or is being rebalanced.
TABLE_SCHEMA	VARCHAR	Schema of the table that will be, was, or is being rebalanced.
TABLE_NAME	VARCHAR	Name of the table that will be, was, or is being rebalanced.
REBALANCE_METHOD	VARCHAR	Method that will be, is, or was used to rebalance the projections of this table. Possible values are: • REFRESH • REPLICATE • ELASTIC_CLUSTER
DURATION_SEC	INTERVAL SEC	Deprecated - populated by NULL. Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS .
SEPARATED_PERCENT	NUMERIC(5,2)	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.
TRANSFERRED_PERCENT	NUMERIC(5,2)	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.
SEPARATED_BYTES	INTEGER	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.
TO_SEPARATE_BYTES	INTEGER	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.
TRANSFERRED_BYTES	INTEGER	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.
TO_TRANSFER_BYTES	INTEGER	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.
IS_LATEST	BOOLEAN	True if this row pertains to the most recent rebalance activity, where elastic_cluster_version = (SELECT version FROM v_catalog.elastic_cluster;)
ELASTIC_CLUSTER_VERSION	INTEGER	The Elastic Cluster has a version, and each time the cluster topology changes, this version is incremented. This column reflects the version to which this row of information pertains. The TO_* fields (TO_SEPARATE_* and TO_TRANSFER_*) are only valid for the current version. To view only rows from the current, latest or upcoming rebalance operation, use: WHERE elastic_cluster_version = (SELECT version FROM v_catalog.elastic_cluster;)
start_timestamp	TIMESTAMPTZ	The time that the rebalance began.
end_timestamp	TIMESTAMPTZ	The time that the rebalance ended.

Privileges

Superuser

See also

- ELASTIC_CLUSTER
- REBALANCE_PROJECTION_STATUS

RECOVERY_STATUS

Provides the status of recovery operations, returning one row for each node.

> **Note**
> You cannot query this or other system tables table during cluster recovery; the cluster must be UP to accept connections.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
RECOVER_EPOCH	INTEGER	Epoch the recovery operation is trying to catch up to.
RECOVERY_PHASE	VARCHAR	Current stage in the recovery process. Can be one of the following: • NULL • current • historical pass *X* , where *X* is the iteration count
SPLITS_COMPLETED	INTEGER	Number of independent recovery SPLITS queries that have run and need to run.
SPLITS_TOTAL	INTEGER	Total number of SPLITS queries that ran. Each query corresponds to one row in the PROJECTION_RECOVERIES table. If SPLITS_TOTAL = 2, then there should be 2 rows added to PROJECTION_RECOVERIES, showing query details.
HISTORICAL_COMPLETED	INTEGER	Number of independent recovery HISTORICAL queries that have run and need to run.
HISTORICAL_TOTAL	INTEGER	Total number of HISTORICAL queries that ran. Each query corresponds to one row in the PROJECTION_RECOVERIES table. If HISTORICAL_TOTAL = 2, then there should be 2 rows added to PROJECTION_RECOVERIES, showing query details.
CURRENT_COMPLETED	INTEGER	Number of independent recovery CURRENT queries that have run and need to run.
CURRENT_TOTAL	INTEGER	Total number of CURRENT queries that ran. Each query corresponds to one row in the PROJECTION_RECOVERIES table. If CURRENT_TOTAL = 2, then there should be 2 rows added to PROJECTION_RECOVERIES, showing query details.
IS_RUNNING	BOOLEAN	True (*t*) if the node is still running recovery; otherwise false (*f*).

Privileges

None

See also

PROJECTION_RECOVERIES

REMOTE_REPLICATION_STATUS

Provides the status of replication tasks to alternate clusters.

Column Name	Data Type	Description
CURRENT_EPOCH	INTEGER	
EPOCH	INTEGER	

LAST_REPLICATED_TIME	TIMESTAMPTZ	
OBJECTS	VARCHAR	
REPLICATED_EPOCH	INTEGER	
REPLICATION_POINT	VARCHAR	
SNAPSHOT_NAME	VARCHAR	

Privileges

None

REPARENTED_ON_DROP

Lists re-parenting events of objects that were dropped from their original owner but still remain in Vertica. For example, a user may leave the organization and need to be removed from the database. When the database administrator drops the user from the database, that user's objects are re-parented to another user.

In some cases, a Vertica user's objects are reassigned based on the GlobalHeirUsername parameter. In this case, a user's objects are re-parented to the user indicated by this parameter.

Column Name	Data Type	Description
REPARENT_TIMESTAMP	TIMESTAMP	The time the re-parenting event occurred.
NODE_NAME	VARCHAR	The name of the node or nodes on which the re-parenting occurred.
SESSION_ID	VARCHAR	The identification number of the re-parenting event.
USER_ID	INTEGER	The unique, system-generated user identification number.
USER_NAME	VARCHAR	The name of the user that caused the re-parenting event. For example, a dbadmin user may have dropped a user thus re-parenting that user's objects.
TRANSACTION_ID	INTEGER	The system-generated transaction identification number. Is NULL if a transaction id does not exist.
OLD_OWNER_NAME	VARCHAR	The the name of the dropped user who used to own the re-parented object.
OLD_OWNER_OID	INTEGER	The unique identification number of the user who used to own the re-parented object.
NEW_OWNER_NAME	VARCHAR	The name of the user who now owns the re-parented objects.
NEW_OWNER_OID	INTEGER	The unique identification number of the user who now owns the re-parented objects.
OBJ_NAME	VARCHAR	The name of the object being re-parented.
OBJ_OID	INTEGER	The unique identification number of the object being re-parented.
SCHEMA_NAME	VARCHAR	The name of the schema in which the object resides.
SCHEMA_OID	INTEGER	The unique identification number of the schema in which the re-parented object resides.

RESHARDING_EVENTS

Monitors historic and ongoing resharding operations.

Column Name	Data Type	Description

EVENT_TIME_STAMP	TIMESTAMP	Date and time of the resharding event.
NODE_NAME	VARCHAR	Node name for which resharding information is listed.
SESSION_ID	VARCHAR	Unique numeric ID assigned by the Vertica catalog that identifies the session for which resharding information is captured. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
USER_ID	INTEGER	Numeric ID of the user who ran the resharding operation.
USER_NAME	VARCHAR	Name of the user who ran the resharding operation.
TRANSACTION_ID	INTEGER	Numeric ID of the specified rehsarding transaction within the session.
RUNNING_STATUS	VARCHAR	Current status of the resharding operation, one of the following strings: • START: The resharding operation has begun on all nodes. • RUNNING: The shard named OLD_SHARD_NAME is currently being resharded on the node. • RESHARDED: The resharding operation on the node is complete for the shard named OLD_SHARD_NAME. • ABORT: The resharding operation was aborted on all nodes. • COMPLETE: The resharding operation has completed for all nodes in the database. **Note** Only the RESHARDED and RUNNING statuses are logged for each node. All other statuses are logged only on the initiator node.
OLD_SHARD_NAME	VARCHAR	Name of the shard to which the node was subscribed previous to the resharding operation. You can query the SHARDS system table for information about the new shard configuration.
OLD_SHARD_OID	INTEGER	Numeric ID of the shard to which the node was subscribed previous to the resharding operation.
OLD_SHARD_LOWER_BOUND	INTEGER	Lower bound of the shard to which the node was subscribed prior to the resharding operation. This value is set only if the resharding operation is complete for the shard specified by OLD_SHARD_OID.
OLD_SHARD_UPPER_BOUND	INTEGER	Upper bound of the shard to which the node was subscribed prior to the resharding operation. This value is set only if the resharding operation is complete for the shard specified by OLD_SHARD_OID.
CATALOG_SIZE	INTEGER	Catalog size (in bytes) on the node for the shard specified by OLD_SHARD_NAME. This value is provided only when the RUNNING_STATUS of the node is RUNNING or RESHARDED.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

See also

- Change the number of shards in the database
- RESHARD_DATABASE
- SHARDS

RESOURCE_ACQUISITIONS

Retains information about resources (memory, open file handles, threads) acquired by each running request. Each request is uniquely identified by its transaction and statement IDs within a given session.

Important

If a request cascades to one or more resource pools beyond the original pool, this table contains multiple records for the same request—one record for each resource pool. The following values are specific to each resource pool:

- Timestamp values: QUEUE_ENTRY_TIMESTAMP , ACQUISITION_TIMESTAMP , and RELEASE_TIMESTAMP
- DURATION_MS
- IS_EXECUTING

You can trace the history of cascade events by querying system table RESOURCE_POOL_MOVE .

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node name for which information is listed.
TRANSACTION_ID	INTEGER	Transaction identifier for this request.
STATEMENT_ID	INTEGER	Unique numeric ID for each statement within a transaction. NULL indicates that no statement is currently being processed.
REQUEST_TYPE	VARCHAR	Type of request issued to a resource pool. End users always see this column set to Reserve, to indicate that the request is query-specific.
POOL_ID / POOL_NAME	INTEGER / VARCHAR	Each resource pool that participated in handling this request: • POOL_ID : A unique numeric ID assigned by the Vertica catalog that uniquely identifies the resource pool. • POOL_NAME : Name of the resource pool.
THREAD_COUNT	INTEGER	Number of threads in use by this request.
OPEN_FILE_HANDLE_COUNT	INTEGER	Number of open file handles in use by this request.
MEMORY_INUSE_KB	INTEGER	Total amount of memory in kilobytes acquired by this query. Column RESERVED_EXTRA_MEMORY _B in system table QUERY_PROFILES shows how much unused memory (in bytes) remains that is reserved for a given query but is unassigned to a specific operator. If operators for a query acquire all memory specified by MEMORY_INUSE_KB , the plan must request more memory from the Vertica Resource Manager.
QUEUE_ENTRY_TIMESTAMP	TIMESTAMPTZ	Timestamp when the request was queued in this resource pool.
ACQUISITION_TIMESTAMP	TIMESTAMPTZ	Timestamp when the request was admitted to run.
RELEASE_TIMESTAMP	TIMESTAMPTZ	Time when Vertica released this resource acquisition.
DURATION_MS	INTEGER	Duration in milliseconds of request execution. If the request cascaded across multiple resource pools, DURATION_MS applies only to this resource pool.
IS_EXECUTING	BOOLEAN	Set to true if the resource pool is still executing this request. A value of false can indicate one of the following: • The request was completed or denied. • The request cascaded to another resource pool.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

Queue wait time

You can calculate how long a resource pool queues a given request before it begins execution by subtracting QUEUE_ENTRY_TIMESTAMP from ACQUISITION_TIMESTAMP . For example:

```
=> SELECT pool_name, queue_entry_timestamp, acquisition_timestamp,
   (acquisition_timestamp-queue_entry_timestamp) AS 'queue wait'
   FROM V_MONITOR.RESOURCE_ACQUISITIONS WHERE node_name ILIKE '%node0001';
```

See also

- RESOURCE_POOL_STATUS
- RESOURCE_POOLS
- RESOURCE_QUEUES
- RESOURCE_REJECTIONS

RESOURCE_POOL_MOVE

Displays the cascade event information on each node.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node name for which information is listed.
MOVE_TIMESTAMP	TIMESTAMPTZ	Time when the query attempted to move to the target pool.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
USER_ID	INTEGER	Identifies the query event user.
USER_NAME	VARCHAR	Name of the user for which Vertica lists query information at the time it records the session.
TRANSACTION_ID	INTEGER	Transaction identifier for the request.
STATEMENT_ID	INTEGER	Unique numeric ID for the statement.
SOURCE_POOL_NAME	VARCHAR	Name of the resource pool where the query was executing when Vertica attempted the move.
TARGET_POOL_NAME	VARCHAR	Name of resource pool where the query attempted to move.
MOVE_CAUSE	VARCHAR	Denotes why the query attempted to move. Valid values: • MOVE RESOURCE POOL COMMAND • RUNTIMECAP EXCEEDED
SOURCE_CAP	INTEGER	Effective RUNTIMECAP value for the source pool. The value represents the lowest of these three values: • session RUNTIMECAP • user RUNTIMECAP • source pool RUNTIMECAP

TARGET_CAP	INTEGER	Effective RUNTIMECAP value for the target pool. The value represents the lowest of these three values: • session RUNTIMECAP • user RUNTIMECAP • target pool RUNTIMECAP
SUCCESS	BOOLEAN	True, if the query successfully moved to the target pool.
RESULT_REASON	VARCHAR	States reason for success or failure of the move.

See also

- QUERY_PROFILES
- RESOURCE_POOL_STATUS
- RESOURCE_POOLS
- RESOURCE_QUEUES
- RESOURCE_REJECTIONS

RESOURCE_POOL_STATUS

Provides current state of built-in and user-defined resource pools on each node. Information includes:

- Current memory usage
- Resources requested and acquired by various requests
- Number of queries executing

For general information about resource pools, see Resource pool architecture .

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node for which information is provided.
POOL_OID	INTEGER	Unique numeric ID that identifies the pool and is assigned by the Vertica catalog.
POOL_NAME	VARCHAR	Name of the resource pool.
IS_INTERNAL	BOOLEAN	Denotes whether a pool is built-in .
MEMORY_SIZE_KB	INTEGER	Value of MEMORYSIZE setting of the pool in kilobytes.
MEMORY_SIZE_ACTUAL_KB	INTEGER	Current amount of memory, in kilobytes, allocated to the pool by the resource manager. The actual size can be less than specified in the DDL, if both the following conditions exist: • The pool has been recently altered in a running system. • The request to shuffle memory is pending.
MEMORY_INUSE_KB	INTEGER	Amount of memory, in kilobytes, acquired by requests running against this pool.
GENERAL_MEMORY_BORROWED_KB	INTEGER	Amount of memory, in kilobytes, borrowed from the GENERAL pool by requests running against this pool. The sum of MEMORY_INUSE_KB and GENERAL_MEMORY_BORROWED_KB should be less than MAX_MEMORY_SIZE_KB .
QUEUEING_THRESHOLD_KB	INTEGER	Calculated as MAX_MEMORY_SIZE_KB * 0.95 . When the amount of memory used by all requests against this resource pool exceeds the QUEUEING_THRESHOLD_KB , new requests against the pool are queued until memory becomes available.

MAX_MEMORY_SIZE_KB	INTEGER	Value, in kilobytes, of the MAXMEMORYSIZE parameter as defined for the pool. After this threshold is reached, new requests against this pool are rejected or queued until memory becomes available. **Note** MAX_MEMORY_SIZE_KB might not reflect the set MAXMEMORYSIZE parameter value if the specified value cannot be reached. For example, if MAXMEMORYSIZE = 10G but less than 2G is available, MAX_MEMORY_SIZE_KB will not reflect the original value in KB. Instead, it will display only 2G in KB, as that is the highest value available to it.
MAX_QUERY_MEMORY_SIZE_KB	INTEGER	Value, in kilobytes, of the MAXQUERYMEMORYSIZE parameter as defined for the pool. The resource pool limits this amount of memory to all queries that execute in it.
RUNNING_QUERY_COUNT	INTEGER	Number of queries currently executing in this pool.
PLANNED_CONCURRENCY	INTEGER	Value of PLANNEDCONCURRENCY parameter as defined for the pool.
MAX_CONCURRENCY	INTEGER	Value of MAXCONCURRENCY parameter as defined for the pool.
IS_STANDALONE	BOOLEAN	If the pool is configured to have MEMORYSIZE equal to MAXMEMORYSIZE , the pool is considered standalone because it does not borrow any memory from the General pool.
QUEUE_TIMEOUT	INTERVAL	The interval that the request waits for resources to become available before being rejected. If you set this value to NONE, Vertica displays it as NULL.
QUEUE_TIMEOUT_IN_SECONDS	INTEGER	Value of QUEUETIMEOUT parameter as defined for the pool. If QUEUETIMEOUT is set to NONE, Vertica displays this value as NULL.
EXECUTION_PARALLELISM	INTEGER	Limits the number of threads used to process any single query issued in this resource pool.
PRIORITY	INTEGER	Value of PRIORITY parameter as defined for the pool. When set to HOLD , Vertica sets a pool's priority to -999 so the query remains queued until QUEUETIMEOUT is reached.
RUNTIMECAP_IN_SECONDS	INTEGER	Defined for this pool by parameter RUNTIMECAP , specifies in seconds the maximum time a query in the pool can execute. If a query exceeds this setting, it tries to cascade to a secondary pool.
RUNTIME_PRIORITY	VARCHAR	Defined for this pool by parameter RUNTIMEPRIORITY , determines how the resource manager should prioritize dedication of run-time resources (CPU, I/O bandwidth) to queries already running in this resource pool.
RUNTIME_PRIORITY_THRESHOLD	INTEGER	Defined for this pool by parameter RUNTIMEPRIORITYTHRESHOLD , specifies in seconds a time limit in which a query must finish before the resource manager assigns to it the resource pool's RUNTIME_PRIORITY setting.
SINGLE_INITIATOR	BOOLEAN	Set for backward compatibility.

QUERY_BUDGET_KB	INTEGER	The current amount of memory that queries are tuned to use. The calculation that Vertica uses to determine this value is described in Query budgeting . **Note** The calculated value can change when one or more running queries needs more than the budgeted amount to run. For a detailed example of query budget calculations, see Do You Need to Put Your Query on a Budget? in the Vertica User Community .
CPU_AFFINITY_SET	VARCHAR	The set of CPUs on which queries associated with this pool are executed. Can be: • A percentage of CPUs on the system • A zero-based list of CPUs (a four-CPU system c of CPUs 0, 1, 2, and 3).
CPU_AFFINITY_MASK	VARCHAR	The bit mask of CPUs available for use in this pool, read from right to left. See Examples below.
CPU_AFFINITY_MODE	VARCHAR	The mode for the CPU affinity, one of the following: • ANY • EXCLUSIVE • SHARED

Examples

The following query returns bit masks that show CPU assignments for three user-defined resource pools. Resource pool bigqueries runs queries on CPU 0, ceo_pool on CPU 1, and testrp on CPUs 0 and 1:

```
=> SELECT pool_name, node_name, cpu_affinity_set, cpu_affinity_mode,
    TO_BITSTRING(CPU_AFFINITY_MASK::VARBINARY) "CPU Affinity Mask"
    FROM resource_pool_status WHERE IS_INTERNAL = 'false' order by pool_name, node_name;
 pool_name |    node_name     | cpu_affinity_set | cpu_affinity_mode | CPU Affinity Mask
-----------+------------------+------------------+-------------------+-------------------
 bigqueries | v_vmart_node0001 | 0                | SHARED            | 00110001
 bigqueries | v_vmart_node0002 | 0                | SHARED            | 00110001
 bigqueries | v_vmart_node0003 | 0                | SHARED            | 00110001
 ceo_pool   | v_vmart_node0001 | 1                | SHARED            | 00110010
 ceo_pool   | v_vmart_node0002 | 1                | SHARED            | 00110010
 ceo_pool   | v_vmart_node0003 | 1                | SHARED            | 00110010
 testrp     | v_vmart_node0001 | 0-1              | SHARED            | 00110011
 testrp     | v_vmart_node0002 | 0-1              | SHARED            | 00110011
 testrp     | v_vmart_node0003 | 0-1              | SHARED            | 00110011
(9 rows)
```

See also

- CREATE RESOURCE POOL
- RESOURCE_ACQUISITIONS
- RESOURCE_POOLS
- RESOURCE_QUEUES
- RESOURCE_REJECTIONS
- Managing workloads
- Monitoring resource pools

RESOURCE_QUEUES

Provides information about requests pending for various resource pools.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The name of the node for which information is listed.
TRANSACTION_ID	INTEGER	Transaction identifier for this request
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID uniquely identifies a statement within a session.
POOL_NAME	VARCHAR	The name of the resource pool
MEMORY_REQUESTED_KB	INTEGER	Amount of memory in kilobytes requested by this request
PRIORITY	INTEGER	Value of PRIORITY parameter specified when defining the pool.
POSITION_IN_QUEUE	INTEGER	Position of this request within the pool's queue
QUEUE_ENTRY_TIMESTAMP	TIMESTAMP	Timestamp when the request was queued

See also

- RESOURCE_ACQUISITIONS
- RESOURCE_POOLS
- RESOURCE_REJECTIONS

RESOURCE_REJECTION_DETAILS

Records an entry for each resource request that Vertica denies. This is useful for determining if there are resource space issues, as well as which users/pools encounter problems.

Column Name	Data Type	Description
REJECTED_TIMESTAMP	TIMESTAMPTZ	Time when Vertica rejected the resource.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_NAME	VARCHAR	Name of the user at the time Vertica recorded the session.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL .
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID, and REQUEST_ID uniquely identifies a statement within a session.
POOL_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theresource pool.
POOL_NAME	VARCHAR	Name of the resource pool
REASON	VARCHAR	Reason for rejecting this request; for example: • Usage of single request exceeds high limit • Timed out waiting for resource reservation • Canceled waiting for resource reservation

RESOURCE_TYPE	VARCHAR	Memory, threads, file handles or execution slots. The following list shows the resources that are limited by the resource manager . A query might need some amount of each resource, and if the amount needed is not available, the query is queued and could eventually time out of the queue and be rejected. • Number of running plans • Number of running plans on initiator node (local) • Number of requested threads • Number of requested file handles • Number of requested KB of memory • Number of requested KB of address space **Note:** Execution slots are determined by MAXCONCURRENCY parameter.
REJECTED_VALUE	INTEGER	Amount of the specific resource requested by the last rejection

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

See also

RESOURCE_REJECTIONS

RESOURCE_REJECTIONS

Monitors requests for resources that are rejected by the Resource manager . Information is valid only as long as the node is up and the counters reset to 0 upon node restart.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
POOL_ID	INTEGER	Catalog-assigned integer value that uniquely identifies theresource pool.
POOL_NAME	VARCHAR	Name of the resource pool.
REASON	VARCHAR	Reason for rejecting this request, for example: • Usage of single request exceeds high limit • Timed out waiting for resource reservation • Canceled waiting for resource reservation
RESOURCE_TYPE	VARCHAR	Memory, threads, file handles or execution slots. The following list shows the resources that are limited by the resource manager . A query might need some amount of each resource, and if the amount needed is not available, the query is queued and could eventually time out of the queue and be rejected. • Number of running plans • Number of running plans on initiator node (local) • Number of requested threads • Number of requested file handles • Number of requested KB of memory • Number of requested KB of address space **Note** Execution slots are determined by MAXCONCURRENCY parameter.
REJECTION_COUNT	INTEGER	Number of requests rejected due to specified reason and RESOURCE_TYPE .

FIRST_REJECTED_TIMESTAMP	TIMESTAMPTZ	Time of the first rejection for this pool.
LAST_REJECTED_TIMESTAMP	TIMESTAMPTZ	Time of the last rejection for this pool.
LAST_REJECTED_VALUE	INTEGER	Amount of the specific resource requested by the last rejection.

Examples

```
=> SELECT node_name, pool_name, reason, resource_type, rejection_count AS count, last_rejected_value AS value FROM resource_rejections;
    node_name     | pool_name |           reason            | resource_type | count |  value
------------------+-----------+-----------------------------+---------------+-------+---------
 v_vmart_node0001 | sysquery  | Request exceeded high limit | Memory(KB)    |     1 | 8248449
(1 row)
```

See also

- CLEAR_RESOURCE_REJECTIONS
- DISK_RESOURCE_REJECTIONS

RESOURCE_USAGE

Monitors system resource management on each node.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
REQUEST_COUNT	INTEGER	The cumulative number of requests for threads, file handles, and memory (in kilobytes).
LOCAL_REQUEST_COUNT	INTEGER	The cumulative number of local requests.
REQUEST_QUEUE_DEPTH	INTEGER	The current request queue depth.
ACTIVE_THREAD_COUNT	INTEGER	The current number of active threads.
OPEN_FILE_HANDLE_COUNT	INTEGER	The current number of open file handles.
MEMORY_REQUESTED_KB	INTEGER	The memory requested in kilobytes.
ADDRESS_SPACE_REQUESTED_KB	INTEGER	The address space requested in kilobytes.
ROS_USED_BYTES	INTEGER	The size of the ROS in bytes.
ROS_ROW_COUNT	INTEGER	The number of rows in the ROS.
RESOURCE_REQUEST_REJECT_COUNT	INTEGER	The number of rejected plan requests.
RESOURCE_REQUEST_TIMEOUT_COUNT	INTEGER	The number of resource request timeouts.
RESOURCE_REQUEST_CANCEL_COUNT	INTEGER	The number of resource request cancelations.
DISK_SPACE_REQUEST_REJECT_COUNT	INTEGER	The number of rejected disk write requests.
FAILED_VOLUME_REJECT_COUNT	INTEGER	The number of rejections due to a failed volume.
TOKENS_USED	INTEGER	For internal use only.
TOKENS_AVAILABLE	INTEGER	For internal use only.

SESSION_MARS_STORE

Shows Multiple Active Result Sets (MARS) storage information.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
SESSION_ID	VARCHAR	Identifier of the Vertica session. This identifier is unique within the cluster for the current session but can be reused in a subsequent session.
USER_NAME	VARCHAR	The username used to create the connection.
RESULTSET_ID	INTEGER	Identifier assigned to the result set.
ROW_COUNT	INTEGER	Number of rows requested by the query.
REMAINING_ROW_COUNT	INTEGER	Number of rows that still need to be returned.
BYTES_USED	INTEGER	The number of bytes requested.

SESSION_PARAMETERS

Provides information about user-defined parameters (UDPARAMETERS) set for the current session.

Column Name	Data Type	Description
SESSION_ID	VARCHAR	The unique identifier for the session.
SCHEMA_NAME	VARCHAR	The name of the schema on which the session is running.
LIB_NAME	VARCHAR	The name of the user library running the UDx, if necessary.
LIB_OID	VARCHAR	The object ID of the library containing the function, if one is running.
PARAMETER_NAME	VARCHAR	The name of the session parameter.
CURRENT_VALUE	VARCHAR	The value of the session parameter.

See also

- Configuration parameter management
- CONFIGURATION_PARAMETERS

SESSION_PROFILES

Provides basic session parameters and lock time out data. To obtain information about sessions, see Profiling database performance .

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
USER_NAME	VARCHAR	The name used to log in to the database or NULL if the session is internal.
CLIENT_HOSTNAME	VARCHAR	The host name and port of the TCP socket from which the client connection was made; NULL if the session is internal.
LOGIN_TIMESTAMP	TIMESTAMP	The date and time the user logged into the database or when the internal session was created. This field is useful for identifying sessions that have been left open for a period of time and could be idle.

LOGOUT_TIMESTAMP	TIMESTAMP	The date and time the user logged out of the database or when the internal session was closed.
SESSION_ID	VARCHAR	A unique numeric ID assigned by the Vertica catalog, which identifies the session for which profiling information is captured. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
EXECUTED_STATEMENT_SUCCESS_COUNT	INTEGER	The number of successfully run statements.
EXECUTED_STATEMENT_FAILURE_COUNT	INTEGER	The number of unsuccessfully run statements.
LOCK_GRANT_COUNT	INTEGER	The number of locks granted during the session.
DEADLOCK_COUNT	INTEGER	The number of deadlocks encountered during the session.
LOCK_TIMEOUT_COUNT	INTEGER	The number of times a lock timed out during the session.
LOCK_CANCELLATION_COUNT	INTEGER	The number of times a lock was canceled during the session.
LOCK_REJECTION_COUNT	INTEGER	The number of times a lock was rejected during a session.
LOCK_ERROR_COUNT	INTEGER	The number of lock errors encountered during the session.
CLIENT_TYPE	VARCHAR	The type of client from which the connection was made. Possible client type values: • ADO.NET Driver • ODBC Driver • JDBC Driver • vsql
CLIENT_VERSION	VARCHAR	Returns the client version.
CLIENT_OS	VARCHAR	Returns the client operating system.
CLIENT_OS_USER_NAME	VARCHAR	The name of the user that logged into, or attempted to log into, the database. This is logged even when the login attempt is unsuccessful.

See also

LOCKS

SESSIONS

Monitors external sessions. Use this table to perform the following tasks:

- Identify users who are running lengthy queries.
- Identify users who hold locks because of an idle but uncommitted transaction.
- Determine the details of the database security used for a particular session, either Secure Socket Layer (SSL) or client authentication.
- Identify client-specific information, such as client version.

> **Note**
> During session initialization and termination, you might see sessions running only on nodes other than the node on which you ran the virtual table query. This is a temporary situation that corrects itself when session initialization and termination complete.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.

USER_NAME	VARCHAR	The name used to log in to the database or NULL if the session is internal.
CLIENT_OS_HOSTNAME	VARCHAR	The hostname of the client as reported by their operating system.
CLIENT_HOSTNAME	VARCHAR	The IP address and port of the TCP socket from which the client connection was made; NULL if the session is internal. Vertica accepts either IPv4 or IPv6 connections from a client machine. If the client machine contains mappings for both IPv4 and IPv6, the server randomly chooses one IP address family to make a connection. This can cause the CLIENT_HOSTNAME column to display either IPv4 or IPv6 values, based on which address family the server chooses.
CLIENT_PID	INTEGER	The process identifier of the client process that issued this connection. Remember that the client process could be on a different machine than the server.
LOGIN_TIMESTAMP	TIMESTAMP	The date and time the user logged into the database or when the internal session was created. This field can help you identify sessions that have been left open for a period of time and could be idle.
SESSION_ID	VARCHAR	The identifier required to close or interrupt a session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
IDLE_SESSION_TIMEOUT	VARCHAR	Specifies how long this session can remain idle before timing out, set by SET SESSION IDLESESSIONTIMEOUT .
GRACE_PERIOD	VARCHAR	Specifies how long a session socket remains blocked while awaiting client input or output for a given query, set by SET SESSION GRACEPERIOD . If the socket is blocked for a continuous period that exceeds the grace period setting, the server shuts down the socket and throws a fatal error. The session is then terminated.
CLIENT_LABEL	VARCHAR	A user-specified label for the client connection that can be set when using ODBC. See Label in ODBC DSN connection properties . An MC output value means there are is a client connection to an MC-managed database for that USER_NAME
TRANSACTION_START	DATE	The date/time the current transaction started or NULL if no transaction is running.
TRANSACTION_ID	INTEGER	A string containing the hexadecimal representation of the transaction ID, if any; otherwise, NULL.
TRANSACTION _DESCRIPTION	VARCHAR	Description of the current transaction.
STATEMENT_START	TIMESTAMP	The timestamp the current statement started execution, or NULL if no statement is running.
STATEMENT_ID	INTEGER	A unique numeric ID assigned by the Vertica catalog, which identifies the currently-executing statement. A value of NULL indicates that no statement is currently being processed.
LAST_STATEMENT_DURATION_US	INTEGER	The duration of the last completed statement in microseconds.
RUNTIME_PRIORITY	VARCHAR	Specifies how many run-time resources (CPU, I/O bandwidth) are allocated to queries that are running in the resource pool.
CURRENT_STATEMENT	VARCHAR	The currently executing statement, if any. NULL indicates that no statement is currently being processed.

LAST_STATEMENT	VARCHAR	NULL if the user has just logged in; otherwise the currently running statement or the most recently completed statement.
SSL_STATE	VARCHAR	Indicates if Vertica used Secure Socket Layer (SSL) for a particular session. Possible values are: • None—Vertica did not use SSL. • Server—Server authentication was used, so the client could authenticate the server. • Mutual—Both the server and the client authenticated one another through mutual authentication. See Security and authentication and TLS protocol .
AUTHENTICATION_ METHOD	VARCHAR	The type of client authentication used for a particular session, if known. Possible values are: • Unknown • Trust • Reject • Hash • Ident • LDAP • GSS • TLS See Security and authentication and Configuring client authentication .
CLIENT_TYPE	VARCHAR	The type of client from which the connection was made. Possible client type values: • ADO.NET Driver • ODBC Driver • JDBC Driver • vsql
CLIENT_VERSION	VARCHAR	Client version.
CLIENT_OS	VARCHAR	Client operating system.
CLIENT_OS_USER _NAME	VARCHAR	The name of the user that logged into, or attempted to log into, the database. This is logged even when the login attempt is unsuccessful.
CLIENT_AUTHENTICATION_NAME	VARCHAR	User-assigned name of the authentication method.
CLIENT_ AUTHENTICATION	INTEGER	Object identifier of the client authentication method.
REQUESTED_PROTOCOL	INTEGER	The requested Vertica client server protocol to be used when connecting.
EFFECTIVE_PROTOCOL	INTEGER	The requested Vertica client server protocol used when connecting.
EXTERNAL_MEMORY_KB	INTEGER	Amount of memory consumed by the Java Virtual Machines associated with the session.

Privileges

A superuser has unrestricted access to all session information. Users can view information only about their own, current sessions.

See also

- CLOSE_SESSION
- CLOSE_ALL_SESSIONS

SPREAD_STATE

Lists Spread daemon settings for all nodes in the cluster.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Which node the settings are for.
TOKEN_TIMEOUT	INTEGER	The timeout period in milliseconds before spread considers a node to be down due to lack of response to a message.

Examples

```
=> SELECT * FROM V_MONITOR.SPREAD_STATE;
    node_name     | token_timeout
------------------+---------------
 v_vmart_node0003 |          8000
 v_vmart_node0001 |          8000
 v_vmart_node0002 |          8000
(3 rows)
```

See also

- Adjusting Spread Daemon timeouts for virtual environments
- SET_SPREAD_OPTION

STORAGE_BUNDLE_INFO_STATISTICS

Indicates which projections have storage containers with invalid bundle metadata in the database catalog. If any ROS or DV container has invalid bundle metadata fields, Vertica increments the corresponding column (ros_without_bundle_info_count or dv_ros_without_bundle_info_count) by one.

To update the catalog with valid bundle metadata, call UPDATE_STORAGE_CATALOG , as an argument to Vertica meta-function DO_TM_TASK . For details, see Writing bundle metadata to the catalog .

Column Name	Data Type	Description
node_name	VARCHAR	Name of this projection's node
projection_oid	INTEGER	Projection's unique catalog identifier
projection_name	VARCHAR	Projection name
projection_schema	VARCHAR	Projection schema name
total_ros_count	INTEGER	Total number of ROS containers for this projection
ros_without_bundle_info_count	INTEGER	Number of ROS containers for this projection with invalid bundle metadata
total_dv_ros_count	INTEGER	Total number of DV (delete vector) containers for this projection
dv_ros_without_bundle_info_count	INTEGER	Number of DV containers for this projection with invalid bundle metadata

STORAGE_CONTAINERS

Monitors information about Vertica storage containers.

Column Name	Data Type	Description
NODE_NAME *	VARCHAR	Node name for which information is listed.
SCHEMA_NAME *	VARCHAR	Schema name for which information is listed.

PROJECTION_ID *	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the projection.
PROJECTION_NAME *	VARCHAR	Projection name for which information is listed on that node.
STORAGE_OID *	INTEGER	Numeric ID assigned by the Vertica catalog, which identifies the storage. The same OID can appear on more than one node.
SAL_STORAGE_ID	VARCHAR	Unique hexadecimal numeric ID assigned by the Vertica catalog, which identifies the storage.
TOTAL_ROW_COUNT *	VARCHAR	Total rows in the storage container listed for that projection. If the database has been re-sharded, this value will be inaccurate until the Tuple mover realigns the storage containers to the new shard layout.
DELETED_ROW_COUNT *	INTEGER	Total rows in the storage container deleted for that projection. If the database has been re-sharded, this value will be inaccurate until the Tuple mover realigns the storage containers to the new shard layout.
USED_BYTES *	INTEGER	Number of bytes in the storage container used to store the compressed projection data. This value should not be compared to the output of the AUDIT function, which returns the raw data size of database objects. If the database has been re-sharded, this value will be inaccurate until the Tuple mover realigns the storage containers to the new shard layout.
START_EPOCH *	INTEGER	Number of the start epoch in the storage container for which information is listed.
END_EPOCH *	INTEGER	Number of the end epoch in the storage container for which information is listed.
GROUPING	VARCHAR	The group by which columns are stored: • ALL : All columns are grouped • PROJECTION : Columns grouped according to projection definition • NONE : No columns grouped, despite grouping in the projection definition • OTHER : Some grouping but neither all nor according to projection (e.g., results from add column)
SEGMENT_LOWER_BOUND	INTEGER	Lower bound of the segment range spanned by the storage container or NULL if the corresponding projection is not elastic.
SEGMENT_UPPER_BOUND	INTEGER	Upper bound of the segment range spanned by the storage container or NULL if the corresponding projection is not elastic.
LOCATION_LABEL	VARCHAR (128)	The location label (if any) for the storage container is stored.
DELETE_VECTOR_COUNT	INTEGER	The number of delete vectors in the storage container. If the database has been re-sharded, this value will be inaccurate until the Tuple mover realigns the storage containers to the new shard layout.
SHARD_ID	INTEGER	Set only for an Eon Mode database, ID of the shard that this container belongs to.
SHARD_NAME	VARCHAR(128)	Set only for an Eon Mode database, name of the shard that this container belongs to.

ORIGINAL_SEGMENT_LOWER_BOUND	INTEGER	The lower bound of a storage container before database re-sharding. This value is set only if the database has been re-sharded and the storage containers have not been realigned with current shard definitions. For details, see RESHARD_DATABASE .
ORIGINAL_SEGMENT_UPPER_BOUND	INTEGER	The upper bound of a storage container before database re-sharding. This value is set only if the database has been re-sharded and the storage container has not been realigned with current shard definitions. For details, see RESHARD_DATABASE .

* Column values cached for faster query performance

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

Examples

The following query identifies any storage containers that have not yet been realigned to the new shard segmentation bounds after running RESHARD_DATABASE :

```
=> SELECT COUNT(*) FROM storage_containers WHERE original_segment_lower_bound IS NOT NULL AND original_segment_upper_bound IS NOT NULL;
```

STORAGE_POLICIES

Monitors the current storage policies in effect for one or more database objects.

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	Schema name for which information is listed.
OBJECT_NAME	VARCHAR	The name of the database object associated through the storage policy.
POLICY_DETAILS	VARCHAR	The object type of the storage policy.
LOCATION_LABEL	VARCHAR (128)	The label for this storage location.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

See also

- Viewing storage locations and policies
- PARTITIONS
- STORAGE_CONTAINERS
- STORAGE_USAGE

STORAGE_TIERS

Provides information about all storage locations with the same label across all cluster nodes. This table lists data totals for all same-name labeled locations.

The system table shows what labeled locations exist on the cluster, as well as other cluster-wide data about the locations.

Column Name	Data Type	Description
LOCATION_LABEL	VARCHAR	The label associated with a specific storage location. The storage_tiers system table includes data totals for unlabeled locations, which are considered labeled with empty strings ('').
NODE_COUNT	INTEGER	The total number of nodes that include a storage location named location_label .

LOCATION_COUNT	INTEGER	The total number of storage locations named location_label . This value can differ from node_count if you create labeled locations with the same name at different paths on different nodes. For example: **v_vmart_node0001:** Create one labeled location, FAST **V_vmart_node0002** : Create two labeled locations, FAST , at different directory paths In this case, node_count value = 2, while location_count value = 3.
ROS_CONTAINER_COUNT	INTEGER	The total number of ROS containers stored across all cluster nodes for location_label .
TOTAL_OCCUPIED_SIZE	INTEGER	The total number of bytes that all ROS containers for location_label occupy across all cluster nodes.

Privileges

None

See also

- DISK_STORAGE
- STORAGE_POLICIES
- STORAGE_USAGE
- Storage functions

STORAGE_USAGE

Provides information about file system storage usage. This is useful for determining disk space usage trends.

Column Name	Data Type	Description
POLL_TIMESTAMP	TIMESTAMPTZ	Time when Vertica recorded the row.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
PATH	VARCHAR	Path where the storage location is mounted.
DEVICE	VARCHAR	Device on which the storage location is mounted.
FILESYSTEM	VARCHAR	File system on which the storage location is mounted.
USED_BYTES	INTEGER	Counter history of number of used bytes.
FREE_BYTES	INTEGER	Counter history of number of free bytes.
USAGE_PERCENT	FLOAT	Percent of storage in use.

Privileges

Superuser

See also

- DISK_STORAGE
- STORAGE_CONTAINERS
- STORAGE_POLICIES
- STORAGE_TIERS
- Storage functions

STRATA

Contains internal details of how the Tuple Mover combines ROS containers in each projection, broken down by stratum and classifies the ROS containers by size and partition. The related STRATA_STRUCTURES table provides a summary of the strata values.

Mergeout describes how the Tuple Mover combines ROS containers.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
SCHEMA_NAME	VARCHAR	The schema name for which information is listed.
PROJECTION_ID	INTEGER	Catalog-assigned numeric value that uniquely identifies the projection.
PROJECTION_NAME	VARCHAR	The projection name for which information is listed on that node.
STRATUM_KEY	VARCHAR	References the partition or partition group for which information is listed.
STRATA_COUNT	INTEGER	The total number of strata for this projection partition.
MERGING_STRATA_COUNT	INTEGER	The number of strata the Tuple Mover can merge out.
STRATUM_CAPACITY	INTEGER	The maximum number of ROS containers for the stratum before they must be merged.
STRATUM_HEIGHT	FLOAT	The size ratio between the smallest and largest ROS container in this stratum.
STRATUM_NO	INTEGER	The stratum number. Strata are numbered starting at 0, for the stratum containing the smallest ROS containers.
STRATUM_LOWER_SIZE	VARCHAR	The smallest ROS container size allowed in this stratum.
STRATUM_UPPER_SIZE	VARCHAR	The largest ROS container size allowed in this stratum.
ROS_CONTAINER_COUNT	INTEGER	The current number of ROS containers in the projection partition.

STRATA_STRUCTURES

This table provides an overview of Tuple Mover internal details. It summarizes how the ROS containers are classified by size. A more detailed view can be found in the STRATA virtual table.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
SCHEMA_NAME	VARCHAR	The schema name for which information is listed.
PROJECTION_NAME	VARCHAR	The projection name for which information is listed on that node.
PROJECTION_ID	INTEGER	Catalog-assigned numeric value that uniquely identifies the projection.
STRATUM_KEY	VARCHAR	References the partition or partition group for which information is listed.
STRATA_COUNT	INTEGER	The total number of strata for this projection partition.
MERGING_STRATA_COUNT	INTEGER	In certain hardware configurations, a high strata could contain more ROS containers than the Tuple Mover can merge out; output from this column denotes the number of strata the Tuple Mover can merge out.

STRATUM_CAPACITY	INTEGER	The maximum number of ROS containers that the strata can contained before it must merge them.
STRATUM_HEIGHT	FLOAT	The size ratio between the smallest and largest ROS container in a stratum.
ACTIVE_STRATA_COUNT	INTEGER	The total number of strata that have ROS containers in them.

Examples

```
=> \pset expanded
Expanded display is on.
=> SELECT node_name, schema_name, projection_name, strata_count,
          stratum_capacity, stratum_height, active_strata_count
          FROM strata_structures WHERE stratum_capacity > 60;
-[ RECORD 1 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node0001
schema_name         | public
projection_name     | shipping_dimension_DBD_22_seg_vmart_design_vmart_design
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
-[ RECORD 2 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node0001
schema_name         | public
projection_name     | shipping_dimension_DBD_23_seg_vmart_design_vmart_design
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
-[ RECORD 3 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node0002
schema_name         | public
projection_name     | shipping_dimension_DBD_22_seg_vmart_design_vmart_design
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
-[ RECORD 4 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node0002
schema_name         | public
projection_name     | shipping_dimension_DBD_23_seg_vmart_design_vmart_design
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
-[ RECORD 5 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node0003
schema_name         | public
projection_name     | shipping_dimension_DBD_22_seg_vmart_design_vmart_design
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
-[ RECORD 6 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node0003
schema_name         | public
projection_name     | shipping_dimension_DBD_23_seg_vmart_design_vmart_design
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
```

```
active_strata_count | 1
-[ RECORD 7 ]-------+-----------------------------------------------------------
node_name           | v_vmartdb_node0004
schema_name         | public
projection_name     | shipping_dimension_DBD_22_seg_vmart_design_vmart_design
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
-[ RECORD 8 ]-------+-----------------------------------------------------------
node_name           | v_vmartdb_node0004
schema_name         | public
projection_name     | shipping_dimension_DBD_23_seg_vmart_design_vmart_design
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
```

SYSTEM

Monitors the overall state of the database.

Column Name	Data Type	Description
CURRENT_EPOCH	INTEGER	The current epoch number.
AHM_EPOCH	INTEGER	The AHM epoch number.
LAST_GOOD_EPOCH	INTEGER	The smallest (min) of all the checkpoint epochs on the cluster.
REFRESH_EPOCH	INTEGER	Deprecated, always set to -1.
DESIGNED_FAULT_TOLERANCE	INTEGER	The designed or intended K-safety level.
NODE_COUNT	INTEGER	The number of nodes in the cluster.
NODE_DOWN_COUNT	INTEGER	The number of nodes in the cluster that are currently down.
CURRENT_FAULT_TOLERANCE	INTEGER	The number of node failures the cluster can tolerate before it shuts down automatically. This is the current K-safety level.
CATALOG_REVISION_NUMBER	INTEGER	The catalog version number.
ROS_USED_BYTES	INTEGER	The ROS size in bytes (cluster-wide).
ROS_ROW_COUNT	INTEGER	The number of rows in ROS (cluster-wide).
TOTAL_USED_BYTES	INTEGER	The total storage in bytes across the database cluster.
TOTAL_ROW_COUNT	INTEGER	The total number of rows across the database cluster.

SYSTEM_RESOURCE_USAGE

Provides history about system resources, such as memory, CPU, network, disk, I/O.

Column Name	Data Type	Description

NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
END_TIME	TIMESTAMP	End time of the history interval.
AVERAGE_MEMORY_USAGE_PERCENT	FLOAT	Average memory usage in percent of total memory (0-100) during the history interval.
AVERAGE_CPU_USAGE_PERCENT	FLOAT	Average CPU usage in percent of total CPU time (0-100) during the history interval.
NET_RX_KBYTES_PER_SECOND	FLOAT	Average number of kilobytes received from network (incoming) per second during the history interval.
NET_TX_KBYTES_PER_SECOND	FLOAT	Average number of kilobytes transmitting to network (outgoing) per second during the history interval.
IO_READ_KBYTES_PER_SECOND	FLOAT	Disk I/O average number of kilobytes read from disk per second during the history interval.
IO_WRITTEN_KBYTES_PER_SECOND	FLOAT	Average number of kilobytes written to disk per second during the history interval.

Privileges

Superuser

SYSTEM_SERVICES

Provides information about background system services that Workload Analyzer monitors.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
SERVICE_TYPE	VARCHAR	Type of service; can be one of: • SYSTEM • TUPLE MOVER
SERVICE_GROUP	VARCHAR	Group name, if there are multiple services of the same type.
SERVICE_NAME	VARCHAR	Name of the service.
SERVICE_INTERVAL_SEC	INTEGER	How often the service is executed (in seconds) during the history interval.
IS_ENABLED	BOOLEAN	Denotes if the service is enabled.
LAST_RUN_START	TIMESTAMPTZ	Denotes when the service was started last time.
LAST_RUN_END	TIMESTAMPTZ	Denotes when the service was completed last time.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

SYSTEM_SESSIONS

Provides information about system internal session history by system task.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.

USER_NAME	VARCHAR	Name of the user at the time Vertica recorded the session.
SESSION_ID	INTEGER	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any. If a session is active but no transaction has begun, TRANSACTION_ID returns NULL.
STATEMENT_ID	VARCHAR	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID and STATEMENT_ID uniquely identifies a statement within a session.
SESSION_TYPE	VARCHAR	Session type, one of: • CLIENT • DBD • MERGEOUT • REBALANCE_CLUSTER • RECOVERY • REFRESH • TIMER_SERVICE • CONNECTION • SUBSESSION • REPARTITION_TABLE • LICENSE_AUDIT • STARTUP • SHUTDOWN • VSPREAD
RUNTIME_PRIORITY	VARCHAR	Specifies how many run-time resources (CPU, I/O bandwidth) are allocated to queries that are running in the resource pool.
DESCRIPTION	VARCHAR	Transaction description in this session.
SESSION_START_TIMESTAMP	TIMESTAMPTZ	Value of session at beginning of history interval.
SESSION_END_TIMESTAMP	TIMESTAMPTZ	Value of session at end of history interval.
IS_ACTIVE	BOOLEAN	Denotes if the session is still running.
SESSION_DURATION_MS	INTEGER	Duration of the session in milliseconds.
CLIENT_TYPE	VARCHAR	Columns not used in SYSTEM_SESSIONS system table. To view values for these columns, see the V_MONITOR schema system tables SESSIONS , USER_SESSIONS , CURRENT_SESSION , and SESSION_PROFILES .
CLIENT_VERSION	VARCHAR	
CLIENT_OS	VARCHAR	
CLIENT_OS_USER_NAME	VARCHAR	The name of the user that logged into, or attempted to log into, the database. This is logged even when the login attempt is unsuccessful.

Privileges

Superuser

TABLE_RECOVERIES

Provides detailed information about recovered and recovering tables during a recovery by table .

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is performing the recovery.
TABLE_NAME	VARCHAR	The name of the table being recovered.
TABLE_OID	INTEGER	The object ID of the table being recovered.
STATUS	VARCHAR	The status of the table. Tables can have the following status: • recovered: The table is fully recovered • recovering: The table is in the process of recovery • error_retry: Vertica has attempted to recover the table, but the recovery failed. Tables that have not yet begun the recovery process do not have a status.
PHASE	VARCHAR	The phase of the recovery .
THREAD_ID	VARCHAR	The ID of the thread that performed the recovery.
START_TIME	TIMESTAMPTZ	The date and time that the table began recovery.
END_TIME	TIMESTAMPTZ	The date and time that the table completed recovery.
RECOVER_PRIORITY	INTEGER	The recovery priority of the table being recovered.
RECOVER_ERROR	VARCHAR	Error that caused the recovery to fail.
IS_HISTORICAL	BOOLEAN	If f , the record contains recovery information for the current process.

Privileges

None

Examples

```
=> SELECT * FROM TABLE_RECOVERIES;
-[RECORD 1]----------------------------------
node_name        | node04
table_oid        | 45035996273708000
table_name       | public.t
status           | recovered
phase            | current replay delete
thread_id        | 7f7a817fd700
start_time       | 2017-12-13 08:47:28.825085-05
end_time         | 2017-12-13 08:47:29.216571-05
recover_priority | -9223372036854775807
recover_error    | Event apply failed
is_historical    | t
-[RECORD 2]----------------------------------------
node_name        | v_test_parquet_ha_node0011
table_oid        | 45035996273937680
table_name       | public.t2_impala230_uncompre_multi_file_libhdfs_1
status           | error-retry
phase            | historical
thread_id        | 7f89a574f700
start_time       | 2018-02-24 11:30:59.008831-05
end_time         | 2018-02-24 11:33:09.780798-05
recover_priority | -9223372036854775807
recover_error    | Could not stop all dirty transactions[txnId = 45035996273718426; ]
is_historical    | t
```

TABLE_RECOVERY_STATUS

Provides node recovery information during a Recovery By Table .

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is performing the recovery.
NODE_RECOVERY_START_TIME	TIMESTAMPTZ	The timestamp for when the node began recovering.
RECOVER_EPOCH	INTEGER	The epoch that the recovery operation is trying to recover to.
RECOVERING_TABLE_NAME	VARCHAR	The name of the table currently recovering.
TABLES_REMAIN	INTEGER	The total number of tables on the node.
IS_RUNNING	BOOLEAN	Indicates if the recovery process is still running.

Privileges

None

Examples

```
=> SELECT * FROM TABLE_RECOVERY_STATUS;
-[ RECORD 1 ]----------+----------------
node_name              | v_vmart_node0001
node_recovery_start_time |
recover_epoch          |
recovering_table_name  |
tables_remain          | 0
is_running             | f
-[ RECORD 2 ]----------+----------------
node_name              | v_vmart_node0002
node_recovery_start_time |
recover_epoch          |
recovering_table_name  |
tables_remain          | 0
is_running             | f
-[ RECORD 3 ]----------+----------------
node_name              | v_vmart_node0003
node_recovery_start_time | 2017-12-13 08:47:28.282377-05
recover_epoch          | 23
recovering_table_name  | user_table
tables_remain          | 5
is_running             | y
```

TABLE_STATISTICS

Displays statistics that have been collected for tables and their respective partitions.

Column Name	Data Type	Description
LOGICAL_STATS_OID	INTEGER	Uniquely identifies a collection of statistics for a given table.
TABLE_NAME	VARCHAR	Name of an existing database table.
MIN_PARTITION_KEY, MAX_PARTITION_KEY	VARCHAR	Statistics for a range of partition keys collected by ANALYZE_STATISTICS_PARTITION , empty if statistics were collected by ANALYZE_STATISTICS .
ROW_COUNT	INTEGER	The number of rows analyzed for each statistics collection.

STAT_COLLECTION_TIME	TIMESTAMPTZ	The timestamp of each statistics collection.

TLS_CONFIGURATIONS

Lists settings for TLS Configuration objects for the server, LDAP, etc.

Column Name	Data Type	Description
NAME	VARCHAR	Name of the TLS Configuration. Vertica includes the following TLS Configurations by default: • server • LDAPLink • LDAPAuth • data_channel
OWNER	VARCHAR	Owner of the TLS Configuration object.
CERTIFICATE	VARCHAR	The certificate associated with the TLS Configuration object.
CA_CERTIFICATES	VARCHAR	The CA certificate(s) used to verify client certificates. In cases where a TLS Configuration uses more than one CA, each CA will have its own row in the table.
CIPHER_SUITES	VARCHAR	The cipher suites to used to secure the connection.

MODE	VARCHAR	How Vertica establishes TLS connections with another host, one of the following, in order of ascending security:
		• DISABLE : Disables TLS. All other options for this parameter enable TLS.
		• ENABLE : Enables TLS. Vertica does not check client certificates.
		• TRY_VERIFY : Establishes a TLS connection if one of the following is true:
		  ◦ the other host presents a valid certificate
		  ◦ the other host doesn't present a certificate
		If the other host presents an invalid certificate, the connection will use plaintext.
		• VERIFY_CA : Connection succeeds if Vertica verifies that the other host's certificate is from a trusted CA. If the other host does not present a certificate, the connection uses plaintext.
		• VERIFY_FULL : Connection succeeds if Vertica verifies that the other host's certificate is from a trusted CA and the certificate's cn (Common Name) or subjectAltName attribute matches the hostname or IP address of the other host.
		Note that for client certificates, cn is used for the username, so subjectAltName must match the hostname or IP address of the other host.
		VERIFY_FULL is unsupported for client-server TLS (the connection type handled by ServerTLSConfig) and behaves as VERIFY_CA .

Examples

In this example, the LDAPAuth TLS Configuration uses two CA certificates:

```
=> SELECT * FROM tls_configurations WHERE name='LDAPAuth';
   name   |  owner  | certificate | ca_certificate | cipher_suites |  mode
----------+---------+-------------+----------------+---------------+---------
 LDAPAuth | dbadmin | server_cert | ca             |               | DISABLE
 LDAPAuth | dbadmin | server_cert | ica            |               | DISABLE
(2 rows)
```

To make more clear the relationship between a TLS Configuration and its CA certificates, you can format the query with LISTAGG :

```
=> SELECT name, owner, certificate, LISTAGG(ca_certificate) AS ca_certificates, cipher_suites, mode
FROM tls_configurations
WHERE name='LDAPAuth'
GROUP BY name, owner, certificate, cipher_suites, mode
ORDER BY 1;
   name   |  owner  | certificate | ca_certificates | cipher_suites |  mode
----------+---------+-------------+-----------------+---------------+---------
 LDAPAuth | dbadmin | server_cert | ca,ica          |               | DISABLE
(1 row)
```

TRANSACTIONS

Records the details of each transaction.

Column Name	Data Type	Description
START_TIMESTAMP	TIMESTAMPTZ	Beginning of history interval.
END_TIMESTAMP	TIMESTAMPTZ	End of history interval.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the user.
USER_NAME	VARCHAR	Name of the user for which transaction information is listed.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL .
DESCRIPTION	VARCHAR	Textual description of the transaction.
START_EPOCH	INTEGER	Number of the start epoch for the transaction.
END_EPOCH	INTEGER	Number of the end epoch for the transaction
NUMBER_OF_STATEMENTS	INTEGER	Number of query statements executed in this transaction.
ISOLATION	VARCHAR	Denotes the transaction mode as "READ COMMITTED" or "SERIALIZABLE".
IS_READ_ONLY	BOOLEAN	Denotes "READ ONLY" transaction mode.
IS_COMMITTED	BOOLEAN	Determines if the transaction was committed. False means ROLLBACK.
IS_LOCAL	BOOLEAN	Denotes transaction is local (non-distributed).
IS_INITIATOR	BOOLEAN	Denotes if the transaction occurred on this node (t).
IS_DDL	BOOLEAN	Distinguishes between a DDL transaction (t) and non-DDL transaction (f).

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

See also

Transactions

TRUNCATED_SCHEMATA

Lists the original names of restored schemas that were truncated due to name lengths exceeding 128 characters.

Column Name	Data Type	Description
RESTORE_TIME	TIMESTAMPTZ	The time that the table was restored.
SESSION_ID	VARCHAR	Identifier for the restoring session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
USER_ID	INTEGER	Identifier of the user for the restore event.
USER_NAME	VARCHAR	Name of the user for which Vertica lists restore information at the time it recorded the session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL.
ORIGINAL_SCHEMA_NAME	VARCHAR	The original name of the schema prior to the restore.
NEW_SCHEMA_NAME	VARCHAR	The name of the schema after it was truncated.

Privileges

None

TUNING_RECOMMENDATIONS

Returns tuning recommendation results from the last call to ANALYZE_WORKLOAD . This information is useful for building filters on the Workload Analyzer result set.

Column Name	Data Type	Description
OBSERVATION_COUNT	INTEGER	Integer for the total number of events observed for this tuning recommendation. For example, if you see a return value of 1, Workload Analyzer is making its first tuning recommendation for the event in 'scope'.
FIRST_OBSERVATION_TIME	TIMESTAMPTZ	Timestamp when the event first occurred. If this column returns a null value, the tuning recommendation is from the current status of the system instead of from any prior event.
LAST_OBSERVATION_TIME	TIMESTAMPTZ	Timestamp when the event last occurred. If this column returns a null value, the tuning recommendation is from the current status of the system instead of from any prior event.
TUNING_PARAMETER	VARCHAR	Objects on which to perform a tuning action. For example, a return value of: • public.t informs the DBA to run Database Designer on table t in the public schema • bsmith notifies a DBA to set a password for user bsmith
TUNING_DESCRIPTION	VARCHAR	Textual description of the tuning recommendation to perform on the tuning_parameter object. For example: • Run database designer on table schema.table • Create replicated projection for table schema.table • Consider incremental design on query • Re-segment projection projection-name on high-cardinality column(s) • Drop the projection projection-name • Alter a table's partition expression • Reorganize data in partitioned table • Decrease the MoveOutInterval configuration parameter setting

TUNING_COMMAND	VARCHAR	Command string if tuning action is a SQL command. For example: Update statistics on a particular schema's table.column: SELECT ANALYZE_STATISTICS('public.table.column'); Resolve mismatched configuration parameter LockTimeout : SELECT * FROM CONFIGURATION_PARAMETERS WHERE parameter_name = 'LockTimeout'; Set the password for user bsmith : ALTER USER (bsmith) IDENTIFIED BY ('new_password');
TUNING_COST	VARCHAR	Cost is based on the type of tuning recommendation and is one of: • LOW : minimal impact on resources from running the tuning command • MEDIUM : moderate impact on resources from running the tuning command • HIGH : maximum impact on resources from running the tuning command Depending on the size of your database or table, consider running high-cost operations after hours instead of during peak load times.

Privileges

Superuser

Examples

See ANALYZE_WORKLOAD .

See also

- Analyzing workloads
- Workload analyzer recommendations

TUPLE_MOVER_OPERATIONS

Monitors the status of Tuple Mover operations on each node.

Column Name	Data Type	Description
OPERATION_START_TIMESTAMP	TIMESTAMP	Start time of a Tuple Mover operation.
NODE_NAME	VARCHAR	Node name for which information is listed.
OPERATION_NAME	VARCHAR	One of the following: • Analyze Statistics • DVMergeout • Mergeout • Partitioning • Rebalance • Recovery Replay Delete

OPERATION_STATUS	VARCHAR	Returns the status of each operation, one of the following: • Empty string: not running • Start • Running • Complete • Update • Abort • Change plan type to Replay Delete
TABLE_SCHEMA	VARCHAR	Schema name for the specified projection.
TABLE_NAME	VARCHAR	Table name for the specified projection.
PROJECTION_NAME	VARCHAR	Name of the projection being processed.
PROJECTION_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the projection.
COLUMN_ID	INTEGER	Identifier for the column for the associated projection being processed.
EARLIEST_CONTAINER_START_EPOCH	INTEGER	Populated for mergeout and purge operations only. For an automatically-invoked mergeout, for example, the returned value represents the lowest epoch of containers involved in the mergeout.
LATEST_CONTAINER_END_EPOCH	INTEGER	Populated for mergeout and purge_partitions operations. For an automatically-invoked mergeout, for example, the returned value represents the highest epoch of containers involved in the mergeout.
ROS_COUNT	INTEGER	Number of ROS containers.
TOTAL_ROS_USED_BYTES	INTEGER	Size in bytes of all ROS containers in the mergeout operation. (Not applicable for other operations.)
PLAN_TYPE	VARCHAR	One of the following: • Analyze Statistics • DVMergeout • Mergeout • Partitioning • Rebalance • Recovery Replay Delete • Replay Delete
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any. If a session is active but no transaction has begun, TRANSACTION_ID returns NULL.
IS_EXECUTING	BOOLEAN	Distinguishes between actively-running (t) and completed (f) tuple mover operations.
RUNTIME_PRIORITY	VARCHAR	Determines how many run-time resources (CPU, I/O bandwidth) the Resource Manager should dedicate to running queries in the resource pool, one of the following: • HIGH • MEDIUM • LOW

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

Examples

```
=> SELECT node_name, operation_status, projection_name, plan_type
   FROM TUPLE_MOVER_OPERATIONS;
 node_name        | operation_status | projection_name  | plan_type
------------------+------------------+------------------+-----------
 v_vmart_node0001 | Running          | p1_b2            | Mergeout
 v_vmart_node0002 | Running          | p1               | Mergeout
 v_vmart_node0001 | Running          | p1_b2            | Replay Delete
 v_vmart_node0001 | Running          | p1_b2            | Mergeout
 v_vmart_node0002 | Running          | p1_b2            | Mergeout
 v_vmart_node0001 | Running          | p1_b2            | Replay Delete
 v_vmart_node0002 | Running          | p1               | Mergeout
 v_vmart_node0003 | Running          | p1_b2            | Replay Delete
 v_vmart_node0001 | Running          | p1               | Mergeout
 v_vmart_node0002 | Running          | p1_b1            | Mergeout
```

See also

- DO_TM_TASK
- PURGE

UDFS_EVENTS

Records information about events involving the S3 , HDFS , GCS , and Azure file systems.

Column Name	Data Type	Description
START_TIME	TIMESTAMPTZ	Event start time
END_TIME	TIMESTAMPTZ	Event end time
NODE_NAME	VARCHAR	Name of the node that reported the event
SESSION_ID	VARCHAR	Identifies the event session, unique within the cluster at any point in time but can be reused when the session closes.
USER_ID	INTEGER	Identifies the event user.
TRANSACTION_ID*	INTEGER	Identifies the event transaction within the SESSION_ID-specified session, if any; otherwise NULL.
STATEMENT_ID*	INTEGER	Uniquely identifies the current statement, if any; otherwise NULL.
REQUEST_ID*	INTEGER	Uniquely identifies the event request in the user session.
FILESYSTEM	VARCHAR	Name of the file system, such as S3
PATH	VARCHAR	Complete file path
EVENT	VARCHAR	The function call that was made. For example: `virtual size_t SAL::S3FileOperator::read(void*, size_t)`
STATUS	VARCHAR	Status of the event: OK, CANCEL, or FAIL
DESCRIPTION	VARCHAR	Other event details, for internal use only
ACTIVITY	VARCHAR	Points to the component that was active and logged the event, for internal use only.
PLAN_ID	VARCHAR	Uniquely identifies the node-specific Optimizer plan for this event.

OPERATOR_ID	INTEGER	Identifier assigned by the Execution Engine operator instance that performs the work

* In combination, TRANSACTION_ID, STATEMENT_ID, and REQUEST_ID uniquely identify an event within a given session.

Privileges

Superuser

UDFS_OPS_PER_HOUR

This table summarizes the S3 file system statistics for each hour.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node the file system is on.
FILESYSTEM	VARCHAR	Name of the file system, such as S3.
START_TIME	TIMESTAMP	Start time for statistics gathering.
END_TIME	TIMESTAMP	Stop time for statistics gathering.
AVG_OPERATIONS_PER_SECOND	INTEGER	Average number of operations per second during the specified hour.
AVG_ERRORS_PER_SECOND	INTEGER	Average number of errors per second during the specified hour.
RETRIES	INTEGER	Number of retry events during the specified hour.
METADATA_READS	INTEGER	Number of requests to write metadata during the specified hour. For example, S3 POST and DELETE requests are metadata writes.
METADATA_WRITES	INTEGER	Number of requests to write metadata during the specified hour. For example, S3 POST and DELETE requests are metadata writes.
DATA_READS	INTEGER	Number of read operations, such as S3 GET requests to download files, during the specified hour.
DATA_WRITES	INTEGER	Number of write operations, such as S3 PUT requests to upload files, during the specified hour.
UPSTREAM_BYTES	INTEGER	Number of bytes received during the specified hour.
DOWNSTREAM_BYTES	INTEGER	Number of bytes sent during the specified hour.

Examples

```
=> \x
Expanded display is on.
=> SELECT * FROM UDFS_OPS_PER_HOUR;
-[ RECORD 1 ]----------+--------------------
node_name              | e1
filesystem             | S3
start_time             | 2018-04-06 04:00:00
end_time               | 2018-04-06 04:00:00
avg_operations_per_second | 0
avg_errors_per_second  | 0
retries                | 0
metadata_reads         | 0
metadata_writes        | 0
data_reads             | 0
data_writes            | 0
upstream_bytes         | 0
downstream_bytes       | 0
...
```

See also

UDFS_OPS_PER_MINUTE

UDFS_OPS_PER_MINUTE

This table summarizes the S3 file system statistics for each minute.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node the file system is on.
FILESYSTEM	VARCHAR	Name of the file system, such as S3.
START_TIME	TIMESTAMP	Start time for statistics gathering.
END_TIME	TIMESTAMP	Stop time for statistics gathering.
AVG_OPERATIONS_PER_SECOND	INTEGER	Average number of operations per second during the specified minute.
AVG_ERRORS_PER_SECOND	INTEGER	Average number of errors per second during the specified minute.
RETRIES	INTEGER	Number of retry events during the specified minute.
METADATA_READS	INTEGER	Number of requests to write metadata during the specified minute. For example, S3 POST and DELETE requests are metadata writes.
METADATA_WRITES	INTEGER	Number of requests to write metadata during the specified minute. For example, S3 POST and DELETE requests are metadata writes.
DATA_READS	INTEGER	Number of read operations, such as S3 GET requests to download files, during the specified minute.
DATA_WRITES	INTEGER	Number of write operations, such as S3 PUT requests to upload files, during the specified minute.
UPSTREAM_BYTES	INTEGER	Number of bytes received during the specified minute.
DOWNSTREAM_BYTES	INTEGER	Number of bytes sent during the specified minute.

Examples

```
=> \x
Expanded display is on.
=> SELECT * FROM UDFS_OPS_PER_MINUTE;
-[ RECORD 1 ]----------+--------------------
node_name              | e1
filesystem             | S3
start_time             | 2018-04-06 04:17:00
end_time               | 2018-04-06 04:18:00
avg_operations_per_second | 0
avg_errors_per_second  | 0
retries                | 0
metadata_reads         | 0
metadata_writes        | 0
data_reads             | 0
data_writes            | 0
upstream_bytes         | 0
downstream_bytes       | 0
...
```

See also

UDFS_OPS_PER_HOUR
UDFS_STATISTICS

Records aggregate information about file-system and object-store operations. For access through LibHDFS++, the table records information about metadata but not data.

An operation can be made up of many individual read, write, or retry requests. SUCCESSFUL_OPERATIONS and FAILED_OPERATIONS count operations; the other counters count individual requests. When an operation finishes, one of the OPERATIONS counters is incremented once, but several other counters could be incremented several times each.

Column Name	Data Type	Description
FILESYSTEM	VARCHAR	Name of the file system, such as S3 or Libhdfs++.
SUCCESSFUL_OPERATIONS	INTEGER	Number of successful file-system operations.
FAILED_OPERATIONS	INTEGER	Number of failed file-system operations.
RETRIES	INTEGER	Number of retry events.
METADATA_READS	INTEGER	Number of requests to read metadata. For example, S3 list bucket and HEAD requests are metadata reads.
METADATA_WRITES	INTEGER	Number of requests to write metadata. For example, S3 POST and DELETE requests are metadata writes.
DATA_READS	INTEGER	Number of read operations, such as S3 GET requests to download files.
DATA_WRITES	INTEGER	Number of write operations, such as S3 PUT requests to upload files.
DOWNSTREAM_BYTES	INTEGER	Number of bytes received.
UPSTREAM_BYTES	INTEGER	Number of bytes sent.
OPEN_FILES	INTEGER	Number of files that are currently open.
MAPPED_FILES	INTEGER	Number of currently-mapped files on S3 file systems. This value shows the number of streaming connections for reading data from S3. This value will be 0 for non-S3 file systems.
READING	INTEGER	The number of currently-running read operations.

WRITING	INTEGER	The number of currently-running writer operations.

Examples

The following query gets the total number of metadata RPCs for Libhdfs++ operations:

```
=> SELECT SUM(metadata_reads) FROM UDFS_STATISTICS WHERE filesystem = 'Libhdfs++';
```

UDX_EVENTS

Records information about events raised from the execution of user-defined extensions.

A UDx populates the __RAW__ column using ServerInterface::logEvent() (C++ only). VMap support is provided by Flex Tables, which must not be disabled.

Column Name	Data Type	Description
REPORT_TIME	TIMESTAMPTZ	Time the event occurred.
NODE_NAME	VARCHAR	Name of the node that reported the event
SESSION_ID	VARCHAR	Identifies the event session, unique within the cluster at any point in time but can be reused when the session closes.
USER_ID	INTEGER	Identifies the user running the UDx.
USER_NAME	VARCHAR	Identifies the user running the UDx.
TRANSACTION_ID*	INTEGER	Identifies the event transaction within the SESSION_ID-specified session, if any; otherwise NULL.
STATEMENT_ID*	INTEGER	Uniquely identifies the current statement, if any; otherwise NULL.
REQUEST_ID*	INTEGER	Uniquely identifies the event request in the user session.
UDX_NAME	VARCHAR	Name of the UDx, as specified in the corresponding CREATE FUNCTION statement.
RAW	VARBINARY	VMap containing UDx-specific values.

UDX_FENCED_PROCESSES

Provides information about processes Vertica uses to run user-defined extensions in fenced mode.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
PROCESS_TYPE	VARCHAR	Indicates what kind of side process this row is for and can be one of the following values: • UDxZygoteProcess — Master process that creates worker side processes, as needed, for queries. There will be, at most, 1 UP UDxZygoteProcess for each Vertica instance. • UDxSideProcess — Indicates that the process is a worker side process. There could be many UDxSideProcesses, depending on how many sessions there are, how many queries, and so on.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
LANGUAGE	VARCHAR	The language of the UDx. For example ' R ' or ' C++ ';
MAX_MEMORY_JAVA_KB	INTEGER	The maximum amount of memory in KB that can be used for the Java heap file on the node.

PID	INTEGER	Linux process identifier of the side process (UDxSideProcess).
PORT	VARCHAR	For Vertica internal use. The TCP port that the side process is listening on.
STATUS	VARCHAR	Set to UP or DOWN, depending on whether the process is alive or not. After a process fails, Vertica restarts it only on demand. So after a process failure, there might be periods of time when no side processes run.

Privileges

None

USER_LIBRARIES

Lists the user libraries that are currently loaded. These libraries contain user-defined extensions (UDxs) that provide additional analytic functions.

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR(8192)	The name of the schema containing the library.
LIB_NAME	VARCHAR(8192)	The name of the library.
LIB_OID	INTEGER	The object ID of the library.
AUTHOR	VARCHAR(8192)	The creator of the library file.
OWNER_ID	INTEGER	The object ID of the library's owner.
LIB_FILE_NAME	VARCHAR(8192)	The name of the shared library file.
MD5_SUM	VARCHAR(8192)	The MD5 checksum of the library file, used to verify that the file was correctly copied to each node. **Note** This use of MD5 is not for cryptographic or authentication purposes. For information on authenticating with MD5 see Hash authentication .
SDK_VERSION	VARCHAR(8192)	The version of the Vertica SDK used to compile the library.
REVISION	VARCHAR(8192)	The revision of the Vertica SDK used to compile the library.
LIB_BUILD_TAG	VARCHAR(8192)	Internal information set by library developer to track the when the library was compiled.
LIB_VERSION	VARCHAR(8192)	The version of the library.
LIB_SDK_VERSION	VARCHAR(8192)	The version of the Vertica SDK intended for use with the library. The developer sets this value manually. This value may differ from the values in the SDK_VERSION and REVISION, which are set automatically during compilation.
SOURCE_URL	VARCHAR(8192)	A URL that contains information about the library.
DESCRIPTION	VARCHAR(8192)	A description of the library.
LICENSES_REQUIRED	VARCHAR(8192)	The licenses required to use the library.
SIGNATURE	VARCHAR(8192)	The signature used to sign the library for validation.

DEPENDENCIES	VARCHAR (8192)	External libraries on which this library depends. These libraries are maintained by Vertica, just like the user libraries themselves.

USER_LIBRARY_MANIFEST

Lists user-defined functions contained in all loaded user libraries.

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	The name of the schema containing the function.
LIB_NAME	VARCHAR	The name of the library containing the UDF.
LIB_OID	INTEGER	The object ID of the library containing the function.
OBJ_NAME	VARCHAR	The name of the constructor class in the library for a function.
OBJ_TYPE	VARCHAR	The type of user defined function (scalar function, transform function)
ARG_TYPES	VARCHAR	A comma-delimited list of data types of the function's parameters.
RETURN_TYPE	VARCHAR	A comma-delimited list of data types of the function's return values.

Privileges

None

USER_SESSIONS

Returns user session history on the system.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_NAME	VARCHAR	Name of the user at the time Vertica recorded the session.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
TRANSACTION_ID	VARCHAR	Identifier for the transaction within the session, if any. If a session is active but no transaction has begun, TRANSACTION_ID returns NULL.
STATEMENT_ID	VARCHAR	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID and STATEMENT_ID uniquely identifies a statement within a session.
RUNTIME_PRIORITY	VARCHAR	Determines the amount of run-time resources (CPU, I/O bandwidth) the Resource Manager should dedicate to queries already running in the resource pool. Valid values are: • HIGH • MEDIUM • LOW Queries with a HIGH run-time priority are given more CPU and I/O resources than those with a MEDIUM or LOW run-time priority.

SESSION_START_TIMESTAMP	TIMESTAMPTZ	Value of session at beginning of history interval.
SESSION_END_TIMESTAMP	TIMESTAMPTZ	Value of session at end of history interval.
IS_ACTIVE	BOOLEAN	Denotes if the operation is executing.
CLIENT_OS_HOSTNAME	VARCHAR	The hostname of the client as reported by their operating system.
CLIENT_HOSTNAME	VARCHAR	The IP address and port of the TCP socket from which the client connection was made; NULL if the session is internal. Vertica accepts either IPv4 or IPv6 connections from a client machine. If the client machine contains mappings for both IPv4 and IPv6, the server randomly chooses one IP address family to make a connection. This can cause the CLIENT_HOSTNAME column to display either IPv4 or IPv6 values, based on which address family the server chooses.
CLIENT_PID	INTEGER	Linux process identifier of the client process that issued this connection. **Note:** The client process could be on a different machine from the server.
CLIENT_LABEL	VARCHAR	User-specified label for the client connection that can be set when using ODBC. See Label in DSN Parameters.
SSL_STATE	VARCHAR	Indicates if Vertica used Secure Socket Layer (SSL) for a particular session. Possible values are: • None – Vertica did not use SSL. • Server – Sever authentication was used, so the client could authenticate the server. • Mutual – Both the server and the client authenticated one another through mutual authentication. See Implementing Security and TLS protocol .
AUTHENTICATION_METHOD	VARCHAR	Type of client authentication used for a particular session, if known. Possible values are: • Unknown • Trust • Reject • Kerberos • Password • MD5 • LDAP • Kerberos-GSS • Ident See Security and authentication and Configuring client authentication .
CLIENT_TYPE	VARCHAR	The type of client from which the connection was made. Possible client type values: • ADO.NET Driver • ODBC Driver • JDBC Driver • vsql
CLIENT_VERSION	VARCHAR	Returns the client version.
CLIENT_OS	VARCHAR	Returns the client operating system.

CLIENT_OS_USER_NAME	VARCHAR	The name of the user that logged into, or attempted to log into, the database. This is logged even when the login attempt is unsuccessful.

Privileges

Non-superuser: No explicit privileges required. You only see records for tables that you have privileges to view.

See also

- CURRENT_SESSION
- SESSION_PROFILES
- SESSIONS
- SYSTEM_SESSIONS

API reference

The Vertica SDK has application programming interfaces (APIs) in the following languages:

- C++ SDK documentation
- Java SDK documentation
- Python SDK documentation
- R SDK

Vertica has the following client API:

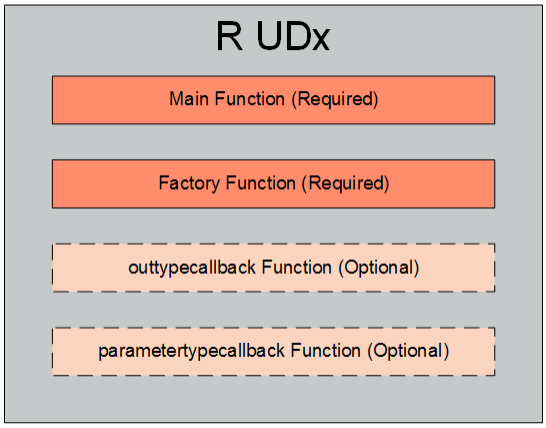
- JDBC API

In this section

- C++ SDK
- Java SDK
- R SDK
- Python SDK
- JDBC API

R SDK

Welcome to the Vertica R SDK API Documentation. This documentation covers all of the functions that make up the R SDK. Using this API, you can create User Defined Extensions (UDxs) that integrate your own features into the Vertica Analytics Platform.



To learn how UDxs work and how to develop, compile, and deploy them, see Developing user-defined extensions (UDxs) .

In this section

- Main function
- Factory function
- Outtypecallback function
- Parametertypecallback function

Main function

The first function called when Vertica runs an R user-defined extension (UDx). It must return a data frame and is defined by the name parameter of the factory function.

Example

```
FunctionName <- function(input_data_frame, parameters_data_frame) {
# Each column from Vertica is turned into a column in the input data frame.
# This is a brief example of multiplying two columns of the input data frame.
output_data_frame <- data.frame(input_data_frame[, 1] * input_data_frame[, 2])
# The function must return a data frame.
return(output_data_frame)
}
```

Arguments

input_data_frame	The data read by the Vertica user-defined extension (UDx). This input must be a data frame.
parameters_data_frame	The parameters passed to the Vertica user-defined extension (UDx). This input must be a data frame. If you are passing parameters from your Vertica UDx, defined with USING PARAMETERS, then you must included this in your R UDx. If you are not passing parameters to your R UDx, then this can be omitted from your R UDx.

Description

The main function takes as input one data frame as the first argument, and if your UDx accepts parameters, a parameter data frame as the second argument. The main function must return a data frame. The factory function converts the intype argument into a data frame for the main function. For example, intype=c("float","float") is converted by the factory function into a two-dimensional matrix.

The function can call other functions defined in your R UDx library file.

Factory function

The factory function consists of an R list which specifies the parameters of your R user-defined extension (UDx).

Example

```
FunctionNameFactory <- function() {
list(name               = MainFunctionName,
     udxtype            = c("scalar"),
     intype             = c("float","int", ...),
     outtype            = c("float", ...),
     outtypecallback    = MyReturnTypeFunc,
     parametertypecallback = MyParamReturnFunc,
     volatility         = c("volatile"),
     strictness         = c("called_on_null_input") )
}
```

Arguments

name	The name of the R function in this UDx that is called when the function is invoked by Vertica, and which returns the value(s) back to Vertica.
udxtype	The type of UDx, one of the following: • scalar • transform
intype	Data types of arguments accepted by the function. UDxs support up to 9800 arguments.
outtype	Data types of the arguments returned by the function.
outtypecallback	(Optional) The callback function to call before sending the data back to Vertica, which defines the types and precision that the main function returns.

parametertypecallback	The callback function to send parameter types and names to Vertica. This parameter is required if your UDx is called from Vertica with USING PARAMETERS.
volatility	(Optional) Indicates whether the function returns the same output given the same input, one of the following: • VOLATILE (default) • IMMUTABLE • STABLE For details, see Setting null input and volatility behavior for R functions .
strictness	(Optional) Indicates whether the function always returns NULL when any input argument is NULL, one of the following: • CALLED_ON_NULL_INPUT (default) • RETURN_NULL_ON_NULL_INPUT • STRICT For details, see Setting null input and volatility behavior for R functions .

Description

User-defined functions in R (R UDx) require a factory function for each main R function that you want to call from within Vertica. The factory function encapsulates all the information required by Vertica to load the R UDx.

Outtypecallback function

Function to define the column names, output data types, length/precision, and scale of the data being returned to Vertica.

Example

```
SalesTaxReturnTypes <- function(arguments.data.frame, parameters.data.frame) {
 output.return.type <- data.frame(datatype = rep(NA, 2),
                                  length   = rep(NA, 2),
                                  scale    = rep(NA, 2),
                                  name     = rep(NA, 2))
 output.return.type$datatype <- c("float", "float")
 output.return.type$name <- c("Sales Tax Rate", "Item Cost with Tax")
 return(output.return.type)
```

Arguments

* arguments.data.frame	Data frame that contains the arguments passed to the UDx from Vertica. This data frame is created and used internally by the UDx process.
* parameters.data.frame	Data frame that contains the parameters defined in the UDx from Vertica. This data frame is created and used internally by the UDx process.
datatype	Output data type.
length	(Optional) Dimension of the output.
scale	(Optional) Proportional dimensions of the output.
name	Column name of the output.

Description

When creating the outtypecallback function, define one row for each value returned. Use the same order as in the outtype defined in the factory function. If any columns are left blank or the outtypecallback function is omitted, then Vertica uses default values.

Important

When specifying LONG VARCHAR or LONG VARBINARY data types, include the space between the two words. For example:

```
datatype = c("long varchar")
```

Parametertypecallback function

Function to define the output data type(s), length/precision, and scale of the parameters sent to the main function.

Example

```
FunctionNameParameterTypes <- function() {
parameters <- list(datatype = c("int", "varchar"),
                   length   = c("NA", "100"),
                   scale    = c("NA", "NA"),
                   name     = c("k", "algorithm"))
return(parameters)
}
```

Arguments

datatype	The data type of the parameter.
length	(Optional) The dimension of the parameter.
scale	The proportional dimensions of the parameter.
name	The name of the parameter.

Description

When creating the parametertypecallback function, you define one row for each parameter being sent to the function. You must specify a parametertypecallback function if your UDx uses parameters.

The parametertypecallback does not accept any input arguments.

Vertica error messages

Welcome to the Vertica Error Codes guide. This guide is mainly for developers who need to understand how Vertica's error codes relate to error states returned by the ODBC and JDBC drivers.

The different ways Vertica reports errors

Vertica reports warnings and errors via two different mechanisms: SQLSTATEs and error messages. SQLSTATEs are intended for use by client applications, such as those accessing Vertica via ODBC or JDBC. Error messages are displayed to interactive users (for example, users connected to Vertica through vsql) and written to error logs.

About SQLSTATE

Vertica reports the success or failure of each statement it executes to client applications using a five-character SQLSTATE value. Many of these values are defined by the SQL standard. Others (identified by the letter "V" in their values) are Vertica-specific.

SQLSTATE values are grouped into classes which are defined by the first two characters in the SQLSTATE value. The last three characters indicate a specific condition within a class. For example, the SQLSTATE class 22 represents all data errors. The specific SQLSTATE value 22012 represents a division by zero error. SQLSTATE classes let an application that does not recognize a specific SQLSTATE value to still get a general idea of the result.

Warning and error messages

Each error and warning message displayed to interactive users or written to a log file by Vertica has its own numeric error code assigned to it. For example:

```
ERROR 3117: Division by zero
WARNING 4098: No projections found
ERROR 5617: Multiple WITH clauses not allowed
```

The error code number is not related to the SQLSTATE value. However, error and warning messages do correspond to a specific SQLSTATE. They are just a more-specific human-readable message compared to the SQLSTATE, which is mainly intended for client applications.

For example, all warning messages displayed by Vertica correspond to the SQLSTATE class 01. The warning message "WARNING 3084: Design Workspace couldn't be dropped" corresponds to the SQLSTATE value 01000 ERRCODE_WARNING.

Error codes do not change between Vertica releases, but individual error and warning messages may be added or removed in new releases. Your client application should not depend on particular error code appearing from one release to the next. Instead, it should use the SQLSTATE value to determine the result of executing a statement.

See the SQL state list for a list of all the SQLSTATE classes and values defined by Vertica. This table also links to lists of error or warning messages that are associated with each SQLSTATE value.

In this section

- SQL state list

SQL state list

The following table lists the SQLSTATE classes and individual SQLSTATE codes.

SQLState	Description	Details
Class 00—Successful Completion		
00000	SUCCESSFUL_COMPLETION	associated error messages
Class 01—Warning		
01000	WARNING	associated warning messages
01003	WARNING_NULL_VALUE_ELIMINATED_IN_SET_FUNCTION	
01004	WARNING_STRING_DATA_RIGHT_TRUNCATION	associated warning messages
01006	WARNING_PRIVILEGE_NOT_REVOKED	associated warning messages
01007	WARNING_PRIVILEGE_NOT_GRANTED	associated warning messages
01008	WARNING_PRIVILEGE_ALREADY_GRANTED	
01009	WARNING_PRIVILEGE_ALREADY_REVOKED	associated warning messages
0100C	WARNING_DYNAMIC_RESULT_SETS_RETURNED	
01V01	WARNING_DEPRECATED_FEATURE	associated warning messages
01V02	WARNING_QUERY_RETRIED	
Class 02—No Data		
02000	NO_DATA	associated error messages
02001	NO_ADDITIONAL_DYNAMIC_RESULT_SETS_RETURNED	
Class 03—SQL Statement Not Yet Complete		
03000	SQL_STATEMENT_NOT_YET_COMPLETE	
Class 08—Client Connection Exception		
08000	CONNECTION_EXCEPTION	associated error messages

08001	SQLCLIENT_UNABLE_TO_ESTABLISH_SQLCONNECTION	associated error messages
08003	CONNECTION_DOES_NOT_EXIST	associated error messages
08004	SQLSERVER_REJECTED_ESTABLISHMENT_OF_SQLCONNECTION	associated error messages
08006	CONNECTION_FAILURE	associated error messages
08007	TRANSACTION_RESOLUTION_UNKNOWN	
08V01	PROTOCOL_VIOLATION	associated error messages
Class 09—Triggered Action Exception		
09000	TRIGGERED_ACTION_EXCEPTION	associated error messages
Class 0A—Feature Not Supported		
0A000	FEATURE_NOT_SUPPORTED	associated error messages
0A005	PLAN_TO_SQL_NOT_SUPPORTED	associated error messages
Class 0B—Invalid Transaction Initiation		
0B000	INVALID_TRANSACTION_INITIATION	associated error messages
Class 0F—Locator Exception		
0F000	LOCATOR_EXCEPTION	
0F001	L_E_INVALID_SPECIFICATION	
Class 0L—Invalid Grantor		
0L000	INVALID_GRANTOR	
0LV01	INVALID_GRANT_OPERATION	associated error messages
Class 0P—Invalid Role Specification		
0P000	INVALID_ROLE_SPECIFICATION	
Class 20—Case Not Found		
20000	CASE_NOT_FOUND	associated error messages
Class 21—Cardinality Violation		
21000	CARDINALITY_VIOLATION	associated error messages
Class 22—Data Exception		
22000	DATA_EXCEPTION	associated error messages
22001	STRING_DATA_RIGHT_TRUNCATION	associated error messages
22002	NULL_VALUE_NO_INDICATOR_PARAMETER	
22003	NUMERIC_VALUE_OUT_OF_RANGE	associated error messages

22004	NULL_VALUE_NOT_ALLOWED	associated error messages
22005	ERROR_IN_ASSIGNMENT	
22007	INVALID_DATETIME_FORMAT	associated error messages
22008	DATETIME_FIELD_OVERFLOW	associated error messages
22009	INVALID_TIME_ZONE_DISPLACEMENT_VALUE	associated error messages
2200B	ESCAPE_CHARACTER_CONFLICT	associated error messages
2200C	INVALID_USE_OF_ESCAPE_CHARACTER	
2200D	INVALID_ESCAPE_OCTET	associated error messages
2200F	ZERO_LENGTH_CHARACTER_STRING	
2200G	MOST_SPECIFIC_TYPE_MISMATCH	
22010	INVALID_INDICATOR_PARAMETER_VALUE	
22011	SUBSTRING_ERROR	associated error messages
22012	DIVISION_BY_ZERO	associated error messages
22015	INTERVAL_FIELD_OVERFLOW	associated error messages
22018	INVALID_CHARACTER_VALUE_FOR_CAST	
22019	INVALID_ESCAPE_CHARACTER	associated error messages
2201B	INVALID_REGULAR_EXPRESSION	associated error messages
2201E	INVALID_ARGUMENT_FOR_LOG	
2201F	INVALID_ARGUMENT_FOR_POWER_FUNCTION	
2201G	INVALID_ARGUMENT_FOR_WIDTH_BUCKET_FUNCTION	associated error messages
22020	INVALID_LIMIT_VALUE	
22021	CHARACTER_NOT_IN_REPERTOIRE	associated error messages
22022	INDICATOR_OVERFLOW	
22023	INVALID_PARAMETER_VALUE	associated error messages
22024	UNTERMINATED_C_STRING	
22025	INVALID_ESCAPE_SEQUENCE	associated error messages
22026	STRING_DATA_LENGTH_MISMATCH	
22027	TRIM_ERROR	
2202E	ARRAY_ELEMENT_ERROR	
22906	NONSTANDARD_USE_OF_ESCAPE_CHARACTER	associated error messages

22V01	FLOATING_POINT_EXCEPTION	
22V02	INVALID_TEXT_REPRESENTATION	associated error messages
22V03	INVALID_BINARY_REPRESENTATION	associated error messages
22V04	BAD_COPY_FILE_FORMAT	associated error messages
22V05	UNTRANSLATABLE_CHARACTER	
22V0B	ESCAPE_CHARACTER_ON_NOESCAPE	associated error messages
22V21	INVALID_EPOCH	associated error messages
22V23	RAISE_EXCEPTION	associated error messages
22V24	COPY_PARSE_ERROR	associated error messages
Class 23—Integrity Constraint Violation		
23000	INTEGRITY_CONSTRAINT_VIOLATION	
23001	RESTRICT_VIOLATION	
23502	NOT_NULL_VIOLATION	associated error messages
23503	FOREIGN_KEY_VIOLATION	associated error messages
23505	UNIQUE_VIOLATION	associated error messages
23514	CHECK_VIOLATION	associated error messages
Class 24—Invalid Cursor State		
24000	INVALID_CURSOR_STATE	
Class 25—Invalid Transaction State		
25000	INVALID_TRANSACTION_STATE	associated error messages
25001	ACTIVE_SQL_TRANSACTION	
25002	BRANCH_TRANSACTION_ALREADY_ACTIVE	
25003	INAPPROPRIATE_ACCESS_MODE_FOR_BRANCH_TRANSACTION	
25004	INAPPROPRIATE_ISOLATION_LEVEL_FOR_BRANCH_TRANSACTION	
25005	NO_ACTIVE_SQL_TRANSACTION_FOR_BRANCH_TRANSACTION	
25006	READ_ONLY_SQL_TRANSACTION	associated error messages
25007	SCHEMA_AND_DATA_STATEMENT_MIXING_NOT_SUPPORTED	
25008	HELD_CURSOR_REQUIRES_SAME_ISOLATION_LEVEL	
25V01	NO_ACTIVE_SQL_TRANSACTION	associated error messages
25V02	IN_FAILED_SQL_TRANSACTION	

Class 26—Invalid SQL Statement Name		
26000	INVALID_SQL_STATEMENT_NAME	
Class 27—Triggered Data Change Violation		
27000	TRIGGERED_DATA_CHANGE_VIOLATION	
Class 28—Invalid Authorization Specification		
28000	INVALID_AUTHORIZATION_SPECIFICATION	associated error messages
28001	ACCOUNT_LOCKED	associated error messages
28002	PASSWORD_EXPIRED	
28003	PASSWORD_IN_GRACE_PERIOD	
Class 2B—Dependent Privilege Descriptors Still Exist		
2B000	DEPENDENT_PRIVILEGE_DESCRIPTORS_STILL_EXIST	
2BV01	DEPENDENT_OBJECTS_STILL_EXIST	associated error messages
Class 2D—Invalid Transaction Termination		
2D000	INVALID_TRANSACTION_TERMINATION	
Class 2F—SQL Routine Exception		
2F000	SQL_ROUTINE_EXCEPTION	
2F002	S_R_E_MODIFYING_SQL_DATA_NOT_PERMITTED	
2F003	S_R_E_PROHIBITED_SQL_STATEMENT_ATTEMPTED	
2F004	S_R_E_READING_SQL_DATA_NOT_PERMITTED	
2F005	S_R_E_FUNCTION_EXECUTED_NO_RETURN_STATEMENT	
Class 34—Invalid Cursor Name		
34000	INVALID_CURSOR_NAME	
Class 38—External Routine Exception		
38000	EXTERNAL_ROUTINE_EXCEPTION	
38001	E_R_E_CONTAINING_SQL_NOT_PERMITTED	
38002	E_R_E_MODIFYING_SQL_DATA_NOT_PERMITTED	
38003	E_R_E_PROHIBITED_SQL_STATEMENT_ATTEMPTED	
38004	E_R_E_READING_SQL_DATA_NOT_PERMITTED	associated error messages
Class 39—External Routine Invocation Exception		
39000	EXTERNAL_ROUTINE_INVOCATION_EXCEPTION	

39001	E_R_I_E_INVALID_SQLSTATE_RETURNED	
39004	E_R_I_E_NULL_VALUE_NOT_ALLOWED	
39V01	E_R_I_E_TRIGGER_PROTOCOL_VIOLATED	
39V02	E_R_I_E_SRF_PROTOCOL_VIOLATED	
Class 3B—Savepoint Exception		
3B000	SAVEPOINT_EXCEPTION	
3B001	S_E_INVALID_SPECIFICATION	
Class 3D—Invalid Catalog Name		
3D000	INVALID_CATALOG_NAME	
Class 3F—Invalid Schema Name		
3F000	INVALID_SCHEMA_NAME	
Class 40—Transaction Rollback		
40000	TRANSACTION_ROLLBACK	
40001	T_R_SERIALIZATION_FAILURE	
40002	T_R_INTEGRITY_CONSTRAINT_VIOLATION	
40003	T_R_STATEMENT_COMPLETION_UNKNOWN	
40V01	T_R_DEADLOCK_DETECTED	associated error messages
Class 42—Syntax Error or Access Rule Violation		
42000	SYNTAX_ERROR_OR_ACCESS_RULE_VIOLATION	
42501	INSUFFICIENT_PRIVILEGE	associated error messages
42601	SYNTAX_ERROR	associated error messages
42602	INVALID_NAME	associated error messages
42611	INVALID_COLUMN_DEFINITION	associated error messages
42622	NAME_TOO_LONG	associated error messages
42701	DUPLICATE_COLUMN	associated error messages
42702	AMBIGUOUS_COLUMN	associated error messages
42703	UNDEFINED_COLUMN	associated error messages
42704	UNDEFINED_OBJECT	associated error messages
42710	DUPLICATE_OBJECT	associated error messages
42712	DUPLICATE_ALIAS	associated error messages

42723	DUPLICATE_FUNCTION	associated error messages
42725	AMBIGUOUS_FUNCTION	associated error messages
42803	GROUPING_ERROR	associated error messages
42804	DATATYPE_MISMATCH	associated error messages
42809	WRONG_OBJECT_TYPE	associated error messages
42830	INVALID_FOREIGN_KEY	associated error messages
42846	CANNOT_COERCE	associated error messages
42883	UNDEFINED_FUNCTION	associated error messages
42939	RESERVED_NAME	associated error messages
42P20	WINDOWING_ERROR	associated error messages
42V01	UNDEFINED_TABLE	associated error messages
42V02	UNDEFINED_PARAMETER	associated error messages
42V03	DUPLICATE_CURSOR	associated error messages
42V04	DUPLICATE_DATABASE	associated error messages
42V05	DUPLICATE_PSTATEMENT	
42V06	DUPLICATE_SCHEMA	associated error messages
42V07	DUPLICATE_TABLE	associated error messages
42V08	AMBIGUOUS_PARAMETER	associated error messages
42V09	AMBIGUOUS_ALIAS	associated error messages
42V10	INVALID_COLUMN_REFERENCE	associated error messages
42V11	INVALID_CURSOR_DEFINITION	associated error messages
42V12	INVALID_DATABASE_DEFINITION	
42V13	INVALID_FUNCTION_DEFINITION	associated error messages
42V14	INVALID_PSTATEMENT_DEFINITION	
42V15	INVALID_SCHEMA_DEFINITION	associated error messages
42V16	INVALID_TABLE_DEFINITION	associated error messages
42V17	INVALID_OBJECT_DEFINITION	associated error messages
42V18	INDETERMINATE_DATATYPE	associated error messages
42V21	UNDEFINED_PROJECTION	associated error messages
42V22	UNDEFINED_NODE	associated error messages

42V23	UNDEFINED_PERMUTATION	
42V24	UNDEFINED_USER	associated error messages
42V25	PATTERN_MATCH_ERROR	associated error messages
42V26	DUPLICATE_NODE	associated error messages
Class 44—WITH CHECK OPTION Violation		
44000	WITH_CHECK_OPTION_VIOLATION	
Class 53—Insufficient Resources		
53000	INSUFFICIENT_RESOURCES	associated error messages
53100	DISK_FULL	associated error messages
53200	OUT_OF_MEMORY	associated error messages
53300	TOO_MANY_CONNECTIONS	
Class 54—Program Limit Exceeded		
54000	PROGRAM_LIMIT_EXCEEDED	associated error messages
54001	STATEMENT_TOO_COMPLEX	associated error messages
54011	TOO_MANY_COLUMNS	associated error messages
54023	TOO_MANY_ARGUMENTS	associated error messages
Class 55—Object Not In Prerequisite State		
55000	OBJECT_NOT_IN_PREREQUISITE_STATE	associated error messages
55006	OBJECT_IN_USE	associated error messages
55V02	CANT_CHANGE_RUNTIME_PARAM	associated error messages
55V03	LOCK_NOT_AVAILABLE	associated error messages
55V04	TM_MARKER_NOT_AVAILABLE	associated error messages
Class 57—Operator Intervention		
57000	OPERATOR_INTERVENTION	
57014	QUERY_CANCELED	associated error messages
57015	SLOW_DELETE	associated error messages
57V01	ADMIN_SHUTDOWN	associated error messages
57V02	CRASH_SHUTDOWN	
57V03	CANNOT_CONNECT_NOW	associated error messages
57V04	DML_COMMIT_DURING_SHUTDOWN	associated error messages

Class 58—System Error		
58030	IO_ERROR	associated error messages
58V01	UNDEFINED_FILE	associated error messages
58V02	DUPLICATE_FILE	
Class V0—PL/vSQL errors		
V0001	PLVSQL_ERROR	associated error messages
V0002	PLVSQL_RAISE	
V0003	NO_DATA_FOUND	
V0004	TOO_MANY_ROWS	
V0005	ASSERT_FAILURE	associated error messages
Class V1—Vertica-specific multi-node errors class		
V1001	LOST_CONNECTIVITY	associated error messages
V1002	K_SAFETY_VIOLATION	associated error messages
V1003	CLUSTER_CHANGE	associated error messages
Class V2—Vertica-specific miscellaneous errors class		
V2000	AUTH_FAILED	associated error messages
V2001	LICENSE_ISSUE	associated error messages
V2002	MOVEOUT_ABORTED	
Class VC—Configuration File Error		
VC001	CONFIG_FILE_ERROR	associated error messages
VC002	LOCK_FILE_EXISTS	
Class VD—DB Designer errors		
VD001	DESIGNER_FUNCTION_ERROR	associated error messages
Class VP—User procedure errors		
VP000	USER_PROC_ERROR	associated error messages
VP001	USER_PROC_EXEC_ERROR	associated error messages
Class VX—Internal Error		
VX001	INTERNAL_ERROR	associated error messages
VX002	DATA_CORRUPTED	associated error messages
VX003	INDEX_CORRUPTED	associated error messages

VX004	PLAN_TO_SQL_INTERNAL_EROR	associated error messages
VX005	INTERNAL_FILE_NOT_FOUND	

In this section

- Messages associated with SQLSTATE 00000
- Messages associated with SQLSTATE 01000
- Messages associated with SQLSTATE 01004
- Messages associated with SQLSTATE 01006
- Messages associated with SQLSTATE 01007
- Messages associated with SQLSTATE 01009
- Messages associated with SQLSTATE 01V01
- Messages associated with SQLSTATE 02000
- Messages associated with SQLSTATE 08000
- Messages associated with SQLSTATE 08001
- Messages associated with SQLSTATE 08003
- Messages associated with SQLSTATE 08004
- Messages associated with SQLSTATE 08006
- Messages associated with SQLSTATE 08V01
- Messages associated with SQLSTATE 09000
- Messages associated with SQLSTATE 0A000
- Messages associated with SQLSTATE 0A005
- Messages associated with SQLSTATE 0B000
- Messages associated with SQLSTATE 0LV01
- Messages associated with SQLSTATE 20000
- Messages associated with SQLSTATE 21000
- Messages associated with SQLSTATE 22000
- Messages associated with SQLSTATE 22001
- Messages associated with SQLSTATE 22003
- Messages associated with SQLSTATE 22004
- Messages associated with SQLSTATE 22007
- Messages associated with SQLSTATE 22008
- Messages associated with SQLSTATE 22009
- Messages associated with SQLSTATE 2200B
- Messages associated with SQLSTATE 2200D
- Messages associated with SQLSTATE 22011
- Messages associated with SQLSTATE 22012
- Messages associated with SQLSTATE 22015
- Messages associated with SQLSTATE 22019
- Messages associated with SQLSTATE 2201B
- Messages associated with SQLSTATE 2201G
- Messages associated with SQLSTATE 22021
- Messages associated with SQLSTATE 22023
- Messages associated with SQLSTATE 22025
- Messages associated with SQLSTATE 22906
- Messages associated with SQLSTATE 22V02
- Messages associated with SQLSTATE 22V03
- Messages associated with SQLSTATE 22V04
- Messages associated with SQLSTATE 22V0B
- Messages associated with SQLSTATE 22V21
- Messages associated with SQLSTATE 22V23
- Messages associated with SQLSTATE 22V24
- Messages associated with SQLSTATE 23502
- Messages associated with SQLSTATE 23503
- Messages associated with SQLSTATE 23505
- Messages associated with SQLSTATE 23514
- Messages associated with SQLSTATE 25000
- Messages associated with SQLSTATE 25006
- Messages associated with SQLSTATE 25V01
- Messages associated with SQLSTATE 28000

- Messages associated with SQLSTATE 28001
- Messages associated with SQLSTATE 2BV01
- Messages associated with SQLSTATE 38004
- Messages associated with SQLSTATE 40V01
- Messages associated with SQLSTATE 42501
- Messages associated with SQLSTATE 42601
- Messages associated with SQLSTATE 42602
- Messages associated with SQLSTATE 42611
- Messages associated with SQLSTATE 42622
- Messages associated with SQLSTATE 42701
- Messages associated with SQLSTATE 42702
- Messages associated with SQLSTATE 42703
- Messages associated with SQLSTATE 42704
- Messages associated with SQLSTATE 42710
- Messages associated with SQLSTATE 42712
- Messages associated with SQLSTATE 42723
- Messages associated with SQLSTATE 42725
- Messages associated with SQLSTATE 42803
- Messages associated with SQLSTATE 42804
- Messages associated with SQLSTATE 42809
- Messages associated with SQLSTATE 42830
- Messages associated with SQLSTATE 42846
- Messages associated with SQLSTATE 42883
- Messages associated with SQLSTATE 42939
- Messages associated with SQLSTATE 42P20
- Messages associated with SQLSTATE 42V01
- Messages associated with SQLSTATE 42V02
- Messages associated with SQLSTATE 42V03
- Messages associated with SQLSTATE 42V04
- Messages associated with SQLSTATE 42V06
- Messages associated with SQLSTATE 42V07
- Messages associated with SQLSTATE 42V08
- Messages associated with SQLSTATE 42V09
- Messages associated with SQLSTATE 42V10
- Messages associated with SQLSTATE 42V11
- Messages associated with SQLSTATE 42V13
- Messages associated with SQLSTATE 42V15
- Messages associated with SQLSTATE 42V16
- Messages associated with SQLSTATE 42V17
- Messages associated with SQLSTATE 42V18
- Messages associated with SQLSTATE 42V21
- Messages associated with SQLSTATE 42V22
- Messages associated with SQLSTATE 42V24
- Messages associated with SQLSTATE 42V25
- Messages associated with SQLSTATE 42V26
- Messages associated with SQLSTATE 53000
- Messages associated with SQLSTATE 53100
- Messages associated with SQLSTATE 53200
- Messages associated with SQLSTATE 54000
- Messages associated with SQLSTATE 54001
- Messages associated with SQLSTATE 54011
- Messages associated with SQLSTATE 54023
- Messages associated with SQLSTATE 55000
- Messages associated with SQLSTATE 55006
- Messages associated with SQLSTATE 55V02
- Messages associated with SQLSTATE 55V03
- Messages associated with SQLSTATE 55V04
- Messages associated with SQLSTATE 57014
- Messages associated with SQLSTATE 57015
- Messages associated with SQLSTATE 57V01
- Messages associated with SQLSTATE 57V03

- Messages associated with SQLSTATE 57V04
- Messages associated with SQLSTATE 58030
- Messages associated with SQLSTATE 58V01
- Messages associated with SQLSTATE V0001
- Messages associated with SQLSTATE V0005
- Messages associated with SQLSTATE V1001
- Messages associated with SQLSTATE V1002
- Messages associated with SQLSTATE V1003
- Messages associated with SQLSTATE V2000
- Messages associated with SQLSTATE V2001
- Messages associated with SQLSTATE VC001
- Messages associated with SQLSTATE VD001
- Messages associated with SQLSTATE VP000
- Messages associated with SQLSTATE VP001
- Messages associated with SQLSTATE VX001
- Messages associated with SQLSTATE VX002
- Messages associated with SQLSTATE VX003
- Messages associated with SQLSTATE VX004

Messages associated with SQLSTATE 00000

This topic lists the messages associated with the SQLSTATE 00000.

SQLSTATE 00000 description

SUCCESSFUL_COMPLETION

Messages associated with this SQLState

Info messages

INFO 2372: Cannot commit; no transaction in progress INFO 2499: Cannot rollback; no transaction in progress

Notice messages

NOTICE 2005: *string*

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 01000

This topic lists the messages associated with the SQLSTATE 01000.

SQLSTATE 01000 description

WARNING

Messages associated with this SQLState

Error messages

ERROR 5922: Insufficient privileges to alter table *string*

Notice messages

NOTICE 2005: *string*
NOTICE 2519: Cannot shut down while users are connected
NOTICE 2690: Compression hint specified for column '*string*' does nothing
because column is GROUPED
NOTICE 3175: Encoding requested for column '*string*' does nothing
NOTICE 3512: Ignoring data path, change unsupported
NOTICE 4927: The *string string* depends on *string*
NOTICE 4929: The *string* [*string*] depends on *string*
NOTICE 5748: API limitation: affinities will be set only for the first
value (out of *value*) CPUs
NOTICE 6008: Resetting CPU affinity set for resource pool *string* to the
default (ANY)
NOTICE 6182: Cannot shut down while DBD sessions are running
NOTICE 7090: Cannot set affinity set for resource pool '*string*' because
the affinity mode is ANY
NOTICE 7338: The column "*string*" of table "*string*" is referenced by table
"*string*", column "*string*"
NOTICE 8689: *value* objects could not be re-parented
NOTICE 9088: Omitting rest of dependent objects, overall *value* objects
depending on *string*
NOTICE 9503: The *string* [*value*] depends on *string*

Warning messages

WARNING 2005: *string*
WARNING 2362: Cannot begin transaction; transaction is already running
WARNING 3084: Design couldn't be dropped
WARNING 3152: Duplicate values in columns marked as UNIQUE will now be
ignored for the remainder of your session or until
reenable_duplicate_key_error() is called
WARNING 3372: Failed to disable profiling: *string*
WARNING 3373: Failed to enable profiling: *string*
WARNING 3539: Incorrect results are possible. Please contact Vertica
Support if unsure
WARNING 3791: Invalid view *string*: *string*
WARNING 4071: NO COMMIT option will be ignored for external table "*string*"
WARNING 4088: No new valid default roles specified. Retaining previous
set of default roles for user *string*
WARNING 4098: No projections found
WARNING 4102: No rows are inserted into table "*string*"."*string*" because ON
COMMIT DELETE ROWS is the default for create temporary
table
WARNING 4116: No super projections created for table *string.*
WARNING 4246: Only GLOBAL scope is supported for clearing *string* profiles
WARNING 4463: Projection *string* is not up to date
WARNING 4468: Projection <*string*> is not available for query processing.
Execute the select start_refresh() function to copy data
into this projection.

The projection must have a sufficient number of buddy
projections and all nodes must be up before starting a
refresh
WARNING 4539: Received no response from *stringstring*
WARNING 4792: Storage option "*string*" will be ignored for external table
"*string*"
WARNING 4871: System view *string* for tuning rule *string* is currently invalid
WARNING 4873: System view for tuning rule *string* does not exist
WARNING 4996: This request may deadlock the system. Please report the
details to technical support
WARNING 5068: Total declared length of columns of one of the
constraints exceeds the limit, truncation may happen

WARNING 5119: UDx code didn't respond when Vertica tried to get function prototype for *string* in library *string*: *string*
WARNING 5448: View *string* is currently invalid
WARNING 5451: Violations of some of foreign key constraints may not be reported because of no privilege on the foreign tables
WARNING 5642: Projection *string* is not persistent or not up to date; it will not be copied
WARNING 5717: No statistics has been exported. Either the DB is empty or you try to export an external table or you do not have access to the available objects
WARNING 5724: Segmentation clause contains a *string* - data loads may be slowed significantly
WARNING 5727: Sort clause contains a *string* - data loads may be slowed significantly
WARNING 5741: View *string* depends on other relations
WARNING 5819: Design could not be reset
WARNING 5821: Detected keys sharing the same case-insensitive key name
WARNING 5860: Due to the data isolation of temp tables with an on-commit-delete-rows policy, the compute_flextable_keys() and compute_flextable_keys_and_build_view() functions cannot access this table's data. The build_flextable_view() function can be used with a user-provided keys table to create a view, but involves a DDL commit which will delete the table's rows
WARNING 5873: Failed to add table *string* of hcatalog schema *string* to catalog: *string*
WARNING 5875: Failed to alter table *string* of hcatalog schema *string* to catalog: *string*
WARNING 5880: Failed to describe table *string* in hcatalog database *string*: *string*
WARNING 5881: Failed to describe table *string* in schema *string*: HCatalog database *string* does not exist
WARNING 5884: Failed to list hcatalog tables of hcatalog schema *string*: *string*
WARNING 5886: Failed to mirror table *string* in schema *string*: *string*
WARNING 5909: Found and ignored keys with names longer than the maximum column-name length limit
WARNING 5912: HASH() arguments contain expressions that reference table columns
WARNING 5917: Ignored some keys since the total key count exceeds the view column limit
WARNING 5922: Insufficient privileges to alter table *string*
WARNING 5923: Insufficient privileges to drop table *string*
WARNING 5991: Projection basename "*string*" hint was ignored. "*string*" is used as the basename
WARNING 5993: Projection is irregularly segmented by column
WARNING 6053: The view creation involved a DDL commit which deleted the table's rows
WARNING 6256: Error while analyzing constraint(s) *string* on *string*
WARNING 6257: Error while creating LTT for analyzing constraints on *string*
WARNING 6356: No partitions have been swapped. Neither table has partitions fall in range
WARNING 6417: Swapped partitions are not immediately moved to a new storage location
WARNING 6515: Projection with name "*string*" already exists in schema "*string*" for anchor table "*string*.*string*"
WARNING 6565: Inequal query trees produced the same hash code query1:

string

query2:

string

WARNING 6594: Unknown RangeTblEntry:*value*
WARNING 6608: *string* is an invalid argument for hint *string*
WARNING 6765: Error parsing distrib value *string* in Distrib hint's arguments. The whole hint will be ignored
WARNING 6773: Failed to compare plans for query '*string*'
WARNING 6797: Hint *string* can accept at most one argument, the hint with *value* arguments is ignored
WARNING 6798: Hint *string* can be specified only once for each join, if multiple instances listed only the first is considered
WARNING 6799: Hint *string* is not feasible and will be ignored
WARNING 6800: Hint *string* must have one argument
WARNING 6801: Hint *string* requires exactly two arguments, the hint with *value* arguments is ignored
WARNING 6813: Inherited privileges are globally disabled; schema parameter is set but has no effect
WARNING 6814: Inherited privileges are globally disabled; table parameter is set but has no effect
WARNING 6815: Inherited privileges are globally disabled; view parameter is set but has no effect
WARNING 6818: Input operations specified for Hint *string* is not feasible and will be ignored
WARNING 6922: Projection name was changed to *string* because it conflicts with the basename of the table *string*
WARNING 6990: Text index has invalid tokenizer
WARNING 6991: The active saved plan for this query has incompatible output column set. Continuing with the original query
WARNING 7000: The projection '*string*' was used to enforce the enabled key constraint '*string*', and may be regenerated to validate a DML statement on the base table
WARNING 7020: Unable to find the following query in the export file: '*string*'
WARNING 7030: Unexpected group clause found during query comparison
WARNING 7031: Unexpected group clause found during query hashing
WARNING 7033: Unexpected sort clause found during query comparison
WARNING 7036: Unknown Expr:*value* during query comparison
WARNING 7037: Unknown Expr:*value* found
WARNING 7038: Unknown Expr:*value* found during query hashing
WARNING 7039: Unknown Inequality OpOid:*value* found during query hashing
WARNING 7040: Unknown Logic OpExpr:*value* found during query comparison
WARNING 7041: Unknown Logic OpExpr:*value* found during query hashing
WARNING 7044: Unknown OpExpr:*value* found during query comparison
WARNING 7045: Unknown OpExpr:*value* found during query hashing
WARNING 7046: Unknown RangeTblEntry:*value* found during query comparison
WARNING 7047: Unknown RangeTblEntry:*value* found during query hashing
WARNING 7048: Unknown Sublink:*value* found during query comparison
WARNING 7049: Unknown Sublink:*value* found during query hashing
WARNING 7052: Unsupported JoinExpr Type found during query comparison
WARNING 7053: Unsupported JoinExpr Type found during query hashing
WARNING 7070: View "*string*" will include privileges from schema "*string*"
WARNING 7074: You appear to be using an old version of HDFS that may have stability problems under high load. See the Vertica documentation for supported HDFS versions
WARNING 7186: Inherited privileges are globally disabled. Privileges will be materialized and visibility of table *string* may change
WARNING 7265: Inherited privileges are globally disabled. Privileges will be materialized and visibility of view *string* may change
WARNING 7709: Could not drop table '*string*'. Please remove manually if necessary.

Detail: *string*
WARNING 7752: Cannot make a local query plan for "try local" hint, please check hash function, segment quantity, segment boundary, etc
WARNING 7793: The cache will be set to 1 instead
WARNING 7942: Unknown expression found during hashing: *value*
WARNING 7994: Not converged before max_iterations [*value*]
WARNING 8012: The parameter [*string*] is not defined for this function
WARNING 8086: Encoding option specified for attribute (*string*) of external table (*string*) will be ignored
WARNING 8275: The following shards missed the *string* due to missing primary subscriber (probably down): *string*
WARNING 8343: Not converged before max_iterations *value*
WARNING 8413: Lambda provided without regularization type: default for regularization is [*string*]; lambda will have no effect
WARNING 8414: max_depth is set to *value* while max_breadth to *value*. This means the size of trees may become limited by *string* first
WARNING 8419: Parameters [*string*] are not supported for optimizer [*string*], only for [*string*]
WARNING 8506: CGD optimizer could not invert covariance matrix which is required for calculating statistics. Coefficients are not affected
WARNING 8526: Regularization type [*string*] may cause optimizer [*string*] to not converge
WARNING 8531: *string* Directory for errors files was not created.

Unable to write errors for this instance of COPY command
WARNING 8547: Not converged before maximum inner iterations *value*
WARNING 8557: Feature *string* not applicable in Eon mode
WARNING 8580: Failed to convert internal form to readable form: *string*
WARNING 8581: Failed to convert to internal form: *string*
WARNING 8582: Failed to describe hcatalog tables of hcatalog schema *string*: *string*
WARNING 8583: Given lambda value [*value*] will result in all zero coefficients; use a value lower than lambda max for this dataset [*value*]
WARNING 8602: Caught exception while parsing JSON
WARNING 8621: The following string provided is not valid JSON: *string*
WARNING 8704: This table is not partitioned. Ignore activePartitionCount changes
WARNING 8737: The meta function set_recover_by_table is disabled in 9.1SP1. set_recover_test_settings() is supposed to be used only for testing purpose by developers
WARNING 8818: Library and dependencies have a combined size of *value* GB,
WARNING 8841: Consider enabling MergeOutCache when reflexive mergeout is disabled. Otherwise background TM Mergeout service may affect the query performance
WARNING 8953: Hint ECSMode is not feasible and will be ignored
WARNING 9002: Setting the token timeout greater than *value* msec may severely decrease cluster responsiveness to node shutdowns and failures
WARNING 9011: MergeOutCache doesn't work when reflexive mergeout is enabled
WARNING 9127: UPDATE/DELETE a table with aggregate projections (LAPs) will automatically run refresh on the LAPs when this transaction commits
WARNING 9210: Target node *string* is down, so depot size has been estimated from depot location on initiator. As soon as the node

comes up, its depot size might be altered depending on its disk size
WARNING 9337: Caught exception while connecting to LDAP: *string*
WARNING 9740: Changed max_depth to [*value*] to avoid a structural limit
WARNING 9974: No DFS file replicas available for retrieval of DFS file located at *string* (Oid: *value*)from its DFS file distribution *value*. This file will be dropped by Recovery
WARNING 9975: No nodes in cluster contain DFSFileBlocks for retrieving DFS file located at *string*: (Oid: *value*).This file will be dropped
WARNING 9990: :c hint in Create directed query CUSTOM has no effect, every constant counts by default
WARNING 9994: IgnoreConst/:v hint in Create directed query OPT has no effect, all constants are ignored by default
WARNING 10003: Unsupported hint for Constant type
WARNING 10251: Failed to disable debugging: *string*
WARNING 10252: Failed to enable debugging: *string*
WARNING 10365: No mergeout ran because mergeout is disabled on table(s) *string*
WARNING 10395: New query label *string* will replace already set label *string*
WARNING 10420: New range-partitioned projection *string* is segmented. Because anchor table *string* has no unsegmented super projections, the new projection cannot be used to process queries
WARNING 10440: Endpoint does not exist (http_code=*value*).Phone home data upload to [*string*] failed: *string*
WARNING 10441: Failed to collect Phone home data
WARNING 10447: Phone home data upload to [*string*] failed: *string*
WARNING 10582: Failed to parse schema [*string*] with error [*string*]
WARNING 10586: Token introspection request to [*string*] failed: *string*
WARNING 10606: Discovery URL request to [*string*] failed: *string*
WARNING 10608: Failed to parse schema of discovery URL response [*string*] with error [*string*]

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 01004

This topic lists the messages associated with the SQLSTATE 01004.

SQLSTATE 01004 description

WARNING_STRING_DATA_RIGHT_TRUNCATION

Messages associated with this SQLState

WARNING 7166: client_label exceeded maximum length; truncated to 255 characters

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 01006

This topic lists the messages associated with the SQLSTATE 01006.

SQLSTATE 01006 description

WARNING_PRIVILEGE_NOT_REVOKED

Messages associated with this SQLState

ERROR 4925: The *string* "*string*" cannot be *string string* "*string*"

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 01007

This topic lists the messages associated with the SQLSTATE 01007.

SQLSTATE 01007 description

WARNING_PRIVILEGE_NOT_GRANTED

Messages associated with this SQLState

WARNING 5682: USAGE privilege on schema "*string*" also needs to be granted to "*string*"

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 01009

This topic lists the messages associated with the SQLSTATE 01009.

SQLSTATE 01009 description

WARNING_PRIVILEGE_ALREADY_REVOKED

Messages associated with this SQLState

NOTICE 2495: Cannot revoke *string*"*string*" privilege(s) for *string string* that you did not grant to "*string*"

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 01V01

This topic lists the messages associated with the SQLSTATE 01V01.

SQLSTATE 01V01 description

WARNING_DEPRECATED_FEATURE

Messages associated with this SQLState

```
WARNING 2693: Configuration parameter string has been deprecated; setting
    it has no effect
WARNING 4736: set_local_segment_threshold has been deprecated; setting
    it has no effect
WARNING 8461: drop_partition has been deprecated and will be removed
    in a future version
WARNING 8462: merge_partitions has been deprecated and will be removed
    in a future version
WARNING 8463: merge_projection_partitions has been deprecated and will
    be removed in a future version
WARNING 8751: iterate_ros_objs has been deprecated and will be removed
    in a future version
WARNING 8757: rollover_minmax_obj has been deprecated and has no
    effect
WARNING 9382: Database branching is removed
WARNING 9508: Function string is deprecated
WARNING 9515: This version of Database Designer is deprecated and will
    be replaced in a future release
WARNING 9536: analyze_correlations is deprecated
WARNING 10146: string has been deprecated. Use 'Export to Delimited'
     instead
WARNING 10450: SHARED DATA and SHARED DATATEMP locations are
     deprecated and will not be supported in future versions
WARNING 10609: INFER_EXTERNAL_TABLE_DDL('path' [USING PARAMETERS
     format='parquet', table_name='default_table'] ) has
     been deprecated. Please use INFER_TABLE_DDL('path'
     [USING PARAMETERS format='parquet',
     table_name='default_table', table_type = 'external'] )
```

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 02000

This topic lists the messages associated with the SQLSTATE 02000.

SQLSTATE 02000 description

NO_DATA

Messages associated with this SQLState

```
ERROR 3741: Invalid range
ERROR 9997: Invalid partition range. Min partition key is greater than
     Max partition key
```

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 08000

This topic lists the messages associated with the SQLSTATE 08000.

SQLSTATE 08000 description

CONNECTION_EXCEPTION

Messages associated with this SQLState

Error messages

ERROR 2029: *string* from stdin failed: *string*
ERROR 2708: Connection to database [*string*] is invalid
ERROR 2896: Could not receive data from server:*string*
ERROR 2908: Could not send data to server: *string*
ERROR 3276: Error while waiting on socket. *value*
ERROR 4342: Password encryption failed
ERROR 5197: Unknown authentication method (*value*) requested by server
ERROR 7662: MD5 password hash requested when MD5 is not allowed
ERROR 8941: Could not establish SSL connection
ERROR 8942: Could not establish SSL connection: *string*
ERROR 8943: Could not establish SSL connection: Remote server
certificate could not be validated with error [*string*]
ERROR 8945: Could not establish TLS connection: TLS is not enabled on
remote cluster
ERROR 9103: Could not set remote cluster hostname: *string*
ERROR 9104: Could not set SSL fd: *string*

Fatal messages

FATAL 5273: Unsupported frontend protocol *value.value*: server supports *value*.0
to *value.value*
FATAL 7580: Unsupported frontend protocol *value.value*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 08001

This topic lists the messages associated with the SQLSTATE 08001.

SQLSTATE 08001 description

SQLCLIENT_UNABLE_TO_ESTABLISH_SQLCONNECTION

Messages associated with this SQLState

ERROR 2322: Cancel() -- connect() failed:
ERROR 2324: Cancel() -- socket() failed:
ERROR 2823: Could not connect to server [*string*]: *string*

Is the server running and accepting

TCP/IP connections on port *string*?
ERROR 2824: Could not connect to server: *string*

Is the server running on host [*string*] and accepting

TCP/IP connections on port *string*?
ERROR 2839: Could not create socket: *string*
ERROR 2865: Could not get client address from socket: *string*
ERROR 2869: Could not get socket error status: *string*
ERROR 2912: Could not set socket to close-on-exec mode: *string*
ERROR 2913: Could not set socket to non-blocking mode: *string*
ERROR 2914: Could not set socket to TCP no delay mode: *string*
ERROR 7801: Could not translate host name "*string*" to address using family
"*string*": *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 08003

This topic lists the messages associated with the SQLSTATE 08003.

SQLSTATE 08003 description

CONNECTION_DOES_NOT_EXIST

Messages associated with this SQLState

ERROR 4717: Server closed the connection unexpectedly

This probably means the server terminated abnormally

before or while processing the request

Note

The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 08004

This topic lists the messages associated with the SQLSTATE 08004.

SQLSTATE 08004 description

SQLSERVER_REJECTED_ESTABLISHMENT_OF_SQLCONNECTION

Messages associated with this SQLState

FATAL 4060: New session rejected due to limit, already *value* sessions active

FATAL 7470: New session rejected because connection limit of *value* on *string* already met for *string*

FATAL 10611: New session rejected because subcluster to which this node belongs is draining connections

Note

The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 08006

This topic lists the messages associated with the SQLSTATE 08006.

SQLSTATE 08006 description

CONNECTION_FAILURE

Messages associated with this SQLState

Commerror messages

COMMERROR 2607: Client has disconnected

COMMERROR 5167: Unexpected EOF on client connection

Error messages

ERROR 2323: Cancel() -- send() failed: *string*
ERROR 2606: Client failed when looking for pending signals
ERROR 2607: Client has disconnected
ERROR 4539: Received no response from *stringstring*
ERROR 8466: Received no response from (*string*), operation (*string*)
ERROR 8503: Cannot get a reply from node: *string*
ERROR 8944: Could not establish TLS connection. Please check the
Vertica log for more details
ERROR 8979: Remote Vertica cluster did not connect over TLS but TLS is
required for Import/Export on this cluster because
ImportExportTLSMode is set to *string*
ERROR 9768: RPC: peer went down during call: *string*

Fatal messages

FATAL 2607: Client has disconnected
FATAL 4777: SSL initialization failure
FATAL 8072: Client hasn't responded during grace period!

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 08V01

This topic lists the messages associated with the SQLSTATE 08V01.

SQLSTATE 08V01 description

PROTOCOL_VIOLATION

Messages associated with this SQLState

Commerror messages

COMMERROR 3337: Expected GSS response, got message type '*string*'
COMMERROR 3339: Expected password response, got message type '*string*'
COMMERROR 3532: Incomplete message from client
COMMERROR 3700: Invalid message length
COMMERROR 5168: Unexpected EOF within message length word
COMMERROR 8088: Expected changepassword response, got message type
'*string*'
COMMERROR 8089: Expected LDAP password response, got message type '*string*'
COMMERROR 9176: X509 can't generate digest for cert_chain; error *value*
COMMERROR 9333: *string*: OpenSSL SYSCALL error: EOF detected
COMMERROR 9334: *string*: TLS error: *string*
COMMERROR 9335: *string*: unrecognized OpenSSL error code: *value*
COMMERROR 9339: Could not accept TLS connection (2): EOF detected
COMMERROR 9340: Could not accept TLS connection (3): *string*
COMMERROR 9341: Could not accept TLS connection (4): EOF detected
COMMERROR 9342: Could not initialize TLS connection: *string*
COMMERROR 9344: Could not set TLS socket: *string*
COMMERROR 9349: TLS renegotiation failure
COMMERROR 9352: Unrecognized OpenSSL error code: *value*

Error messages

ERROR 2055: *string* Unexpected message type *string* reading from stdin
ERROR 2257: Bind message has *value* parameter formats but *value* parameters
ERROR 2258: Bind message has *value* result formats but query has *value* columns
ERROR 3334: Expected a RowDescription Message
ERROR 3335: Expected a SendExport Message
ERROR 3575: Insufficient data left in message
ERROR 3631: Invalid CLOSE message subtype *value*
ERROR 3651: Invalid DESCRIBE message subtype *value*
ERROR 3699: Invalid message format
ERROR 3701: Invalid message type
ERROR 3702: Invalid message type *value*
ERROR 3755: Invalid string in message
ERROR 3887: Lost synchronization with server: length *value*
ERROR 4074: No data left in message
ERROR 4718: Server did not identify with a pid & key
ERROR 5181: Unexpected message type 0x*hex value*
ERROR 5208: Unknown message from server

Fatal messages

FATAL 3667: Invalid frontend message type *value*
FATAL 3668: Invalid frontend message type *string*
FATAL 3739: Invalid protocol used.Message type *value* (*character*)
FATAL 3753: Invalid startup packet layout: expected terminator as last byte
FATAL 4073: No data files to load
FATAL 5776: Cannot send load balance request after SSL negotiation

Warning messages

WARNING 5872: Expected to flush an end-of-batch client message but received a message of type *value*. Attempting to recover...
WARNING 6863: MARS operation not supported for your client version. Parameter not changed

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 09000

This topic lists the messages associated with the SQLSTATE 09000.

SQLSTATE 09000 description

TRIGGERED_ACTION_EXCEPTION

Messages associated with this SQLState

Error messages

ERROR 2005: *string*
ERROR 7353: A problem during the execution of writeModel.

Detail: *string*
ERROR 7356: A problem occurred during the execution of a BFGS iteration.

Detail: *string*
ERROR 7358: A problem occurred during the execution of a MinMax iteration.

Detail: *string*
ERROR 7359: A problem occurred during the execution of a Newton

iteration.

Detail: *string*

ERROR 7360: A problem occurred during the execution of a robust_zscore iteration.

Detail: *string*

ERROR 7361: A problem occurred during the execution of a zscore iteration.

Detail: *string*

ERROR 7362: A problem occurred during the execution of an iteration.

Detail: *string*

ERROR 7373: Cannot check if all columns are numeric.

Detail: *string*

ERROR 7377: Cannot compute input column list.

Detail: *string*

ERROR 7378: Cannot find the current user.

Detail: *string*

ERROR 7383: Cannot make inf clause.

Detail: *string*

ERROR 7384: Cannot make null clause.

Detail: *string*

ERROR 7395: Could not compute evaluation metrics.

Detail: *string*

ERROR 7396: Could not create view '*string*'.

Detail: *string*

ERROR 7497: Problem in balance.

Detail: *string*

ERROR 7498: Problem in balance: Unknown exception

ERROR 7503: Problem in detect_outliers.

Detail: *string*

ERROR 7504: Problem in detect_outliers: Unknown exception

ERROR 7505: Problem in initializing centers table.

Detail: *string*

ERROR 7506: Problem in kmeans.

Detail: *string*

ERROR 7507: Problem in kmeans: Unknown exception

ERROR 7516: Problem in normalize.

Detail: *string*

ERROR 7517: Problem in normalize: Unknown exception

ERROR 7524: Problem in writing initial model to DFS.

Detail: *string*

ERROR 7593: Could not randomly pick *value* distinct centers after trying *value* times

ERROR 7626: A problem during the execution of computing the number of detected outliers.

Detail: *string*

ERROR 7731: Problem in calculating alpha for Linear Regression.

Detail: *string*

ERROR 7732: Problem in calculating Hessian matrix for Linear Regression.

Detail: *string*

ERROR 7733: Problem in strongWolfeLineSearch.

Detail: *string*

ERROR 7734: Problem in zoom.

Detail: *string*

ERROR 7761: A problem during the execution of computing the mad.

Detail: *string*

ERROR 7762: A problem during the execution of computing the median.

Detail: *string*

ERROR 7763: A problem occurred during the execution of a mad computation.

Detail: *string*

ERROR 7764: A problem occurred during the execution of median computation.

Detail: *string*

ERROR 7769: Kmeans++ exceeded max number of retries for choosing initial center

ERROR 7805: No input columns provided

ERROR 7826: Numeric overflow occurred during execution of kmeans++

ERROR 7827: Numeric overflow occurred when computing total sum of squares

ERROR 7828: Numeric overflow occurred when computing total within-cluster sum of squares

ERROR 7829: Numeric overflow occurred when computing within-cluster sum of squares

ERROR 7844: Input table *string* is empty

ERROR 7845: No input columns left after excluding

ERROR 7851: No rows remain after filtering rows with null and infinity values

ERROR 7949: A problem occurred during the execution

of mode computation.

Detail: *string*

ERROR 7961: Problem in impute.

Detail: *string*

ERROR 7962: Problem in impute: Unknown exception

ERROR 7975: A problem occurred during training of SVM model.

Detail: *string*

ERROR 7988: Input table is empty

ERROR 7992: No rows remain after filtering rows containing NULL, NaN or INF

ERROR 8000: Problem in normalize_fit.

Detail: *string*

Detail: *string*

ERROR 8001: Problem in normalize_fit: Unknown exception

ERROR 8020: Unexpected empty result from query

ERROR 8045: A blob container cannot be initialized more than once

ERROR 8046: A problem during the execution of computing the max weight.

Detail: *string*

ERROR 8047: A problem during the execution of sse_linear_reg

ERROR 8055: A problem occurred during the execution of computing weight table.

Detail: *string*

ERROR 8056: A problem occurred during the execution of saving weight table to vectors.

Detail: *string*

ERROR 8078: Could not link model *string* to catalog object.

Details:*string*

ERROR 8102: Invalid value [*string*] for parameter [*string*]. Valid values are [*string*]

ERROR 8113: Model name cannot be empty

ERROR 8134: Problem in factor function.

Detail: *string*

ERROR 8241: A problem occurred during the execution of

a mode imputation with partition by.

Detail: *string*

ERROR 8242: A problem occurred during the execution of

a mode imputation without partition by.

Detail: *string*

ERROR 8243: A problem occurred during the execution of a mean

imputation with partition by.

Detail: *string*

ERROR 8244: A problem occurred during the execution of a mean

imputation.

Detail: *string*

ERROR 8291: A problem occurred during the execution of coordinate descent covariance.

Detail: *string*

ERROR 8293: A problem occurred during the execution of offsets query (averages of all columns).

Detail: *string*

ERROR 8301: Parameter [*string*] cannot be redefined

ERROR 8306: Problem in creating model [*string*].

Detail: *string*

ERROR 8307: Problem in creating model [*string*]: Unknown exception

ERROR 8308: Problem in cross_validation.

Detail: *string*
ERROR 8309: Problem in cross_validation: Unknown exception
ERROR 8315: Type [*string*] of parameter [*string*] is not supported
ERROR 8318: Empty string cannot be passed into extra_levels parameter
ERROR 8319: Invalid JSON for extra_levels: [*string*]
ERROR 8320: Problem in one_hot_encoder_fit.

Detail: *string*
ERROR 8321: Problem in one_hot_encoder_fit: Unknown exception
ERROR 8344: Parameter [*string*] was provided and cannot be included again in cv_hyperparams
ERROR 8369: Could not train any tree. Increase sampling_size
ERROR 8399: A problem in model summary.

Detail: *string*
ERROR 8405: Could not create catalog object for model [*string*]

Detail: *string*
ERROR 8406: Could not create catalog object for temporary model

Detail: *string*
ERROR 8407: Could not link temporary model to catalog object.

Details:*string*
ERROR 8420: Problem in creating temporary model.

Detail: *string*
ERROR 8421: Problem in creating temporary model: Unknown exception
ERROR 8499: A problem occurred during training of a Naive Bayes model.

Detail: *string*
ERROR 8523: One of the folds does not have enough data to train. Please try to reduce the number of folds
ERROR 8529: Unsupported model format version for upgrade
ERROR 8565: Relation [*string*] has no valid rows
ERROR 8566: Relation [*string*] is empty
ERROR 8572: Error while initializing model upgrade task.

Details:*string*
ERROR 8577: Error while upgrade models.

DETAILS:*string*
ERROR 8590: Schema *value* not found
ERROR 8626: A problem occurred during training of PCA model.

Detail: *string*
ERROR 8627: A problem occurred during training of SVD model.

Detail: *string*
ERROR 8732: Input relation [*string*] is empty
ERROR 9090: Subcluster is already set to default
ERROR 9263: A problem occurred during the execution of a kmeans iteration.

Detail: *string*
ERROR 9267: Could not randomly pick *value* distinct centers after trying *value* times:Number of clusters must not be bigger than the number of distinct rows without null or infinity values
ERROR 9278: Only found less than 2 centers during a bisection. You may want to try again; or there may be fewer than k distinct

data points in the table
ERROR 9279: Problem in writing initial kmeans model to DFS.

Detail: *string*
ERROR 9415: Invalid model category: *string*
ERROR 9416: Invalid model type: *string*
ERROR 9450: Failed to import model(s): *string*
ERROR 9477: Unknown exception in importing model(s)
ERROR 9807: The columns [*string*] is empty
ERROR 9808: The total number of categories in all input columns must be less than *value*, but it is *value*
ERROR 9970: Problem in one_hot_encoder_fit.

Detail: *stringstringstring*
ERROR 10127: Could not train any tree, increase sampling_size
ERROR 10300: A problem occurred during the execution of a Moving Average iteration.

Detail: *string*
ERROR 10435: A problem occurred while deleting temp DFS files.

Detail: *string*

Warning messages

WARNING 2005: *string*
WARNING 7784: Could not remove blob named *string*.

Detail: Unexpected exception
WARNING 7785: Could not remove the blob named *string*.

Detail: *string*
WARNING 7858: Only found *value* non-empty clusters. You may want to try
again or use a better set of initial centers; or there
may be fewer than k distinct datapoints in the table
WARNING 8346: The total number of models to be trained is very large
[*value*]. It could take very long time to finish
WARNING 8507: Could not upgrade model to latest version, during
import. Please run upgrade_model().

DETAIL:*string*
WARNING 8508: Could not upgrade model to latest version, during
import. Please run upgrade_model(). Unknown Exception
WARNING 8571: Cannot upgrade an incomplete model: [*value*]. Skipping
this model
WARNING 8573: Error while performing upgrade for model: *string*.

Details:Model file: *string* does not exist
WARNING 8574: Error while performing upgrade for model[*value*].

Details: Model does not exist
WARNING 8575: Error while performing upgrade for model[*value*].

Details:*string*
WARNING 8576: Error while performing upgrade.

Details:*string*
WARNING 8714: Error while performing upgrade for model: *string*.

Details:Model root directory does not exist
WARNING 9322: Only found 1 cluster during bisection step *value*. You may
want to try again; or there may be fewer than k distinct
data points in the table
WARNING 10413: Kmeans not converged in the kmeans trial *value* of
bisection step *value* (out of *value*) before
kmeans_max_iterations *value*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 0A000

This topic lists the messages associated with the SQLSTATE 0A000.

SQLSTATE 0A000 description

FEATURE_NOT_SUPPORTED

Messages associated with this SQLState

Error messages

ERROR 2009: *string* can not be used in function *string*
ERROR 2013: *string* clause is not supported for expressions
ERROR 2014: *string* Concatenated GZIP/BZIP is not supported with
NATIVE/NATIVE VARCHAR formats
ERROR 2036: *string* is not a legal time unit

ERROR 2058: *string* VIEW is not supported
ERROR 2089: A correlated column in a subquery expression is not supported
ERROR 2114: ADD COLUMN over temporary tables is not supported
ERROR 2130: Aggregate function *string* (*value*) is not supported
ERROR 2133: Aggregate function calls cannot contain subqueries
ERROR 2138: Aggregate functions can only be called on columns of a table
ERROR 2161: ALL subquery with a correlated expression is not supported
ERROR 2165: ALTER COLUMN TYPE over temporary tables is not supported
ERROR 2166: ALTER TABLE does not support ADD COLUMN with other clauses
ERROR 2167: ALTER TABLE does not support ALTER COLUMN TYPE with other clauses
ERROR 2168: ALTER TABLE does not support DROP COLUMN with other clauses
ERROR 2169: ALTER TABLE does not support SET SCHEMA with other clauses
ERROR 2178: An expression containing a correlated subquery with aggregate function is not supported
ERROR 2183: Analytic functions are not allowed in an ORDER BY on a UNION/INTERSECT/EXCEPT
ERROR 2184: Analytic functions are not supported in the ORDER BY of an analytic function OVER clause
ERROR 2190: Analytics query with having clause expression that involves aggregates and subquery is not supported
ERROR 2192: ANALYZE_CONSTRAINTS is currently not supported in non-default locales
ERROR 2208: Another Design/Deployment is in progress
ERROR 2210: ANTI join with segmented inner not supported
ERROR 2220: Argument *string* must not contain subqueries
ERROR 2226: Argument to seeded random_must be a constant
ERROR 2329: Cannot accept a value of type any
ERROR 2330: Cannot accept a value of type anyarray
ERROR 2331: Cannot accept a value of type anyelement
ERROR 2332: Cannot accept a value of type internal
ERROR 2333: Cannot accept a value of type language_handler
ERROR 2334: Cannot accept a value of type opaque
ERROR 2335: Cannot accept a value of type trigger
ERROR 2340: Cannot add IDENTITY/AUTO-INCREMENT columns
ERROR 2350: Cannot alter type of column "*string*" since it is referenced in the constraint "*string*"
ERROR 2351: Cannot alter type of column "*string*" since it is referenced in the default expression of column "*string*"
ERROR 2352: Cannot alter type of column "*string*" since it is referenced in the partition expression
ERROR 2360: Cannot assign to system column "*string*"
ERROR 2363: Cannot broadcast non-subquery outer input to a join
ERROR 2368: Cannot change owner of temporary table
ERROR 2377: Cannot convert column "*string*" from "*string*" to type "*string*"
ERROR 2392: Cannot delete from a view
ERROR 2399: Cannot display a value of type any
ERROR 2400: Cannot display a value of type anyelement
ERROR 2401: Cannot display a value of type internal
ERROR 2402: Cannot display a value of type language_handler
ERROR 2403: Cannot display a value of type opaque
ERROR 2404: Cannot display a value of type trigger
ERROR 2407: Cannot drop a table column when a node is down
ERROR 2411: Cannot drop column "*string*" since it is referenced in the partition expression
ERROR 2412: Cannot drop column "*string*" since it is referenced in the primary key constraint
ERROR 2425: Cannot export virtual *string string*

ERROR 2425: Cannot export virtual *string string*
ERROR 2443: Cannot insert into a view
ERROR 2458: Cannot mergeout uncommitted data in the presence of savepoints
ERROR 2503: Cannot set a subfield to DEFAULT
ERROR 2504: Cannot set an array element to DEFAULT
ERROR 2532: Cannot update a view
ERROR 2533: Cannot Update/Merge tables involved in prejoins with Limit clause on unsorted output
ERROR 2549: Cannot use DISTINCT with user-defined transform functions
ERROR 2552: Cannot use meta function or non-deterministic function in PARTITION BY expression
ERROR 2556: Cannot use SAVEPOINT with uncommitted tuple mover enabled
ERROR 2558: Cannot use subquery in EXECUTE parameter
ERROR 2559: Cannot use subquery in expressions within COPY
ERROR 2560: Cannot use subquery in PARTITION BY expression
ERROR 2561: Cannot use subquery in SEGMENTED BY expression
ERROR 2562: Cannot use Vertica's built-in file source and a UDSource in the same query
ERROR 2569: Catalog object *string* does not exist
ERROR 2602: Clause "NO PROJECTION" conflicts with the column list
ERROR 2603: Clause "NO PROJECTION" is supported only on temporary tables
ERROR 2618: CoerceToDomain is not supported
ERROR 2619: CoerceToDomainValue is not supported
ERROR 2628: Column "*string*" in PARTITION BY expression is not allowed, since it contains NULL values
ERROR 2646: Column *string* has the NOT NULL constraint set and has no default value defined
ERROR 2648: Column *string* in PARTITION BY expression is not allowed, since it is not present in some projections
ERROR 2652: Column *string* occurred multiple times in the definition of Projection *string*
ERROR 2667: Column name list is not allowed in CREATE TABLE / AS EXECUTE
ERROR 2672: Column type int2 is not supported
ERROR 2673: Column type int4 is not supported
ERROR 2676: Command *string* is not supported
ERROR 2679: COMMENT not supported for system objects
ERROR 2680: COMMENT not supported for this object type
ERROR 2692: Conditional UNION/INTERSECT/EXCEPT statements are not implemented
ERROR 2698: Conflicting or redundant column options
ERROR 2699: Conflicting or redundant options
ERROR 2725: Copy cannot return rejected rows from executor nodes
ERROR 2726: Copy cannot return rejected rows from more than one file
ERROR 2739: COPY force not null is available only in CSV mode, but CSV mode is not supported
ERROR 2740: COPY force quote is available only in CSV mode, but CSV mode is not supported
ERROR 2741: COPY FROM does not support the BINARY option
ERROR 2742: COPY FROM does not support the CSV option
ERROR 2743: COPY FROM does not support the OIDS option
ERROR 2744: COPY LOCAL does not support rejected row numbers with exceptions or rejected data options
ERROR 2751: COPY quote is available only in CSV mode, but CSV mode is not supported
ERROR 2770: Correlated EXISTS/NOT EXISTS subquery containing having clause with aggregates is not supported
ERROR 2772: Correlated EXISTS/NOT EXISTS subquery with limit 0 is not supported
ERROR 2773: Correlated EXISTS/NOT EXISTS with aggregate COUNT is not

ERROR 2773: Correlated EXISTS/NOT EXISTS with aggregate COUNT is not supported
ERROR 2776: Correlated EXISTS/NOT EXISTS with User Defined Aggregate is not supported
ERROR 2777: Correlated expression in ON clause is not supported
ERROR 2778: Correlated expression in set operator subquery is not supported
ERROR 2779: Correlated expressions in SELECT list of subquery are not supported
ERROR 2780: Correlated subqueries cannot have more than one level
ERROR 2781: Correlated subqueries with analytics in the select list is not supported
ERROR 2782: Correlated subqueries with no group by and a non-strict expression containing an aggregate in the select list is not supported
ERROR 2783: Correlated subquery column in select/gby/oby not supported
ERROR 2784: Correlated subquery could not be flattened as a join
ERROR 2785: Correlated subquery could not get flattened, a correlated expression could not be treated as a join
ERROR 2786: Correlated subquery expression without aggregates and with limit is not supported
ERROR 2787: Correlated subquery expressions under OR not supported
ERROR 2788: Correlated subquery in expression with operator <> is not supported
ERROR 2790: Correlated subquery with aggregate and limit 0 is not supported
ERROR 2792: Correlated subquery with aggregate function COUNT is not supported
ERROR 2793: Correlated subquery with distinct/group by is not supported
ERROR 2794: Correlated subquery with having clause expression that involves aggregates and subquery is not supported
ERROR 2795: Correlated subquery with NOT IN is not supported
ERROR 2796: Correlated subquery with outer joins and uncorrelated exists is not supported
ERROR 2797: Correlated subquery with User Defined Aggregate is not supported
ERROR 2854: Could not find array type for data type *string*
ERROR 2942: CREATE ASSERTION is not supported
ERROR 2980: Data type not supported
ERROR 2981: Data type not supported (*value*)
ERROR 2983: Database "*string*" does not exist
ERROR 2987: Database references are not supported: "*string.string.string*"
ERROR 3026: Defining query must have a from clause
ERROR 3115: DistinctExpr not supported
ERROR 3116: Distrib overrides are too restrictive. Can not find completed Join Order
ERROR 3118: DML on projection/view is not supported
ERROR 3119: DML query with a predicate that could not be pushed below joins and does not refer solely to the target table is not supported
ERROR 3123: DROP ASSERTION is not supported
ERROR 3126: DROP COLUMN over temporary tables is not supported
ERROR 3132: DROP SEQUENCE does not support CASCADE
ERROR 3141: Dropping local and global objects in one statement is not supported
ERROR 3157: Dynamic load not supported
ERROR 3163: Embedded SQL involving local objects is not supported
ERROR 3174: ENCODED BY is supported in CREATE TABLE ... AS SELECT statement only
ERROR 3246: Error parsing distrib overrides (unexpected end of

override); *string*
ERROR 3247: Error parsing distrib value; *string*
ERROR 3277: Error writing to [*string*]
ERROR 3291: Event ANY_ROW is not supported
ERROR 3317: Executing when OPT:PLAN_ALL_NODES_ACTIVE option is set
ERROR 3343: Explicit JOIN clause contains a join predicate between relations previously joined
ERROR 3351: Expressions in COPY may not contain aggregate functions
ERROR 3352: Expressions in COPY may not contain analytic or Time Series Aggregate Functions
ERROR 3353: Expressions not supported in Times Series Aggregate Function
ERROR 3357: External tables only support files or a User Defined Source
ERROR 3404: FieldStore is not supported
ERROR 3417: Final phase output size mismatch
ERROR 3420: First argument of date_part must be a constant string
ERROR 3434: For INSERT SELECT statement, replicated/broadcasted source data not supported
ERROR 3436: For SELECT DISTINCT, ORDER BY expressions must appear in the SELECT clause
ERROR 3451: Function *string* can't be used as a case expression
ERROR 3452: Function *string* can't be used in a boolean
ERROR 3453: Function *string* can't be used in a WHEN clause
ERROR 3454: Function *string* can't be used in as a segment expression
ERROR 3455: Function *string* can't be used with an operator
ERROR 3488: Group By, Order By, Aggregates, Having & limits not allowed in update/delete
ERROR 3510: IGNORE NULLS argument must be a Boolean constant
ERROR 3553: INHERITS not supported
ERROR 3566: Input of anonymous composite types is not implemented
ERROR 3600: Interpolated predicates can accept arguments of the same type only
ERROR 3601: Interpolated predicates can be part of AND expressions only
ERROR 3613: Interval units "*string*" not supported
ERROR 3821: Joins with an interpolated predicate can have a conjunctive expression containing equality predicates. The equality predicates cannot have expressions or column references with different modifiers
ERROR 3822: Joins with an interpolated predicate cannot have expressions or column references with different modifiers in any of the expressions
ERROR 3857: Library built with unsupported version of Vertica SDK [Version: *string*, Revision: *string*]
ERROR 3876: Locale must be a constant
ERROR 3900: MATCH PARTIAL is not supported
ERROR 3972: Multi-column subquery expressions can only be used with the =, <=> and <> operators
ERROR 3973: Multi-column subquery type ALL can only be used with the = and <=> operators
ERROR 3974: Multi-column subquery type ANY can only be used with the =, <=> and <> operators
ERROR 4106: No single-source bulk loads have been executed in this session
ERROR 4147: Node issuing the query cannot be marked as down
ERROR 4160: Non-equality correlated subquery expression is not supported
ERROR 4170: Not a Star or Snow-Flake Query block
ERROR 4171: Not a Star or Snow-Flake Query block; dimension table not a star or snowflake

ERROR 4172: Not a Star or Snow-Flake Query block; no fact table found
ERROR 4173: Not a Star or Snow-Flake Query block; there are multiple fact tables
ERROR 4197: NULL value found in a column used by a subquery
ERROR 4228: ON COMMIT DROP not supported in CREATE TABLE
ERROR 4238: Only a temporary table projection can be pinned
ERROR 4248: Only inner joins are allowed in the projection defining query
ERROR 4256: Only relations and subqueries are allowed in the FROM clause
ERROR 4258: Only super user can call export_catalog with an output file name
ERROR 4259: Only super user can get the rebalance data script
ERROR 4263: Only superuser can drop system schema
ERROR 4264: Only superuser can rebalance data
ERROR 4265: Only superuser can rebalance data for replicated projections
ERROR 4266: Only superuser can rebalance data for segmented projections
ERROR 4280: Operator *string* (*value*) is not supported
ERROR 4281: Operator *string* is not supported for row expressions
ERROR 4298: ORDER BY on a UNION/INTERSECT/EXCEPT result must be on one of the result columns
ERROR 4299: ORDER mode not supported
ERROR 4306: OUTER join with broadcasted outer data not supported
ERROR 4307: OUTER or SEMI join - done through CROSS join and FILTER - with replicated outer and segmented inner not supported
ERROR 4308: OUTER relation in OUTER join is not the fact table nor a snowflake dimension table
ERROR 4309: Outer replicated/segmented input to a join cannot be resegmented
ERROR 4310: LEFTOUTER/SEMI/ANTI join with replicated/broadcasted outer data not supported
ERROR 4328: PARTITION AUTO can only be used with single-phase user defined transform functions
ERROR 4329: PARTITION AUTO cannot be used with pattern matching
ERROR 4331: PARTITION BY expression cannot return a tuple
ERROR 4332: PARTITION BY expression has an unknown type
ERROR 4333: PARTITION BY expression may not contain aggregate functions
ERROR 4335: Partitioning expression not supported for temporary tables
ERROR 4336: Partitioning not supported for temporary tables
ERROR 4352: Pattern "E" is not supported
ERROR 4375: PINNED clause conflicts with KSAFE setting
ERROR 4376: PINNED clause is not supported in CREATE TABLE statement
ERROR 4412: Prepared statements are currently unsupported
ERROR 4465: Projection *string* of local temporary table cannot be created under user schema *string*
ERROR 4471: Projection choices are too restrictive - cannot create correct join between tables
ERROR 4584: RENAME COLUMN over temporary tables is not supported
ERROR 4586: replicate_catalog has been shut off
ERROR 4628: Row Expressions are not supported in this context
ERROR 4644: Scalar array expression cannot contain column references or subqueries
ERROR 4645: Scalar array op *string* (*value*) is not supported
ERROR 4664: Segmentation clause can not have offset in CREATE TABLE statement
ERROR 4665: Segmentation clause with offset conflicts with KSAFE setting
ERROR 4666: Segmentation expression must have integer type

ERROR 4671: SELECT FOR UPDATE cannot be applied to a function
ERROR 4672: SELECT FOR UPDATE cannot be applied to a join
ERROR 4673: SELECT FOR UPDATE cannot be applied to NEW or OLD
ERROR 4674: SELECT FOR UPDATE is not allowed with EXTERNAL TABLES
ERROR 4675: SELECT FOR UPDATE is not allowed with libraries
ERROR 4676: SELECT FOR UPDATE is not allowed with sequences
ERROR 4677: SELECT FOR UPDATE is not allowed with UNION/INTERSECT/EXCEPT
ERROR 4678: SELECT FOR UPDATE is not allowed with views
ERROR 4680: Self joins in UPDATE statements are not allowed
ERROR 4703: Sequence cannot be moved between system schema and user schema
ERROR 4711: Sequence or IDENTITY/AUTO_INCREMENT column in merge query is not supported
ERROR 4714: Sequences are not allowed in default expressions of local temp tables
ERROR 4715: Sequences cannot be called in views
ERROR 4716: Sequences cannot be created under system schemas
ERROR 4728: Set Operator *string* ALL not supported
ERROR 4730: Set Operator queries without a FROM clause are not supported
ERROR 4733: SET SCHEMA over temporary tables is not supported
ERROR 4747: SetToDefault is not supported
ERROR 4786: Statement *string* is not supported
ERROR 4808: Subqueries are not supported as the left hand argument to another subquery
ERROR 4809: Subqueries are not supported in the ORDER BY of a timeseries OVER clause
ERROR 4810: Subqueries are not supported in the ORDER BY of an analytic function OVER clause
ERROR 4812: Subqueries are not supported in the PARTITION BY of an analytic function OVER clause
ERROR 4816: Subqueries in the ON clause are not supported
ERROR 4817: Subqueries in the SELECT or ORDER BY are not supported if the query has aggregates and the subquery is not part of the GROUP BY
ERROR 4818: Subqueries in the SELECT or ORDER BY are not supported if the subquery is not part of the GROUP BY
ERROR 4820: Subqueries in UPDATE/DELETE/MERGE is not supported
ERROR 4821: Subqueries not allowed in target of insert
ERROR 4822: Subqueries referring to no outer columns in HAVING clause when query has aggregates and no GROUP BY are not supported
ERROR 4824: Subquery aggregate expression that refers a correlated column is not supported
ERROR 4839: Subquery type ARRAY is not supported
ERROR 4842: Subquery without a from clause is not supported
ERROR 4850: Support for UPDATE/DELETE/MERGE is not enabled
ERROR 4854: SyncMarkers are not supported
ERROR 4865: System table *string* cannot be created under user schema *string*
ERROR 4869: System view "*string*" cannot be dropped
ERROR 4870: System view *string* cannot be created under user schema *string*
ERROR 4884: Table *string* cannot be created under system schema *string*
ERROR 4897: Table cannot be moved between system schema and user schema
ERROR 4910: Table revalidation error
ERROR 4918: Temporary Sequences are not supported
ERROR 4933: The argument types in a subquery expression in the where/having clause do not match
ERROR 4938: The constant value following the LIMIT clause cannot be negative

ERROR 4939: The constant value following the OFFSET clause cannot be negative
ERROR 4948: The fourth input argument of TIME_SLICE must be START or END
ERROR 4960: The ORDER BY ... USING clause is not supported
ERROR 4966: The second parameter of export_catalog is invalid: *string*
ERROR 4968: The slice length parameter of TIME_SLICE must be a positive integer
ERROR 5005: Time Series Aggregate Function with interpolation scheme LINEAR may only have an INTEGER or FLOAT type as its first argument
ERROR 5016: Time units "*string*" not supported
ERROR 5023: Timeseries output functions are not supported in the ORDER BY of a timeseries OVER clause
ERROR 5028: Timestamp units "*string*" not supported
ERROR 5110: Type *string* (*value*) is not supported
ERROR 5160: Uncorrelated EXISTS subqueries in HAVING clause when query has aggregates and no GROUP BY are not supported
ERROR 5195: UNIQUE predicate is not supported
ERROR 5262: Unsafe use of string constant with Unicode escapes
ERROR 5264: Unsupported access to session-scoped (LOCAL) object
ERROR 5270: Unsupported COPY command clause
ERROR 5275: Unsupported Join in From clause
ERROR 5276: Unsupported Join in From clause: FULL OUTER JOINS not supported
ERROR 5277: Unsupported Join in From clause: UNION JOINS not supported
ERROR 5278: Unsupported join of two non-alike segmented projections
ERROR 5280: Unsupported mix of Joins
ERROR 5284: Unsupported query syntax
ERROR 5289: Unsupported subquery expression
ERROR 5291: Unsupported use of aggregates
ERROR 5292: Unsupported use of cursors
ERROR 5293: Unsupported use of DISTINCT clause
ERROR 5294: Unsupported use of FROM clause
ERROR 5295: Unsupported use of GROUP BY or DISTINCT clause
ERROR 5296: Unsupported use of HAVING clause
ERROR 5297: Unsupported use of LIMIT/OFFSET clause
ERROR 5298: Unsupported use of ORDER BY clause
ERROR 5299: Unsupported use of outer joins
ERROR 5300: Unsupported use of query/subquery without FROM clause
ERROR 5301: Unsupported use of sub-queries
ERROR 5302: Unsupported use of target relation
ERROR 5303: Unsupported use of UDF in WHERE clause
ERROR 5304: Unsupported use of UNION/INTERSECT/EXCEPT
ERROR 5314: UPDATE may not refer to tables in prejoin projections
ERROR 5366: User defined aggregate cannot be used in query with other distinct aggregates
ERROR 5388: User has insufficient privilege on *string string*
ERROR 5392: User must have the DBDUSER role to run the database designer
ERROR 5396: User projection *string* cannot be created under system schema *string*
ERROR 5402: User-defined transform functions are not supported in the ORDER BY clause
ERROR 5407: VALINDEX column must be the first column in ORDER BY list
ERROR 5426: Vertica currently allows a maximum of *value* physical storage containers per projection
ERROR 5427: Vertica does not support GRANT / REVOKE ON LANGUAGE
ERROR 5428: Vertica does not support GRANT / REVOKE ON TABLESPACE
ERROR 5447: View *string* cannot be created under system schema *string*
ERROR 5456: Volatile functions may not be used in fillers when other

computed columns refer to them
ERROR 5465: Window frame exclusion is not supported
ERROR 5537: Cannot alter user-defined type "*string*" of column "*string*"
ERROR 5550: COPY from UDSource does not support rejected row numbers with exceptions or rejected data options
ERROR 5551: COPY LOCAL cannot process more than ONE NATIVE or NATIVE VARCHAR file at a time
ERROR 5562: Creating temp tables by LIKE clause is not supported
ERROR 5595: Invalid argument type *string* in function *string*
ERROR 5607: Language of replacement library [*string*] must match language of existing library [*string*]
ERROR 5681: Unsupported base type *string* for User-defined type *string*
ERROR 5698: Cannot export statistics for the specified object
ERROR 5725: Size specification not supported for User Defined Type *string*
ERROR 5731: The second parameter must be a table/projection/column name
ERROR 5758: Can not drop Filesystem proc *string*
ERROR 5759: Can not drop library "*string*": referenced by storage locations
ERROR 5763: Can't create a managed external table with non-file sources
ERROR 5781: Cannot use meta function or non-deterministic function in SEGMENTED BY expression
ERROR 5859: Due to the data isolation of temp tables with an on-commit-delete-rows policy, the compute_flextable_keys() and compute_flextable_keys_and_build_view() functions cannot access this table's data
ERROR 5864: Error parsing table (invalid table): *string*
ERROR 5914: HCatalog schema *string* not permitted in search path
ERROR 5990: Projection *string* cannot be created under hcatalog schema *string*
ERROR 5992: Projection cannot be created for HCatalog table *string*
ERROR 6005: Remote table *string*.*string* found in design query
ERROR 6019: Sequence *string* cannot be created under hcatalog schema *string*
ERROR 6020: Sequence *string* cannot be moved between system schema and hcatalog schema *string*
ERROR 6038: Table *string* cannot be created under hcatalog schema *string*
ERROR 6039: Table *string* cannot be moved under hcatalog schema *string*
ERROR 6092: Unsupported access to flex table: No *string* support
ERROR 6108: View *string* cannot be created under hcatalog schema *string*
ERROR 6142: Alter partition not supported for tables with aggregate projections
ERROR 6145: Analytic functions are not allowed in projections
ERROR 6200: Command CREATE INDEX is not supported
ERROR 6233: DISTINCT Aggregates are not allowed in projections
ERROR 6244: EE option DISABLE_AUTOPARTITION may not be used in conjunction with aggregate projections, or projections with expressions
ERROR 6265: Expressions in the projection SELECT list may not be repeated
ERROR 6266: Expressions on aggregates are not allowed in aggregate projections
ERROR 6322: LIMIT may not be used in projection definitions unless the OVER clause is supplied
ERROR 6380: PARTITION BEST cannot be used with pattern matching
ERROR 6382: PARTITION NODES cannot be used with pattern matching
ERROR 6401: Setting the cascade to pool to the built-in pool "*string*" is not supported
ERROR 6427: The LIMIT clause does not support OFFSET ... and OVER() clauses simultaneously
ERROR 6455: Unsupported access to table with projection expressions or aggregates

ERROR 6464: Use of LIMIT with the OVER(...) clause and DISTINCT is not supported within the same SELECT block
ERROR 6465: Use of LIMIT with the OVER(...) clause and ORDER BY is not supported within the same SELECT block
ERROR 6467: User defined aggregate cannot be used in query with MLAs
ERROR 6476: Cannot directly modify index table constraint
ERROR 6477: Cannot modify index table columns directly
ERROR 6501: Access policy cannot be created on table "*string*" since it has an aggregate projection
ERROR 6503: Access policy cannot be created on table "*string*" since it has a projection with expressions
ERROR 6505: Cannot create Aggregate projection on tables with access policy
ERROR 6507: Cannot create projection with expression on tables with access policy
ERROR 6522: Cannot alter column width on variable length columns with BLOCKDICT_COMP encoding
ERROR 6525: Direct query to projection "*string*" are not supported: "*string*"
ERROR 6538: Unable to *string*: "*string*"
ERROR 6644: Arguments to Transform Functions in Live Aggregate Projections cannot be constants
ERROR 6685: Cannot specify anything other than expressions on table columns in the SELECT list
ERROR 6690: Cannot use multi phase UDTs with live aggregate projections
ERROR 6742: drop_location for non-retired DATA locations is not supported
ERROR 6785: For NULLAWARE ANTI JOIN, columns of left-hand-side relation must not appear on the right hand side of ON clause predicates
ERROR 6786: For NULLAWARE ANTI JOIN, columns of right-hand-side relation must appear on the right hand side of ON clause predicates
ERROR 6787: For NULLAWARE ANTI JOIN, columns of right-hand-side relation must not appear on the left hand side of ON clause predicates
ERROR 6817: Input filename cannot be NULL
ERROR 6822: Inputs to batch transform function have to be in the same order as the outputs of the prepass transform function
ERROR 6823: Inputs to batch Transform function may not be expressions
ERROR 6865: Materialized WITH queries are not supported with directed queries
ERROR 6867: Meta-functions cannot be used in directed queries
ERROR 6882: Null value in ON clause is not supported for Nullaware anti join
ERROR 6888: Only equality join is supported in ON clause of Nullaware anti join
ERROR 6891: Order BY clauses are not allowed in aggregate projection definitions
ERROR 6894: Output filename cannot be NULL
ERROR 6974: System/Virtual tables, projections or views not supported for optimizer-generated annotated queries
ERROR 6983: Tables or projections with access policies not supported for optimizer-generated annotated queries
ERROR 6992: The batch and prepass transform functions must be partitioned by the same values
ERROR 6996: The DB admin has disallowed using the MARS feature
ERROR 7054: Unsupported operation in ON clause of Nullaware anti join
ERROR 7103: Java user defined functions are not supported in aggregate projections
ERROR 7113: Queries with syntactic hints are not supported with

directed queries
ERROR 7123: Statement including date/time literals based on current date/time are not supported for directed queries
ERROR 7124: Statement including hints are not supported for directed queries
ERROR 7133: UNI Join with non-subquery inner not supported
ERROR 7169: Constant true/false predicate with ignore constant hint is not supported
ERROR 7188: Object type *value* not supported for access policies
ERROR 7201: The constant value following the LIMIT clause cannot be both zero and ignoreconst
ERROR 7239: Computing flex table keys with non-binary collation locales is not supported
ERROR 7294: *string* expressions must not return a set
ERROR 7307: Cannot use aggregate functions in *string* expressions
ERROR 7308: Cannot use analytic or time series aggregate functions in *string* expressions
ERROR 7318: Default expressions with subqueries are not supported in a MERGE statement
ERROR 7344: *string* expressions may not refer to other columns with *string* expressions
ERROR 7346: *string* queries may not refer to a temporary table
ERROR 7347: *string* queries must refer to tables
ERROR 7368: ALTER NOTIFIER: Unsupported parameter '*string*'
ERROR 7371: Cannot alter a column's *string* when a node is down
ERROR 7372: Cannot assign value to "*string*" column
ERROR 7394: Columns in COPY may not contain virtual columns
ERROR 7403: describeProjection is not supported in fenced UDx
ERROR 7404: describeTable is not supported in fenced UDx
ERROR 7410: Epsilon must be a non-negative float number and smaller than one-million
ERROR 7422: External tables cannot have *string* expressions
ERROR 7430: Initial centers in table '*string*' are not distinct
ERROR 7439: Invalid type '*string*' for column *string*
ERROR 7440: Invalid type '*string*' for column *string* of initial_centers_table
ERROR 7442: listProjections is not supported in fenced UDx
ERROR 7444: listTableProjections is not supported in fenced UDx
ERROR 7446: listTables is not supported in fenced UDx
ERROR 7450: max_iterations must be a positive integer and less than one-million
ERROR 7455: MD5 can't be used in FIPS mode
ERROR 7466: Must specify either a valid initialization method or initial centers table
ERROR 7477: Notifier action is immutable. To set the new action URL you need to drop the notifier, and create a new one
ERROR 7486: num_clusters must be a positive integer
ERROR 7487: Number of clusters must not be bigger than the number of rows without null or infinity values
ERROR 7492: Only 'euclidean' is supported for distance_method
ERROR 7539: Sequences are not allowed in *string* expressions of local temp tables
ERROR 7550: TableSample does not work with update statements
ERROR 7553: Temporary tables cannot have subqueries as *string* expressions
ERROR 7558: The response_column is included in the predictor_columns. You should exclude it to have a meaningful model
ERROR 7559: The selected normalization method is not supported. Only support MinMax, zscore, robust_zscore at present
ERROR 7560: The selected outlier detection method is not supported. Only support robust_zscore based outlier detection at present
ERROR 7606: Number of clusters must not be bigger than the number of

distinct rows without null or infinity values
ERROR 7634: TableSample is not supported with Aggregate projections
ERROR 7646: Cursor [*string*] does not support multiple shards
ERROR 7647: Cursor [*string*] does not support multiple storage containers
ERROR 7648: Cursor [*string*] does not support this operation
ERROR 7649: Cursor [*string*] does not support writes
ERROR 7650: describeBlob is not supported in fenced UDx
ERROR 7652: describeFunction does not support function lookup by name / arguments yet
ERROR 7653: describeFunction is not supported in fenced UDx
ERROR 7654: describeType is not supported in fenced UDx
ERROR 7655: describeType is not supported yet
ERROR 7658: listBlobs is not supported in fenced UDx
ERROR 7659: listBlobs is not supported yet
ERROR 7660: listDerivedTables is not supported in fenced UDx
ERROR 7670: UDx cursors do not currently support deleted data
ERROR 7677: Looking up cursor by table is not yet supported
ERROR 7686: ALTER TABLE does not support ADD CONSTRAINT on text index
ERROR 7687: ALTER TABLE does not support ALTER PARTITION on text index
ERROR 7691: DROP PARTITION is not supported on text index
ERROR 7703: ALTER NOTIFIER: MAXMEMORYSIZE cannot be empty
ERROR 7739: Slices are not supported for VMaps
ERROR 7790: Order BY is not allowed within the OVER() clause of User Defined Transforms in projection definitions
ERROR 7792: Resource pool *string* cannot be modified
ERROR 7818: Unsupported type '*string*' for column '*string*'
ERROR 7840: Column [*string*] is duplicated in exclude_columns
ERROR 7841: Column [*string*] is duplicated in predictor_columns
ERROR 7846: Relation *string* is empty
ERROR 7848: Table '*string*' contains *value* rows, should be *value*
ERROR 7878: Too many columns in MLA grouping sets: *value*, while only up to *value* MLA columns are allowed
ERROR 7889: ALTER MODEL does not support multiple clauses
ERROR 7898: Model *string* cannot be moved under hcatalog schema *string*
ERROR 7900: Model cannot be moved to system schema
ERROR 7926: Catalog object *string* is not either a table or a schema
ERROR 7927: Column "*string*" is referenced in a column set using expression of projection "*string*" but is not stored in the projection
ERROR 7930: COMPLEX JOIN without a boolean marker column not supported
ERROR 7931: COMPLEX JOIN without a complex_join_marker() column at the end of the select list not supported
ERROR 7932: COMPLEX JOIN without a subquery not supported
ERROR 7934: Refresh_columns on multiple tables is only supported for "rebuild" mode
ERROR 7935: Refreshed column cannot be any projection's sort key
ERROR 7937: Refreshed column cannot be referenced in projection's segmentation expression
ERROR 7938: Refreshed column cannot be referenced in table's partition expression
ERROR 7964: The selected missing value imputation method is not supported.

Only support mean and mode based missing value imputation at present
ERROR 7974: *string* is not supported in fenced UDx
ERROR 7976: Add column with default subquery is not supported for unsegmented projections
ERROR 7980: Cannot alter model in an incomplete state
ERROR 8008: Refresh_columns with rebuild mode is not supported for unsegmented projections

ERROR 8013: The selected normalization method is not supported. Valid options are 'minmax', 'zscore', and 'robust_zscore'
ERROR 8023: Unsupported column type [*string*] for column [*string*]
ERROR 8034: Refresh_columns on enforced PK/Unique constraint columns not supported
ERROR 8038: Cannot alter type of column "*string*" since it is referenced in the SET USING expression of column "*string*"
ERROR 8040: Cannot refresh column containing grouped containers
ERROR 8043: Refreshed column cannot be any projection's grouped column
ERROR 8060: Cannot COPY "*string*" from the client using the keyword LOCAL
ERROR 8123: Not enough number of appropriate samples to run the algorithm (excluding samples with NULL, NaN, or Infinity values)
ERROR 8140: Projections on shared storage cannot use explicit node lists
ERROR 8141: Projections on shared storage cannot use explicit range segmentation
ERROR 8155: SET SCHEMA over flex keys tables is not supported
ERROR 8156: SET SCHEMA over text index tables is not supported
ERROR 8180: UDx concurrency request is not supported for empty OVER clause
ERROR 8202: Cannot run refresh_columns or add column with default subquery when any node is recovering
ERROR 8212: Block Dictionary Compression is not supported for data types longer than *value* bytes
ERROR 8213: Block Dictionary Compression is not supported for data types longer than 65000 bytes
ERROR 8357: Cannot use meta function in partition GROUP BY expression
ERROR 8358: Cannot use non-deterministic function in partition GROUP BY expression
ERROR 8362: PARTITION BY expression must appear at least once in partition GROUP BY expression
ERROR 8363: Partition GROUP BY expression can only reference columns by way of inclusion of the PARTITION BY expression
ERROR 8379: num_clusters must be less than or equal to 10000
ERROR 8386: Cannot use subquery in partition GROUP BY expression
ERROR 8388: PARTITION BY expression may not contain window functions
ERROR 8389: Partition GROUP BY expression cannot return a tuple
ERROR 8390: Partition GROUP BY expression has an unknown type
ERROR 8391: Partition GROUP BY expression may not contain aggregate functions
ERROR 8392: Partition GROUP BY expression may not contain window functions
ERROR 8410: HAVING clauses are not allowed in projections
ERROR 8474: Model *string* cannot be created under hcatalog schema *string*
ERROR 8475: Model cannot be created under system schemas
ERROR 8490: Temporary relation is not supported for this function
ERROR 8534: Cannot grant privileges on system table to PSEUDOSUPERUSER role
ERROR 8535: Cannot grant privileges on system table to SYSMONITOR role
ERROR 8546: Invalid privilege on system table. Only SELECT is supported
ERROR 8558: Feature *string* only supported in Eon mode
ERROR 8559: Feature *string* unsupported in Eon mode
ERROR 8638: Drop column not supported on multiphase UDT
ERROR 8681: Range segmented projection is deprecated
ERROR 8691: Explain JSON not supported on copy statements. Re-run the statement without the JSON keyword
ERROR 8739: Uncorrelated EXISTS subqueries are not supported when the query either has HAVING clause subqueries involving aggregates or Multilevel aggregation, and also when the

query has either OUTER JOINS or NOT IN subqueries
ERROR 8765: analyze_correlations is not supported
ERROR 8876: Comparisons with ANY/ALL arrays are unsupported in this context
ERROR 8890: Refresh_columns with partition range is not supported for multiple tables
ERROR 8932: Cannot alter type of column "*string*" since it is referenced by the *string* projection "*string*"
ERROR 8933: Cannot drop column "*string*" since it is referenced by the *string* projection "*string*"
ERROR 8975: Only a single outer SELECT statement is supported
ERROR 9041: Cannot alter column "*string*" type to an inlined complex type
ERROR 9042: Cannot alter inlined complex type of column "*string*"
ERROR 9045: Cannot drop Inlined Complex Types
ERROR 9047: Comparison of incompatible Row Expressions
ERROR 9049: Constraint not supported for complex type column *string*
ERROR 9052: CREATE TYPE for Complex Type is not yet supported
ERROR 9065: Row Expressions in Group By are not supported for queries with Multi-Level Aggregates
ERROR 9076: Add Column is not supported for unsegmented projections when the node used as initiator is not participating to the replica shard
ERROR 9134: Access policy cannot be created on Complex Type column "*string*"
ERROR 9175: UPDATE/DELETE/MERGE a temporary table with aggregate projections is not supported
ERROR 9192: Only optimized MERGE is supported on target table having projection with aggregation or expression
ERROR 9215: Array slice on multiple indexes not supported yet
ERROR 9222: Default COPY parsers do not support *string*
ERROR 9228: PARTITION BY expression cannot return a collection
ERROR 9232: Skip is not yet supported for array types
ERROR 9237: Refreshed column cannot be referenced in any UDT/expression projection
ERROR 9240: UPDATE/DELETE/MERGE a table with UDT projections is not supported
ERROR 9241: A complex type can only have up to depth *value*
ERROR 9264: Bind DN cannot be NULL
ERROR 9265: Bind Password cannot be NULL
ERROR 9268: Filter Group cannot be NULL
ERROR 9269: Filter User cannot be NULL
ERROR 9270: Group Members cannot be NULL
ERROR 9271: Group Name cannot be NULL
ERROR 9273: LDAP Uri cannot be NULL
ERROR 9280: Scope cannot be NULL
ERROR 9281: Search Base cannot be NULL
ERROR 9285: User Name cannot be NULL
ERROR 9302: *string* is currently unsupported for this function
ERROR 9309: System and virtual objects and their dependents are currently unsupported for this function
ERROR 9336: Cannot commit DMLs which trigger refreshing of Live Aggregate Projections when any node is recovering
ERROR 9345: Dry Run Connect Failed!
ERROR 9346: LDAPLink: Could not perform bind for ldapbinddn "*string*" on server "*string*": error code *value*: *string*
ERROR 9360: Cannot add new sort order to a Live Aggregate or Expression projection
ERROR 9366: Projections must select data from only one table
ERROR 9384: Live-aggregate projection is not supported for ADD COLUMN ... PROJECTIONS (...) or ALTER COLUMN ... ENCODING PROJECTIONS (...)

ERROR 9485: Constraint not supported for *string* type column *string*
ERROR 9501: Projection *string* has no direct reference to table column "*string*"
ERROR 9505: A complex type can only have up to depth *value*. Column *string* does not satisfy this limit
ERROR 9545: Webhdfs TRUNCATE operation is not supported in this hdfs version
ERROR 9607: Cannot create subcluster level internal resource pool "*string*"
ERROR 9618: Cannot analyze statistics of a collection column
ERROR 9619: Cannot apply complex alias "*string*" to a scalar target
ERROR 9629: CREATE TABLE AS SELECT with Complex Types in SELECT is not supported
ERROR 9631: Incompatible column types in USING clause
ERROR 9639: Setting parameter '*string*' of internal resource pool at subcluster level is not supported
ERROR 9654: SET Operations with Row Expressions in SELECT are not supported
ERROR 9745: Expressions with multiple SELECT statements cannot be used in *string* query definitions
ERROR 9780: LIMIT in a recursive query is not implemented
ERROR 9783: Multi-value expressions are not supported in this context
ERROR 9784: Mutual recursion between WITH items is not implemented
ERROR 9785: Nested WITH RECURSIVE statements are not supported
ERROR 9786: OFFSET in a recursive query is not implemented
ERROR 9787: ORDER BY in a recursive query is not implemented
ERROR 9802: Complex types of this type are not yet implemented
ERROR 9820: Recursive WITH item "*string*" needs explicitly defined column aliases
ERROR 9824: *string* is not supported for arguments of *string* types
ERROR 9848: Using complex types for filler columns is not yet supported
ERROR 9850: WITH item "*string*" referencing recursive item "*string*" is not supported
ERROR 9910: Complex type *string* output format length [*value*] exceeded limit [*value*]
ERROR 9920: Updating elements of array column "*string*" is not supported
ERROR 9952: Corrupted argument mode; this should be unreachable
ERROR 9953: Corrupted stored procedure language code *value*; this should be unreachable
ERROR 9954: Corrupted stored procedure language; this should be unreachable
ERROR 9957: INOUT parameters are not supported in this language
ERROR 9961: OUT parameters are not supported in this language
ERROR 9973: There are *value* columns to process. This function cannot yet process more than *value* columns
ERROR 9999: Partition range on Live Aggregate Projection is not supported
ERROR 10057: Non-optimized DELETE/UPDATE/MERGE is not supported for partition ranged projection for table partition data type as *string*
ERROR 10060: PARTITION BY expression cannot include user-defined data types
ERROR 10109: Cannot apply complex alias "*string*" to an array target
ERROR 10110: Incompatible Complex Type Case results
ERROR 10117: Corrupted assignment type *value*
ERROR 10118: Corrupted diag option *value*
ERROR 10119: Corrupted diag type *value*
ERROR 10121: Corrupted fetch direction type *value*
ERROR 10122: Corrupted fetch type *value*
ERROR 10123: Corrupted raise level *value*
ERROR 10124: Corrupted raise option *value*

ERROR 10125: Corrupted type kind *value*
ERROR 10139: Multi-dimension arrays are not supported
ERROR 10143: Unsupported use of SET type as a complex type field
ERROR 10157: Bare RAISE must be inside an exception block
ERROR 10163: Cannot TYPE-alias cursor variables
ERROR 10165: CONTEXT option is not yet supported
ERROR 10169: Corrupted control flow type *value*
ERROR 10193: Identifiers can be at most 128 characters long
ERROR 10207: ROW_COUNT option is not yet supported
ERROR 10219: validateAssignableType called on wrong type kind *value*
ERROR 10232: Ordering is not supported on *string* types
ERROR 10245: Corrupted expression type *value*
ERROR 10315: Cannot set *string* for column "*string*" since it is used in projection "*string*"
ERROR 10318: Declaring records is not yet supported
ERROR 10319: Declaring row types is not yet supported
ERROR 10323: Executing FOREACH (ARRAY) is not yet supported
ERROR 10333: ROWTYPE is not yet supported
ERROR 10334: Stored procedures have been disabled by the superuser
ERROR 10335: The RECORD type is not yet supported
ERROR 10337: Type "*string*" is not yet supported for stored procedures
ERROR 10338: Unsupported complex type
ERROR 10355: OUT and INOUT arguments are not yet supported
ERROR 10375: Grouping sets are not supported for multi-argument aggregate functions
ERROR 10384: Only NEXT/FORWARD direction is currently supported for FETCH/MOVE
ERROR 10386: WHERE CURRENT OF (CURSOR) syntax is not supported
ERROR 10408: Cursor arguments ("*string*") may not themselves be cursors
ERROR 10412: FOREACH (ARRAY) is not yet supported
ERROR 10442: Feature *string* only supported in Eon mode with Communal Location
ERROR 10443: GROUP BY does not support function match_columns
ERROR 10446: ORDER BY does not support function match_columns
ERROR 10464: Invalid use of SELECT match_columns() in recursive WITH item "*string*", unable to identify result set columns
ERROR 10471: OR REPLACE syntax cannot be used to replace external procedures
ERROR 10486: Feature *string* is removed in production builds
ERROR 10490: VFS paths cannot be specified by users: *string*
ERROR 10500: Cannot alter column type to a complex type
ERROR 10503: Cannot use complex expressions in topk or UDT projections
ERROR 10506: COPY LOCAL is not yet supported in internal statements
ERROR 10510: EXPORT TO VERTICA does not support exporting complex types
ERROR 10511: Expressions cannot be used for columns of complex types
ERROR 10513: Inserting into indiviudal fields of complex types is not supported
ERROR 10518: Native table must have at least one column whose datatype is either a primitive type or a 1D array of a primitive type
ERROR 10522: PARTITION BY expression cannot return a complex type
ERROR 10527: Projection must have at least one column whose datatype is either a primitive type or a 1D array of a primitive type
ERROR 10530: Unsupported operation on table with complex types
ERROR 10542: Tuning limited to SELECT queries
ERROR 10569: Projections whose anchor tables have complex types are not supported for encoding design
ERROR 10572: Tables with complex types cannot be added as design tables

ERROR 10583: Flex table with complex real columns must have at least
one real column whose datatype is either a primitive type
or a 1D array of a primitive type
ERROR 10604: Default values not supported in multi-row-insert
ERROR 10643: Cannot add a column of complex type to a table with
default columns
ERROR 10644: Cannot drop column "*string*" since it is the last non-complex
column
ERROR 10645: Cannot use multi-row insert: column '*string*' has unsupported
type '*string*'

Warning messages

WARNING 2660: Column column *string* is no longer at position *value* in table *string*
WARNING 2856: Could not find column *string* in table *string*
WARNING 4486: Projections are always created and persisted in the
default Vertica locale. The current locale is *string*
WARNING 5300: Unsupported use of query/subquery without FROM clause
WARNING 6517: Access policy is requested from unsupported object: "*string*"
WARNING 6530: Invalid statistics file. Total number of bounds
specified: '*value*' do not match the buckets value: '*value*'
for column '*string*'. Buckets value specified should exactly
be same as total number of bounds
WARNING 6531: Invalid table name: '*string*'. Make sure the schema/table
exists in database
WARNING 6532: Invalid value '*value*' for attribute '*string*' under bound
'*string*', column '*string*'. Please use a positive value.
WARNING 6533: Invalid value '*value*' for attribute '*string*' under column
'*string*'. Please use a positive value.
WARNING 6862: MARS operation not supported for this client type.
Parameter not changed
WARNING 7288: Transaction isolation level not supported. Parameter not
changed
WARNING 8496: Historical queries are not supported on temporary tables
or projections whose underlying tables are temporary
WARNING 8525: Regularization type [*string*] does not use provided parameter
[*string*], it only affects [*string*]
WARNING 8679: Projection *string* not used in the plan because the
projection is deprecated for node down plans
WARNING 9165: Live Aggregate Projection "*string*" will be created for
temporary table "*string*". Data in "*string*" will be neither
updated nor deleted
WARNING 9239: UDT Projection "*string*" will be created for "*string*". Data in
"*string*" will be neither updated nor deleted
WARNING 9656: Specifying segmentation on specific nodes is deprecated
WARNING 10466: match_columns() function ignores column alias "*string*"
WARNING 10564: Did not add tables *string*
WARNING 10570: Skipped projections *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 0A005

This topic lists the messages associated with the SQLSTATE 0A005.

SQLSTATE 0A005 description

PLAN_TO_SQL_NOT_SUPPORTED

Messages associated with this SQLState

ERROR 6679: Cannot generate annotated query when query contains table that does not have a projection
ERROR 6700: Constant NULLs in NOT-IN clause / Nullaware Anti Join not supported for optimizer-generated annotated queries
ERROR 6739: DML not supported for optimizer-generated annotated queries
ERROR 6770: EXPORT not supported for optimizer-generated annotated queries
ERROR 6866: Meta-function not supported for optimizer-generated annotated queries
ERROR 6889: Optimizer can only generate annotated query for SELECT queries
ERROR 6973: System tables not supported for optimizer-generated annotated queries
ERROR 7114: Query with syntactic hints not supported for optimizer-generated annotated queries
ERROR 7132: Under non-default locale, Multi-level aggregates in set operator sub-queries (except UNION ALL) not supported for optimizer-generated annotated queries
ERROR 7187: Multi-level aggregate queries with more than *value* grouping sets not supported for optimizer-generated annotated queries
ERROR 7449: Match clause not supported for optimizer-generated annotated queries
ERROR 7666: Query with geometry type not supported for optimizer-generated annotated queries
ERROR 7696: Queries with UDX functions with user defined parameter type of *string* is not supported
ERROR 9684: Complex Types not supported for optimizer-generated annotated queries
ERROR 9685: Complex Types outputs are not supported in directed queries

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 0B000

This topic lists the messages associated with the SQLSTATE 0B000.

SQLSTATE 0B000 description

INVALID_TRANSACTION_INITIATION

Messages associated with this SQLState

ERROR 2321: Can't start a Transaction in this context

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 0LV01

This topic lists the messages associated with the SQLSTATE 0LV01.

SQLSTATE 0LV01 description

INVALID_GRANT_OPERATION

Messages associated with this SQLState

Error messages

ERROR 2005: *string*
ERROR 2120: Admin option for a role cannot be granted to *string*"public"
ERROR 2601: Circular assignation of roles is not allowed
ERROR 3484: Grant option for a privilege cannot be granted to "public"
ERROR 3485: Grant option for a privilege cannot be granted to (and thus revoked from) "public"
ERROR 3486: Grant options cannot be granted back to your own grantor
ERROR 3616: Invalid *string* statement
ERROR 3719: Invalid option specified for *string* statement
ERROR 3723: Invalid privilege type "*string*"
ERROR 3724: Invalid privilege type *string* for aggregate function
ERROR 3725: Invalid privilege type *string* for analytic function
ERROR 3726: Invalid privilege type *string* for database
ERROR 3727: Invalid privilege type *string* for function
ERROR 3728: Invalid privilege type *string* for library
ERROR 3729: Invalid privilege type *string* for procedure
ERROR 3730: Invalid privilege type *string* for relation
ERROR 3731: Invalid privilege type *string* for resource pool
ERROR 3732: Invalid privilege type *string* for schema
ERROR 3733: Invalid privilege type *string* for sequence
ERROR 3734: Invalid privilege type *string* for storage location
ERROR 3735: Invalid privilege type *string* for transform
ERROR 5601: Invalid privilege type *string* for filter function
ERROR 5602: Invalid privilege type *string* for parser function
ERROR 5603: Invalid privilege type *string* for source function
ERROR 6680: Cannot GRANT *string* authentication to LDAP role
ERROR 6681: Cannot GRANT *string* authentication to LDAP user
ERROR 6682: Cannot GRANT/REVOKE LDAP role to/from LDAP user or role
ERROR 7163: Cannot materialize schema privileges. Table *string* is not set to include schema privileges
ERROR 7227: Cannot materialize schema privileges. View *string* is not set to include schema privileges
ERROR 7894: Invalid privilege type *string* for model
ERROR 10649: Invalid privilege type *string* for key
ERROR 10650: Invalid privilege type *string* for tlsconfig

Notice messages

NOTICE 2121: Admin option for role "*string*" was already granted to role "*string*"
NOTICE 2122: Admin option for role "*string*" was already granted to user "*string*"
NOTICE 2123: Admin option for role "*string*" was not already granted to *string* "*string*"
NOTICE 4617: Role "*string*" was already granted to role "*string*"
NOTICE 4618: Role "*string*" was already granted to user "*string*"
NOTICE 4619: Role "*string*" was not already granted to *string* "*string*"
NOTICE 6190: Client Authentications "*string*" was already granted to role "*string*"
NOTICE 6191: Client Authentications "*string*" was already granted to user "*string*"
NOTICE 6192: Client Authentications "*string*" was NOT granted to role "*string*"
NOTICE 6193: Client Authentications "*string*" was NOT granted to user "*string*"

Warning messages

WARNING 3745: Invalid role name *string*
WARNING 4613: Role "*string*" cannot be set as default
WARNING 4614: Role "*string*" does not exist
WARNING 9923: LDAPLink: Circular assignation of roles is not allowed

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 20000

This topic lists the messages associated with the SQLSTATE 20000.

SQLSTATE 20000 description

CASE_NOT_FOUND

Messages associated with this SQLState

ERROR 10421: No matching case found

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 21000

This topic lists the messages associated with the SQLSTATE 21000.

SQLSTATE 21000 description

CARDINALITY_VIOLATION

Messages associated with this SQLState

ERROR 10266: Query returned 0 rows where 1 was expected
ERROR 10332: Query returned multiple rows where 1 was expected

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22000

This topic lists the messages associated with the SQLSTATE 22000.

SQLSTATE 22000 description

DATA_EXCEPTION

Messages associated with this SQLState

ERROR 3646: Invalid Datum pointer
ERROR 4163: Non-positive value supplied to randomint: *value*
ERROR 4921: Test Error @*string*
ERROR 4922: Test Error from @*string*
ERROR 7474: Non-positive value supplied to randomint_crypto: *value*
ERROR 8489: Query failed because of an empty result

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22001

This topic lists the messages associated with the SQLSTATE 22001.

SQLSTATE 22001 description

STRING_DATA_RIGHT_TRUNCATION

Messages associated with this SQLState

ERROR 2991: Date '*string*'*string* too long for type *string*(*value*)
ERROR 3426: Float '*string*'*string* too long for type *string*
ERROR 3589: Integer '*string*'*string* is too long for type *string*(*value*)
ERROR 3605: Interval '*string*'*string* too long for type *string*(*value*)
ERROR 4208: Numeric '*string*' is too long for type *string*
ERROR 4315: Padded octet length (*value*) exceeds the *value* octet limit
ERROR 4604: Result (*value* characters) exceeds the field width (*value*)
ERROR 4800: String of *value* octets is too long for type *string*(*value*)
ERROR 5004: Time '*string*'*string* too long for type *string*(*value*)
ERROR 5024: Timestamp '*string*'*string* too long for type *string*(*value*)
ERROR 5032: Timestamptz '*string*'*string* too long for type *string*(*value*)
ERROR 5035: Timetz '*string*'*string* too long for type *string*(*value*)
ERROR 5417: Value too long for type character varying(*value*)
ERROR 5418: Value too long for type character(*value*)
ERROR 7401: Date '*string*' too long for buffer of length (*value*)
ERROR 7431: Interval '*string*' too long for buffer of length (*value*)
ERROR 7568: Time '*string*' too long for buffer of length (*value*)
ERROR 7569: Timestamp '*string*' too long for buffer of length (*value*)
ERROR 7570: TimestampTz '*string*' too long for buffer of length (*value*)
ERROR 7571: TimeTz '*string*' too long for buffer of length (*value*)
ERROR 8276: Type *string*(*value*) is too short to hold UUID values, need at least *value*
ERROR 8682: String of *value* octets is too long for type *string*(*value*) for column *string*
ERROR 9227: Output array isn't big enough

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22003

This topic lists the messages associated with the SQLSTATE 22003.

SQLSTATE 22003 description

NUMERIC_VALUE_OUT_OF_RANGE

Messages associated with this SQLState

ERROR 2429: Cannot find matching query in the system
ERROR 2828: Could not convert '*string*'*string* to an int8
ERROR 3425: Float "*value*" is out of range for type *string*
ERROR 3675: Invalid input for *string*, exceeds 32 bits: "*string*"
ERROR 3676: Invalid input for *string*, exceeds 64 bits: "*string*"
ERROR 3786: Invalid value for float: "*string*"
ERROR 4200: Number of buckets must be a positive integer
ERROR 4361: Percentile value must be a number between 0 and 1
ERROR 4704: Sequence exceeded max value
ERROR 4705: Sequence exceeded min value
ERROR 4756: Smoothing factor must between 0 and 1
ERROR 4795: String "*string*"*string* is out of range as a float8
ERROR 4796: String "*string*"*string* is out of range as an int8
ERROR 4845: Sum() overflowed
ERROR 5408: Value "*string*" is out of range for type *string*
ERROR 5409: Value "*string*" is out of range for type int8
ERROR 5411: Value exceeds range of type *string*
ERROR 5412: Value is too long for type *string*: "*string*"
ERROR 6063: Total number of significant digits for value *string* is more
than what is defined. Buffer size is *value* while actual
length of word is *value* instead
ERROR 7623: Value "*string*" is out of range for type int64
ERROR 7697: Arithmetic overflow accumulating numeric, operand *value*
ERROR 7698: Arithmetic overflow adding numerics, operands *value*, *value*
ERROR 7699: Arithmetic overflow subtracting numerics, operands *value*,
value
ERROR 7986: Evaluation of expression to be inserted exceeded range of
type numeric(*value*,*value*)
ERROR 10537: Requested join-id out of range

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22004

This topic lists the messages associated with the SQLSTATE 22004.

SQLSTATE 22004 description

NULL_VALUE_NOT_ALLOWED

Messages associated with this SQLState

Error messages

ERROR 2110: ACL arrays must not contain null values
ERROR 2501: Cannot set a NOT NULL column (*string*) to a NULL value in *string* statement
ERROR 2502: Cannot set a NOT NULL column (*string*) to a NULL value in INSERT/UPDATE statement
ERROR 2514: Cannot set NOT NULL columns (*string*) to a NULL value in INSERT/UPDATE statement
ERROR 4195: NULL value detected in data partitioning expression
ERROR 4340: Partitioning on NULL
ERROR 8361: NULL value detected in partition group by expression
ERROR 10533: 'query' is a required argument to action 'start_session'
ERROR 10538: Tuning action cannot be NULL
ERROR 10539: Tuning args cannot be NULL
ERROR 10541: Tuning join-id is a required argument to the flip_join_order action
ERROR 10543: Tuning query is required to start a tuning session
ERROR 10546: Tuning session name cannot be NULL
ERROR 10549: Tuning session-name is a required argument

Warning messages

WARNING 9249: Using PARTITION expression that may result in NULL values

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22007

This topic lists the messages associated with the SQLSTATE 22007.

SQLSTATE 22007 description

INVALID_DATETIME_FORMAT

Messages associated with this SQLState

ERROR 2171: AM/PM hour (*value*) must be between 1 and 12
ERROR 2364: Cannot calculate day of year without year information
ERROR 3439: Format *string* is invalid for an Interval value
ERROR 3535: Inconsistent use of year *value* and "BC"
ERROR 3647: Invalid day-of-week '*string*'
ERROR 3679: Invalid input syntax for *string*: "*string*"
ERROR 3721: Invalid partition key
ERROR 3785: Invalid value for *string*: "*string*"

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22008

This topic lists the messages associated with the SQLSTATE 22008.

SQLSTATE 22008 description

DATETIME_FIELD_OVERFLOW

Messages associated with this SQLState

ERROR 2992: Date/time field value out of range: "*string*"
ERROR 4065: next_day(infinity, DOW) is not defined

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22009

This topic lists the messages associated with the SQLSTATE 22009.

SQLSTATE 22009 description

INVALID_TIME_ZONE_DISPLACEMENT_VALUE

Messages associated with this SQLState

ERROR 3768: Invalid timezone interval displacement ERROR 5044: Timezone displacement out of range: "*string*"

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 2200B

This topic lists the messages associated with the SQLSTATE 2200B.

SQLSTATE 2200B description

ESCAPE_CHARACTER_CONFLICT

Messages associated with this SQLState

ERROR 2699: Conflicting or redundant options

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 2200D

This topic lists the messages associated with the SQLSTATE 2200D.

SQLSTATE 2200D description

INVALID_ESCAPE_OCTET

Messages associated with this SQLState

ERROR 3285: ESCAPE strings must be a single octet, not "*string*"

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22011

This topic lists the messages associated with the SQLSTATE 22011.

SQLSTATE 22011 description

SUBSTRING_ERROR

Messages associated with this SQLState

ERROR 4034: Negative count not allowed
ERROR 4035: Negative length not allowed
ERROR 4036: Negative or zero substring start position not allowed
ERROR 4039: Negative substring length not allowed
ERROR 4222: Occurrence number must be > 0
ERROR 4784: Start position cannot be 0

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22012

This topic lists the messages associated with the SQLSTATE 22012.

SQLSTATE 22012 description

DIVISION_BY_ZERO

Messages associated with this SQLState

ERROR 3117: Division by zero

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22015

This topic lists the messages associated with the SQLSTATE 22015.

SQLSTATE 22015 description

INTERVAL_FIELD_OVERFLOW

Messages associated with this SQLState

ERROR 3606: Interval field value out of range: "*string*"

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22019

This topic lists the messages associated with the SQLSTATE 22019.

SQLSTATE 22019 description

INVALID_ESCAPE_CHARACTER

Messages associated with this SQLState

Error messages

ERROR 2729: COPY DELIMITER for column *string* must be a single character
ERROR 2730: COPY delimiter must be a single character
ERROR 2731: COPY ENCLOSED BY cannot be a whitespace character
ERROR 2732: COPY ENCLOSED BY for column *string* cannot be a whitespace character
ERROR 2733: COPY ENCLOSED BY for column *string* must be a single character
ERROR 2734: COPY ENCLOSED BY must be a single character
ERROR 2736: COPY ESCAPE AS for column *string* must be a single character
ERROR 2737: COPY ESCAPE must be a single character
ERROR 2758: COPY TRIM for column *string* must be an empty string or a single character
ERROR 2759: COPY trim must be an empty string or a single character
ERROR 3284: ESCAPE strings must be a single character, not "*string*"
ERROR 10607: ESCAPE strings must be a single character, not "*string*"

Warning messages

WARNING 3284: ESCAPE strings must be a single character, not "*string*"

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 2201B

This topic lists the messages associated with the SQLSTATE 2201B.

SQLSTATE 2201B description

INVALID_REGULAR_EXPRESSION

Messages associated with this SQLState

ERROR 3742: Invalid regexp match_param: '*character*'
ERROR 4552: Regexp match or recursion limit exceeded (rc *value*)
ERROR 4553: Regexp pattern error at offset *value*: *string*
ERROR 4554: Regexp pattern study error: *string*
ERROR 5064: Too many regular expression subexpressions

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 2201G

This topic lists the messages associated with the SQLSTATE 2201G.

SQLSTATE 2201G description

INVALID_ARGUMENT_FOR_WIDTH_BUCKET_FUNCTION

Messages associated with this SQLState

ERROR 2939: Count must be greater than zero
ERROR 3888: Lower and upper bounds must be finite
ERROR 3889: Lower bound cannot equal upper bound
ERROR 4277: Operand, lower bound and upper bound cannot be NaN

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22021

This topic lists the messages associated with the SQLSTATE 22021.

SQLSTATE 22021 description

CHARACTER_NOT_IN_REPERTOIRE

Messages associated with this SQLState

ERROR 4551: Regexp encountered an invalid UTF-8 character

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22023

This topic lists the messages associated with the SQLSTATE 22023.

SQLSTATE 22023 description

INVALID_PARAMETER_VALUE

Messages associated with this SQLState

Error messages

ERROR 2008: *string* can not be set to a negative number
ERROR 2028: *string* exceptions and rejected_data can not be the same filename
ERROR 2042: *string* must be a positive integer
ERROR 2048: *string* Path [*string*] is a directory
ERROR 2049: *string* Path [*string*] is a socket
ERROR 2056: *string* Unrecognized format '*string*' for column *value*
ERROR 2071: '*string*' is not a valid size description
ERROR 2077: [*string*] cannot be dropped. There will be no storage locations for data files
ERROR 2078: [*string*] cannot be dropped. There will be no storage locations for temporary files
ERROR 2079: [*string*] cannot be retired. There will be no storage locations for data files
ERROR 2080: [*string*] cannot be retired. There will be no storage locations for temporary files
ERROR 2081: [*string*] is not a valid storage location on node *string*
ERROR 2108: ACL array contains wrong data type
ERROR 2109: ACL arrays must be one-dimensional
ERROR 2158: All columns of soft unique key statistics must be from the same table
ERROR 2196: analyze_statistics: Cannot analyze statistics of a virtual table *string*
ERROR 2197: analyze_statistics: invalid accuracy *value*. A number between 0 and 100 is required
ERROR 2254: Bad snapshot name '*string*' (cannot contain / or start with a .)
ERROR 2298: Can not lock/unlock super user account
ERROR 2300: Can not reuse any recent passwords
ERROR 2301: Can not reuse current password
ERROR 2302: Can not reuse the previous *value* passwords
ERROR 2317: Can't purge projection(s); AHM is at epoch 0
ERROR 2319: Can't set a REJECTED file on node '*string*', which the current query is not executing on
ERROR 2320: Can't set an EXCEPTIONS file on node '*string*', which the current query is not executing on
ERROR 2365: Cannot calculate week number without year information
ERROR 2370: Cannot close a protected session
ERROR 2414: Cannot drop extended statistics on a projection (*string*)

ERROR 2452: Cannot load data from node *string* as it is down
ERROR 2457: Cannot merge partitions in multiple tables at the same time
ERROR 2468: Cannot partition by value multiple tables at the same time
ERROR 2478: Cannot release savepoint; no transaction in progress
ERROR 2500: Cannot set *string* maxConcurrency to unlimited
ERROR 2509: Cannot set maxMemorySize of *string* pool to *string* [*value* KB], as it is above 75% [75% = *value* KB]
ERROR 2510: Cannot set maxMemorySize of *string* pool to none, as this could prevent moveout from running
ERROR 2511: Cannot set maxMemorySize of recovery pool to *string* [*value* KB], as it is below 25% [*value* KB]
ERROR 2513: Cannot set memorySize of general pool
ERROR 2523: Cannot specify exceptions or rejected-data files ON ANY NODE
ERROR 2540: Cannot use 0 for a key, used internally
ERROR 2548: Cannot use both COPY LOCAL and ON ANY NODE: LOCAL files are stored on the client, not on any Vertica node
ERROR 2621: Collection type must be specified
ERROR 2624: Column "*string*" does not exist
ERROR 2653: Column *string* of projection *string* has ACCESSRANK < 0
ERROR 2695: Conflicting "datestyle" keywords
ERROR 2720: Conversion to timezone "*string*" failed
ERROR 2722: COPY .. LOCAL cannot store *string* on a Vertica node
ERROR 2723: COPY ... LOCAL can read files from the client only
ERROR 2724: COPY ... LOCAL can read files with same compression only
ERROR 2727: COPY column option *string* not supported with format *string*
ERROR 2728: COPY delimiter *value* not appear in the NULL specification
ERROR 2735: COPY ENCLOSING CHARACTER *value* not appear in the NULL specification
ERROR 2748: COPY NULL must be an empty string or a single character for FIXED WIDTH data
ERROR 2749: COPY option *string* not supported
ERROR 2750: COPY option *string* not supported with format *string*
ERROR 2752: COPY RECORD TERMINATOR must be at least ONE character long
ERROR 2753: COPY REJECTMAX should be >= 0
ERROR 2756: COPY skip characters should be >= 0
ERROR 2757: COPY skip should be >= 0
ERROR 2760: COPY WITH PARSER Error (column *value*): Parser specified a column of type [*string*]; table needs [*string*]
ERROR 2761: COPY WITH PARSER Error: Parser specified *value* column(s); table needs *value* column(s)
ERROR 2765: COPY: width and length of null string does not match for column *string*
ERROR 2766: COPY: width for column *string* has to be greater than 0
ERROR 2830: Could not convert to timezone "*string*"
ERROR 2932: Couldn't find the specified task
ERROR 2950: Current design does not meet the requirements for K = *value*.

Current design is valid for K *string value*.

string
ERROR 2963: CURRENT_TIME(*value*) precision must not be negative
ERROR 2965: CURRENT_TIMESTAMP(*value*) precision must not be negative
ERROR 2983: Database "*string*" does not exist
ERROR 2993: Datepart "*string*" not recognized
ERROR 2994: Datepart is invalid
ERROR 3006: DDL statement interfered with snapshot; an object no longer exists
ERROR 3007: DDL statement interfered with this statement
ERROR 3012: DECIMAL precision *value* must be between 1 and *value*

ERROR 3012: DECIMAL precision *value* must be between 1 and *value*
ERROR 3013: DECIMAL scale *value* must be between 0 and precision *value*
ERROR 3032: Delimiter and record terminator cannot be the same value
ERROR 3033: Delimiter and record terminator for *string* cannot be the same value
ERROR 3138: drop_statistics: Can not drop statistics for a virtual table/projection *string*
ERROR 3168: ENCLOSED BY and delimiter *value* not be the same value
ERROR 3170: ENCLOSED BY and record terminator *value* not be the same value
ERROR 3178: ENFORCELENGTH cannot be specified for *string*
ERROR 3280: ESCAPE AS and delimiter *value* not be the same value
ERROR 3281: ESCAPE AS and NULL specification *value* not be the same value
ERROR 3282: ESCAPE AS and record terminator *value* not be the same value
ERROR 3383: Failed to parse object name string
ERROR 3423: Fixed width record size (*value*) is too large. Record size has to be lesser than *value* (0x*value*)
ERROR 3424: Fixed width record size is too large. Record size has to be lesser than *value* (0x*value*)
ERROR 3440: Format cannot be specified for *string*
ERROR 3503: ICU *string* error: '*string*'
ERROR 3505: ICU does not support locale '*string*'
ERROR 3513: Illegal argument to change_runtime_priority: NULL
ERROR 3514: Illegal argument to set_config_parameter: NULL
ERROR 3524: In the SAMPLE STORAGE n or SAMPLE STORAGE n,b clause, n must be a constant greater than or equal to 0
ERROR 3525: In the SAMPLE STORAGE n PERCENT or SAMPLE STORAGE n PERCENT,b clause, n must be a constant greater than or equal to 0 and less than or equal to 100
ERROR 3526: In the SAMPLE STORAGE n PERCENT,b clause, n must be a constant greater than or equal to 0 and less than or equal to 100, while b must be a constant greater than or equal to 0
ERROR 3527: In the SAMPLE STORAGE n,b clause, both n and b must be constants greater than or equal to 0
ERROR 3528: In the SAMPLE STORAGE n,b or SAMPLE STORAGE n PERCENT,b clause, b must be a constant greater than or equal to 0
ERROR 3612: Interval units "*string*" not recognized
ERROR 3632: Invalid collection type *string* specified
ERROR 3652: Invalid Directives type: *string*
ERROR 3686: Invalid interval value for timezone
ERROR 3688: Invalid K value: *value*. K cannot be less than zero
ERROR 3689: Invalid K value: *value*. Maximum K value for *value* nodes is: *value*
ERROR 3692: Invalid limit type (*string*): must be HIGH or LOW
ERROR 3695: Invalid list syntax for "datestyle"
ERROR 3707: Invalid node: [*string*]
ERROR 3720: Invalid paramid: *value*
ERROR 3721: Invalid partition key
ERROR 3741: Invalid range
ERROR 3743: Invalid resource type (*string*)
ERROR 3746: Invalid runtime priority string
ERROR 3750: Invalid service name for '*string*'
ERROR 3772: Invalid typmod
ERROR 3777: Invalid Usage type: *string*
ERROR 3780: Invalid user/role name "*string*"
ERROR 3783: Invalid value *string=string*
ERROR 3787: Invalid value for parameter
ERROR 3788: Invalid value for parameter *string*: *string*
ERROR 3789: Invalid value for search path: "*string*"

ERROR 3730: Invalid value for search path: *string*
ERROR 3840: Keyword '*string*' (*string*=*string*) is not supported
ERROR 3845: Latency should be > 0
ERROR 3852: Length for type *string* cannot exceed *value*
ERROR 3853: Length for type *string* must be at least 1
ERROR 3877: LOCALTIME(*value*) precision must not be negative
ERROR 3879: LOCALTIMESTAMP(*value*) precision must not be negative
ERROR 3912: maxMemorySize of *string* [*value* KB] is not in bounds [max is *value* KB]
ERROR 3920: memoryCap of *string* (*value* KB) would exceed [*value* KB]
ERROR 3922: memorySize *string* [*value* KB] would exceed maxMemorySize *string* [*value* KB]
ERROR 3923: memorySize of *string* [*value* KB] would exceed [*value* KB]
ERROR 3967: More than one *string* specified for a node
ERROR 4027: Must supply a CATALOGPATH
ERROR 4028: Must supply a HOSTNAME
ERROR 4037: Negative run time cap is not allowed
ERROR 4038: Negative runTimeCap is not allowed
ERROR 4089: No objects specified
ERROR 4175: Not allowed to close session
ERROR 4186: NULL is an invalid K value
ERROR 4187: NULL is invalid object name for analyze_extended_statistics
ERROR 4190: NULL is invalid scope type for analyze_extended_statistics
ERROR 4191: NULL is invalid statistics type for analyze_extended_statistics
ERROR 4194: NULL string and record terminator *value* not be the same value
ERROR 4211: NUMERIC precision *value* must be between 1 and *value*
ERROR 4212: NUMERIC scale *value* must be between 0 and precision *value*
ERROR 4222: Occurrence number must be > 0
ERROR 4250: Only ONE exception file should be specified for a LOCAL copy
ERROR 4252: Only ONE rejected data file should be specified for a LOCAL copy
ERROR 4318: Parameter *string* in default profile can not be set to DEFAULT
ERROR 4319: Parameter *string* may not exceed 9999
ERROR 4330: PARTITION BY clause must contain table columns in a valid expression
ERROR 4334: Partition key too long
ERROR 4347: Path cannot be an empty string
ERROR 4406: Precision for type float must be at least 1 bit
ERROR 4407: Precision for type float must be less than 54 bits
ERROR 4408: Precision must be less than *value*; result would be numeric(*value*,*value*)
ERROR 4454: Projection *string* cannot be analyzed, because it is not up to date
ERROR 4456: Projection *string* cannot drop statistics, because it is not up to date
ERROR 4529: Rebalance skew percent must be in the range [0,100]
ERROR 4556: Regexp starting position must be greater than zero
ERROR 4595: Resource pool "*string*" is an internal pool and cannot be dropped
ERROR 4606: Retention settings must be less than 2TB
ERROR 4639: Run time cap cannot exceed 1 year
ERROR 4642: runTimeCap cannot exceed 1 year
ERROR 4647: Scaling factor must be greater than zero
ERROR 4648: Scaling factor must be less than 33
ERROR 4653: Schema *string* is virtual
ERROR 4701: Sequence *string* is already owned by *string*
ERROR 4702: SEQUENCE CACHE should be greater than 0

ERROR 4708: SEQUENCE MAXVALUE is too large and will overflow
ERROR 4709: SEQUENCE MINVALUE is too small and will underflow
ERROR 4710: SEQUENCE MINVALUE should be lesser than MAXVALUE
ERROR 4712: SEQUENCE START WITH should be between MINVALUE and MAXVALUE
ERROR 4723: SET *string* takes only one argument
ERROR 4766: Specified too few widths for the given number of columns
ERROR 4770: Specify at least one table-column for soft unique key statistics
ERROR 4800: String of *value* octets is too long for type *string*(*value*)
ERROR 4802: STROKE collations are not supported
ERROR 4807: Subnet mask is empty
ERROR 4862: System pool priority must be between -110 and 110 inclusive
ERROR 4885: Table *string* does not exist
ERROR 4892: Table *string* is not partitioned
ERROR 4893: Table *string* is session scoped
ERROR 4894: Table *string* is virtual
ERROR 4923: That password is not acceptable
ERROR 4937: The confidence level must be between 0 and 100 inclusive.

string
ERROR 4961: The permissible error must between 0 and 100 inclusive.

string
ERROR 5002: Throughput should be > 0
ERROR 5014: Time units "*string*" not recognized
ERROR 5015: Time units "*string*" not recognized
ERROR 5019: TIME(*value*)*string* precision must not be negative
ERROR 5026: Timestamp units "*string*" not recognized
ERROR 5027: Timestamp units "*string*" not recognized
ERROR 5030: TIMESTAMP(*value*)*string* precision must not be negative
ERROR 5034: TIMESTAMPTZ(*value*) precision must not be negative
ERROR 5036: TIMETZ(*value*) precision must not be negative
ERROR 5038: Timezone "*string*" not recognized
ERROR 5039: Timezone "*string*" uses leap seconds
ERROR 5048: Timezone value "*string*" is more than *value* hours
ERROR 5067: Total data collector memory retention of *value* is too large given system memory size
ERROR 5106: TuningRecommendations data collection is disabled
ERROR 5118: UDL specified no execution nodes; at least one execution node must be specified
ERROR 5198: Unknown authentication method: "*string*"
ERROR 5209: Unknown node: *string*
ERROR 5211: Unknown or unsupported object: *string*
ERROR 5215: Unknown value *string*=*string*
ERROR 5220: Unrecognized "datestyle" keyword: "*string*"
ERROR 5229: Unrecognized format '*string*'
ERROR 5248: Unrecognized privilege type: "*string*"
ERROR 5258: Unrecognized timezone name: "*string*"
ERROR 5271: Unsupported format code: *value*
ERROR 5316: Usage cannot be an empty string
ERROR 5317: Usage of [*string*] cannot be changed from *string* to *string*
ERROR 5319: Usage of [*string*] cannot be changed to *string*. There will be no storage locations for data files
ERROR 5320: Usage of [*string*] cannot be changed to *string*. There will be no storage locations for temporary files
ERROR 5360: User "*string*" does not exist
ERROR 5393: User pool priority must be between -100 and 100 inclusive
ERROR 5437: Vertica should not be run with less than 1GB of RAM

ERROR 5520: *string* compresses network traffic. *string* does NOT compress network traffic. Please change the configuration to be consistent

ERROR 5521: *string* does NOT compress network traffic. *string* compresses network traffic. Please change the configuration to be consistent

ERROR 5538: Cannot COPY user-defined types directly. Please compute them using copy expressions

ERROR 5542: Cannot INSERT or COPY user-defined types directly. Please compute them using appropriate user-defined functions

ERROR 5545: Cluster layout must include all non-ephemeral nodes and should also not include any ephemeral nodes

ERROR 5549: Conversion from *string* to DataType *string* failed. Invalid value

ERROR 5571: Empty storage tier label is not allowed

ERROR 5576: Every permanent node should only be listed once

ERROR 5598: Invalid or unavailable type 'LONG VARBINARY'

ERROR 5599: Invalid or unavailable type 'LONG VARCHAR'

ERROR 5613: Length for type *string* must be between 1 and *value*

ERROR 5631: Object *string* does not exist or is not of supported type

ERROR 5632: Object *string* is not a table

ERROR 5634: Path [*string*] is a directory

ERROR 5644: Projection basename "*string*" is not a prefix of projection name "*string*"

ERROR 5647: Provided Node "*string*" does not exist

ERROR 5648: Provided Node "*string*" is not permanent

ERROR 5668: Target table name can not be empty

ERROR 5681: Unsupported base type *string* for User-defined type *string*

ERROR 5685: User Defined Filter expected but found *string*

ERROR 5686: User Defined Parser expected but found *string*

ERROR 5687: User Defined Source expected but found *string*

ERROR 5703: Couldn't find the specified task, or the Resource Manager has not recieved the request

ERROR 5728: Specified too many widths (*value*) for the given number of columns (*value*)

ERROR 5740: '*string*' is not a valid value for database option *string*

ERROR 5746: analyze_statistics: invalid number of buckets *value*. A number > 0 is required

ERROR 5750: Attempt to configure CPU affinity mode conflicts with configuration of resource pool '*string*'

ERROR 5751: Attempt to configure CPU affinity set to '*string*' conflicts with configuration of resource pool '*string*'

ERROR 5752: Attempt to configure CPU affinity set to '*string*' in exclusive mode would not leave any CPUs available for system queries

ERROR 5753: Attempt to configure CPU affinity to exclusive mode would not leave any CPUs available for system queries

ERROR 5762: Can only specify user defined file system for DATA and/or TEMP storage locations

ERROR 5768: Cannot do RETURNREJECTED and REJECTED DATA AS TABLE in the same query; rejected records can only be saved to one location

ERROR 5778: Cannot specify both a rejected file and a rejected table in the same statement

ERROR 5779: Cannot specify both an exceptions file and a rejected table in the same statement

ERROR 5804: CPU #*value* is not available to this server, because of server-level processor pinning

ERROR 5918: Improperly formatted broadcast address [*string*]

ERROR 5925: Interface IP address (family *string*) "*string*" is invalid

ERROR 5933: Invalid state for UDFilter: REJECT

ERROR 5934: Invalid state for UDSource: INPUT_NEEDED

ERROR 5935: Invalid state for UDSource: REJECT

ERROR 5935: Invalid state for UDSource: REJECT
ERROR 5954: memoryCap of *string* [*value* KB] would exceed [*value* KB]
ERROR 5962: Must request a positive key count to materialize: *value*
ERROR 5976: Object already exists: *string*. Can't create a rejections table with the same name
ERROR 6006: Request for *value* percent of *value* CPUs rounds to zero CPUs
ERROR 6007: Request for reservation of CPU #*value* conflicts with another pool's reservation
ERROR 6010: Resource pool '*string*' not found
ERROR 6044: Table already exists: *string*. Can't create a rejections table with the same name
ERROR 6045: The CPU affinity mode cannot be SHARED or EXCLUSIVE if the affinity set is empty
ERROR 6073: Unable to allocate *value* CPUs for resource pool in *string* affinity mode
ERROR 6088: Unknown control mode *string*
ERROR 6090: Unknown database option '*string*'
ERROR 6097: User-Defined Load function indicated that it consumed *value* bytes, when only *value* were available
ERROR 6123: A Resource Pool cannot cascade to itself
ERROR 6126: Access policy cannot be altered on unknown table "*string*"
ERROR 6127: Access policy cannot be copied from unknown object "*string*"
ERROR 6128: Access policy cannot be copied to unknown object "*string*"
ERROR 6129: Access policy cannot be created on temporary table "*string*"
ERROR 6130: Access policy cannot be created on unknown table "*string*"
ERROR 6131: Access policy cannot be created with empty expression
ERROR 6132: Access policy cannot be dropped on unknown object "*string*"
ERROR 6134: Access policy for COLUMN "*string*" already exists on "*string*"
ERROR 6135: Access policy for ROWS already exists on "*string*"
ERROR 6150: Can not alter access policy on table "*string*" for COLUMN "*string*": it doesn't exist
ERROR 6151: Can not alter access policy on table "*string*" for ROWS: it doesn't exist
ERROR 6152: Can not copy access policy from table "*string*" for COLUMN "*string*": it doesn't exist
ERROR 6153: Can not copy access policy from table "*string*" for ROWS: it doesn't exist
ERROR 6154: Can not drop access policy on table "*string*" for COLUMN "*string*": it doesn't exist
ERROR 6155: Can not drop access policy on table "*string*" for ROWS: it doesn't exist
ERROR 6159: Can't drop indexed column
ERROR 6160: Can't *string* between these two tables: Text indices don't line up
ERROR 6161: Can't specify a schema for LOCAL TEMP objects
ERROR 6167: Cannot directly modify a text-index table
ERROR 6170: Cannot index external table '*stringstringstring*'
ERROR 6172: Cannot move data from a non retired storage location on node *string*
ERROR 6175: Cannot perform the specified administrative operation on a table with a text index
ERROR 6178: Cannot resolve node address [*string*] using address family *string*
ERROR 6179: Cannot resolve node address [*string*] using family *string*
ERROR 6180: Cannot resolve node control address [*string*] using address family *string*
ERROR 6181: Cannot resolve node control address [*string*] using family *string*
ERROR 6194: Column "*string*" doesnt exists in "*string*". Cannot create access policy
ERROR 6202: Correlation STRENGTH must be in the range [-1.0,1.0]
ERROR 6275: General pool priority must be between -100 and 100 inclusive

inclusive
ERROR 6283: Illegal argument to get_config_parameter: NULL
ERROR 6288: Indexed table must have a projection that is segmented by HASH(*string*), sorted by *string*, and contains every column listed in the CREATE TEXT INDEX statement
ERROR 6295: Invalid declaration; 'tls' method requires HOST TLS
ERROR 6296: Invalid ip address and/or network mask - "*string*"
ERROR 6297: Invalid number of control nodes *value*
ERROR 6298: Invalid parameter - "*string* = *string*"
ERROR 6299: Invalid resource pool specified for cascade to
ERROR 6302: IP address family conflict: node address family is *string*, control address family is *string*
ERROR 6329: Malformed message description
ERROR 6368: Only simple "expression" is allowed in access policy
ERROR 6397: Sequences cannot be used in access policy
ERROR 6414: Subnet IP address (family *string*) "*string*" is invalid
ERROR 6421: Table name can not be empty
ERROR 6456: Unsupported authentication method - "*string*"
ERROR 6475: Can't swap partitions between these two tables: Text indices don't line up
ERROR 6492: No text index column named "*string*". Indexed text column in text index table must exist with name "*string*"
ERROR 6504: Bad expression for Access Policy
ERROR 6509: Cascade to cannot be used to make a resource pool loop
ERROR 6519: Bad expression for Access Policy - "*string*"
ERROR 6520: Can not copy access policy from table "*string*" for COLUMN "*string*": "*string*"
ERROR 6521: Can not copy access policy from table "*string*" for ROWS: "*string*"
ERROR 6545: NULL is invalid object name for analyze_external_row_count
ERROR 6546: NULL is invalid object name for drop_external_row_count)
ERROR 6609: *string* process has reserved resources. Release it first
ERROR 6641: Argument of nth_value must be greater than zero
ERROR 6661: Can not expire password on LDAP user account
ERROR 6663: Can not modify password on LDAP user account
ERROR 6664: Can not modify profile on LDAP user account
ERROR 6666: Can not set Security Algorithm on LDAP user account
ERROR 6689: Cannot store TEMP data on HDFS storage locations
ERROR 6803: Ident authentication not allowed for remote connection types
ERROR 6934: Resource Allocation for this process not found
ERROR 6944: Schema "*string*" is already set to *string* privileges
ERROR 6979: Table *string* does not contain a key constraint '*string*'
ERROR 6980: Table *string* is already set to inherit privileges
ERROR 6981: Table *string* is already set to not inherit privileges
ERROR 7042: Unknown node
ERROR 7061: User-Defined Load function indicated that it consumed *value* bytes from the record lengths buffer, when only *value* were available
ERROR 7069: View "*string*" is already set *string* inherit privileges
ERROR 7104: Length of *string* type function argument cannot exceed *value* [*string*]
ERROR 7108: Parameter '*string*' cannot be NULL
ERROR 7140: DataBuffer indicated that it read *value* bytes, while LengthBuffer indicated that *value* data bytes were read
ERROR 7141: DataBuffer indicated that it wrote *value* bytes, while LengthBuffer indicated that *value* data bytes were written
ERROR 7145: Invalid end-of-file state for *string*: INPUT_NEEDED
ERROR 7146: Invalid state for UDParser: OUTPUT_NEEDED
ERROR 7154: UDChunker indicated that it consumed bytes, but returned INPUT_NEEDED
ERROR 7156: Access policy cannot be created on system table "*string*"
ERROR 7159: Backup type is invalid

ERROR 7159: Backup type is invalid
ERROR 7160: Cannot expand glob pattern due to error: *string*
ERROR 7162: Cannot load data due to error: *string*
ERROR 7174: Cyclic access policy detected for relation: *string*
ERROR 7179: ERROR TOLERANCE is not supported for *string* loads
ERROR 7180: ERROR TOLERANCE is not supported for ORC files
ERROR 7184: Illegal argument to create_storage_containers: NULL
ERROR 7185: Illegal argument: Please enter positive number for creating ROS containers
ERROR 7189: Occurrence parameter must be non-negative
ERROR 7194: Restore type is invalid
ERROR 7209: View *string* is already owned by *string*
ERROR 7225: Backup epoch [*value*] must be earlier than the current epoch [*value*]
ERROR 7240: Constraint '*string*' on table *string* is not a primary or unique key, nor a check constraint
ERROR 7254: ERROR TOLERANCE is not supported for Parquet files
ERROR 7269: Invalid portion: *string* [*value*]
ERROR 7271: Negative queueTimeout is not allowed
ERROR 7304: Cannot load data from [*string*]: all specified nodes are down
ERROR 7306: Cannot specify exceptions or rejected-data files on multiple nodes
ERROR 7312: Column *string* does not exist in Table *string*
ERROR 7313: Column *string* in Table *string* does not have SET USING expression
ERROR 7321: Invalid action *string* specified
ERROR 7322: Invalid pool name *string* specified
ERROR 7334: Size *value* out of range
ERROR 7335: Table *string* does not have any column with a SET USING expression
ERROR 7387: Cannot remove session's idle timeout
ERROR 7417: Error in blob creation: nChunks must be greater than 0
ERROR 7432: Invalid action URI '*string*': adapter not supported
ERROR 7434: Invalid cursor request source: *value*
ERROR 7436: Invalid number of arguments
ERROR 7451: Maxconnections cannot be greater than total number of allowable connections
ERROR 7452: Maxconnections cannot be set for a superuser
ERROR 7454: Maximum message size for notifier cannot be more than *value*
ERROR 7456: Memory size cannot be zero
ERROR 7459: Message cannot be empty
ERROR 7465: Must set mode in which connection limit is applicable
ERROR 7467: Negative idlesessiontimeout is not allowed
ERROR 7468: New idlesessiontimeout *string* would exceed database wide limit of *string*
ERROR 7469: New idlesessiontimeout *string* would exceed user limit of *string*
ERROR 7472: No channel is set
ERROR 7473: No notifier is set
ERROR 7476: Notifier "*string*" does not exist
ERROR 7481: Notifier memory size (*value*) cannot be less than the maximum message size (*value*)
ERROR 7482: Notifier memory size (*value*) should be less than 2TB
ERROR 7491: Object name cannot be null
ERROR 7536: Sample Count should be between 0 and 100
ERROR 7552: Temporary data cursor request requires type information
ERROR 7564: There is no node with such name: [*string*]
ERROR 7578: UDSource requested 0 threads on node [*string*] which is targeted for execution
ERROR 7587: Valid value for outlierThreshould is a positive float number
ERROR 7588: Valid value for sampling_ratio is a positive float number between 0 and 1

ERROR 7615: Requested too many nodes
ERROR 7624: When an empty map of nodes is supplied, expect nNodes > 0
ERROR 7707: Chunk index *value* out of range 0-*value*
ERROR 7714: Error in blob creation: maxSize cannot be negative
ERROR 7715: Error in blob creation: minSize cannot be negative
ERROR 7716: Error in blob creation: minSize must be less than maxSize
ERROR 7727: Invalid notifier adapter configuration: *string*
ERROR 7741: Tried to read (*value*) past end of chunk (*value*)
ERROR 7742: Tried to read (at *value*) past end of chunk
ERROR 7749: Unknown notifier adapter
ERROR 7770: miniBatch must be a positive integer
ERROR 7771: No resource grant exists with this id
ERROR 7799: Connection address family '*string*' is invalid; expected one of: 'ipv4','ipv6'
ERROR 7800: Connection address family '*string*' is not valid for host IP address '*string*'
ERROR 7868: Must specify shared storage for built-in shared file system
ERROR 7871: Projection "*string*" refers to column "*string*" referenced in the added column's default expression
ERROR 7877: Target table "*string*" is not anchor table for Projection "*string*"
ERROR 7899: Model *string* is already owned by *string*
ERROR 7910: Invalid column *string* definition for column *string*
ERROR 7911: Snapshot type is invalid
ERROR 7918: Invalid load stack: *string*
ERROR 7919: Invalid state for UDChunker::*string*(): *string*
ERROR 7924: UDChunker aligned a 0-byte chunk; chunks must be non-empty
ERROR 7933: Invalid refresh columns mode "*string*". Specify either "update" or "rebuild"
ERROR 7993: No table specified
ERROR 8065: Cannot load data from node *string* as it is not in the elastic throughput group selected for this session
ERROR 8068: Cannot remove session's grace period
ERROR 8073: Column *string* must be qualified as multiple tables are specified
ERROR 8087: Error in blob creation: name cannot be empty
ERROR 8097: Invalid path *string* for file system *string*
ERROR 8098: Invalid projection name '*string*' while parsing an optimizer directive
ERROR 8104: LIMIT can't be specified for non depot locations
ERROR 8110: Minimum value for *string* is *string*
ERROR 8160: Size cannot be an empty string
ERROR 8167: Table *string* does not exist for specified column *string*
ERROR 8168: Table *string* does not have any column in the specified column list
ERROR 8169: Table *string* for column *string* is not specified in the table list
ERROR 8175: The new period *string* would exceed the *string* limit of *string*
ERROR 8190: 'directory' parameter cannot be empty
ERROR 8191: 'directory' parameter must be specified
ERROR 8192: Compression '*string*' is invalid
ERROR 8193: Directory [*string*] exists. Please remove it or specify another directory
ERROR 8196: Temporary directory [*string*] exists. Please retry the query
ERROR 8197: Unable to look up UDFS for File System [*string*]
ERROR 8198: Unable to verify if directory [*string*] exists due to '*string*'
ERROR 8199: Unable to verify if temporary directory [*string*] exists due to '*string*'
ERROR 8215: 'compression' value must be of string type
ERROR 8216: 'directory' value must be of string type
ERROR 8217: 'rowGroupSizeMB' value must be of Integer type

ERROR 8218: 'tempsuffix' value must be of string type
ERROR 8246: Cannot create subscription in *string* state
ERROR 8249: Cannot make a subscription for DOWN node *string* as PRIMARY
ERROR 8253: Catalog name must not be specified in object names: *string*
ERROR 8254: Invalid or empty object name: *string*
ERROR 8271: Please specify schema name when using *
ERROR 8285: Missing parameter: parentName. Please provide parentName parameter
ERROR 8286: Missing parameter: parentType. Please provide parentType parameter
ERROR 8294: Cannot identify the type of [*string*]
ERROR 8297: Invalid JSON string [*string*]
ERROR 8302: Parameter check failed: *string*
ERROR 8402: Cannot create subscription for "*string*" when node is in EXECUTE state
ERROR 8411: Hyper parameter [*string*] is empty
ERROR 8412: Hyper parameter [*string*] is invalid
ERROR 8415: Metric [*string*] is not supported for algorithm [*string*]
ERROR 8425: Regularization type [*string*] is not supported for given optimizer [*string*]. Use [*string*]
ERROR 8433: The "count" member of JSON object [*string*] must be a positive integer
ERROR 8434: The "first" and "step" members of JSON object [*string*] must be numbers
ERROR 8452: Invalid file path: *string*
ERROR 8476: Usage:

Requires a Service Type followed by the Service Name as
found by calling list_services

Supported Service Types: 'TM'

Example:

select disable_service('TM','TM Mergeout');
ERROR 8477: Usage:

Requires a Service Type followed by the Service Name as
found by calling list_services

Supported Service Types: 'TM'

Example:

select enable_service('TM','TM Mergeout');
ERROR 8479: Usage:

Requires a Service Type to list services

Supported Service Types: 'TM', 'SYSTEM'

Example:

select list_services('TM');
ERROR 8488: Node *string* is not valid
ERROR 8502: Cannot drop subscription for replica shard
ERROR 8505: Catalog name must not be specified in wildcard: *string*
ERROR 8514: Intercept Mode [*string*] cannot be used when pre-centering the data. Use [*string*]
ERROR 8524: Patterns may not exclude a table [*string*] from an included schema [*string*]

schema [*string*]
ERROR 8530: *string* could not create directory for *string* '*string*'
ERROR 8532: *string* input file and *string* can not be the same file [*string*]
ERROR 8533: *string* path for *string* ['*string*'] must be a directory for multiple sources
ERROR 8549: Schema *string* is session scoped
ERROR 8598: *string* does NOT encrypt network traffic. *string* encrypts network traffic. Please change the configuration to be consistent
ERROR 8599: *string* encrypts network traffic. *string* does NOT encrypt network traffic. Please change the configuration to be consistent
ERROR 8611: Invalid cipher type: *value*
ERROR 8612: Invalid crypto operation type: *value*
ERROR 8670: Invalid value *value*
ERROR 8686: id_column is necessary when seed is specified
ERROR 8688: "*string*" is not a valid filesystem name
ERROR 8706: '*string*' value must be constant
ERROR 8709: Cannot parse metric. Expect only one JSON object but found more than one [*string*]
ERROR 8710: Cannot parse metric. Unexpectd JSON object [*string*]
ERROR 8711: Cannot parse metric. Unexpected JSON array [*string*]
ERROR 8712: Class label [*string*] used in cv_metrics does not exist
ERROR 8717: MAXQUERYMEMORYSIZE can be set only for general and user-defined pools
ERROR 8718: MAXQUERYMEMORYSIZE cannot exceed the maximum size of the pool
ERROR 8719: MAXQUERYMEMORYSIZE of *string* [*value* KB] would exceed the system limit [*value* KB]
ERROR 8725: id_column must be a single column
ERROR 8726: Invalid metric: *string*
ERROR 8727: Unexpected arguments [*string*] for metric [*string*]
ERROR 8728: Unsupported averaging method [*string*]. Expected one of: micro, macro and weighted
ERROR 8740: Unsupported parameter value. Please use 'true' or 'false'
ERROR 8764: 'fileSizeMB' value must be of Integer type
ERROR 8796: fileSizeMB '*value*' is invalid. Must be 0 (no limit) or a positive Integer value
ERROR 8797: Invalid node name
ERROR 8799: Invalid shard name: *string*
ERROR 8807: Usage of [*string*] cannot be changed
ERROR 8815: '*string*' value must be of string type
ERROR 8821: *string* '*string*' is invalid. dirMode requires user's read, write, and execute permissions (rwx------)
ERROR 8822: *string* '*string*' is invalid. Must be a positive octal value up to 1777 or a permission string of type rwxrwxrwx
ERROR 8838: Cannot set MAXCONCURRENCY of *string* pool to 0
ERROR 8839: Cannot set parameter *string* on built-in pool "*string*"
ERROR 8851: Incorrect CIDR format. Details: [*string*]
ERROR 8853: Load balance policy type '*string*' does not exist. Please supply a valid load balance policy name, such as 'ROUNDROBIN'
ERROR 8865: Routes must be unique. Specified route *string* is equivalent to route *string* in rule '*string*'
ERROR 8889: Refresh_columns on partitions is only supported for "rebuild" mode
ERROR 8902: Invalid argument(s): *value*, *value*, *value*
ERROR 8904: *string*: Can not analyze statistics of a virtual table/projection *string*
ERROR 8910: Empty partition key is invalid for *string*
ERROR 8915: Invalid accuracy value for *string*
ERROR 8917: NULL is invalid accuracy value for *string*
ERROR 8918: NULL is invalid value for *string*

ERROR 8918: NULL is invalid column name for *string*
ERROR 8919: NULL is invalid object name for *string*
ERROR 8920: NULL is invalid partition key for *string*
ERROR 8921: NULL is invalid statistics type for *string*
ERROR 8924: RemoteStorageForTemp is set to 1, but cannot find any remote storage locations for temp usage
ERROR 8928: Usage type not supported. Only COMMUNAL, USER and TEMP locations are supported
ERROR 8930: Alter load balance group sub-command not recognized
ERROR 8931: Cannot add objects of type *string* to a Load Balance Group of type *string*
ERROR 8934: Cannot drop objects of type *string* from a Load Balance Group of type *string*
ERROR 8940: Could not decode catalog object type for statement type *value*
ERROR 8954: HybridStream must be initialized with at least a buffer or a file path
ERROR 8960: Invalid catalog object type *value* (*string*) for any kind of Load Balance Group
ERROR 9005: The filter clause cannot be empty when adding '*string*' objects to Load Balance Group '*string*' of type '*string*' because that group does not already have a filter
ERROR 9006: Unknown spread option: *string*
ERROR 9007: 'pageSizeKB' value must be of Integer type
ERROR 9008: 'rowBatchSize' value must be of Integer type
ERROR 9012: PageSizeKB '*value*' is invalid. Must be a positive Integer value
ERROR 9013: RowBatchSize '*value*' is invalid. Must be a positive Integer value
ERROR 9014: RowgroupSizeMB '*value*' is invalid. Must be a positive Integer value
ERROR 9070: Too many threads specified for the index tool; *value* specified; *value* maximum
ERROR 9100: Cannot make a subscription for SECONDARY node *string* as PRIMARY
ERROR 9109: FAULTGROUP is not supported in EON mode
ERROR 9110: Invalid parameter: node_name
ERROR 9111: Invalid parameter: shard_name
ERROR 9113: Missing parameter: node_name
ERROR 9184: Invalid value for option TokenTimeout. Timeout must be integer from *value* to *value* msec or 0 (auto)
ERROR 9220: Cannot create array of size *value*
ERROR 9225: Negative array index
ERROR 9234: Type is not an array type
ERROR 9258: No parquet file was found in [*string*]
ERROR 9275: More than one parquet file was found in [*string*], please specify only one parquet file
ERROR 9282: Seed cannot be 1, since a val of 1 resets srand()
ERROR 9303: '*string*' is an unsupported object type for privileges
ERROR 9307: Name cannot be NULL
ERROR 9313: Type cannot be NULL
ERROR 9327: string_to_array: delimiter cannot be empty
ERROR 9328: string_to_array: NULL delimiter not supported
ERROR 9329: string_to_array parse error: *string*
ERROR 9353: *string* can not be greater than PASSWORD_MAX_LENGTH *value*
ERROR 9354: PASSWORD_MAX_LENGTH must be within the range from *value* to *value*
ERROR 9355: PASSWORD_MIN_DIGITS + PASSWORD_MIN_SYMBOLS + PASSWORD_MIN_LETTERS *value* can not be greater than PASSWORD_MAX_LENGTH *value*
ERROR 9356: PASSWORD_MIN_DIGITS + PASSWORD_MIN_SYMBOLS + PASSWORD_MIN_LOWERCASE_LETTERS +

PASSWORD_MIN_LOWERCASE_LETTERS + PASSWORD_MIN_UPPERCASE_LETTERS *value* can not be greater than PASSWORD_MAX_LENGTH *value*

ERROR 9413: Depot policy for object *string* does not exist

ERROR 9414: Depot policy for object *string* does not exist for subcluster *string*

ERROR 9417: Partition level policies for object *string* already exists on *string*. Drop them before creating table level policies

ERROR 9420: Table level policy for object *string* already exists on *string*. Drop them before creating partition level policies

ERROR 9422: The specified model name can not have more than 3 components: database.schema.model

ERROR 9424: Usage:

Requires a Service Type followed by the Service Name as found by calling list_services

Supported Service Types: 'TM', 'SYSTEM'

Example:

select hurry_service('TM','PartitionTables');

ERROR 9454: Invalid VerticaTypeForComplexType [*string*] specified

ERROR 9491: *string*: Invalid stats type '*string*'. Valid values are 'BASE', 'HISTOGRAMS' and 'ALL'

ERROR 9509: Invalid parameters

ERROR 9538: Control set size out of bounds -1 <= *value* <= *value*

ERROR 9539: Control set size out of bounds, valid value ranges [1, *value*]

ERROR 9540: Control set size out of bounds: valid values are [1, *value*]

ERROR 9541: Maximum *value* control nodes allowed, and there are already *value* (expected) control nodes in the cluster

ERROR 9544: Subcluster *string* does not have any nodes

ERROR 9553: AES keys with passwords cannot also include key text

ERROR 9569: Couldn't load any certificates

ERROR 9570: Couldn't parse subject '*string*'; missing '='

ERROR 9590: Private key and certificate public key do not match: *string*

ERROR 9591: RSA keys with passwords are not currently supported

ERROR 9608: Cannot create subcluster with control set size *value*, maximum *value* control nodes allowed and there are already *value* (expected) control nodes in the cluster

ERROR 9611: Increase in pool size to *string* [*value* KB] causes general pool to fall below minimum [25% = *value* KB] *string*

ERROR 9632: LABEL should be specified for depot locations

ERROR 9645: COPY column option *string* not supported for column of scalar type (*string*)

ERROR 9646: COPY parameter *string* cannot have the same value as *string*

ERROR 9647: COPY parameter *string* cannot overlap parameter *string*

ERROR 9648: COPY parameter *string* must be a single character

ERROR 9686: LDAPLinkSearchTimeout must be greater than 0

ERROR 9697: Cannot replicate properties of subcluster *string*

ERROR 9721: Password cannot be changed until password minimum lifetime has passed

ERROR 9722: *string* can not be greater than PASSWORD_LIFE_TIME

ERROR 9723: *string* can not be greater than PASSWORD_MIN_LENGTH *value*

ERROR 9724: Cannot clone a subcluster's properties to itself

ERROR 9727: Password is not significantly different than previous password

ERROR 9743: No node or subcluster specified

ERROR 9757: File path is not specified

ERROR 9758: File size is not specified

ERROR 9759: Invalid bound for collection type *string*

ERROR 9797: Sync to cloud is not enabled
ERROR 9805: Invalid CPU affinity parameters for pool '*string*' on subcluster *string*
ERROR 9806: Invalid salt length
ERROR 9823: *string* is not a valid object for this function
ERROR 9834: If a client certificate is specified, a CA bundle must also be set
ERROR 9835: If TLS is enabled with server certificate validation, a CA bundle must be set
ERROR 9844: Table level policies for object *string* already exists on *string*. Drop them before creating projection level policies
ERROR 9846: Unsupported TLS mode '*string*'
ERROR 9900: Value to search type [*string*] cannot be implicitly cast to the collection element type [*string*]
ERROR 9915: Invalid return type for function [*string*]
ERROR 9916: max_binary_size should be a positive number
ERROR 9944: string_to_array: collection_close cannot be empty
ERROR 9945: string_to_array: collection_close should be of length 1
ERROR 9946: string_to_array: collection_delimiter cannot be empty
ERROR 9947: string_to_array: collection_open cannot be empty
ERROR 9948: string_to_array: collection_open should be of length 1
ERROR 9949: User parameter *string* cannot have the same value as *string*
ERROR 9993: Cannot set node IP address to be [*string*] because that IP address is used by at least one UP node
ERROR 10018: 'compressionBlockSizeKB' value must be of Integer type
ERROR 10019: 'compressionStrategy' value must be of string type
ERROR 10020: 'rowIndexStride' value must be of Integer type
ERROR 10021: 'stripeSizeMB' value must be of Integer type
ERROR 10024: compressionBlockSizeKB '*value*' is invalid. Must be a positive Integer value
ERROR 10025: CompressionStrategy '*string*' is invalid
ERROR 10081: rowIndexStride '*value*' is invalid. Must be a positive Integer value
ERROR 10083: stripeSizeMB '*value*' is invalid. Must be a positive Integer value
ERROR 10111: Both config section and key cannot be empty
ERROR 10148: 'binaryTypesFormat' value must be of string type
ERROR 10149: 'enclosedBy' value must be of string type
ERROR 10150: 'escapeAs' value must be of string type
ERROR 10151: 'fileExtension' value must be of string type
ERROR 10152: 'nullAs' value must be of string type
ERROR 10153: 'recordTerminator' value must be of string type
ERROR 10158: binaryTypesFormat '*string*' is invalid
ERROR 10197: Invalid value for 'addHeader'
ERROR 10244: Cannot specify node-names nodes for SHARED [*string*] location
ERROR 10254: Grants for SHARED USER location can only be changed for all nodes
ERROR 10286: SHARED [*string*] location can only be dropped for all nodes
ERROR 10287: Specifying individual nodes is not supported for SHARED [*string*] location
ERROR 10295: Usage of [*string*] cannot be changed from SHARED *string* to SHARED *string*
ERROR 10354: max_binary_size should be enough to fit at least one element
ERROR 10376: Invalid timestamp argument: *string*
ERROR 10379: The number of plans should be greater than 0 and less than equal to *value*
ERROR 10381: [*string*] is not a valid storage location on node [*string*]
ERROR 10434: 'match_columns' parameter cannot be empty
ERROR 10445: No column names match any match_columns expression

ERROR 10467: match_columns() function takes only a constant as a valid input argument
ERROR 10483: Cannot create storage locations using VFS filesystem
ERROR 10529: SIGNED BY ca does not sign the provided certificate
ERROR 10573: Unknown storage tag [*string*]
ERROR 10600: Schema name "*string*"."*string*" is too long
ERROR 10602: FOR SUBCLUSTER can only be used for altering user RESOURCE POOL
ERROR 10610: Invalid salt value: contains a null char 00
ERROR 10612: Notifier Channel cannot be empty
ERROR 10615: 'filename' value must be of string type
ERROR 10618: fileName is not allowed when PARTITION BY expression is non-empty
ERROR 10619: fileName must not be set to empty
ERROR 10620: Invalid DEPENDS string for CREATE LIBRARY
ERROR 10623: Minimum message size for notifier cannot be less than *value*
ERROR 10628: 'omitNullFields' value must be of Boolean type
ERROR 10638: Notification policy parameter cannot be null

Info messages

INFO 6296: Invalid ip address and/or network mask - "*string*"
INFO 6456: Unsupported authentication method - "*string*"
INFO 6457: Unsupported host type - "*string*"

Notice messages

NOTICE 9798: Table *string* is already owned by *string*, nothing was done

Warning messages

WARNING 2075: @INCLUDE without filename in timezone file "*string*", line *value*
WARNING 2415: Cannot drop extended statistics on projection *string*. Dropping base statistics only
WARNING 2964: CURRENT_TIME(*value*) precision reduced to maximum allowed, *value*
WARNING 2966: CURRENT_TIMESTAMP(*value*) precision reduced to maximum allowed, *value*
WARNING 3607: INTERVAL leading field precision increased to *value*
WARNING 3608: INTERVAL leading field precision reduced to *value*
WARNING 3610: INTERVAL SECOND precision reduced to *value*
WARNING 3673: Invalid hint identifier '*string*'
WARNING 3710: Invalid number for timezone offset in timezone file "*string*", line *value*
WARNING 3759: Invalid syntax in timezone file "*string*", line *value*
WARNING 3767: Invalid timezone file name "*string*"
WARNING 3787: Invalid value for parameter
WARNING 3878: LOCALTIME(*value*) precision reduced to maximum allowed, *value*
WARNING 3880: LOCALTIMESTAMP(*value*) precision reduced to maximum allowed, *value*
WARNING 3960: Missing timezone abbreviation in timezone file "*string*", line *value*
WARNING 3961: Missing timezone offset in timezone file "*string*", line *value*
WARNING 5020: TIME(*value*)*string* precision reduced to maximum allowed, *value*
WARNING 5029: TIMESTAMP(*value*) precision reduced to maximum allowed, *value*
WARNING 5031: TIMESTAMP(*value*)*string* precision reduced to maximum allowed, *value*
WARNING 5037: TIMETZ(*value*) precision reduced to maximum allowed, *value*
WARNING 5041: Timezone abbreviation "*string*" is multiply defined
WARNING 5042: Timezone abbreviation "*string*" is too long (maximum *value* characters) in timezone file "*string*", line *value*
WARNING 5045: Timezone file recursion limit exceeded in file "*string*"
WARNING 5046: Timezone offset *value* is not a multiple of 900 sec (15 min)

WARNING 5046: Timezone offset *value* is not a multiple of 900 sec (15 min) in timezone file "*string*", line *value*
WARNING 5047: Timezone offset *value* is out of range in timezone file "*string*", line *value*
WARNING 5136: Unable to log this tuning analysis event
WARNING 5605: Invalid projection createtype '*string*'
WARNING 5645: Projection basename cannot be empty
WARNING 5646: Projection createtype cannot be empty
WARNING 5693: Using 1 year for QUEUETIMEOUT
WARNING 6189: Client Authentication "*string*" is disabled
WARNING 6203: Could not access *string* "*string*": *value*
WARNING 6289: Indexed table must have a projection that is segmented by HASH(*string*), sorted by *string*, and contains every column listed in the CREATE TEXT INDEX statement (no projections have been created yet)
WARNING 6302: IP address family conflict: node address family is *string*, control address family is *string*
WARNING 6314: LDAP URL [*string*] seems to be malformed*string*. Client Authentication "*string*" is disabled
WARNING 6383: Path "*string*" exists, but it is directory, not a file
WARNING 6384: Path "*string*" exists, but it is not a directory
WARNING 6387: Projection replace oid is empty or malformed
WARNING 6473: You cannot use both parameters '*string*' and '*string*'. Client Authentication '*string*' is disabled
WARNING 6566: Invalid constant hint:'*string*'
WARNING 6569: Invalid table hint identifier '*string*'
WARNING 6619: /*+syntactic_join*/ hint omitted, ignoring join hints
WARNING 6802: Hint _oidref is empty or malformed
WARNING 6824: Invalid explain hint identifier '*string*'
WARNING 6825: Invalid join hint identifier '*string*'
WARNING 6835: Invalid with hint identifier '*string*'
WARNING 7083: /*+syntactic_join*/ hint omitted, ignoring union hints
WARNING 7102: Invalid union hint identifier '*string*'
WARNING 7168: Column Access Policies on flex tables may not be completely secure
WARNING 7541: Skipnode hint needs node name
WARNING 7542: Skipnode hint omitted in sub-query
WARNING 7584: Using 1 year as idlesessiontimeout
WARNING 7674: Idlesessiontimeout cannot be less than 15 minutes for superuser. Using 15 minutes as idlesessiontimeout
WARNING 8187: Using *string* (maximum) as *string*
WARNING 8303: Parameter check warning: *string*
WARNING 8916: No remote storage location found for temp usage
WARNING 9566: Could not load certificate from provided value; Error in PEM_read_bio_X509: *string*
WARNING 9753: Collection type bound will not be used
WARNING 10493: *string* "*string*" is already owned by *string*
WARNING 10617: Exporting to single file. fileSizeMB will be set to 0 (no limit)

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22025

This topic lists the messages associated with the SQLSTATE 22025.

SQLSTATE 22025 description
INVALID_ESCAPE_SEQUENCE

Messages associated with this SQLState

ERROR 3656: Invalid escape sequence
ERROR 3657: Invalid escape string

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22906

This topic lists the messages associated with the SQLSTATE 22906.

SQLSTATE 22906 description

NONSTANDARD_USE_OF_ESCAPE_CHARACTER

Messages associated with this SQLState

WARNING 4166: Nonstandard use of \' in a string literal at or near "*string*"
WARNING 4167: Nonstandard use of \\ in a string literal at or near "*string*"
WARNING 4168: Nonstandard use of escape in a string literal at or near "*string*"

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22V02

This topic lists the messages associated with the SQLSTATE 22V02.

SQLSTATE 22V02 description

INVALID_TEXT_REPRESENTATION

Messages associated with this SQLState

ERROR 2825: Could not convert "*string*"*string* to a boolean
ERROR 2826: Could not convert "*string*"*string* to a float8
ERROR 2827: Could not convert "*string*"*string* to an int8
ERROR 3677: Invalid input for *string*: "*string*"
ERROR 3680: Invalid input syntax for boolean: "*string*"
ERROR 3681: Invalid input syntax for integer: "*string*"
ERROR 3682: Invalid input syntax for numeric: "*string*"
ERROR 3711: Invalid number: "*string*"
ERROR 3712: Invalid numeric format *string*
ERROR 3714: Invalid numeric value: "*string*"
ERROR 3751: Invalid Session ID format
ERROR 3757: Invalid syntax for float: "*string*"
ERROR 3758: Invalid syntax for numeric: "*string*"
ERROR 3894: Malformed record literal: "*string*"
ERROR 4169: Not a number: "*string*"
ERROR 4198: Number exceeds format: "*string*"
ERROR 5930: Invalid numeric format *string*. Expected precision is *value* and scale is *value*
ERROR 8264: Invalid input syntax for type *string*: "*string*"
ERROR 9988: Invalid input syntax for type *string*: "*string*"

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22V03

This topic lists the messages associated with the SQLSTATE 22V03.

SQLSTATE 22V03 description

INVALID_BINARY_REPRESENTATION

Messages associated with this SQLState

ERROR 2829: Could not convert integer *valuestring* to a boolean
ERROR 3536: Incorrect binary data format in bind parameter *value*
ERROR 3623: Invalid binary input syntax: '*string*...'
ERROR 3624: Invalid bitstring "*string*"
ERROR 3671: Invalid hex string "*string*"
ERROR 3678: Invalid input syntax for *string*
ERROR 3716: Invalid octal string format "*string*"
ERROR 3717: Invalid octal string format (octal string length
ERROR 5416: Value too long for type *string*(*value*)
ERROR 5936: Invalid string format "*string*"

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22V04

This topic lists the messages associated with the SQLSTATE 22V04.

SQLSTATE 22V04 description

BAD_COPY_FILE_FORMAT

Messages associated with this SQLState

ERROR 2031: *string* Header size (*value*) is corrupted
ERROR 2032: *string* Header size (*value*) is too small
ERROR 2035: *string* Input record *value* has been rejected (*string*)
ERROR 2053: *string* Row size (*value*) is corrupted
ERROR 2054: *string* Unexpected EOF while reading header. Expected *value* but
read *value*
ERROR 2738: COPY file signature not recognized
ERROR 2767: COPY: Wrong Header size *value*. Expected *value*
ERROR 3562: Input has extra trailing bytes
ERROR 3640: Invalid COPY file header (unsupported Version Number)
ERROR 4206: Number of fields is *value*, expected *value*
ERROR 4627: Row delimiter not found; corrupt file input (read *value*
bytes from input)
ERROR 5495: Wrong size *value* for bool column *value* (*string*)
ERROR 5496: Wrong size *value* for date column *value* (*string*)
ERROR 5497: Wrong size *value* for float column *value* (*string*)
ERROR 5498: Wrong size *value* for integer column *value* (*string*)
ERROR 5499: Wrong size *value* for Interval column *value* (*string*)
ERROR 5500: Wrong size *value* for Numeric column *value* (*string*)
ERROR 5501: Wrong size *value* for Time column *value* (*string*)
ERROR 5502: Wrong size *value* for Timestamp column *value* (*string*)
ERROR 5503: Wrong size *value* for TimestampTz column *value* (*string*)
ERROR 5504: Wrong size *value* for TimeTz column *value* (*string*)
ERROR 6363: Not able to skip *value* rows for source [*string*]
ERROR 7293: *string* [*value*] records have been rejected
ERROR 8394: Wrong size *value* for Uuid column *value* (*string*)

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22V0B

This topic lists the messages associated with the SQLSTATE 22V0B.

SQLSTATE 22V0B description

ESCAPE_CHARACTER_ON_NOESCAPE

Messages associated with this SQLState

ERROR 2746: COPY NO ESCAPE cannot also contain an ESCAPE clause
ERROR 2747: COPY NO ESCAPE for column *string* cannot also contain an ESCAPE clause

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22V21

This topic lists the messages associated with the SQLSTATE 22V21.

SQLSTATE 22V21 description

INVALID_EPOCH

Messages associated with this SQLState

ERROR 2144: AHM can't advance past the cluster last full backup epoch. (Last Backup Epoch: *value*)
ERROR 2145: AHM can't advance past the cluster last backup time. (Last Backup time: *string*)
ERROR 2146: AHM can't advance past the cluster last good epoch (LGE) time (Cluster LGE time: *string*)
ERROR 2147: AHM can't advance past the cluster last good epoch (LGE). (Cluster LGE: *value*)
ERROR 2148: AHM can't advance past the latest epoch time (Latest epoch time: *string*)
ERROR 2153: AHM must be less than the current epoch (Current Epoch: *value*)
ERROR 2318: Can't run historical queries at epochs prior to the Ancient History Mark
ERROR 3000: DDL interfered with this statement
ERROR 3184: Epoch specified is not in historical epoch range
ERROR 3559: Input epoch must be greater than or equal to the earliest epoch (earliest epoch: *value*)
ERROR 3560: Input epoch must be greater than the current AHM (Current AHM: *value*)
ERROR 3561: Input epoch must be less than or equal to the AHM epoch (AHM epoch: *value*)
ERROR 3567: Input time can't be rounded down to an epoch higher than the current AHM epoch (Current AHM epoch: *value*, Current AHM time: *string*)
ERROR 3568: Input time must be greater than or equal to the earliest epoch time (Earliest epoch time: *string*)
ERROR 3569: Input time must be greater than the current AHM time (Current AHM time: *string*)
ERROR 3570: Input time must be less than or equal to the AHM epoch time (AHM epoch time: *string*)
ERROR 3654: Invalid epoch
ERROR 3844: Last good epoch not set
ERROR 3926: MergeOut start epoch (=*value*) greater than end epoch (=*value*)
ERROR 4940: The current AHM is already *value*
ERROR 5013: Time specified is not in historical epoch range
ERROR 7639: AHM can't advance due to lack of full backup

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22V23

This topic lists the messages associated with the SQLSTATE 22V23.

SQLSTATE 22V23 description

RAISE_EXCEPTION

Messages associated with this SQLState

Error messages

ERROR 5783: Client error: *string* (in function *string*() at *string*:*value*)
ERROR 7084: ABORTRECOVERY is only allowed in UNSAFE mode
ERROR 7136: USER GENERATED ERROR
ERROR 7137: USER GENERATED ERROR: *string*

Warning messages

WARNING 2005: *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 22V24

This topic lists the messages associated with the SQLSTATE 22V24.

SQLSTATE 22V24 description

COPY_PARSE_ERROR

Messages associated with this SQLState

Error messages

ERROR 6703: Corrupt ORC source!
ERROR 6723: Data type *string* is not supported for the ORC file format
ERROR 6777: Failed to read ORC source [*string*]
ERROR 6778: Failed to read ORC source [*string*]: *string*
ERROR 7087: Attempt to load *value* columns from an ORC source [*string*] that has *value* columns
ERROR 7127: The table has *value* columns, but the ORC source [*string*] has *value* columns
ERROR 7157: Attempt to load *value* columns from an ORC source [*string*] that has *value* columns and *value* partition columns
ERROR 7202: The table has *value* columns, but the ORC source [*string*] has *value* columns and *value* partition columns
ERROR 7241: Corrupt parquet source!
ERROR 7246: Data type *string* is not supported for the PARQUET file format
ERROR 7257: Failed to read parquet source [*string*]
ERROR 7258: Failed to read parquet source [*string*]: *string*
ERROR 7985: Decimal type with precision *value* (> 38) is not supported
ERROR 8862: No ORC Column Readers found for column [*value*]
ERROR 9135: Attempt to load *value* columns from a parquet source [*string*] that has *value* columns
ERROR 9156: Datatype mismatch: column *value* in the parquet source [*string*] has primitive type *string*, expected complex type
ERROR 9171: The table has *value* columns, but the parquet source [*string*] has *value* columns
ERROR 9252: Corrupted Schema of Parquet file
ERROR 9258: No parquet file was found in [*string*]
ERROR 9266: Byte array as Numeric is not supported right now
ERROR 9289: Datatype mismatch: column *value* in the parquet source [*string*] does not have the expected ARRAY type at *string*
ERROR 9290: Datatype mismatch: column *value* in the parquet source [*string*] does not have the expected MAP type at *string*
ERROR 9295: Datatype mismatch: column *value* in the parquet source [*string*] has ARRAY type. Expected ROW type at *string*
ERROR 9296: Datatype mismatch: column *value* in the parquet source [*string*] has MAP type. Expected ROW type at *string*
ERROR 9299: Source has unsupported MAP type at *string*
ERROR 9325: Source has unsupported ARRAY type at *string*
ERROR 9399: Datatype mismatch: column *value* in the parquet source [*string*] has *stringstringvalue* type*stringstring*, expected primitive type *string*
ERROR 9400: Datatype mismatch: column *value* in the parquet source [*string*] has *stringstringvalue stringstringstring*, expected *string*
ERROR 9401: Datatype mismatch: column *value* in the parquet source [*string*] has different *string* type field count. Source has *value* at *string*, expected *value*
ERROR 9402: Datatype mismatch: column *value* in the parquet source [*string*] has different ROW type. Source has primitive type *string* at

string, expected complex type
ERROR 9520: Attempt to load *value* columns from a parquet source [*string*] that has *value* columns and *value* partition columns
ERROR 9524: The table has *value* columns, but the parquet source [*string*] has *value* columns and *value* partition columns
ERROR 9699: Datatype mismatch: column "*string*" in the ORC source [*string*] has type *string*, expected *string*
ERROR 9700: Datatype mismatch: column "*string*" in the ORC source [*string*] has type *string*, expected ARRAY type
ERROR 9701: Datatype mismatch: column "*string*" in the ORC source [*string*] has type *string*, expected MAP type
ERROR 9702: Datatype mismatch: column "*string*" in the ORC source [*string*] has type *string*, expected ROW type
ERROR 9703: Datatype mismatch: column "*string*" in the ORC source [*string*] has type LIST of *string*, expected *string*
ERROR 9705: Datatype mismatch: Field "*string*" in the ORC source [*string*] has type *string*, expected *string*
ERROR 9706: Datatype mismatch: Field "*string*" in the ORC source [*string*] has type *string*, expected ARRAY type
ERROR 9707: Datatype mismatch: Field "*string*" in the ORC source [*string*] has type *string*, expected MAP type
ERROR 9708: Datatype mismatch: Field "*string*" in the ORC source [*string*] has type *string*, expected ROW type
ERROR 9709: Datatype mismatch: Struct type "*string*" in the ORC source [*string*] has fewer child fields than expected
ERROR 9710: Datatype mismatch: Struct type "*string*" in the ORC source [*string*] has more child fields than expected
ERROR 9842: Selected column *string* has a complex type
ERROR 9969: Partition Columns cannot be array type *string*
ERROR 10055: No orc file was found in [*string*]
ERROR 10591: Failed to read avro source [*string*]
ERROR 10592: Failed to read avro source [*string*]: *string*
ERROR 10594: No avro file was found in [*string*]
ERROR 10598: Failed to load schema from avro file [*string*]

Warning messages

WARNING 9317: Data type TIME of column [*string*] is not supported in Vertica right now. Creating this external table will cause errors.
WARNING 9435: Data type UNSIGNED INT64 of column [*string*] is not supported in Vertica. Numeric(20,0) is used instead

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 23502

This topic lists the messages associated with the SQLSTATE 23502.

SQLSTATE 23502 description

NOT_NULL_VIOLATION

Messages associated with this SQLState

Error messages

ERROR 2416: Cannot drop NOT NULL constraint on column "*string*" when it is referenced in PARTITION BY expression
ERROR 2417: Cannot drop NOT NULL constraint on column "*string*" when it is referenced in primary key constraint
ERROR 4182: NOT NULL constraint on column "*string*" already exists
ERROR 4183: NOT NULL constraint on column "*string*" does not exist

Warning messages

WARNING 2623: Column "*string*" definition changed to NOT NULL

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 23503

This topic lists the messages associated with the SQLSTATE 23503.

SQLSTATE 23503 description

FOREIGN_KEY_VIOLATION

Messages associated with this SQLState

ERROR 4165: Nonexistent foreign key value detected in FK-PK join [*string*]; value [*string*]

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 23505

This topic lists the messages associated with the SQLSTATE 23505.

SQLSTATE 23505 description

UNIQUE_VIOLATION

Messages associated with this SQLState

ERROR 3147: Duplicate MERGE key detected in join [*string*]; value [*string*]
ERROR 3149: Duplicate primary/unique key detected in join [*string*]; value [*string*]
ERROR 4840: Subquery used as an expression returned more than one row
ERROR 6744: Duplicate key values in table '*string*': '*string*' -- violates constraint '*string*'
ERROR 6745: Duplicate key values: '*string*' -- violates constraint '*string*'
ERROR 7695: Null value in primary key column: '*string*' -- violates constraint '*string*'

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 23514

This topic lists the messages associated with the SQLSTATE 23514.

SQLSTATE 23514 description

CHECK_VIOLATION

Messages associated with this SQLState

ERROR 7230: Check constraint '*string*' *string* violation: '*string*'
ERROR 7231: Check constraint '*string*' *string* violation in table '*string*': '*string*'
ERROR 7232: Check constraint '*string*' *string* violation: '*string*'

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 25000

This topic lists the messages associated with the SQLSTATE 25000.

SQLSTATE 25000 description

INVALID_TRANSACTION_STATE

Messages associated with this SQLState

Error messages

ERROR 8058: Active subscriptions changed during query planning
ERROR 8231: Invalid number of shards to operate on => *value*
ERROR 8233: Node is not the primary for shard (*value*)
ERROR 8269: Node is no longer the primary for shard (*value*)
ERROR 8298: Multiple shards when partitioning ROSs
ERROR 8383: Node is not the primary for shard (*value*)
ERROR 8568: The shard (*value*) is not in the active shard set of this mergeout session
ERROR 10309: Initiator node(*string*) changed from primary node to secondary node during backup task

Rollbacktxn messages

ROLLBACKTXN 8222: No active subscriptions found for the session
ROLLBACKTXN 8647: Transaction Catalog in an inconsistent state

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 25006

This topic lists the messages associated with the SQLSTATE 25006.

SQLSTATE 25006 description

READ_ONLY_SQL_TRANSACTION

Messages associated with this SQLState

ERROR 2448: Cannot issue this command in a read-only transaction

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 25V01

This topic lists the messages associated with the SQLSTATE 25V01.

SQLSTATE 25V01 description

NO_ACTIVE_SQL_TRANSACTION

Messages associated with this SQLState

ERROR 2342: Cannot advance epoch without a transaction

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 28000

This topic lists the messages associated with the SQLSTATE 28000.

SQLSTATE 28000 description

INVALID_AUTHORIZATION_SPECIFICATION

Messages associated with this SQLState

Error messages

ERROR 2701: Conflicting, redundant or unsupported option: *string*
ERROR 2702: Conflicting, redundant or unsupported option: groupElts
ERROR 2959: Current user cannot be dropped
ERROR 4293: Option "*string*" not recognized
ERROR 4722: Session user cannot be dropped
ERROR 4846: Superuser cannot be dropped
ERROR 5387: User does not exist
ERROR 6441: Trying to change password, but user "*string*" does not exist
ERROR 6442: Trying to find authentication mehtod, but user "*string*" does not exist
ERROR 6443: Trying to update login history, but user "*string*" does not exist
ERROR 6494: SecurityAlgorithm valid options are NONE, MD5 and SHA512
ERROR 6671: Cannot change password on LDAP user "*string*"
ERROR 6841: LDAP role "*string*" cannot be dropped unless it is orphaned (sourced from a different LDAP than currently configured)
ERROR 6842: LDAP user "*string*" cannot be renamed
ERROR 6843: LDAP user "*string*" has not database password
ERROR 6844: LDAP user cannot be dropped unless it is orphaned
ERROR 7667: SecurityAlgorithm valid options are NONE and SHA512

Fatal messages

FATAL 2247: Authentication failed for user "*string*": invalid authentication method
FATAL 2248: Authentication failed for username "*string*"
FATAL 3495: GSSAPI authentication failed for user "*string*"
FATAL 3781: Invalid username or password
FATAL 3828: Kerberos 5 authentication failed for user "*string*"
FATAL 3846: LDAP authentication failed for user "*string*"
FATAL 4131: No Vertica user name specified in startup packet
FATAL 5130: Unable to determine authentication mechanism; check "ClientAuthentication" methods granted to user or role
FATAL 5592: Ident authentication failed for user "*string*"
FATAL 6431: TLS authentication failed for user "*string*"
FATAL 6796: Hash/password authentication is not supported for LDAP Users and/or Roles - user "*string*"
FATAL 6807: Implicit trust is not supported for LDAP Users and/or Roles - user "*string*"
FATAL 10585: OAuth authentication failed for user "*string*"
FATAL 10595: Token introspection failed

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 28001

This topic lists the messages associated with the SQLSTATE 28001.

SQLSTATE 28001 description

ACCOUNT_LOCKED

Messages associated with this SQLState

FATAL 4974: The user account "*string*" is locked
FATAL 4975: The user account "*string*" is locked due to too many invalid logins
FATAL 6998: The LDAP user account "*string*" is locked

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 2BV01

This topic lists the messages associated with the SQLSTATE 2BV01.

SQLSTATE 2BV01 description

DEPENDENT_OBJECTS_STILL_EXIST

Messages associated with this SQLState

Error messages

ERROR 3052: Dependent privileges exist
ERROR 3128: DROP failed due to dependencies
ERROR 3130: DROP PROFILE failed due to dependencies
ERROR 3131: DROP ROLE failed due to dependencies
ERROR 6240: DROP AUTHENTICATION failed due to dependencies
ERROR 7303: Cannot drop table "*string*" because other objects depend on it
ERROR 7345: *string* failed due to dependencies

Warning messages

WARNING 7945: Column *string* is dropped which will break some dependent access policies on the table. Certain queries may fail

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 38004

This topic lists the messages associated with the SQLSTATE 38004.

SQLSTATE 38004 description

E_R_E_READING_SQL_DATA_NOT_PERMITTED

Messages associated with this SQLState

WARNING 7339: User may not have read privilege to the external table storage location: [*string*]

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 40V01

This topic lists the messages associated with the SQLSTATE 40V01.

SQLSTATE 40V01 description

T_R_DEADLOCK_DETECTED

Messages associated with this SQLState

ERROR 3010: Deadlock: *string* - *string*
ERROR 3011: Deadlock: [Txn %#llx] *string* - *string* error *string*

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42501

This topic lists the messages associated with the SQLSTATE 42501.

SQLSTATE 42501 description

INSUFFICIENT_PRIVILEGE

Messages associated with this SQLState

Error messages

ERROR 2065: *string*: Invalid table/projection/column *string*
ERROR 2198: analyze_statistics: Requires modify permissions for
table/projection/column *string*
ERROR 2347: Cannot alter predefined role "*string*"
ERROR 2348: Cannot alter superuser *string*'s default roles
ERROR 2349: Cannot alter superuser roles
ERROR 2389: Cannot create system built-in tuning rule
ERROR 2419: Cannot drop system built-in tuning rule
ERROR 2460: Cannot move user *string* to general pool, they lack privileges
ERROR 2481: Cannot remove memoryCap
ERROR 2482: Cannot remove runTimeCap
ERROR 2484: Cannot remove tempSpaceCap
ERROR 2812: Could not add location [*string*]: Permission denied
ERROR 2935: Couldn't nice(*value*) thread: *value*
ERROR 2953: Current password must be supplied to set new password
ERROR 2958: Current user can't change runtime priority of another
user's task
ERROR 2960: Current user doesn't have the privilege to change the task
runtime priority to be higher than its resource pool
ERROR 3577: Insufficient permissions on projection "*string*"
ERROR 3578: Insufficient permissions on schema "*string*"
ERROR 3579: Insufficient permissions on table "*string*"
ERROR 3580: Insufficient privilege: USAGE on SCHEMA '*string*' not granted
for current user
ERROR 3581: Insufficient privileges for projection *string*
ERROR 3582: Insufficient privileges for table *string*
ERROR 3583: Insufficient privileges on *string*
ERROR 3584: Insufficient privileges on *string*, modify privileges
(INSERT|UPDATE|DELETE|TRUNCATE) needed
ERROR 3722: Invalid passphrase: *string*
ERROR 3919: memoryCap of *value* KB would exceed user limit of *value* KB
ERROR 3989: Must be owner of *string string*
ERROR 3990: Must be owner of *string* [*string*]
ERROR 3991: Must be superuser to alter database
ERROR 3992: Must be superuser to alter profile
ERROR 3993: Must be superuser to alter tuning rule
ERROR 3994: Must be superuser to alter user default roles
ERROR 3998: Must be superuser to clear Query/EE profiles

ERROR 4000: Must be superuser to create interface
ERROR 4002: Must be superuser to create profile
ERROR 4003: Must be superuser to create subnet
ERROR 4004: Must be superuser to create tuning rule
ERROR 4005: Must be superuser to create users
ERROR 4006: Must be superuser to drop an interface
ERROR 4008: Must be superuser to drop profile
ERROR 4009: Must be superuser to drop resource pool
ERROR 4010: Must be superuser to drop role
ERROR 4011: Must be superuser to drop subnet
ERROR 4012: Must be superuser to drop tuning rule
ERROR 4013: Must be superuser to drop users
ERROR 4014: Must be superuser to modify resource pools
ERROR 4015: Must be superuser to rename interface
ERROR 4016: Must be superuser to rename profile
ERROR 4017: Must be superuser to rename role
ERROR 4018: Must be superuser to rename subnet
ERROR 4019: Must be superuser to run *string*
ERROR 4020: Must be superuser to run analyze_workload*string*()
ERROR 4059: New runTimeCap *value* ms would exceed user limit of *value* ms
ERROR 4061: New tempSpaceCap *value* KB would exceed user limit of *value* KB
ERROR 4178: Not enough privileges for projection *string*
ERROR 4179: Not enough privileges for table *string*
ERROR 4260: Only superuser can check privileges on other users
ERROR 4261: Only superuser can create roles
ERROR 4366: Permission denied
ERROR 4367: Permission denied for *string string*
ERROR 4368: Permission denied for *string* [*string*]
ERROR 4369: Permission denied to create temporary tables
ERROR 4546: RecvFiles on *string*: Can't write to file [*string*]
ERROR 4741: setThreadCPUNiceValue: couldn't nice(*value*) thread: *value*
ERROR 4742: setThreadIONiceValue: couldn't ionice(*value*) thread: *value*
ERROR 5389: User has insufficient privileges on schema *string*
ERROR 5488: Workspace schema *string* does not exist
ERROR 5517: Your Vertica license is invalid or has expired
ERROR 5618: Must be superuser to alter fault group
ERROR 5619: Must be superuser to create fault group
ERROR 5620: Must be superuser to drop fault group
ERROR 5635: Path to file [*string*] contains a symbolic link
ERROR 5716: Must have create permissions in schema *string* to drop type
ERROR 5956: Must be superuser to ALTER NODE
ERROR 5957: Must be superuser to create filesystem
ERROR 5958: Must be superuser to create location
ERROR 5959: Must be superuser to CREATE NODEs
ERROR 5961: Must be superuser to supply 'user_name' argument to HAS_ROLE() function

HINT: Non-superusers run HAS_ROLE('role_name')
ERROR 5975: Not enough privileges for *string*
ERROR 6125: Access Policies are not enabled
ERROR 6343: Must be superuser to *string* authentication
ERROR 6345: Must be superuser to alter authentication
ERROR 6348: Must be superuser to create authentication
ERROR 6350: Must be superuser to drop authentication
ERROR 6351: Must be superuser to rename authentication
ERROR 6514: Must be superuser to run *value* <name not available>
ERROR 6540: *string*: Invalid external table *string*
ERROR 6871: Must be superuser to rename user
ERROR 6872: Must be superuser to run move_statement_to_resource_pool
ERROR 6896: Parameter *string* can be cleared only via a UDx

ERROR 6897: Parameter *string* can be set only via a UDx
ERROR 6902: Permission denied on [*string.string*]
ERROR 6903: Permission denied, must be superuser to create a directed query
ERROR 6904: Permission denied, must be superuser to drop a directed query
ERROR 6905: Permission denied, must be superuser to save a query
ERROR 6906: Permission denied, must be superuser to update directed query status
ERROR 7376: Cannot close other user's sessions
ERROR 7463: Must be superuser to create notifier
ERROR 7464: Must be superuser to drop notifier
ERROR 7998: Permission denied for model *string*
ERROR 7999: Permission denied for schema *string*
ERROR 8569: Can't create the file [*string*]
ERROR 8855: Must be superuser to create load balance groups
ERROR 8856: Must be superuser to create routing rules
ERROR 8857: Must be superuser to drop a load balance group
ERROR 8858: Must be superuser to drop a routing rule
ERROR 8929: Alter address sub-command not recognized
ERROR 8935: Cannot override ImportExportTLSMode when set to *string*
ERROR 8958: Insufficient privileges on *string*, alter privileges needed
ERROR 8959: Insufficient privileges on *string*, drop privileges needed
ERROR 8963: Load balance group type "*string*" cannot have a filter set
ERROR 8964: Must be superuser to alter load balance group
ERROR 8965: Must be superuser to alter network address
ERROR 8966: Must be superuser to alter routing rule
ERROR 9085: Must be superuser to alter subcluster
ERROR 9086: Must be superuser to create subcluster
ERROR 9087: Must be superuser to drop subcluster
ERROR 9359: User may not have read privilege to the file: [*string*]
ERROR 9373: Must be superuser to alter other users
ERROR 9375: Must have superuser privileges to set configuration parameter *string* at the user level
ERROR 9662: Cannot set resource pool: user *string* lacks privileges on*string* resource pool *string*
ERROR 9959: Only database superusers can create external procedures
ERROR 9960: Only database superusers can drop external procedures
ERROR 9989: Must be superuser/dbadmin/sysmonitor to view other users
ERROR 10145: *string* "*string*" does not exist or the current user doesn't have access to it
ERROR 10310: Only superuser can debug a query
ERROR 10468: Must be superuser or have the UDXDEVELOPER role to create library
ERROR 10469: Must be superuser or have the UDXDEVELOPER role to drop library
ERROR 10514: Insufficient privileges on *string*; usage privliges needed
ERROR 10517: Must be superuser to execute ALTER PROCEDURE ... OWNER
ERROR 10554: Must be *string* to alter access policy
ERROR 10555: Must be *string* to copy access policy
ERROR 10556: Must be *string* to create access policy
ERROR 10557: Must be *string* to drop access policy
ERROR 10558: Only superusers or the owner of both tables can swap partitions with access policies
ERROR 10559: Only superusers or the source table owner can copy/move partitions with access policies

Notice messages

NOTICE 3583: Insufficient privileges on *string*

Warning messages

WARNING 5149: Unable to set role "*string*"
WARNING 5818: Deployment script will not be generated since the user does not have appropriate permissions to write to [*string*]
WARNING 5820: Design script will not be generated since the user does not have appropriate permissions to write to [*string*]

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42601

This topic lists the messages associated with the SQLSTATE 42601.

SQLSTATE 42601 description

SYNTAX_ERROR

Messages associated with this SQLState

Error messages

ERROR 2005: *string*
ERROR 2069: '*string*' is not a table name in the current search_path
ERROR 2085: A column cannot occur in an equality predicate and an interpolation predicate
ERROR 2093: A join can have only one set of interpolated predicates
ERROR 2100: A query with Time Series Aggregate Function *string* must have a timeseries clause
ERROR 2156: All columns are evaluated by expressions. At least one column should be read from input
ERROR 2157: All columns in select list must be columns used by projection
ERROR 2164: Alter Column Type driver: Unrecognized command type
ERROR 2180: Analytic function *string* must have an OVER clause
ERROR 2187: Analytic functions cannot be nested
ERROR 2191: ANALYZE CONSTRAINT is not supported
ERROR 2203: Anchor table not found
ERROR 2214: Argument *value* has invalid type *value* in ANALYZE_WORKLOAD
ERROR 2215: Argument *value* in ANALYZE_WORKLOAD must be constant
ERROR 2223: Argument in ANALYZE_CONSTRAINTS must be constant
ERROR 2230: Arguments of row IN must all be row expressions
ERROR 2346: Cannot alter a sequence with START
ERROR 2381: Cannot create a sequence with RESTART
ERROR 2444: Cannot insert into or update IDENTITY/AUTO_INCREMENT column "*string*"
ERROR 2445: Cannot insert into system column "*string*"
ERROR 2446: Cannot insert multiple commands into a prepared statement
ERROR 2521: Cannot specify anything other than user defined transforms *string* in the *string* list
ERROR 2525: Cannot specify more than one user-defined transform function in the SELECT list
ERROR 2526: Cannot specify more than one window clause with a user defined transform
ERROR 2534: Cannot use "PR" with "S"/"PL"/"MI"/"SG"
ERROR 2535: Cannot use "S" with "MI"
ERROR 2536: Cannot use "S" with "PL"
ERROR 2537: Cannot use "S" with "PL"/"MI"/"SG"/"PR"
ERROR 2538: Cannot use "S" with "SG"
ERROR 2539: Cannot use "V" with a decimal point
ERROR 2545: Cannot use aggregate function in VALUES
ERROR 2627: Column "*string*" in ENCODED BY clause is not found in the table

ERROR 2641: Column "*string.string*" must appear in the PARTITION BY list of Timeseries clause or be used in a Time Series Aggregate Function
ERROR 2642: Column *string* cannot be evaluated
ERROR 2645: Column *string* has other computed columns in its expression
ERROR 2647: Column *string* in ORDER BY list is not found in TABLE
ERROR 2659: Column alias list for "*string*" has too many entries
ERROR 2669: COLUMN OPTION is not supported
ERROR 2670: Column options are not supported
ERROR 2696: Conflicting INTERVAL subtypes
ERROR 2697: Conflicting NULL/NOT NULL declarations for column "*string*" of table "*string*"
ERROR 2699: Conflicting or redundant options
ERROR 2712: Constraint "*string*" for relation "*string*" already exists
ERROR 2715: Constraint declared INITIALLY DEFERRED must be DEFERRABLE
ERROR 2754: COPY requires a data source; either a FROM clause or a WITH SOURCE for a user-defined source
ERROR 2764: COPY: Expression for column *string* cannot be coerced
ERROR 2946: CREATE TABLE AS specifies too many column names
ERROR 2947: CREATE VIEW specifies more column names than columns
ERROR 2986: Database name is required (too few dotted names): *string*
ERROR 3023: Default values specified for IDENTITY/AUTO_INCREMENT column "*string*" of table "*string*"
ERROR 3125: Drop Column driver: Unrecognized command type
ERROR 3142: Duplicate column "*string*" in create table statement
ERROR 3143: Duplicate column *string* in constraint
ERROR 3151: Duplicate tables in projection not allowed
ERROR 3155: Duplicated parameters *string* not allowed
ERROR 3158: Each *string* query must have the same number of columns
ERROR 3164: Empty column name is invalid
ERROR 3165: Empty constraint name is invalid
ERROR 3171: ENCODED BY is not supported in CREATE PROJECTION statement when column renaming list is defined
ERROR 3172: ENCODED BY is not supported in CREATE PROJECTION statement with column definition list
ERROR 3173: ENCODED BY is not supported in CREATE TABLE AS SELECT statement when column list is defined
ERROR 3176: End epoch (*value*) number out of range
ERROR 3177: End epoch (*value*) precedes start epoch (*value*)
ERROR 3183: Epoch number out of range
ERROR 3185: Epoch time out of range
ERROR 3344: EXPORT ... SELECT may not specify INTO
ERROR 3348: Expression "(<*string*> - <*string*>) <interval qualifier>" is not supported
ERROR 3349: Expression for column *string* cannot be coerced
ERROR 3458: Function *string* is not allowed in Time Series queries
ERROR 3461: Function *string* requires at least one argument
ERROR 3500: HAVING / GROUP BY not allowed with Time Series query
ERROR 3511: IGNORE NULLS can only be used with FIRST_VALUE or LAST_VALUE or NTH_VALUE
ERROR 3517: Improper %*value* reference (too few dotted names): *string*
ERROR 3518: Improper %*value* reference (too many dotted names): *string*
ERROR 3519: Improper qualified column name: *string*
ERROR 3520: Improper qualified name (too many dot): *string*
ERROR 3521: Improper qualified name (too many dots): *string*
ERROR 3522: Improper qualified name (too many dotted names): *string*
ERROR 3523: Improper relation name (too many dotted names): *string*
ERROR 3538: Incorrect parameter type provided: *string* is supposed to be of type *string*
ERROR 3548: Indirection is not allowed in a target column

ERROR 3549: Indirection is not allowed in the name of a FILLER column
ERROR 3571: INSERT ... SELECT may not specify INTO
ERROR 3572: INSERT has more expressions than target columns
ERROR 3573: INSERT has more target columns than expressions
ERROR 3599: Interpolated predicates are allowed only in ON CLAUSE of ANSI Join syntax
ERROR 3602: Interpolated predicates should refer to columns from both relations of the join
ERROR 3615: INTO is only allowed on first SELECT of UNION/INTERSECT/EXCEPT
ERROR 3619: Invalid argument type *value* in ANALYZE_CONSTRAINTS
ERROR 3672: Invalid hexadecimal number at or near "*string*"
ERROR 3709: Invalid number at or near "*string*"
ERROR 3775: Invalid Unicode escape character '*character*'
ERROR 3776: Invalid Unicode hex number "*string*"
ERROR 3812: Join condition in merge query must include at least one table attribute
ERROR 3865: LIMIT #,# syntax is not supported
ERROR 3944: Misplaced DEFERRABLE clause
ERROR 3945: Misplaced INITIALLY DEFERRED clause
ERROR 3946: Misplaced INITIALLY IMMEDIATE clause
ERROR 3947: Misplaced NOT DEFERRABLE clause
ERROR 3949: Missing argument
ERROR 3958: Missing savepoint name
ERROR 3976: Multiple assignments to same column "*string*"
ERROR 3978: Multiple decimal points
ERROR 3979: Multiple default values specified for column "*string*" of table "*string*"
ERROR 3980: Multiple DEFERRABLE/NOT DEFERRABLE clauses not allowed
ERROR 3981: Multiple FOR UPDATE clauses are not allowed
ERROR 3982: Multiple INITIALLY IMMEDIATE/DEFERRED clauses not allowed
ERROR 3984: Multiple LIMIT clauses are not allowed
ERROR 3985: Multiple OFFSET clauses are not allowed
ERROR 3986: Multiple ORDER BY clauses are not allowed
ERROR 4023: Must specify memorySize parameter
ERROR 4024: Must specify one new name for each schema
ERROR 4025: Must specify one new name for each table
ERROR 4026: Must specify one new name for each view
ERROR 4062: NEW used in query that is not in a rule
ERROR 4066: No actions specified
ERROR 4070: No columns specified in select list
ERROR 4072: No constraints defined
ERROR 4105: No second argument needed when analyzing all constraints
ERROR 4136: Node "*string*" does not exist
ERROR 4161: Non-integer constant in *string*
ERROR 4164: Nonexistent columns: '*string*'
ERROR 4203: Number of columns defined in CREATE TABLE statement is less than in SELECT query output
ERROR 4204: Number of columns defined in CREATE TABLE statement is more than in SELECT query output
ERROR 4205: Number of columns in the PROJECTION statement must be the same as the number of columns in the SELECT statement
ERROR 4225: OLD used in query that is not in a rule
ERROR 4227: ON COMMIT clause may only be specified for TEMPORARY tables
ERROR 4237: Only a single "S" is allowed
ERROR 4239: Only ASC is allowed in ORDER BY list of auto projection for CREATE TABLE
ERROR 4240: Only columns are allowed in ORDER BY list of auto projection for CREATE TABLE
ERROR 4241: Only columns are allowed in SELECT list of projection

ERROR 4247: Only inner joins are allowed in a projection defining query
ERROR 4268: Only tables are allowed in FROM clause of projection
ERROR 4291: Operator too long at or near "*string*"
ERROR 4294: Option *string* conflicts with prior options
ERROR 4296: Options not set
ERROR 4297: ORDER BY column in timeseries OVER clause must be Timestamp type
ERROR 4325: Parameters can only contain constants or constant expressions
ERROR 4327: Parsing error "*string*" at or near "*string*"
ERROR 4328: PARTITION AUTO can only be used with single-phase user defined transform functions
ERROR 4348: Path Number must be in [0, *value*]
ERROR 4350: Pattern "0" must come before "PR"
ERROR 4351: Pattern "9" must come before "PR"
ERROR 4383: plannedConcurrency must be greater than 0
ERROR 4487: Projections can only be sorted in ascending order
ERROR 4669: SELECT * with no tables specified is not valid
ERROR 4670: SELECT DISTINCT ON is not standard SQL, use just SELECT DISTINCT
ERROR 4706: Sequence functions accept constant strings arguments only
ERROR 4732: Set Operators are not allowed in a projection
ERROR 4761: Sort key *string* should be in the target list
ERROR 4828: Subquery has too few columns
ERROR 4829: Subquery has too many columns
ERROR 4831: Subquery in FROM may not have SELECT INTO
ERROR 4833: Subquery in FROM must have an alias
ERROR 4835: Subquery must return a column
ERROR 4836: Subquery must return only one column
ERROR 4837: Subquery not allowed in a projection
ERROR 4838: Subquery not allowed in SELECT list and/or ORDER BY clause for Time Series queries
ERROR 4855: Syntactic Optimizer requires joins written using ANSI JOIN syntax
ERROR 4856: Syntax error at or near "*string*"
ERROR 4947: The foreign key in this constraint has already been defined as a foreign key for relation "*string*"
ERROR 4955: The number of target columns (*value*) does not match the number of columns (*value*) in the EXPORT statement
ERROR 4956: The number of target columns (*value*) is less than the number of columns (*value*) in the EXPORT statement
ERROR 5007: Time Series Aggregate Functions cannot be nested
ERROR 5008: Time Series queries cannot refer to column of outer query
ERROR 5009: Time Series queries cannot refer to column of outer query: "*string*.*string*"
ERROR 5011: Time slice length must be a positive integer constant
ERROR 5012: Time slice length must be an interval constant
ERROR 5161: Unequal number of entries in row expression
ERROR 5162: Unequal number of entries in row expressions
ERROR 5272: Unsupported From clause expression
ERROR 5285: Unsupported SET option
ERROR 5286: Unsupported SET option *string*
ERROR 5287: Unsupported SHOW option *string*
ERROR 5290: Unsupported transaction option *string*
ERROR 5305: Unterminated /* comment at or near "*string*"
ERROR 5306: Unterminated bit string literal at or near "*string*"
ERROR 5307: Unterminated dollar-quoted string at or near "*string*"
ERROR 5308: Unterminated hexadecimal string literal at or near "*string*"
ERROR 5310: Unterminated quoted identifier at or near "*string*"

ERROR 5311: Unterminated quoted string at or near "*string*"
ERROR 5323: Usage: clear_profiling(*string* , *string*)
ERROR 5324: Usage: disable_profiling(*string*)
ERROR 5325: Usage: enable_profiling(*string*)
ERROR 5326: Use "*string*(*)" to call this aggregate function
ERROR 5383: User Defined Transform Functions are allowed only in a SELECT list
ERROR 5386: User defined transform will return *value* columns, whereas *value* aliases provided
ERROR 5401: User-defined transform function *string* must have an OVER clause
ERROR 5413: Value must be either "units" or "plain"
ERROR 5415: Value must be either ON or OFF
ERROR 5452: Virtual tables are not allowed in FROM clause of projection
ERROR 5492: Wrong number of parameters for prepared statement "*string*"
ERROR 5493: Wrong number of parameters on left side of OVERLAPS expression
ERROR 5494: Wrong number of parameters on right side of OVERLAPS expression
ERROR 5505: You can specify a node name only once in a create projection statement, node *string* appears more than once
ERROR 5518: Zero-length delimited identifier at or near "*string*"
ERROR 5566: Dimension tables may not have data that shorter lived than the fact table
ERROR 5577: Expression for user-defined type column *string* cannot be coerced
ERROR 5600: Invalid predicate in projection-select. Only PK=FK equijoins are allowed
ERROR 5617: Multiple WITH clauses not allowed
ERROR 5629: Not a Star or Snow-Flake Query
ERROR 5630: Nullable FKs are not allowed in projection definition
ERROR 5664: Subqueries not allowed in projection definition
ERROR 5670: The number of alias columns must be the same as the number of selected columns
ERROR 5679: Unrecognized order by expression
ERROR 5691: User-defined function *string* is not a supported scalar function
ERROR 5711: Invalid function arguments
ERROR 5916: If specified, maximum error percentage must be a numeric constant
ERROR 5929: Invalid maximum error percentage specified
ERROR 6034: Syntax Error: '*string*' is a built in type
ERROR 6048: The minimum value that may be specified for maximum error percentage is 0.88
ERROR 6061: Too many arguments to *string*
ERROR 6119: 'deleted' hint takes no arguments
ERROR 6120: 'latestdata' hint arguments must be strings
ERROR 6124: A single argument must be supplied to *string*
ERROR 6136: Aggregate projection without group-by columns is not supported
ERROR 6137: Aggregate projections must have at least one aggregate (SUM, COUNT, MIN, or MAX)
ERROR 6140: All sort keys should be in the target list of the projection
ERROR 6146: At least two columns must be specified
ERROR 6196: Columns in ORDER-BY must be defined in the SELECT statement
ERROR 6197: Columns/Expressions in GROUP BY/PARTITION BY must be first and in the same order with columns in SELECT
ERROR 6198: ORDER BY columns/expressions in the OVER() clause must be

the first SELECT columns/expressions not specified by PARTITION BY clause, and must be specified in SELECT list order

ERROR 6199: Columns/Expressions in the PARTITION BY clause may not be repeated in the ORDER BY clause

ERROR 6232: DISTINCT Aggregates are not allowed in projection

ERROR 6293: Interpolated join predicates cannot have expressions or functions over join columns

ERROR 6294: Invalid Argument

ERROR 6321: Limit in Top-K query must be a positive number

ERROR 6324: Limit/Offset is only allowed in topk projections

ERROR 6340: Multiple ORDER BY clauses in partition TopK/Limit query are not allowed

ERROR 6341: Multiple PARTITION clauses in TopK/Limit query are not allowed

ERROR 6367: Only one table or projection is allowed in FROM clause of top-k projection

ERROR 6369: Only SUM, MIN, MAX and COUNT are allowed in aggregate projections

ERROR 6372: ORDER BY is not allowed in aggregate projection. The aggregate projection is automatically ordered on group by columns

ERROR 6373: ORDER BY is not allowed in top-k projections. The top-k projection is automatically ordered on partition by columns

ERROR 6379: PARTITION BEST can only be used with single-phase user defined transform functions

ERROR 6381: PARTITION NODES can only be used with single-phase user defined transform functions

ERROR 6394: SEGMENTED BY / UNSEGMENTED is not allowed in aggregate projection. The aggregate projection is automatically segmented on group by columns

ERROR 6395: SEGMENTED BY / UNSEGMENTED is not allowed in top-k projections. The top-k projection is automatically segmented on partition by columns

ERROR 6539: User Defined Transform *string* returns a string on which collation can't be applied in non default locale

ERROR 6543: Grouping functions only allowed with aggregates

ERROR 6544: MultiLevel Aggregates are not allowed in projection

ERROR 6640: Argument of hint *string* must be a positive integer

ERROR 6654: Batch/Prepass may only be used in the over(...) clause of a user defined transform in a Live Aggregate Projection

ERROR 6686: Cannot specify LIMIT with OVER(...) clause with a user defined transform

ERROR 6696: Column "*string*.*string*" refers to table "*string*" which is out of scope

ERROR 6697: Columns/Expressions in the partition clause have to appear in the SELECT list

ERROR 6701: Constants with the same parameter hint must be identical

ERROR 6800: Hint *string* must have one argument

ERROR 6853: Live aggregate projections may only be one of the following types: Group-by, Top-K, or UDT

ERROR 6878: No previously saved query to associate with

ERROR 6899: Parameters cannot have NULL values

ERROR 6969: Subqueries for aggregate projections need to contain User Defined Transforms

ERROR 6972: Syntax Error. Struct member '*string*' has missing base type

ERROR 7014: UD Parameters value length [*value*] is more than the max allowed length [*value*]

ERROR 7051: Unsupported hint for input expr

ERROR 7109: Parameter '*string*' is required

ERROR 7147: Option *string* is repeated. Please specify each option only

once
ERROR 7153: Syntax Error. At least one parameter has to be specified
ERROR 7164: Cannot use ORC format with FLEX tables
ERROR 7228: Cannot use *string* compression with *string* files
ERROR 7229: Cannot use PARQUET format with FLEX tables
ERROR 7327: SET USING cannot be specified for IDENTITY/AUTO_INCREMENT column "*string*" of table "*string*"
ERROR 7391: Cannot use an UDSource with *string* files
ERROR 7420: Expression list has more than one *string* column
ERROR 7488: Number of key/value pairs [*value*] exceeds the maximum allowed
ERROR 7538: Sequence manipulation functions not supported in INSERT SELECT with implicit column that has default query
ERROR 7744: Try local hint only supports SELECT queries without table join
ERROR 7746: Try local hint requires all nodes to be up
ERROR 7755: This projection is not valid with "try_local" hint
ERROR 7875: SET USING/DEFAULT cannot be specified for IDENTITY/AUTO_INCREMENT column "*string*" of table "*string*"
ERROR 7909: Columns/Expressions in PARTITION BY must be first and in the same order with columns in SELECT
ERROR 7965: At most one WHEN MATCHED clause is allowed in MERGE
ERROR 7966: At most one WHEN NOT MATCHED clause is allowed in MERGE
ERROR 7967: MERGE/INSERT has more expressions than target columns
ERROR 7968: MERGE/INSERT has more target columns than expressions
ERROR 7982: Cannot use expressions in OVER(...) clause
ERROR 7989: Invalid column in MERGE statement: *string*. The insert filter of a WHEN NOT MATCHED clause cannot reference the target table *string*
ERROR 8011: SELECT expressions must have column labels
ERROR 8114: Multiple SET USING values specified for column "*string*" of table "*string*"
ERROR 8116: No '=' found in '*string*', while parsing for *string*
ERROR 8164: Subquery in MERGE statement are allowed at SOURCE only
ERROR 8205: Hint *string* cannot be used on NULL
ERROR 8956: If provided synopsis_version must be 2
ERROR 8961: Invalid synopsis_version specified
ERROR 8989: synopsis_version must be a numeric constant
ERROR 9015: Sequence manipulation functions allowed only in outer SELECT list
ERROR 9016: Sequence manipulation functions not allowed in subqueries
ERROR 9040: Cannot add new sort order to an existing projection
ERROR 9060: Mismatch with the existing sort order
ERROR 9062: No changes made to the existing sort order
ERROR 9362: Only a table or projection is allowed in the FROM clause of a top-k projection
ERROR 9363: Only a table or projection is allowed in the FROM clause of an aggregate projection
ERROR 9364: Only one table or projection is allowed in the FROM clause of an aggregate projection
ERROR 9365: Only table or projection is allowed in FROM clause of aggregate projection
ERROR 9579: Invalid AES key. AES keys should be supplied as a hex string
ERROR 9582: Key length [*value*] is invalid. Supported AES key lengths are 128, 192, and 256
ERROR 9583: Key type [*string*] is not supported
ERROR 9591: RSA keys with passwords are not currently supported
ERROR 9642: Subquery must return only one scalar column
ERROR 9714: Syntax error: keyword SECONDARY cannot be used in CREATE SUBCLUSTER LIKE
ERROR 9720: Top-level alias name missing

ERROR 9730: Top-level alias name missing
ERROR 9759: Invalid bound for collection type *string*
ERROR 9788: Recursive query "*string*" does not have the form non-recursive-term UNION [ALL] recursive-term
ERROR 9789: Recursive query "*string*" must not contain data-modifying statements
ERROR 9790: Recursive reference to query "*string*" must not appear more than once
ERROR 9791: Recursive reference to query "*string*" must not appear within a subquery
ERROR 9792: Recursive reference to query "*string*" must not appear within an outer join
ERROR 9793: Recursive reference to query "*string*" must not appear within EXCEPT
ERROR 9794: Recursive reference to query "*string*" must not appear within INTERSECT
ERROR 9795: Recursive reference to query "*string*" must not appear within its non-recursive term
ERROR 9868: LIMIT OVER clause is not allowed in Time Series queries
ERROR 10140: Negative array index *value*
ERROR 10159: Cannot *string* labelled non-loop blocks
ERROR 10160: Cannot *string* unlabelled non-loop blocks
ERROR 10175: Could not find *string*'s target block
ERROR 10185: FETCH statement cannot return multiple rows
ERROR 10188: FOR (RANGE) target name cannot include array indices
ERROR 10189: FOR (RANGE) target name cannot include labels
ERROR 10190: Found no SQL expression where one was required
ERROR 10191: Found unexpected SQL expression
ERROR 10194: IF statements require exactly one condition
ERROR 10196: Invalid TLS Verify mode [*string*]
ERROR 10203: Parameter with the name "*string*" has already been declared
ERROR 10209: SQLSTATE digits may only be numbers or capital letters
ERROR 10210: SQLSTATE must be exactly 5 characters long
ERROR 10221: Variable with the name "*string*" already exists in the same scope
ERROR 10255: Instantiation of nonexistent parameter "*string*"
ERROR 10257: Invalid exception condition name
ERROR 10269: Redundant instantiation of parameter "*string*"
ERROR 10298: Variable "*string*" must be initialized, since it's declared NOT NULL
ERROR 10313: WITH clause query may not have SELECT INTO
ERROR 10325: FOR (RANGE) target must be a single variable
ERROR 10331: Query returned *value* columns for assignment where *value* expected
ERROR 10345: CONTEXT is not yet supported
ERROR 10352: For '*string*', nested array columns are not supported
ERROR 10356: ROW_COUNT is not yet supported
ERROR 10385: Query returned *value* columns where 1 was expected
ERROR 10409: Cursor variables ("*string*") cannot be assigned into
ERROR 10414: Mismatch between the number of format parameters (*value*) and the number of supplied arguments (*value*)
ERROR 10415: Mismatch between the number of parameters (*value*) and the number of supplied arguments (*value*)
ERROR 10424: The option *string* is only available for GET CURRENT DIAGNOSTICS
ERROR 10425: The option *string* is only available for GET STACKED DIAGNOSTICS
ERROR 10448: PL/vSQL parser failed at *string* of source string: *string*
ERROR 10456: TYPE-aliased variable must name a PL/vSQL variable or a SQL column
ERROR 10460: Cannot specify 'CREATE OR REPLACE' and 'IF NOT EXISTS' in

ERROR 10460: Cannot specify 'CREATE OR REPLACE' and 'IF NOT EXISTS' in the same create function statement
ERROR 10461: Cannot specify 'CREATE OR REPLACE' and 'IF NOT EXISTS' in the same create procedure statement
ERROR 10534: Error encountered while tuning annotated query: *string*
ERROR 10535: Error encountered while verifying target query: *string*
ERROR 10536: No '=' found in '*string*', while parsing
ERROR 10540: Tuning doesn't support the action '*string*'
ERROR 10561: MATCH_COLUMNS() can either be specified as an element in a SELECT list, or passed in to a function that allows an unlimited number of arguments
ERROR 10613: Password with provided salt cannot be processed: *string*
ERROR 10625: DEFAULT can only appear once in the FROM clause of an UPDATE
ERROR 10627: Syntax error near "DEFAULT"
ERROR 10629: Argument '*string*' was found more than once while parsing
ERROR 10635: Invalid key-value pair: the correct format is 'key1=value1;key2=value2'
ERROR 10636: Invalid keyword: '*string*'
ERROR 10651: Queries with syntactic hints are not supported

Warning messages

WARNING 2030: *string* has been deprecated as *string string* Vertica option
WARNING 2238: At least two arguments are required
WARNING 2239: At most one path number can be entered
WARNING 3706: Invalid node name in hint
WARNING 3738: Invalid projection name in hint: *string*
WARNING 3841: Label can accept only one argument
WARNING 3959: Missing the path number
WARNING 5524: A projection can have only one basename
WARNING 5525: A projection can have only one createtype
WARNING 5714: Missing the random seed
WARNING 5730: The second argument, sampling method, should be always be 1 -- naive sampling(biased)
WARNING 5733: The third argument must be large than 0
WARNING 5734: Three arguments at most: sampling seed, sampling method (optional, default 1), sampling size (optional,default 10)
WARNING 6122: A projection can replace only one original
WARNING 6518: At most one projection offset number can be entered
WARNING 6562: Hint PROJS cannot be used with hint SKIP_PROJS for the same table instance. Hint PROJS will be ignored
WARNING 6563: Hint SKIP_PROJS cannot be used with hint PROJS for the same table instance. Hint SKIP_PROJS will be ignored
WARNING 6568: Invalid projection name in hint: *string*. The whole hint will be ignored
WARNING 6834: Invalid use of _oidrefs hint
WARNING 7750: WHERE clause syntax is incompatible with "try local" hint, hint is ignored
WARNING 8747: At least one argument is required

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42602

This topic lists the messages associated with the SQLSTATE 42602.

SQLSTATE 42602 description

INVALID_NAME

Messages associated with this SQLState

Error messages

ERROR 2278: Built-in function with the same name already exists: *string*
ERROR 2383: Cannot create projections due to naming conflicts with existing projections
ERROR 3378: Failed to generate a unique relation or sequence name
ERROR 3674: Invalid identifier name (*value* octets) "*string*"
ERROR 3703: Invalid name syntax
ERROR 3747: Invalid savepoint identifier *string*
ERROR 4159: Non-ASCII characters in names are prohibited
ERROR 4267: Only table column names and filler column names can appear in the list of columns to copy
ERROR 4451: Projection "*string*" does not exist
ERROR 4506: Query weight must be positive
ERROR 5360: User "*string*" does not exist
ERROR 5403: User/role "*string*" already exists
ERROR 5569: Either column "*string*" does not exist or table alias "*string*" is not allowed in "WHEN MATCHED THEN UPDATE SET"
ERROR 5769: Cannot drop the main vertica license
ERROR 5968: No such license *string* to drop
ERROR 5970: Node *string* is not a control node
ERROR 6089: Unknown control node *string*
ERROR 6676: Cannot drop global heir user "*string*". Try changing / clearing the GlobalHeirUsername config parameter
ERROR 7117: Refresh table name error: *string*
ERROR 8304: Paramter has the same name as argument: *string*
ERROR 8403: Cannot drop the only AutoPass license
ERROR 8642: Invalid model name: *string*
ERROR 9284: Unsupported argument type [*value*] for argument *string*
ERROR 9306: Improper qualified name: *string*
ERROR 9547: Control node *string* is not in the same subcluster as node *string*
ERROR 9549: Node *string* is not in the same subcluster as the new node
ERROR 9744: Unknown node *string*
ERROR 9932: Node *string* has invalid IP address
ERROR 10548: Tuning session name may contain only alphanumeric or underscore characters

Warning messages

WARNING 2398: Cannot determine the best encoding options for some columns in table *string*.*string* due to insufficient data
WARNING 3059: DEPRECATED syntax. Segment expression "*string*" is a projection column name, segmenting on attribute "*string*"*stringstringstring* instead

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42611

This topic lists the messages associated with the SQLSTATE 42611.

SQLSTATE 42611 description

INVALID_COLUMN_DEFINITION

Messages associated with this SQLState

Error messages

ERROR 7342: *string* expression for column "*string*" may not refer to itself
ERROR 7389: Cannot set *string* for column "*string*" since it is referenced in default expression of column "*string*"
ERROR 7783: Cannot set *string* for column "*string*" since it is referenced in *string* expression of column "*string*"

Warning messages

WARNING 6099: Using LONG column '*string*' in a constraint
WARNING 6100: Using PARTITION expression that returns a *string* value
WARNING 6101: Using PARTITION expression that returns a LONG value
WARNING 6102: Using PARTITION expression that returns a LONG value: '*string*'

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42622

This topic lists the messages associated with the SQLSTATE 42622.

SQLSTATE 42622 description

NAME_TOO_LONG

Messages associated with this SQLState

ERROR 2462: Cannot open FileColumn because path is too long *string*
ERROR 3507: Identifier "*string*" is *value* octets long. Maximum limit is *value* octets
ERROR 10547: Tuning session name cannot have more than *value* characters

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42701

This topic lists the messages associated with the SQLSTATE 42701.

SQLSTATE 42701 description

DUPLICATE_COLUMN

Messages associated with this SQLState

ERROR 2629: Column "*string*" is already of type "*string*"
ERROR 2638: Column "*string*" specified more than once
ERROR 2654: Column *string* specified more than once
ERROR 2655: Column *string* specified more than once in options list
ERROR 2662: Column name "*string*" already exists
ERROR 2663: Column name "*string*" appears more than once in USING clause
ERROR 2664: Column name "*string*" does not exist
ERROR 3144: Duplicate column *string* in ORDER BY list
ERROR 3145: Duplicate column name
ERROR 3150: Duplicate projection column name (projection: *string*)
ERROR 3154: Duplicated parameter "*string*" in parameter list
ERROR 5450: View definition can not contain duplicate column names "*string*"
ERROR 5878: Failed to create table *string*: duplicate column name *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42702

This topic lists the messages associated with the SQLSTATE 42702.

SQLSTATE 42702 description

AMBIGUOUS_COLUMN

Messages associated with this SQLState

ERROR 2604: Clause *string* "*string*" is ambiguous
ERROR 2671: Column reference "*string*" is ambiguous
ERROR 2681: Common column name "*string*" appears more than once in left table
ERROR 2682: Common column name "*string*" appears more than once in right table
ERROR 5904: Flex table "*string*" has no internal "*string*" column
ERROR 6144: Ambiguous column reference in projection
ERROR 9649: Field reference "*string.string*" is ambiguous
ERROR 10341: Column reference *string* is ambiguous
ERROR 10342: Column reference *string.string* is ambiguous
ERROR 10343: Column reference *string.string.string* is ambiguous
ERROR 10344: Column reference *string.string.string.string* is ambiguous

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42703

This topic lists the messages associated with the SQLSTATE 42703.

SQLSTATE 42703 description

UNDEFINED_COLUMN

Messages associated with this SQLState

ERROR 2359: Cannot assign to field "*string*" of column "*string*" because there
    is no such column in data type *string*
ERROR 2360: Cannot assign to system column "*string*"
ERROR 2624: Column "*string*" does not exist
ERROR 2625: Column "*string*" does not exist;

    Vertica does not support 'SELECT <table_name> FROM
    <table_name>'
ERROR 2633: Column "*string*" named as primary key does not exist
ERROR 2634: Column "*string*" not found in data type *string*
ERROR 2635: Column "*string*" of relation "*string*" does not exist
ERROR 2636: Column "*string*" specified in USING clause does not exist in
    left table
ERROR 2637: Column "*string*" specified in USING clause does not exist in
    right table
ERROR 2643: Column *string* does not exist
ERROR 2644: Column *string* does not exist in table
ERROR 2651: Column *string* must be loaded or computed
ERROR 2656: Column *string*.*string* does not exist
ERROR 2664: Column name "*string*" does not exist
ERROR 2870: Could not identify column "*string*" in record data type
ERROR 4450: Projection "*string*" does not contain column "*string*"
ERROR 6695: Column "*string*"."*string*" must have CHAR, VARCHAR, LONG VARCHAR,
    VARBINARY, LONG VARBINARY, or USER DEFINED type
ERROR 7009: Tokenizer must have a "*string*" column output for stemming
ERROR 7167: Column "*string*" does not exist on table "*string*"
ERROR 7367: After excluding columns, no input column remains
ERROR 7959: Column [*string*] does not exist in relation [*string*]
ERROR 7984: Column [*string*] is duplicated in parameter *string*
ERROR 9057: Field "*string*" does not exist

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42704

This topic lists the messages associated with the SQLSTATE 42704.

SQLSTATE 42704 description

UNDEFINED_OBJECT

Messages associated with this SQLState

ERROR 2067: '*string*' is not a known granularity for audits.

    string
ERROR 2068: '*string*' is not a known TM task.

    string
ERROR 2070: '*string*' is not a valid granularity for *string*.

    string
ERROR 2073: '*string*' is not supported by index tool
ERROR 2274: Bootstrap error (most likely in Bootstrap.cpp):
    Unregistered name *string*
ERROR 2275: Bootstrap error (most likely in Bootstrap.cpp):
    Unregistered oid *value*
ERROR 2710: Constraint "*string*" does not exist
ERROR 2711: Constraint "*string*" does not exist on table "*string*"
ERROR 2854: Could not find array type for data type *string*

ERROR 2654: Could not find array type for data type *string*
ERROR 2983: Database "*string*" does not exist
ERROR 3000: DDL interfered with this statement
ERROR 3001: DDL statement interfered with *string*.nextval
ERROR 3007: DDL statement interfered with this statement
ERROR 3256: Error reported by client: *string*
ERROR 3442: Found eligible *value* processes to invite, but no matching nodes in catalog
ERROR 3637: Invalid Component Name '*string*'
ERROR 3655: Invalid epoch range
ERROR 3698: Invalid mergeout task identifier (Possible values are: [0, *value*])
ERROR 3715: Invalid object name
ERROR 3748: Invalid scope in ANALYZE_WORKLOAD*string*: schema or table *string* was altered
ERROR 3749: Invalid scope in ANALYZE_WORKLOAD: schema or table *string* does not exist
ERROR 3756: Invalid Sub-Component Name '*string*'
ERROR 3769: Invalid TM operation
ERROR 3779: Invalid user ID: *value*
ERROR 3842: Language does not exist: *string*
ERROR 3855: Library "*string*" does not exist
ERROR 3862: Library with name '*string*' does not exist
ERROR 4046: Network Interface "*string*" does not exist
ERROR 4047: Network Interface "*string*" is setup on another node
ERROR 4101: No role "*string*" exists
ERROR 4109: No storages in the specified epoch range
ERROR 4110: No such node *string*
ERROR 4111: No such object
ERROR 4112: No such projection
ERROR 4113: No such projection '*string*'
ERROR 4123: No user or role "*string*" exists
ERROR 4129: No value found for parameter "*string*"
ERROR 4130: No value found for parameter *value*
ERROR 4136: Node "*string*" does not exist
ERROR 4137: Node *string* does not exist
ERROR 4216: Object '*string*' is not a projection
ERROR 4217: Object '*string*' is not a table
ERROR 4218: Object '*string*' is not a table or projection
ERROR 4223: OID *value* is not a sequence
ERROR 4224: OID *value* is not a Table or a View
ERROR 4446: Profile "*string*" does not exist
ERROR 4447: Profile '*string*' does not exist
ERROR 4451: Projection "*string*" does not exist
ERROR 4594: Resource pool "*string*" does not exist
ERROR 4596: Resource pool '*string*' does not exist
ERROR 4614: Role "*string*" does not exist
ERROR 4616: Role "*string*" not found
ERROR 4650: Schema "*string*" does not exist
ERROR 4656: Schema, table, or projection "*string*" does not exist.

string
ERROR 4697: Sequence "*string*" does not exist
ERROR 4713: Sequence with name '*string*' does not exist
ERROR 4806: Subnet "*string*" does not exist
ERROR 4876: Table "*string*" does not exist
ERROR 4926: The *string* "*string*" does not exist
ERROR 4928: The *string* ["*string*"] does not exist
ERROR 5105: Tuning rule "*string*" does not exist
ERROR 5108: Type "*string*" does not exist

ERROR 5115: Type with OID *value* does not exist
ERROR 5227: Unrecognized drop object type: *value*
ERROR 5360: User "*string*" does not exist
ERROR 5362: User or Role "*string*" not found
ERROR 5365: User available location ["*string*"] does not exist on node ["*string*"]
ERROR 5446: View "*string*" does not exist
ERROR 5459: Window "*string*" does not exist
ERROR 5532: Can not find any eligible locations in tier *string*
ERROR 5585: Fault Group "*string*" does not exist
ERROR 5614: Library *string* does not exist
ERROR 5797: Could not find the JVM resource pool
ERROR 5913: HCatalog database *string* does not exist
ERROR 5931: Invalid Policy Name '*string*'
ERROR 5965: New node cannot be placed in a non-existent Fault Group "*string*"
ERROR 5967: No database found. Create/use a database before start using DFS
ERROR 5969: No table or projection named *string* exists
ERROR 5974: Node doesn't exist
ERROR 5977: Object does not exist
ERROR 6071: Type *value* with odbc_subtype *value* is not supported
ERROR 6149: Authentication "*string*" does not exist
ERROR 6156: Can not use empty authentication
ERROR 6285: Index "*string*" does not exist
ERROR 6354: No auth "*string*" exists
ERROR 6355: No matching node with name *string*
ERROR 6358: No such table '*string*'
ERROR 6423: Target standby node "*string*" does not exist
ERROR 6567: Invalid input parameters.

string
ERROR 7101: Invalid role oid encountered
ERROR 7275: Registry::getSequenceCache failed to get Sequence Cache in planning phase for [*value*]
ERROR 7476: Notifier "*string*" does not exist
ERROR 7897: Model "*string*" does not exist
ERROR 7901: Model with name '*string*' does not exist
ERROR 8112: Model [*string*] does not exist or access was denied
ERROR 8166: Subscription identified by (*string*,*string*) does not exist
ERROR 8174: The model does not exist
ERROR 8248: Cannot make a *string* subscription as PRIMARY
ERROR 8252: Subscription of node *string* to shard *string* is already PRIMARY
ERROR 8282: Invalid DFS Root Type
ERROR 8287: No Model found. Create a model before start using DFS
ERROR 8288: Trouble finding model files
ERROR 8324: Subscription doesn't exist or not in PASSIVE state
ERROR 8560: Model [*string*] has unexpectedly changed
ERROR 8584: Model "*value*" does not exist
ERROR 8852: Load Balance Group "*string*" does not exist
ERROR 8860: Network Address "*string*" does not exist
ERROR 8866: Routing Rule "*string*" does not exist
ERROR 8952: Fault Group "*string*" is not in load balance group "*string*"
ERROR 8968: Network Addresses "*string*" is not in load balance group "*string*"
ERROR 8973: Object of type *string* named "*string*" does not exist
ERROR 9057: Field "*string*" does not exist
ERROR 9073: User Defined Type or Complex Type "*string*" does not exist
ERROR 9078: Cannot drop a subcluster with Drop Fault Group statement
ERROR 9079: Cannot drop default Subcluster
ERROR 9089: Subcluster "*string*" does not exist

ERROR 9095: *string* "*string*" does not exist
ERROR 9122: Subcluster *string* does not exist
ERROR 9124: Subcluster with name *string* does not exist
ERROR 9308: Object '*string*' of type '*string*' does not exist or is not visible
to the current user
ERROR 9492: Could not find set type for data type *string*
ERROR 9535: Subcluster "*string*" is not in load balance group "*string*"
ERROR 9561: Certificate "*string*" does not exist
ERROR 9572: Current subcluster does not exist
ERROR 9580: Key "*string*" does not exist
ERROR 9717: Type "*string*" is for internal use only
ERROR 9718: Type *string* for internal use only
ERROR 9719: Type *string* is for internal use only
ERROR 9803: Field "*string*.*string*" does not exist
ERROR 9810: CA Bundle "*string*" does not exist
ERROR 10162: Cannot drop TLS Configuration objects yet
ERROR 10179: Dropping TLS Configuration objects is not supported yet
ERROR 10199: Key "*string*" is not an RSA key
ERROR 10290: Subcluster or node with name *string* does not exist
ERROR 10346: Could not find *string* type for data type *string*
ERROR 10603: There is no subcluster-specific resource pool '*string*'
ERROR 10646: Cryptographic key "*string*" does not exist
ERROR 10653: TLS Configuration "*string*" does not exist

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42710

This topic lists the messages associated with the SQLSTATE 42710.

SQLSTATE 42710 description

DUPLICATE_OBJECT

Messages associated with this SQLState

Error messages

ERROR 2101: A sequence named "*string*" already exists
ERROR 2105: A table named "*string*" already exists
ERROR 2107: A view named "*string*" already exists
ERROR 2273: Bootstrap error (most likely in Bootstrap.cpp): Oid *value*
is already registered
ERROR 2276: Bootstrap error (most likely in Bootstrap.cpp):Name *string* is
already registered
ERROR 2713: Constraint *string* already exists
ERROR 3153: Duplicated local temp table found in design queries: *string*
ERROR 3327: Existing object "*string*" is not a view
ERROR 3881: Location [*string*] already exists for node *string*
ERROR 4043: Network Interface "*string*" already exists
ERROR 4135: Node "*string*" already exists
ERROR 4213: Object "*string*" already exists
ERROR 4445: Profile "*string*" already exists
ERROR 4482: Projection with base name "*string*" already exists
ERROR 4564: Relation "*string*" already exists
ERROR 4565: Relation "*string*" already exists in schema "*string*"
ERROR 4593: Resource pool "*string*" already exists
ERROR 4621: Role "*string*" already exists
ERROR 4804: Subnet "*string*" already exists
ERROR 4805: Subnet "*string*" already exists for [*string*]

ERROR 5582: Fault Group "*string*" already exists
ERROR 5584: Fault Group "*string*" cannot depend on itself directly or indirectly
ERROR 5615: Location [*string*] conflicts with existing location [*string*] on node *string*
ERROR 5623: Network Interface "*string*" already exists for [*string*]
ERROR 6009: Resource pool *string* already exists
ERROR 6148: Authentication "*string*" already exists
ERROR 6157: Can't create an index in schema "*string*": Schema cannot be found
ERROR 6158: Can't create an index named "*string*": Object already exists
ERROR 6188: Client authentication "*string*" already exists
ERROR 6735: Directed query with identifier "*string*" already exists
ERROR 6736: Directed query with identifier "*string*" does not exist
ERROR 7059: User "*string*" already exists
ERROR 7713: Error in blob creation: A blob with name '*string*' already exists
ERROR 7963: Relation [*string*] specified in parameter 'output_view' already exists
ERROR 8059: Cannot add more than one cache location
ERROR 8201: The specified output Table/View [*string*] already exists
ERROR 8251: Node *string* is already subscribed to shard *string* in *string* state
ERROR 8398: A model named "*string*" already exists
ERROR 8861: Network Addresses "*string*" and "*string*" are on the same node. Network Addresses in Load Balance Groups must refer to distinct nodes
ERROR 8951: Fault Group "*string*" is already in load balance group "*string*"
ERROR 8967: Network Addresses "*string*" is already in load balance group "*string*"
ERROR 8972: Object of type *string* named "*string*" already exists
ERROR 9036: A complex type named "*string*" already exists
ERROR 9037: A field named "*string*" already exists
ERROR 9121: Subcluster "*string*" already exists
ERROR 9534: Subcluster "*string*" is already in load balance group "*string*"
ERROR 9551: *string* "*string*" already exists
ERROR 9659: View definition can not contain duplicate field names "*string*"
ERROR 10477: Cannot submitTask: task queue for task type *value* is not initialized

Warning messages

WARNING 6112: Unable to guarantee the same base name for all segmented buddy projections

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42712

This topic lists the messages associated with the SQLSTATE 42712.

SQLSTATE 42712 description

DUPLICATE_ALIAS

Messages associated with this SQLState

ERROR 4901: Table name "*string*" specified more than once
ERROR 5696: WITH query name "*string*" specified more than once

Note

The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42723

This topic lists the messages associated with the SQLSTATE 42723.

SQLSTATE 42723 description

DUPLICATE_FUNCTION

Messages associated with this SQLState

ERROR 2278: Built-in function with the same name already exists: *string*
ERROR 4220: Object with same name and number of parameters already exists: *string*
ERROR 4428: Procedure/Function with same name and number of parameters already exists in schema *string*
ERROR 4429: Procedure/Function with same name and number of parameters already exists: *string*
ERROR 10457: *string* with same name and number of parameters already exists: *string*
ERROR 10497: Built-in function/procedure with the same name already exists: *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42725

This topic lists the messages associated with the SQLSTATE 42725.

SQLSTATE 42725 description

AMBIGUOUS_FUNCTION

Messages associated with this SQLState

ERROR 3459: Function *string* is not unique
ERROR 4289: Operator is not unique: *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42803

This topic lists the messages associated with the SQLSTATE 42803.

SQLSTATE 42803 description

GROUPING_ERROR

Messages associated with this SQLState

ERROR 2134: Aggregate function calls in subqueries cannot refer to columns in parent (outer) query
ERROR 2135: Aggregate function calls may not be nested
ERROR 2140: Aggregates not allowed in GROUP BY clause
ERROR 2141: Aggregates not allowed in JOIN conditions
ERROR 2142: Aggregates not allowed in WHERE clause
ERROR 2219: Argument *string* must not contain aggregates
ERROR 2543: Cannot use aggregate function in EXECUTE parameter
ERROR 2544: Cannot use aggregate function in function expression in FROM
ERROR 2640: Column "*string.string*" must appear in the GROUP BY clause or be used in an aggregate function
ERROR 4300: ORDER/GROUP BY expression not found in targetlist
ERROR 4634: Rule WHERE condition may not contain aggregate functions
ERROR 4667: SEGMENTED BY expression may not contain aggregate functions
ERROR 4841: Subquery uses ungrouped column "*string.string*" from outer query
ERROR 6139: All non-aggregate expressions in the projection SELECT list must appear in the GROUP BY clause
ERROR 6187: Cannot use aggregate expressions in the projection SELECT list without a GROUP BY clause
ERROR 6277: Grouping functions not allowed in GROUP BY clause
ERROR 6278: Grouping functions not allowed in JOIN conditions
ERROR 6279: Grouping functions not allowed in WHERE clause
ERROR 7085: Aggregate function calls cannot contain grouping functions
ERROR 7099: Grouping functions cannot be nested
ERROR 7182: Grouping function arguments need to be group by expressions

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42804

This topic lists the messages associated with the SQLSTATE 42804.

SQLSTATE 42804 description

DATATYPE_MISMATCH

Messages associated with this SQLState

Error messages

ERROR 2217: Argument *string* must be type float, not type *string*
ERROR 2218: Argument *string* must be type integer, not type *string*
ERROR 2222: Argument *string* must not return a set
ERROR 2224: Argument of *string* must be type boolean, not type *string*
ERROR 2225: Argument of *string* must not return a set
ERROR 2231: Array assignment requires type *string* but expression is of type *string*
ERROR 2232: Array assignment to "*string*" requires type *string* but expression is of type *string*
ERROR 2234: Array subscript must have type integer
ERROR 2358: Cannot assign to field "*string*" of column "*string*" because its type *string* is not a composite type
ERROR 2527: Cannot subscript type *string* because it is not an array
ERROR 2630: Column "*string*" is of type *string* but default expression is of type *string*
ERROR 2631: Column "*string*" is of type *string* but expression is of type *string*
ERROR 2846: Could not determine actual result type for function "*string*" declared to return type *string*
ERROR 3429: For '*string*', types *string* and *string* are inconsistent
ERROR 3545: Index expression may not return a set
ERROR 3801: IS DISTINCT FROM requires = operator to yield boolean
ERROR 3943: Mismatched types in VALUES LESS THAN expressions
ERROR 4284: Operator *string* must not return a set
ERROR 4285: Operator *string* must return type boolean, not type *string*
ERROR 4317: Parameter $*value* of type *string* cannot be coerced to the expected type *string*
ERROR 4803: Subfield "*string*" is of type *string* but expression is of type *string*
ERROR 7221: Argument of type *string* must be type boolean, not *string*
ERROR 7222: Argument of type *string* must be type boolean, not type *string*
ERROR 7309: Column "*string*" is of type *string* but set using expression is of type *string*
ERROR 7929: Column to be added/refreshed is of type *string* but default/set-using expression is of type *string*
ERROR 9799: Cannot subscript an empty array
ERROR 9801: Cannot subscript type *string* with *value* subscript levels
ERROR 9908: Cannot subscript column "*string*" of type *string* because it is not an array
ERROR 10202: Operator *string* not supported for complex types with arrays
ERROR 10220: Values being compared do not have the same schema

Warning messages

WARNING 6480: Column "*string*" is of type *string* but expression in Access Policy is of type *string*. It will be coerced at execution time

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42809

This topic lists the messages associated with the SQLSTATE 42809.

SQLSTATE 42809 description

WRONG_OBJECT_TYPE

Messages associated with this SQLState

ERROR 2011: *string* cannot use the WITHIN GROUP clause
ERROR 2037: *string* is not a supported analytic function
ERROR 2131: Aggregate function calls cannot contain analytic function calls
ERROR 2132: Aggregate function calls cannot contain sequence function calls
ERROR 2668: Column notation .*string* applied to type *string*, which is not a composite type
ERROR 2755: COPY requires relation *string* to be a Table, not a *string*
ERROR 2810: Could not add location [*string*]: Directory not empty
ERROR 2811: Could not add location [*string*]: Not a directory
ERROR 3114: DISTINCT specified, but *string* is not an aggregate function
ERROR 3327: Existing object "*string*" is not a view
ERROR 3421: First argument to modularhash_wrapper must be an integer constant
ERROR 3422: First argument to modularhash_wrapper must be of type integer, not *string*
ERROR 3463: Function *string*(*string*) is not an aggregate
ERROR 3669: Invalid function given
ERROR 3965: modularhash_wrapper must have two arguments: an integer constant and a call to modularhash_internal
ERROR 3966: modularhash_wrapper second argument is not modularhash_internal or a constant
ERROR 4215: Object "*string*" is not a projection
ERROR 4270: Op ANY/ALL (array) requires array on right side
ERROR 4271: Op ANY/ALL (array) requires operator not to return a set
ERROR 4272: Op ANY/ALL (array) requires operator to yield boolean
ERROR 4542: Record type has not been registered
ERROR 4657: Second argument to *string* must be a non-negative integer constant
ERROR 4931: The argument to *string* cannot be null
ERROR 4932: The argument to *string* must be a constant
ERROR 4987: Third argument to *string* must be a constant
ERROR 5111: Type *string* is not composite
ERROR 6036: Table "*string*" is not a flex table
ERROR 6691: CAST and current_date/current_time functions are not supported as analytic functions
ERROR 6951: Skip lazy projection creation for table *string.string* since the object referenced by the hint is either invalid or of the wrong type (expected *string*)
ERROR 9898: The first argument of type [*string*] is not an array type
ERROR 9899: The second argument of type [*string*] is not a scalar type
ERROR 9936: Cannot coerce array to a definite data type
ERROR 10268: Query returned null where a value was expected
ERROR 10314: Cannot assign null into NOT NULL variable
ERROR 10433: *string* is a stored procedure
ERROR 10576: Distinct aggregates cannot use the within group order by clause

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42830

This topic lists the messages associated with the SQLSTATE 42830.

SQLSTATE 42830 description

INVALID_FOREIGN_KEY

Messages associated with this SQLState

ERROR 3438: Foreign keys not specified
ERROR 3531: Incompatible data types between primary and foreign key columns: fk: *string*, pk: *string*
ERROR 4207: Number of primary and foreign keys must be the same

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42846

This topic lists the messages associated with the SQLSTATE 42846.

SQLSTATE 42846 description

CANNOT_COERCE

Messages associated with this SQLState

ERROR 2015: *string* could not convert type *string* to *string*
ERROR 2366: Cannot cast type *string* to *string*
ERROR 4986: Third argument of *string* could not be converted from type *string* to type *string*
ERROR 6510: Column "*string*" is of type *string* but expression in Access Policy is of type *string*. Cannot coerce expression
ERROR 7310: Column "*string*" is of type *string* but the *string* expression is of type *string*
ERROR 9398: Could not convert type *string* to *string*

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42883

This topic lists the messages associated with the SQLSTATE 42883.

SQLSTATE 42883 description

UNDEFINED_FUNCTION

Messages associated with this SQLState

ERROR 2126: Aggregate *string*(*string*) does not exist
ERROR 2127: Aggregate *string*(*) does not exist
ERROR 3007: DDL statement interfered with this statement
ERROR 3456: Function *string* does not exist
ERROR 3457: Function *string* does not exist, or permission is denied for *string*
ERROR 3462: Function *string* with the specified arguments does not exist
ERROR 3930: Meta-function *string* cannot be used in COPY
ERROR 3931: Meta-function *string* cannot be used in INSERT
ERROR 3932: Meta-function *string* cannot be used in UPDATE
ERROR 3933: Meta-function *string* cannot be used with FROM
ERROR 3934: Meta-function ("*string*") can be used only in the Select clause
ERROR 4067: No binary input function available for type *string*
ERROR 4068: No binary output function available for type *string*
ERROR 4083: No input function available for type *string*
ERROR 4091: No output function available for type *string*
ERROR 4286: Operator does not exist: *string*
ERROR 4290: Operator requires run-time type coercion: *string*
ERROR 5394: User procedure call (*value*) is not supported with FROM
ERROR 5910: Function *string* with the specified type and arguments does not exist
ERROR 6333: Meta-functions or non-deterministic functions cannot be used in projection column expressions
ERROR 6809: Inappropriate usage of meta-function ("*string*")
ERROR 7270: Invalid schema.table.column, table.column, or column specification
ERROR 7323: Meta-functions cannot be used in *string* expressions
ERROR 7324: Meta-functions cannot be used in *string* query definitions
ERROR 7340: VOLATILE functions cannot be used in a *string* expression when adding a column
ERROR 7341: VOLATILE functions cannot be used in a *string* query when adding a column
ERROR 7589: VOLATILE functions cannot be used in a *string* expression
ERROR 7590: VOLATILE functions cannot be used in a *string* query
ERROR 7657: Function "*string*" does not exist
ERROR 7688: Calling metafunctions '*string*' and '*string*' within the same query is not supported
ERROR 7692: Multiple calls to metafunction '*string*' within the same query are not supported
ERROR 7970: Meta-function *string* cannot be used in MERGE/UPDATE

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42939

This topic lists the messages associated with the SQLSTATE 42939.

SQLSTATE 42939 description

RESERVED_NAME

Messages associated with this SQLState

ERROR 2297: Can not drop default profile
ERROR 2299: Can not rename default profile
ERROR 2418: Cannot drop role "*string*"
ERROR 2488: Cannot rename role *string*
ERROR 2489: Cannot rename system column epoch
ERROR 2665: Column name "*string*" is reserved
ERROR 2666: Column name *string* is reserved
ERROR 3778: Invalid use of reserved the column name "*string*"
ERROR 4030: Names starting with "v_" are reserved names
ERROR 4953: The name "*string*" is a reserved name
ERROR 4962: The prefix "sys_" is reserved for system tuning rule
ERROR 6195: Column name "*string*" is reserved in aggregate projections
ERROR 7441: Invalid use of reserved column name "*string*"

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42P20

This topic lists the messages associated with the SQLSTATE 42P20.

SQLSTATE 42P20 description

WINDOWING_ERROR

Messages associated with this SQLState

ERROR 2011: *string* cannot use the WITHIN GROUP clause
ERROR 2041: *string* may only have one sort expression in the WITHIN GROUP clause
ERROR 2043: *string* must contain an ORDER BY clause within its analytic clause
ERROR 2044: *string* must NOT contain an ORDER BY clause or WINDOWING clause within its analytic clause
ERROR 2045: *string* must NOT contain WINDOWING clause within its analytic clause
ERROR 2047: *string* only supports the Integer, Float, Interval and Numeric data types
ERROR 2182: Analytic functions are allowed only in a SELECT list and/or ORDER BY clause
ERROR 2185: Analytic functions are not supported in the PARTITION BY of an OVER clause
ERROR 2187: Analytic functions cannot be nested
ERROR 2188: Analytic functions must have a FROM clause
ERROR 2189: Analytic functions not allowed in *string*
ERROR 2305: Can't cast the window bound into Int
ERROR 2306: Can't cast the window bound into the same data type of the ORDER BY column
ERROR 2465: Cannot override ORDER BY clause of window "*string*"
ERROR 2466: Cannot override PARTITION BY clause of window "*string*"
ERROR 2524: Cannot specify frame clause of window "*string*"
ERROR 3435: For range moving window, OrderBy expression must be one of Int, Float, Time, Timestamp, Interval, Date or Numeric
ERROR 3446: Frame clause not allowed without windowing order by
ERROR 3839: Keyword "ALL" is invalid in analytic functions
ERROR 4362: PERCENTILE_CONT/PERCENTILE_DISC must have the WITHIN GROUP clause
ERROR 4363: PERCENTILE_CONT/PERCENTILE_DISC must NOT contain an ORDER BY clause or WINDOWING clause within its analytic clause
ERROR 4811: Subqueries are not supported in the PARTITION BY of a timeseries OVER clause

ERROR 5006: Time Series Aggregate Functions are not supported in the PARTITION BY of a timeseries OVER clause
ERROR 5010: Time Series timestamp alias/Time Series Aggregate Functions not allowed in *string*
ERROR 5460: Window "*string*" is already defined
ERROR 5461: Window frame cannot end with PRECEDING if start is CURRENT ROW
ERROR 5462: Window frame cannot end with PRECEDING or CURRENT ROW if start is FOLLOWING
ERROR 5463: Window frame cannot end with UNBOUNDED PRECEDING
ERROR 5464: Window frame cannot start with UNBOUNDED FOLLOWING
ERROR 5466: Window frame logical offset must be a non-negative number to be consistent with the sort column type
ERROR 5467: Window frame logical offset must be an Interval (Day to Second or Year to Month) to be consistent with the sort column type
ERROR 5468: Window frame logical offset must be an Interval (Day to Second) to be consistent with the sort column type
ERROR 5469: Window frame logical offset must be an interval to be consistent with the sort column type
ERROR 5470: Window frame logical offset must be Int when the sort column type is Int
ERROR 5471: Window frame logical offset must be the same type as the sort column type (Interval Day to Second)
ERROR 5472: Window frame logical offset must be the same type as the sort column type (Interval Year to Month)
ERROR 5473: Window frame logical or physical offset must be a constant
ERROR 5474: Window frame logical or physical offset must be non-negative number or interval
ERROR 5475: Window frame physical offset must be non-negative number
ERROR 5477: Window ordering clause can only contain a single sort key if RANGE is used
ERROR 5478: Windowing not supported for User Defined Analytic functions
ERROR 6276: Grouping functions cannot appear within Analytic functions
ERROR 6900: Partition Prepass/Batch may only be used with Live Aggregate Projection
ERROR 7838: Windowing not supported for analytic usage of BOOL_XOR

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V01

This topic lists the messages associated with the SQLSTATE 42V01.

SQLSTATE 42V01 description

UNDEFINED_TABLE

Messages associated with this SQLState

ERROR 2308: Can't find anchor table
ERROR 2312: Can't find table
ERROR 2313: Can't find table "*string*"
ERROR 2714: Constraint *string* does not exist
ERROR 2948: CTAS: table "*string*" was dropped in another session (DDL interference)
ERROR 3007: DDL statement interfered with this statement
ERROR 3642: Invalid CTAS query: *string*
ERROR 3760: Invalid table name
ERROR 3761: Invalid table name "*string*"
ERROR 3762: Invalid table name *string*
ERROR 3953: Missing FROM-clause entry for table "*string*"
ERROR 3954: Missing FROM-clause entry in subquery for table "*string*"
ERROR 4416: Primary table "*string*" does not exist
ERROR 4446: Profile "*string*" does not exist
ERROR 4566: Relation "*string*" does not exist
ERROR 4567: Relation "*string*" in FOR UPDATE clause not found in FROM clause
ERROR 4568: Relation "*string.string*" does not exist
ERROR 4570: Relation with OID *value* does not exist
ERROR 4614: Role "*string*" does not exist
ERROR 4650: Schema "*string*" does not exist
ERROR 4697: Sequence "*string*" does not exist
ERROR 4806: Subnet "*string*" does not exist
ERROR 4876: Table "*string*" does not exist
ERROR 4883: Table "*string.string*" does not exist
ERROR 4885: Table *string* does not exist
ERROR 4898: Table does not exist (oid=*value*)
ERROR 4911: Table with OID *value* does not exist
ERROR 4912: Table/View with name '*string*' does not exist
ERROR 5105: Tuning rule "*string*" does not exist
ERROR 5360: User "*string*" does not exist
ERROR 5563: DDL statement interfered with this operation
ERROR 6149: Authentication "*string*" does not exist
ERROR 6220: CREATE INDEX failed
ERROR 6221: DDL interfered with this statement; table concurrently dropped
ERROR 6300: Invalid text-index parameters
ERROR 6418: Table "*string*" no longer exists
ERROR 6420: Table '*stringstringstring*' does not exist
ERROR 6729: DDL Interfered! table "*string*" does not exist
ERROR 6730: DDL statement interfered with this statement; table was dropped
ERROR 7285: Table "*string*" does not exist in HCatalog schema *string*
ERROR 7476: Notifier "*string*" does not exist
ERROR 7847: Relation [*string*] does not exist or access was denied
ERROR 8973: Object of type *string* named "*string*" does not exist
ERROR 8974: Object of type *string* OID "*value*" does not exist
ERROR 10521: Object of type *string* named "*string*" does not exist or the current user does not have access to it

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V02

This topic lists the messages associated with the SQLSTATE 42V02.

SQLSTATE 42V02 description

UNDEFINED_PARAMETER

Messages associated with this SQLState

Error messages

ERROR 3638: Invalid configuration parameter *string*; aborting configuration change

ERROR 4321: Parameter *value* is not set

ERROR 4984: There is no parameter $*value*

ERROR 5202: Unknown configuration parameter

ERROR 7357: A problem occurred during the execution of a detect_outliers iteration.

Detail: *string*

ERROR 7363: A problem occurred during the execution of balance.

Detail: *string*

ERROR 7891: init_method not allowed with argument initial_centers_table

ERROR 8017: Type mismatch of an optional argument: *string* must be a valid integer

ERROR 8018: Type mismatch of an optional argument: *string* must be a valid numeric

ERROR 8019: Type mismatch of an optional argument: *string* must be a valid string

ERROR 8051: A problem occurred during the execution of

computing approximate mad table.

Detail: *string*

ERROR 8052: A problem occurred during the execution of

computing approximate median table.

Detail: *string*

ERROR 8053: A problem occurred during the execution of

computing approximate meanAD table.

Detail: *string*

ERROR 8054: A problem occurred during the execution of

computing final mad table.

Detail: *string*

ERROR 8289: Type mismatch of an optional argument: parentName must be a valid string

ERROR 8290: Type mismatch of an optional argument: parentType must be a valid string

ERROR 8316: Type mismatch of an optional argument: *string* must be a valid boolean

ERROR 8741: Unsupported parameter. Only support 'recoverByTable' currently

ERROR 9388: Cannot get configuration parameter *string* at *string* level; no value to return

ERROR 9389: Cannot set configuration parameter *string* at *string* level; aborting configuration change

Warning messages

WARNING 9943: Parameter *string* was not registered by function *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V03

This topic lists the messages associated with the SQLSTATE 42V03.

SQLSTATE 42V03 description

DUPLICATE_CURSOR

Messages associated with this SQLState

Error messages

ERROR 2968: Cursor "*string*" already exists

Warning messages

WARNING 2615: Closing existing cursor "*string*"

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V04

This topic lists the messages associated with the SQLSTATE 42V04.

SQLSTATE 42V04 description

DUPLICATE_DATABASE

Messages associated with this SQLState

ERROR 2706: Connection to database [*string*] already exists

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V06

This topic lists the messages associated with the SQLSTATE 42V06.

SQLSTATE 42V06 description

DUPLICATE_SCHEMA

Messages associated with this SQLState

ERROR 4649: Schema "*string*" already exists

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V07

This topic lists the messages associated with the SQLSTATE 42V07.

SQLSTATE 42V07 description

DUPLICATE_TABLE

Messages associated with this SQLState

ERROR 4753: Skip lazy projection creation since super projection for table *string.string* already exists ERROR 6952: Skip lazy projection creation since a projection enforcing key constraint '*string*' for table *string.string* already exists ERROR 8050: A problem occurred during execution of Balance - the desired temp table name [*string*] already exists

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V08

This topic lists the messages associated with the SQLSTATE 42V08.

SQLSTATE 42V08 description

AMBIGUOUS_PARAMETER

Messages associated with this SQLState

ERROR 2848: Could not determine data type of parameter $*value* ERROR 3534: Inconsistent types deduced for parameter $*value*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V09

This topic lists the messages associated with the SQLSTATE 42V09.

SQLSTATE 42V09 description

AMBIGUOUS_ALIAS

Messages associated with this SQLState

ERROR 4908: Table reference "*string*" is ambiguous ERROR 4909: Table reference *value* is ambiguous

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V10

This topic lists the messages associated with the SQLSTATE 42V10.

SQLSTATE 42V10 description

INVALID_COLUMN_REFERENCE

Messages associated with this SQLState

ERROR 2046: *string* not allowed in *string* clause
ERROR 2050: *string* position *value* is not in select list
ERROR 2221: Argument *string* must not contain variables
ERROR 3467: Function expression in FROM may not refer to other relations of same query level
ERROR 3820: JOIN/ON clause refers to "*string*", which is not part of JOIN
ERROR 4832: Subquery in FROM may not refer to other relations of same query level
ERROR 4877: Table "*string*" has *value* columns available but *value* columns specified
ERROR 5194: UNION/INTERSECT/EXCEPT member statement may not refer to other relations of same query level
ERROR 9637: Record "*string*" has *value* fields available but *value* fields specified
ERROR 9641: Subquery "*string*" has *value* outputs available but *value* outputs specified

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V11

This topic lists the messages associated with the SQLSTATE 42V11.

SQLSTATE 42V11 description

INVALID_CURSOR_DEFINITION

Messages associated with this SQLState

ERROR 2522: Cannot specify both SCROLL and NO SCROLL

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V13

This topic lists the messages associated with the SQLSTATE 42V13.

SQLSTATE 42V13 description

INVALID_FUNCTION_DEFINITION

Messages associated with this SQLState

ERROR 2038: *string* is not a supported Time Series Aggregate Function
ERROR 2139: Aggregates may not return sets
ERROR 2173: An error occurred on node [*string*] when setting up the function [*string*]: [*string*]
ERROR 2177: An error occurred when setting up function "*string*"
ERROR 2397: Cannot determine result data type
ERROR 2451: Cannot load data from 0 sources; please specify 1 or more (on node [*string*])
ERROR 2494: Cannot RETURNREJECTED with multiple files or data sources
ERROR 3113: DISTINCT is supported only for single-argument aggregates
ERROR 3476: Functions in language *string* can be created only in fenced mode
ERROR 3604: Interpolation scheme *string* for Time Series Aggregate Function *string* is not supported
ERROR 3708: Invalid null argument for TSA function *string*
ERROR 3843: Language(*string*) does not match the language associated with the library(*string*)
ERROR 3854: Length of a string in a return type must be greater than zero
ERROR 3860: Library file is not loaded
ERROR 3861: Library not found: *string*
ERROR 3929: Meta functions cannot be used in UDx definitions
ERROR 4086: No language specified
ERROR 4095: No procedure source specified
ERROR 4096: No procedure user specified
ERROR 4251: Only one expression is allowed
ERROR 4257: Only simple "RETURN expression" is allowed
ERROR 4409: Precision of a numeric in a return type must be greater than zero
ERROR 4608: Return type *string* is not supported for SQL functions
ERROR 4609: Return type mismatch in a function declared to return *string*
ERROR 4610: Return type mismatch in function declared to return *string*
ERROR 4746: Setting up function "*string*" failed
ERROR 4858: Syntax error in syntax definition at offset *value*
ERROR 4949: The interpolation argument for Time Series Aggregate Function *string* must be a constant string
ERROR 5476: Window functions cannot return sets
ERROR 5777: Cannot set up function [*string*] on node: *string*
ERROR 6072: UDFileSystem only supports C++ unfenced mode
ERROR 6536: The getPrototype(*value*) and getReturnType(*value*) method must have matching number of returnTypes
ERROR 6537: The return types declared by getPrototype() and getReturnType() do not match
ERROR 6850: Library [*string*] built with incompatible SDK version [*string*]. Please rebuild with SDK version [*string*] and recreate the library
ERROR 6999: The library [*string*] for the function [*string*] was compiled with an incompatible SDK Version [*string*]
ERROR 7270: Invalid schema.table.column, table.column, or column specification
ERROR 8976: Only MIN/MAX and BOOL_AND/BOOL_OR are allowed to use DISTINCT
ERROR 9958: No procedure security option specified
ERROR 9963: Security options not supported for this kind of procedure
ERROR 9965: Users not supported for this kind of procedure

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V15

This topic lists the messages associated with the SQLSTATE 42V15.

SQLSTATE 42V15 description

INVALID_SCHEMA_DEFINITION

Messages associated with this SQLState

ERROR 2945: CREATE specifies a schema (*string*) different from the one being created (*string*)
ERROR 3365: Failed to create default projections for table "*string*"."*string*": *string*
ERROR 3586: Insufficient projections to answer query
ERROR 4097: No projections eligible to answer query
ERROR 4878: Table "*string*" has an out-of-date super projection "*string*"
ERROR 6774: Failed to create default key projections for table "*string*"."*string*": *string*
ERROR 7142: Failed to create hcatalog schema: *string*
ERROR 7876: Table "*string*" has an out-of-date projection "*string*"
ERROR 8633: Failed to create default key projections for table *value*
ERROR 8634: Failed to create default key projections for table *value*, constraint *value*
ERROR 8936: Cannot plan query because no super projections are safe

Note

The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V16

This topic lists the messages associated with the SQLSTATE 42V16.

SQLSTATE 42V16 description

INVALID_TABLE_DEFINITION

Messages associated with this SQLState

Error messages

ERROR 2104: A table cannot have only IDENTITY/AUTO-INCREMENT columns
ERROR 2420: Cannot drop the constraint. (Table "*string*" has a foreign key constraint referencing the specified primary key constraint)
ERROR 2622: Column "*string*" cannot be declared SETOF
ERROR 2626: Column "*string*" from table "*string*" in the SEGMENTED BY expression is required to be present in the projection, but is not
ERROR 2712: Constraint "*string*" for relation "*string*" already exists
ERROR 3508: IDENTITY/AUTO-INCREMENT columns are not allowed in temporary tables
ERROR 3874: Local temporary table constraint cannot reference a non-local table
ERROR 3901: MATCH types other than SIMPLE (the default) are not supported for foreign key constraints
ERROR 3987: Multiple primary keys for table "*string*" are not allowed
ERROR 4162: Non-local table constraint cannot reference a local temporary table
ERROR 4229: ON DELETE actions other than NO ACTION are not supported for foreign key constraints
ERROR 4234: ON UPDATE actions other than NO ACTION are not supported for foreign key constraints
ERROR 4413: Primary constraint for relation "*string*" already exists
ERROR 4415: Primary keys not specified

ERROR 4469: Projection anchor table is not partitioned
ERROR 4550: Referenced primary key constraint does not exist
ERROR 4876: Table "*string*" does not exist
ERROR 4881: Table "*string*" is not partitioned
ERROR 4885: Table *string* does not exist
ERROR 4892: Table *string* is not partitioned
ERROR 4899: Table is not partitioned
ERROR 4900: Table must have at least one column
ERROR 5269: Unsupported constraint type
ERROR 5548: Constraint not supported for user defined type column *string*
ERROR 5874: Failed to add table *string* of hcatalog schema *string* to catalog: no columns
ERROR 5876: Failed to alter table *string* of hcatalog schema *string* to catalog: no columns
ERROR 5879: Failed to describe hcatalog table
ERROR 5948: Local temporary objects may not specify a schema name
ERROR 6171: Cannot merge multiple partitions on an aggregate projection
ERROR 6396: SEGMENTED BY expression contains expressions not present in the SELECT list
ERROR 6434: TM task is not applicable to projections with aggregates
ERROR 6839: Key constraints on an external table cannot be enabled or disabled
ERROR 7122: Staging table's partition key and target table's partition key have different datatype
ERROR 7234: Check constraint '*string*': references to column '*string*' are not allowed in check predicates
ERROR 7235: Check constraint '*string*': subqueries are not allowed in check predicates
ERROR 7236: Check constraint predicate is too long; *value* bytes, maximum length is *value*
ERROR 7237: Check constraints on an external table cannot be enabled or disabled
ERROR 7238: Column "*string*" referenced by check constraint '*string*' is not in the table
ERROR 7260: Function "*string*" referenced by check constraint '*string*' is a meta-function
ERROR 7261: Function "*string*" referenced by check constraint '*string*' is an aggregate function
ERROR 7262: Function "*string*" referenced by check constraint '*string*' is not immutable
ERROR 7263: Function "*string*" referenced by check constraint '*string*' returns a set rather than a single value
ERROR 8091: Failed to parse JSON format Hive table description string: *string*
ERROR 9049: Constraint not supported for complex type column *string*
ERROR 9050: Constraint not supported for inlined complex type column *string*
ERROR 9051: Correlation constraint not supported for user defined types and complex types
ERROR 9202: Tables with complex data types cannot have default columns
ERROR 9203: Tables with complex data types cannot have identity columns
ERROR 9369: Internal complex types cannot be reused
ERROR 10116: Constraint not supported for primitive type column *string*
ERROR 10329: Only Parquet or Orc External Tables can use *string*
ERROR 10502: Cannot have SET USING expressions on complex type columns

Warning messages

WARNING 7233: Check constraint '*string*' does not reference any columns in the table
WARNING 10227: Mergeout is disabled on *string*

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V17

This topic lists the messages associated with the SQLSTATE 42V17.

SQLSTATE 42V17 description

INVALID_OBJECT_DEFINITION

Messages associated with this SQLState

Error messages

ERROR 2387: Cannot create projections involving external table *string*
ERROR 2493: Cannot retrieve information for design *string* in workspace *string*
ERROR 3075: Design type *string* is invalid
ERROR 3078: Optimization objective *string* is invalid
ERROR 3199: Error during deployment querying deployment projections table for workspace *string*
ERROR 3200: Error during deployment querying design projections table for design *string* in workspace *string*
ERROR 3201: Error during deployment while querying deployment projections table for workspace *string*
ERROR 3204: Error during drop design from deployment for workspace *string*
ERROR 3206: Error during extend catalog while querying deployments table for workspace *string*
ERROR 3207: Error during getDesignTablesFromDeployment in workspace *string*
ERROR 3213: Error during remove deployment drops from deployment *string* for workspace *string*
ERROR 3227: Error in querying *string.string*
ERROR 3269: Error while checking whether there are only incremental design deployed for deployment *string* in workspace *string*
ERROR 3271: Error while querying designs table for workspace *string*
ERROR 3968: More than one IDENTITY/AUTO_INCREMENT column defined for table "*string*"
ERROR 3983: Multiple instances of deployment *string* in workspace *string*
ERROR 4128: No valid projections found
ERROR 4230: ON DELETE rule may not use NEW
ERROR 4231: ON INSERT rule may not use OLD
ERROR 4232: ON SELECT rule may not use NEW
ERROR 4233: ON SELECT rule may not use OLD
ERROR 4635: Rule WHERE condition may not contain references to other relations
ERROR 4636: Rules with WHERE conditions may only have SELECT, INSERT, UPDATE, or DELETE actions
ERROR 4919: Temporary table projections are not allowed for this operation
ERROR 4982: There is no deployment *string* in workspace *string*
ERROR 4989: This function cannot be called on design *string* located in design workspace *string*
ERROR 4990: This function cannot be called on design *string*, when its design mode is *string*
ERROR 5367: User defined aggregate must return exactly one column.Function *string* returns *value*
ERROR 5369: User defined analytic must return exactly one column
ERROR 5384: User defined transform must provide names or aliases for

return columns
ERROR 5385: User defined transform must return at least one column
ERROR 5527: An error occurred on node *string* when setting up the type, message:

string
ERROR 5721: Purge is not allowed on temporary tables
ERROR 6166: Cannot create top-k projection: projection columns and limit are too big for the top-k buffer
ERROR 6483: Invalid enum value for parameter name
ERROR 6627: Aggregate projections may only contain User Defined Transforms with partition by BATCH in the outer query
ERROR 6628: Aggregate projections may only contain User Defined Transforms with partition by PREPASS or BATCH
ERROR 6970: Subqueries in aggregate projections may only contain User Defined Transforms with partition by PREPASS
ERROR 6993: The batch and prepass User Defined Transform Functions' signatures are not compatible for use in a live aggregate projection
ERROR 6994: The batch User Defined Transform Function does not have identical input and output signature
ERROR 7788: Only projections with single phase PREPASS User Defined Transforms can have user specified sort order or segmentation
ERROR 7789: ORDER BY / SEGMENTED BY / UNSEGMENTED are not allowed in projections with two phase User Defined Transforms. The projection is automatically ordered and segmented on partition by columns
ERROR 9606: Cannot add new node to empty subclusters, maximum *value* control nodes allowed, and there are already *value* (expected) control nodes in the cluster
ERROR 9847: User defined transform output column sorting must be the prefix
ERROR 9992: Cannot alter partition range on non-partition-ranged projection
ERROR 10002: Table not partitioned
ERROR 10009: Partition min value cannot be NULL
ERROR 10011: Partition range expression data type must match table partition expression data type
ERROR 10012: Partition range expression must be folded into constant
ERROR 10141: Partition range expression cannot contain subquery
ERROR 10222: Cannot alter partition range on local temp projection
ERROR 10262: Partition range literal must resolve to a string value

Warning messages

WARNING 6095: UseLongStrings has been deprecated

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V18

This topic lists the messages associated with the SQLSTATE 42V18.

SQLSTATE 42V18 description

INDETERMINATE_DATATYPE

Messages associated with this SQLState

ERROR 2847: Could not determine data type of column $*value*
ERROR 2848: Could not determine data type of parameter $*value*
ERROR 3609: Interval must be single datetime field

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V21

This topic lists the messages associated with the SQLSTATE 42V21.

SQLSTATE 42V21 description

UNDEFINED_PROJECTION

Messages associated with this SQLState

Error messages

ERROR 2310: Can't find projection
ERROR 2311: Can't find projection *value*
ERROR 2430: Cannot find projection column *value*
ERROR 3005: DDL statement interfered with refresh operation
ERROR 3007: DDL statement interfered with this statement
ERROR 3736: Invalid projection name
ERROR 3737: Invalid projection name *string*
ERROR 4451: Projection "*string*" does not exist
ERROR 4452: Projection "*string*" does not exist or was just dropped
ERROR 4460: Projection *string* does not exist
ERROR 4474: Projection does not exist
ERROR 4876: Table "*string*" does not exist
ERROR 4905: Table or projection "*string*" does not exist
ERROR 5563: DDL statement interfered with this operation
ERROR 7528: Projecton "*string*" does not exist
ERROR 7863: Can't find projection: *value*
ERROR 8223: No up-to-date super projections available to refresh projection
ERROR 9056: DDL Interfered! Projection "*string*" does not exist

Warning messages

WARNING 8763: *value* projections are not loaded because they are unsupported
WARNING 10578: Cannot generate query plan for table "*string*" as no projections are defined for it

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V22

This topic lists the messages associated with the SQLSTATE 42V22.

SQLSTATE 42V22 description

UNDEFINED_NODE

Messages associated with this SQLState

ERROR 4136: Node "*string*" does not exist
ERROR 8416: Node *string* was not found on communal storage
ERROR 8864: Node directory missing from communal storage *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V24

This topic lists the messages associated with the SQLSTATE 42V24.

SQLSTATE 42V24 description

UNDEFINED_USER

Messages associated with this SQLState

Error messages

ERROR 5360: User "*string*" does not exist
ERROR 7213: Current user no longer exists
ERROR 10581: Definer of stored procedure does not exist

Fatal messages

FATAL 5360: User "*string*" does not exist

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V25

This topic lists the messages associated with the SQLSTATE 42V25.

SQLSTATE 42V25 description

PATTERN_MATCH_ERROR

Messages associated with this SQLState

ERROR 2227: Argument to test_pattern_event_eval must be > 0 and less than the total number of events
ERROR 2228: Argument to test_pattern_event_eval must be a constant
ERROR 2553: Cannot use more than one pattern
ERROR 2555: Cannot use pattern test functions with pattern match functions
ERROR 3025: Defining more than 52 events is not supported
ERROR 3288: Event "*string*" in PATTERN clause is not defined in the DEFINE clause
ERROR 3289: Event ANY_ROW cannot be used under *, +, ?, or | when the select list contains the pattern function event_name()
ERROR 3290: Event ANY_ROW is a reserved event and cannot be user defined
ERROR 3294: Event expressions cannot contain analytic functions
ERROR 3295: Event expressions cannot contain correlated expressions
ERROR 3296: Event expressions cannot contain subqueries
ERROR 3297: Event name "*string*" defined more than once
ERROR 4353: Pattern events must be mutually exclusive
ERROR 4354: Pattern match query cannot contain having clause, group clause, aggregates, or distinct
ERROR 4355: Pattern match query cannot contain timeseries clause
ERROR 4356: Pattern matching recursion limit reached
ERROR 4358: PatternMatchingMaxPartition must be greater than 0
ERROR 4359: PatternMatchingMaxPartitionMatches must be greater than 0
ERROR 4360: PatternMatchingPerMatchWorkspaceSize must be greater than 0 and less than 1024
ERROR 4494: Queries with user-defined transform functions (*string*) cannot have a MATCH clause
ERROR 4507: Query with analytic function *string* cannot have a MATCH clause
ERROR 4509: Query with pattern matching function *string* must include a MATCH clause
ERROR 4605: RESULTS GROUPED BY MATCH is not supported
ERROR 5283: Unsupported pattern operator
ERROR 7607: Pattern matching DFA Algorithm work space size limit reached
ERROR 8585: Pattern matching just-in-time processing stack memory limit reached

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 42V26

This topic lists the messages associated with the SQLSTATE 42V26.

SQLSTATE 42V26 description

DUPLICATE_NODE

Messages associated with this SQLState

ERROR 4058: New node matches existing node *string*
ERROR 4063: New values for node *string* matches existing node *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 53000

This topic lists the messages associated with the SQLSTATE 53000.

SQLSTATE 53000 description

INSUFFICIENT_RESOURCES

Messages associated with this SQLState

ERROR 2245: Attempted to create too many ROS containers for projection *string*
ERROR 2843: Could not create thread for recoverProjectionLocal
ERROR 2844: Could not create thread for SubsessionHandler
ERROR 2845: Could not create thread for SubsessionHandler Hurry
ERROR 2997: DBDesigner memory usage (*value* bytes) exceeded system limit
ERROR 3300: Exceeded temp space cap, requested *value* with *value* remaining (used *value*) bytes
ERROR 3416: Filter tried to allocate too much memory (*value*, out of *value* allowed)
ERROR 3587: Insufficient resources to execute plan on pool *string* [*string*]
ERROR 3921: MemoryPool *string* used more memory than allowed
ERROR 3937: MIN/MAX window function could not operate in memory
ERROR 4764: Source tried to allocate too much memory (*value*, out of *value* allowed)
ERROR 5000: Thread limit *value*, but statement needs *value* threads
ERROR 5001: ThreadManager failed to create thread *string*: *string*
ERROR 5022: Timer service failed to run *value*: *string*
ERROR 5065: Too many ROS containers exist for the following projections: *string*
ERROR 5921: Insufficient memory available for database designer
ERROR 5924: Insufficient resources to get resource from *string* pool [*string*]
ERROR 6941: Result set size (*value* KB) is too big. Try increasing TempSpaceCap (currently *value* KB)
ERROR 7423: Failed to acquire resources for blob '*string*'
ERROR 7700: Attempted to move/copy too many ROS containers for projection *string*
ERROR 8722: The minimal memory required by the query [*value* KB] exceeds the query cap size [*value* KB]
ERROR 8875: Command queue cannot create the watchdog thread
ERROR 8877: ICQ cannot create execution threads (*value* requested, *value* created)
ERROR 8949: Exceeded temp space maximum for the container cache
ERROR 8950: Exceeded temp space maximum, requested *value* with *value* remaining (used *value*) bytes
ERROR 8955: HybridStream reached its maximum capacity of *value* bytes
ERROR 9511: Memory budget is not enough for partition projection operation with group expression

Current resource pool: *string*, memory budget: *value* KB, memory required: *value* KB, Projection: *value*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 53100

This topic lists the messages associated with the SQLSTATE 53100.

SQLSTATE 53100 description

DISK_FULL

Messages associated with this SQLState

ERROR 2475: Cannot rebalance cluster. Insufficient disk space on the following nodes: *string*
ERROR 2927: Could not write to [*string*]: *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 53200

This topic lists the messages associated with the SQLSTATE 53200.

SQLSTATE 53200 description

OUT_OF_MEMORY

Messages associated with this SQLState

Error messages

ERROR 2296: Calloc of *value* bytes for *string* failed
ERROR 2344: Cannot allocate sufficient memory for COPY statement (*value* requested, *value* permitted)
ERROR 3499: Hash table out of memory
ERROR 3811: Join [*string*] inner partition did not fit in memory; value [*string*]
ERROR 3813: Join did not fit in memory
ERROR 3814: Join inner did not fit in memory
ERROR 3815: Join inner did not fit in memory [*string*]
ERROR 3819: Join table did not fit in memory
ERROR 3895: Malloc of *value* bytes for *string* failed
ERROR 4176: Not enough memory for test directive numTopKHeaps
ERROR 4302: Out of memory
ERROR 4303: Out of memory when expanding glob: *string*
ERROR 4357: Pattern partition will not fit into memory
ERROR 4381: Plan memory limit exhausted: [*string*]
ERROR 4524: Realloc of *value* bytes for *string* failed
ERROR 5062: Too many hash table entries
ERROR 5063: Too many matches in a single partition
ERROR 5952: Malloc of *value* bytes in Block Memory Manager failed
ERROR 7979: Cannot allocate all variables
ERROR 8628: Cannot allocate variables
ERROR 8708: Cannot parse class_weights parameter [*string*]. Must be two positive values separated by a comma
ERROR 9207: Blob was not allowed to spill to disk after reaching *value* bytes
ERROR 9741: Setenv of *string* failed

Fatal messages

FATAL 4302: Out of memory

Warning messages

WARNING 4495: Query *value* exceeded memory usage limit. Design result for this query might be suboptimal

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 54000

This topic lists the messages associated with the SQLSTATE 54000.

SQLSTATE 54000 description

PROGRAM_LIMIT_EXCEEDED

Messages associated with this SQLState

Error messages

ERROR 2052: *string* Row size *value* is too large
ERROR 2472: Cannot prepare statement - too many prepared statements
ERROR 3626: Invalid buffer enlargement request size *value*
ERROR 4282: Operator *string* may give a *value*-octet Varbinary result; the limit is *value* octets
ERROR 4283: Operator *string* may give a *value*-octet Varchar result; the limit is *value* octets
ERROR 4557: regexp_replace result is too long
ERROR 4913: Target lists can have at most *value* entries
ERROR 5043: Timezone directory stack overflow
ERROR 5060: Too many data partitions
ERROR 5263: Unsupported access to external table
ERROR 5265: Unsupported access to virtual schema
ERROR 5266: Unsupported access to virtual table
ERROR 5267: Unsupported access to virtual view
ERROR 5749: Array size exceeds the maximum allowed (*value*)
ERROR 6064: Transaction commit delta is too large (*value*)
ERROR 6076: Unable to fork to start spread: *value*
ERROR 6117: Size of compressed serialized plan (*value* bytes) is too large
ERROR 6147: Attempted to return result set too large; exceeds remote execution buffer size of *value*
ERROR 8177: Total size of intermediate columns is too large
ERROR 9688: Function *string* may give a *value*-octet result; the limit is *value* octets
ERROR 9864: EE5 Temp Relation Full
ERROR 9867: Leaf page size is not large enough
ERROR 10530: Unsupported operation on table with complex types

Warning messages

WARNING 3866: Line is too long in timezone file "*string*", line *value*

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 54001

This topic lists the messages associated with the SQLSTATE 54001.

SQLSTATE 54001 description

STATEMENT_TOO_COMPLEX

Messages associated with this SQLState

ERROR 4588: Request size too big. Please try to simplify the query
ERROR 4963: The query contains a SET operation tree that is too complex to analyze
ERROR 4964: The query contains an expression that is too complex to analyze
ERROR 7913: MLA CUBE has too many columns: *value*
ERROR 7914: MLA grouping sets have *value* elements taking too much memory: *value* bytes
ERROR 7915: Too many grouping sets generated: *value*
ERROR 7916: Too many grouping sets generated: *value*
ERROR 8031: Too many grouping elements: *value*, request size too big
ERROR 8037: Query plan is too large to serialize
ERROR 8617: Request size too big
ERROR 8922: Query Rejected
ERROR 9357: Query plan is too complex for the execution engine to compile

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 54011

This topic lists the messages associated with the SQLSTATE 54011.

SQLSTATE 54011 description

TOO_MANY_COLUMNS

Messages associated with this SQLState

ERROR 2106: A table/projection/view can only have up to *value* columns -- this create statement has *value*
ERROR 2118: Adding column causes row size (*value*) to exceed MaxRowSize (*value*)
ERROR 2136: Aggregate function cannot have *value* input argument(s)
ERROR 2137: Aggregate function cannot have *value* return value(s)
ERROR 2181: Analytic function cannot have *value* return value(s)
ERROR 2291: Call to ColumnTypes.addAny() is not allowed in Aggregate functions
ERROR 3466: Function cannot have *value* return value(s)
ERROR 4481: Projection row size (*value*) exceeds MaxRowSize (*value*)
ERROR 4630: Row size exceeds MaxRowSize: *value* > *value*
ERROR 4875: Table "*string*" can only have up to *value* columns -- adding one will exceed this limit
ERROR 5898: File system cannot have *value* input argument(s)
ERROR 5899: File system cannot have *value* return value(s)
ERROR 9069: Table can only have up to *value* columns including complex type fields -- this create statement has *value*
ERROR 10652: Table can only have up to *value* columns including complex type fields

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 54023

This topic lists the messages associated with the SQLSTATE 54023.

SQLSTATE 54023 description

TOO_MANY_ARGUMENTS

Messages associated with this SQLState

ERROR 2441: Cannot have more than *value* segmentation columns
ERROR 2469: Cannot pass more than *value* arguments to a function
ERROR 4431: Procedures cannot have more than *value* parameters
ERROR 4646: Scalar/Transform functions cannot have more than *value* parameters
ERROR 5055: Too many arguments
ERROR 5056: Too many arguments to evaluate_delete_performance function

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 55000

This topic lists the messages associated with the SQLSTATE 55000.

SQLSTATE 55000 description

OBJECT_NOT_IN_PREREQUISITE_STATE

Messages associated with this SQLState

Error messages

ERROR 2005: *string*
ERROR 2088: A concurrent load into the partition or a concurrent mergeout operation interfered with this statement
ERROR 2149: AHM can't be set
ERROR 2150: AHM can't be set while retentive refresh is running
ERROR 2151: AHM can't be set. (*value* nodes are down, out of *value*.)
ERROR 2152: AHM can't be set. (*value* nodes are down.)
ERROR 2159: All nodes must be UP to rebalance a cluster
ERROR 2163: Already released
ERROR 2175: An error occurred when loading library file on node *string*, message:

string
ERROR 2200: AnalyzeStatsPlanMaxColumns configuration parameter '*value*' invalid; must be greater than zero
ERROR 2201: AnalyzeStatsSampleBands configuration parameter '*value*' invalid; must be greater than zero
ERROR 2241: Attempt to create view using an invalid relation
ERROR 2242: Attempt to run multi-node KV plan
ERROR 2294: CALL_USE_SESSION_NODES used without setting nodes
ERROR 2303: Can not tell if tables have data, too few responses (*value*) to be conclusive
ERROR 2371: Cannot commit DML/DDL while a node is shutting down
ERROR 2378: Cannot convert column "*string*" to type "*string*"
ERROR 2380: Cannot create a library without an initialized LibraryPath on node: *string*
ERROR 2388: Cannot create projections on a temporary table that has data
ERROR 2409: Cannot drop any more columns in *string*
ERROR 2410: Cannot drop column "*string*" since it is referenced in the default expression of column "*string*"
ERROR 2422: Cannot Drop: *string string* depends on *string string*
ERROR 2424: Cannot execute query with temporary table because a node has recovered since the start of this session
ERROR 2448: Cannot issue this command in a read-only transaction
ERROR 2459: Cannot modify temporary table *string* because a node has

recovered or rebalance data took place since the start of this *string*
ERROR 2483: Cannot remove snapshots without an initialized SnapshotPath
ERROR 2496: Cannot revoke EXECUTE permission from the owner: *string*
ERROR 2497: Cannot revoke EXECUTE permission from the super user
ERROR 2505: Cannot set column "*string*" in table "*string*" to NOT NULL since it contains null values
ERROR 2512: Cannot set memoryCap for session whose current user has been dropped
ERROR 2516: Cannot set runTimeCap for session whose current user has been dropped
ERROR 2517: Cannot set tempSpaceCap for session whose current user has been dropped
ERROR 2541: Cannot use addAny() with any other input column types
ERROR 2542: Cannot use addAny() with any other output column types
ERROR 2563: Cannot validate DV storage
ERROR 2564: Cannot validate storage
ERROR 2587: Changes cannot be made to [*string*]. It has been retired
ERROR 2624: Column "*string*" does not exist
ERROR 2691: Concurrent DDL interfered with this statement
ERROR 2762: COPY: Cannot load into IDENTITY column "*string*"
ERROR 2763: COPY: Cannot specify parsing options for IDENTITY column "*string*"
ERROR 2903: Could not reset epoch because DML locks are held
ERROR 2904: Could not reset epoch because projections exist
ERROR 2933: Couldn't force partition projection *string*
ERROR 2934: Couldn't force partition projections *string*
ERROR 2954: Current phase of recovery failed due to missed event at epoch %#llx
ERROR 2955: Current set of up nodes do not satisfy dependencies
ERROR 2956: Current set of up nodes do not satisfy dependencies for table *string*
ERROR 2961: Current user has been dropped so no defaults are available
ERROR 2962: Current user has been dropped so no roles are available
ERROR 3000: DDL interfered with this statement
ERROR 3002: DDL statement interfered with alter column type
ERROR 3007: DDL statement interfered with this statement
ERROR 3136: drop_partition failed for *string* on node *string*. The projection contains unpartitioned data
ERROR 3196: Error deserializing objects
ERROR 3229: Error loading library file:[*string*]
ERROR 3254: Error reading from file
ERROR 3278: Error writing to file
ERROR 3318: Execution aborted by node state change
ERROR 3807: JobTracker::getMarkedStorages(): Unknown job *value*
ERROR 3808: JobTracker::jobComplete(*string*): Unknown job *value*
ERROR 3809: JobTracker::setDetails(*value*,*value*,*value*): Unknown job *value*
ERROR 3810: JobTracker::setJobDescription(*string*): Unknown job *value*
ERROR 3838: Key *value* already in use
ERROR 3852: Length for type *string* cannot exceed *value*
ERROR 3924: merge_partitions() failed on *string* because of unpartitioned data
ERROR 3925: Mergeout failed: projection *string* is not up-to-date
ERROR 4032: Naming conflict: *string* exists
ERROR 4092: No plan received at node
ERROR 4120: No transaction running on node
ERROR 4127: No valid cache found
ERROR 4138: Node *string* is not available for queries
ERROR 4144: Node has not been set up for plan execution
ERROR 4146: Node is not active or recovering, cannot plan query

ERROR 4146: Node is not active or recovering, cannot plan query
ERROR 4148: Node not prepared to accept plan
ERROR 4151: Node unprepared for rebalance
ERROR 4403: Portal "*string*" cannot be run
ERROR 4457: Projection *string* checkpoint epoch lags snapshot epoch
ERROR 4459: Projection *string* create epoch is greater than the epoch in the query
ERROR 4462: Projection *string* has HSE > snapshot epoch and buddy *string* has HSE <= snapshot epoch
ERROR 4467: Projection (name: *string*, oid: *value*) is newly added during current recovery
ERROR 4592: reset_epoch is disabled because the EnableResetEpoch configuration parameter is 0
ERROR 4611: Returned string value '[*string*]' with length [*value*] is greater than declared field length of [*value*] of field [*string*] at output column index [*value*]
ERROR 4698: Sequence "*string*" has been created by an IDENTITY/AUTO_INCREMENT column and cannot be dropped
ERROR 4700: Sequence *string* has not been accessed in the session
ERROR 4765: Specified K-safety for projection creation is insufficient to support currently down nodes
ERROR 4791: Storage extends beyond specified segment range
ERROR 4793: Stream error: *string*
ERROR 4860: System is not k-safe. DDL is disallowed
ERROR 4861: System is not k-safe. DDL/DML is disallowed
ERROR 4879: Table "*string*" has projections in non-up-to-date state
ERROR 4880: Table "*string*" has projections that are not up-to-date that can refresh from buddy
ERROR 4881: Table "*string*" is not partitioned
ERROR 4885: Table *string* does not exist
ERROR 4892: Table *string* is not partitioned
ERROR 4903: Table or projection no longer exists
ERROR 4934: The attribute "*string*" in table "*string*" needs to be included in projection "*string*" because it is used in the partitioning expression
ERROR 4941: The data type requires length/precision specification
ERROR 4965: The restore violates K safety
ERROR 4972: The types/sizes of source column (index *value*, length *value*) and destination column (index *value*, length *value*) do not match
ERROR 5084: Tried to add field '*string*' that already exists
ERROR 5085: Tried to add unknown node '*string*' to user-defined query plan
ERROR 5132: Unable to evaluate the delete performance after dropping this column for projection "*string*"
ERROR 5151: Unable to validate data in *string*: *string*
ERROR 5204: Unknown data type
ERROR 5210: Unknown object: *string*
ERROR 5321: Usage of [*string*] cannot be changed. It has been retired
ERROR 5522: A concurrent operation interfered with this statement
ERROR 5534: Can't create table in specified target schema
ERROR 5535: Can't find target table's schema
ERROR 5543: Cannot use column type 'any' with any other input column types
ERROR 5544: Cannot use column type 'any' with any other output column types
ERROR 5583: Fault Group "*string*" already exists in a fault group
ERROR 5587: Fault Group "*string*" not found in Fault Group "*string*"
ERROR 5590: Found *value* unsegmented projections with basename *string*; inconsistent with permanent nodes count *value*
ERROR 5626: Node "*string*" already exists in a fault group
ERROR 5627: Node "*string*" not found in Fault Group "*string*"
ERROR 5660: Source table can not be temp, virtual, system, or external

ERROR 5662: Storage tier *string* has not been found on all nodes
ERROR 5667: Target table can not be temp, virtual, system, or external
ERROR 5674: TM interfered with this statement
ERROR 5705: Dvmergeout failed: projection *string* is not up-to-date
ERROR 5712: JobTracker::reportStart: Unknown job *value*
ERROR 5735: Tier *string* is referenced by storage policies. Can not make storage location changes as requested
ERROR 5760: Can only change setting when all started nodes are UP
ERROR 5765: Cannot change control node away from self because other nodes depend on this node to be their control node
ERROR 5766: Cannot change final control node away from self until at least one other node is promoted to be a control node
ERROR 5772: Cannot manually alter automatically generated fault groups
ERROR 5786: Column *value* does not have corresponding storages yet. A concurrent add column operation might be running
ERROR 5883: Failed to list hcatalog tables
ERROR 6001: Recovery failed because DVROS straddles discard epoch
ERROR 6035: Table "*string*" has no non-null records under the column key_name
ERROR 6037: Table "*string_string*" cannot be found or was not created internally
ERROR 6065: Tried to allocate and initialize a *value*-byte string with *value* zero bytes; VString is too small
ERROR 6066: Tried to copy a *value*-byte string to *value*-byte VString object; VString is too small
ERROR 6105: View "*string*" is already linked to flex table "*string*". Linked views will not be overwritten
ERROR 6106: View "*string*" is already linked to this table. Linked views will not be overwritten
ERROR 6107: View "*string_string*" cannot be found or was not created internally
ERROR 6110: A design/deployment process is currently executing in this design space
ERROR 6121: A concurrent opertation interfered with this statement
ERROR 6176: Cannot replace node *string* because it is already a STANDBY
ERROR 6177: Cannot replace node *string* because it is not DOWN
ERROR 6184: Cannot swap partition between same table
ERROR 6185: Cannot transition node *string* to *string* because it still has data
ERROR 6186: Cannot transition node *string* to STANDBY because its loss would cause the cluster to shutdown
ERROR 6207: Could not create internal data-storage directory '*string*': *value*
ERROR 6357: No standby nodes are currently available
ERROR 6359: Node *string* has not been replaced
ERROR 6360: Node *string* is a *string* node and cannot store data
ERROR 6374: Original node *string* is not currently available in STANDBY mode
ERROR 6419: Table *string* can not be temp, virtual, system, or external
ERROR 6422: Target node *string* is a *string* node, not a STANDBY node
ERROR 6424: Target standby node *string* is not currently available
ERROR 6446: UDx set BOOLEAN column *value* to non-boolean value *value*
ERROR 6478: Can only take object-level snapshot from local storage locations
ERROR 6524: DDL interfered with this statement. Table is not partitioned or partition expression got changed
ERROR 6553: Compact storage failed: projection *string* is not up-to-date
ERROR 6633: An enabled constraint can only be declared on a global temporary table during CREATE TABLE
ERROR 6634: An enabled constraint cannot be created on a temporary table with existing data
ERROR 6646: Attempted to commit when column *value* has *value* rows while column 0 has *value* rows

ERROR 6647: Attempted to use a *string* as a *string*
ERROR 6648: Attempted to write past the end of a column
ERROR 6658: Can not *string* to the same table
ERROR 6665: Can not set priority for text index, please use its source table
ERROR 6675: Cannot drop column "*string*" since it is the last non-IDENTITY, non-AUTO_INCREMENT column
ERROR 6677: Cannot execute query because table recovery status change
ERROR 6784: Final attempt at a database snapshot upgraded storage ids on the following nodes: *string*
ERROR 6811: Incorrect use of setter in processBlock
ERROR 6812: Incorrect use of setter in processPartition for [*value*] column
ERROR 6826: Invalid node Oid *value*
ERROR 6937: Restore: Cannot overwrite object *string*
ERROR 6957: Source items are not the correct encoding for direct copies
ERROR 6958: Source items are not the correct size for direct copies
ERROR 6976: Table "*string*" has not been recovered. Please try later
ERROR 6982: Table can not be temp, virtual, system, or external
ERROR 6989: Terminate() must be overidden for a User Defined Aggregate
ERROR 6997: The key *string* doesn't exist
ERROR 7001: The sessionParamReader for namespace '*string*' doesn't exist
ERROR 7002: The sessionParamWriter for namespace '*string*' doesn't exist
ERROR 7016: UDx set BOOLEAN column *value*, row *value* to non-boolean value *value*
ERROR 7017: Unable to acquire side process info
ERROR 7072: WebHCat query [*string*] failed: *string*
ERROR 7093: Comment length of parameter '*string*' is '*value*' which exceeds the maximum allowed '*value*'
ERROR 7119: Source and target table do not match: *string*
ERROR 7121: Staging table and target table do not match: *string*
ERROR 7173: Current set of up nodes do not satisfy dependencies for DFS file distribution *value*
ERROR 7193: RecoverByContainer::recover can't advance the cpe by recovering containers
ERROR 7212: Cannot restore data to node *string*
ERROR 7290: Trying to set the column "*string*" to size of *value*. All data type lengths in table "*string*" must not be greater than *value*, the current maximum raw size for flex table values
ERROR 7299: Cannot alter type of column "*string*" since it is referenced in the default expression of table "*string*", column "*string*"
ERROR 7300: Cannot alter type of column "*string*" since it is referenced in the set using expression of table "*string*", column "*string*"
ERROR 7301: Cannot drop column "*string*" since it is referenced in the default expression of table "*string*", column "*string*"
ERROR 7302: Cannot drop column "*string*" since it is referenced in the set using expression of table "*string*", column "*string*"
ERROR 7343: *string* expression of IDENTITY/AUTO_INCREMENT column "*string*" cannot be altered
ERROR 7390: Cannot set idlesessiontimeout for session whose current user has been dropped
ERROR 7458: Mergeout failed: found missed *string* on table *string*
ERROR 7537: Sequence "*string*" has been created by an IDENTITY/AUTO_INCREMENT column and cannot be used in a *string* expression
ERROR 7583: User or Role *value* cannot be found
ERROR 7619: SBJobTracker::setAdditionalInfo: Unknown oid "*value*"
ERROR 7671: Wrong checksum for library file *string*
ERROR 7766: Cannot drop column "*string*" since it is referenced in the set

using expression of column "*string*"
ERROR 7811: SendFiles on node *string*: file [*string*] has changed
ERROR 7821: Cannot rename user "*string*" since they use MD5 password format
ERROR 7846: Relation *string* is empty
ERROR 7856: Not all requested nodes are available at this moment
ERROR 7857: Not enough available nodes at this moment
ERROR 7866: DDL statement interfered with this statement. ProjCol *value* cannot be found
ERROR 7978: Can only use KV hint with SELECT queries
ERROR 7983: Cannot use KV Hint with a subquery
ERROR 8061: Cannot create COMMUNAL storage location with usage *string*
ERROR 8062: Cannot create COMMUNAL storage location without sharding enabled
ERROR 8064: Cannot drop last subscription for a shard
ERROR 8069: Cannot set grace period for session whose current user has been dropped
ERROR 8139: Projection must be created in the same schema as its anchor table
ERROR 8147: SALColumn found during restore. Restoring objects from on older vertica version is not supported
ERROR 8159: Size can be changed only for DEPOT locations
ERROR 8173: Text index must be created in the same schema as its source table
ERROR 8181: Unable to read file *string*
ERROR 8185: Usage of [*string*] cannot be changed. It is a depot location
ERROR 8186: User lacks privileges on resource pool '*string*'
ERROR 8210: Object *value* appears in the newborns of the Transaction Catalog.

Object is Global: *string*

Object has Shard Tag: *value*

Object was created in this TXN tier: *string*
ERROR 8222: No active subscriptions found for the session
ERROR 8228: Cannot load snapshot containing DOWN source node *string* into the current database
ERROR 8229: Cannot restore or replicate while a node is recovering
ERROR 8232: No snapshot to load in the current session. Did load_snapshot_prep() succeed?
ERROR 8235: Sequence "*string*" has been created by an IDENTITY/AUTO_INCREMENT column and name is internal to Vertica, can not be changed directly
ERROR 8236: Sequence "*string*" has been created by an IDENTITY/AUTO_INCREMENT column, the schema associate with it cannot be changed directly
ERROR 8237: Snapshot contains internally inconsistent epochs; unable to load
ERROR 8239: Unable to map excluded source node *string*
ERROR 8257: Cannot find an UP node with an ACTIVE subscription to the replica shard
ERROR 8261: Dvmergeout failed: not a participating subscriber
ERROR 8266: Mergeout failed: not a participating subscriber
ERROR 8305: Partition projection failed: not a participating subscriber

subscriber
ERROR 8348: Shards *string* are missing participating subscription
ERROR 8360: Need to split partitions before copy/move/swap partitions
ERROR 8365: Unable to copy/move/swap partitions because some projection(s) contain unpartitioned data
ERROR 8400: Can not partition projection by given range due to the group by expression
ERROR 8423: Recovery failed due to missed event at epoch %#llx
ERROR 8430: Some storage containers must be split before the specified partition range can be dropped
ERROR 8435: The number of nodes in communal storage (*value*) is less than the number of nodes specified (*value*)
ERROR 8453: Rebalance: Refreshing projection *string*
ERROR 8459: Node *string* is shut down
ERROR 8483: Execution aborted by cluster configuration changes at catalog version *value*, more recent than our latest sync at version *value* on this node
ERROR 8501: Cannot create miniroses on node *string*
ERROR 8528: Subscription identified by (*string*,*string*) is already marked for removal
ERROR 8562: Node failure during planning
ERROR 8567: ROSs were moved/deleted during TM operation
ERROR 8570: Can't find the opened file [*string*]
ERROR 8592: Storage container *value* of table *string* has been moved to table *string*
ERROR 8594: Cannot do recovery by container for aggregate projection
ERROR 8603: cluster_config.json file does not exist!
ERROR 8605: Could not find spread security details in file
ERROR 8606: Database config parameter change interfered with this statement
ERROR 8613: Key id or contents are not of proper size. Id: *string* (*value*), Key hex len: *value*
ERROR 8620: Spread encryption key [*string*] not found
ERROR 8635: Attempted access to parameter '*string*' of type '*string*' using incompatible accessor '*string*'
ERROR 8639: Error checking compatibility in cluster_config.json. Details -- *string*
ERROR 8640: Error parsing version information in string *string* - did not match expected format v%*value*.%*value*.%*value*-%*value*
ERROR 8655: DDL statement interfered with this statement; subscriptions changed
ERROR 8676: DML/DDL interfered with ALTER COLUMN TYPE
ERROR 8684: Cannot execute query with temporary table because session subscriptions have changed since the start of this session
ERROR 8685: Cannot modify temporary table *string* because session subscriptions have changed since the start of this *string*
ERROR 8690: Current column data does not conform to the new type
ERROR 8702: Schema *string* is already owned by *string*
ERROR 8713: Database can only be revived from an older release and from a database version on or after Vertica9.1.0-0. The current database version is *string*, while the configuration file version is *string*
ERROR 8715: Fail to compute network routing checksum
ERROR 8731: Cannot obtain row count for projection *string*
ERROR 8736: Partial recovery on projection *string* is detected (cpe = *value*, startEpoch = *value*). Need to restart recovery
ERROR 8738: The storage location "*string*" has changed during the plan
ERROR 8742: No response from nodes creating DFS files: *string*
ERROR 8743: No response from required nodes (*string*) in '*string*'
ERROR 8745: Node *string* is not participating for shard *string*. Probably caused by concurrent subscription changes

by concurrent subscription changes
ERROR 8748: Compact storage failed: storage containers have been dropped/moved
ERROR 8752: Mergeout failed: storage containers have been dropped/moved
ERROR 8754: Recovery failed because SC *value* [0x*value*, 0x*value*] straddles endEpoch *value* to discard
ERROR 8755: Recovery failed because SC straddles discard epoch
ERROR 8800: Metafunction in multi-statement command is not supported
ERROR 8806: UDx rejected row with invalid index *value*
ERROR 8842: Could not upload the library [*string*] to communal storage. The tar file [*string*] does not exist
ERROR 8879: The storage container *value* to reocver have been dropped by tuple mover operations
ERROR 8884: Cannot create Depot storage location. Target node is down and there is no depot location on the initiator to estimate the size
ERROR 8891: Unable to refresh column on partitions already merged
ERROR 8892: Unable to refresh column on partitions because some projection(s) contain unpartitioned data
ERROR 8912: FileColumnReader: unable to open position index [*string*]: *string*
ERROR 8925: Unable to analyze statistics on partitions already merged
ERROR 8926: Unable to analyze statistics on partitions because some projection(s) contain unpartitioned data
ERROR 8927: Update storage catalog failed: not a participating subscriber
ERROR 8948: DDL interferes with this statement: requested storages no longer exist
ERROR 8969: Node *string* does not subscribe to shard *string* during backup, which is different from current node subscriptions
ERROR 8970: Node subscription for node: *string* has changed between backup and restore
ERROR 8984: Shard count in the restore db is different from backup db (backup db: *value*, restore db: *value*)
ERROR 8988: Subscription of node *string* is changing
ERROR 8991: The node *string* is no longer UP
ERROR 8992: The number of active node subscription has changed from *value* to *value*
ERROR 9034: *string* expression of complex type column "*string*" cannot be altered
ERROR 9082: Error loading library file [*string*]: *string*
ERROR 9084: Missing library tarball on communal storage [*string*]
ERROR 9099: Cannot find participating nodes to run the query
ERROR 9105: DDL interferes with this statement: table *value* no longer exists
ERROR 9115: Node is not up yet
ERROR 9119: Primary subcluster "*string*" cannot be promoted
ERROR 9120: Secondary subcluster "*string*" cannot be demoted
ERROR 9157: Dependent storage containers are dropped during dvmergeout
ERROR 9193: Subscription (node *string*) to Shard *string*(*value*) has changed or is changing or has changed
ERROR 9316: Current set of up nodes do not satisfy dependencies for DFS file retrieval from DFS file distribution *value*
ERROR 9351: Unable to take snapshot when subscription to *string* is not ACTIVE
ERROR 9381: Column "*string*" does not exist in table *string*
ERROR 9396: Node *string* has resubscribed to shard *string*. Probably caused by concurrent subscription changes or node restart
ERROR 9412: Depot location has not been found on all nodes
ERROR 9429: All nodes must be node_state UP and node_type PERMANENT/EPHEMERAL. Node *string* is *string*

ERROR 9432: Communal storage location [*string*] is an invalid path for file system [*string*]
ERROR 9433: Communal storage location is empty
ERROR 9434: Concurrent database migration jobs are running against the same communal storage location [*string*]
ERROR 9438: Database must be elastic to start migration
ERROR 9439: Database on *string* appears to be in use. Lease expires at [*string*]
ERROR 9440: Depot location [*string*] is an invalid path for file system [*string*]
ERROR 9441: Depot location is empty
ERROR 9442: Down/recovering nodes are detected. All nodes must be UP to start migration
ERROR 9462: Node *string* are added during database migration
ERROR 9463: Path [*string*] is not a local path. Please enter a local path
ERROR 9464: Path [*string*] is not an absolute path. Please enter an absolute path
ERROR 9475: The state of nodes are changing during database migration
ERROR 9494: Error writing error log to [*string*]
ERROR 9513: Refresh cannot be finished when nodes are recovering
ERROR 9514: The states of UP nodes have been changed
ERROR 9522: DDL interferes with this statement: projection *value* no longer exists
ERROR 9523: DDL interferes with this statement: table *string* is no longer partitioned
ERROR 9525: A different database[*string* (revive ID *string*)] exists on the communal storage location [*string*]
ERROR 9526: A different database[*string* (revive ID *string*)] has already existed on the communal storage location [*string*]
ERROR 9627: Complex type *string* of column *string* is not supported
ERROR 9630: HDFS path [*string*] detected. Hdfs URIs are not supported in database migration. Please use webhdfs or swebhdfs URIs in migrate_enterprise_to_eon
ERROR 9633: Local path [*string*] detected. Please set a valid location for communal storage
ERROR 9644: Cannot Drop: ResourcePool *string* depends on ResourcePool *string*
ERROR 9658: Unbundled storage containers, projections with grouped ROSes/columns, or storage containers with inconsistent segmentation bounds are found. See error log at [*string*]
ERROR 9661: Application with code *value* isn't registered
ERROR 9666: Process should not be called since processWithMetadata is defined
ERROR 9681: Spread Decryption failed (*value* bytes)
ERROR 9698: Cannot replicate subcluster properties: two subclusters do not have the same number of nodes
ERROR 9716: Two subcluster must be both primary or both secondary
ERROR 9725: Cannot drop subscription of shard *string* on node *string* because the node is the primary subscriber of the shard
ERROR 9729: Inconsistent projections, grouped ROSes, or projections with grouped columns have been found. See error log at [*string*]
ERROR 9732: Cannot replicate properties from subcluster *string*
ERROR 9733: Cannot replicate properties to subcluster *string*
ERROR 9734: Cannot replicate properties: increasing number of control nodes of subcluster *string* from *value* to *value* will result in exceeding maximum allowed *value* control nodes in the cluster
ERROR 9735: Cannot replicate properties: two subclusters have different control set size values
ERROR 9906: A concurrent loading operation interfered with this statement
ERROR 9930: TS: transfer reject: *string*

ERROR 9931: Cannot add new nodes to the subcluster because every control node in the subcluster has an invalid IP address
ERROR 9971: Projection *string* is aggregate projection and cannot be split
ERROR 9986: Type is not a complex type
ERROR 10004: ArrayWriter cannot be used by this function type
ERROR 10008: Invalid use of *string*::*string*
ERROR 10112: Cannot set NOT NULL field to NULL
ERROR 10128: DataChunks: cannot dispatch chunk '*string*' (*string*): no dispatcher '*string*'
ERROR 10239: Cannot find session ID *string*
ERROR 10283: Set types cannot be an array element
ERROR 10328: Nested internal statements with different session catalog content are not yet supported
ERROR 10394: Location cannot be dropped as it stores data or DFS files
ERROR 10410: DDL interferes with this statement: storage location [*value*] no longer exists
ERROR 10422: Running DML statements is not possible in read-only mode
ERROR 10427: Transaction commit aborted since the database has lost the quorum
ERROR 10428: Transaction commit aborted since the database is currently in read-only mode
ERROR 10476: Cannot submit task - scheduler is deinitialized
ERROR 10480: JobTracker::setJobExecutorNodeName(*string*): Unknown job *value*
ERROR 10484: Error: expected 1 argument, saw *value*
ERROR 10485: Error: invalid password for executing test function
ERROR 10491: Session subscriptions do not match catalog; likely result of concurrent subscription changes
ERROR 10508: DDL statement interfered with this statement: Complex type mismatch
ERROR 10544: Tuning session "*string*" not active
ERROR 10545: Tuning session is already active
ERROR 10630: Cannot create SSL context for HTTP server
ERROR 10631: Cannot start HTTP server

Fatal messages

FATAL 2975: Data directory [*string*] has group or world access
FATAL 2976: Data directory [*string*] has wrong ownership
FATAL 6303: JVM resources not available on all nodes; unable to proceed

Needs_Remote_Table messages

NEEDS_REMOTE_TABLE 5964: Needs remote table

Notice messages

NOTICE 8522: No node in UP state has written a cluster config file

Warning messages

WARNING 2316: Can't match imported node '*string*' to node in current database
WARNING 4177: Not enough nodes are up for Projection <*string*> to be available, marking it as out of date
WARNING 6870: Multiple values for the parameter *string*. The parameter will not be set
WARNING 8009: Running KV query inside existing transaction. There will be a performance implication. Consider rerunning the query outside of an existing transaction
WARNING 8151: Session resource pool does not exist
WARNING 8470: Attempting to load snapshot with inconsistently applied upgrade tasks
WARNING 8661: Storage may need to be reorganized in accordance with the partition GROUP BY policy

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 55006

This topic lists the messages associated with the SQLSTATE 55006.

SQLSTATE 55006 description

OBJECT_IN_USE

Messages associated with this SQLState

Error messages

ERROR 2307: Can't drop self
ERROR 3003: DDL statement interfered with Database Designer
ERROR 3004: DDL statement interfered with query replan
ERROR 3007: DDL statement interfered with this statement
ERROR 3896: Manual mergeout not supported while tuple mover is running
ERROR 4122: No up-to-date super projection left on the anchor table of projection *string*
ERROR 4139: Node *string* transitioned to state UP during this statement
ERROR 4145: Node is active and cannot be altered
ERROR 4455: Projection *string* cannot be dropped because K-safety would be violated
ERROR 4470: Projection cannot be dropped because history after AHM would be lost
ERROR 4488: Projections cannot be dropped or data would be lost due to down nodes
ERROR 4527: Rebalance is already running
ERROR 4528: Rebalance is already scheduled to run in the background
ERROR 4971: The status of one or more nodes changed during query planning
ERROR 6052: The system must retain at least one control node after the drop
ERROR 6162: Cannot alter a control node to be a STANDBY node
ERROR 6163: Cannot alter initiator node to STANDBY
ERROR 7219: A DDL statement interfered with this statement: constraint '*string*' has been disabled or dropped on table '*string*'
ERROR 7220: A DDL statement interfered with this statement: constraint '*string*' has been enabled on table '*string*'
ERROR 7402: DDL statement interfered with this statement. Text indices don't line up
ERROR 7616: SBJobTracker::createMarker: existing projOid *value*
ERROR 7888: DDL statement (*string*) interfered with this statement
ERROR 9031: Projection CPE is *value*, expected refresh start epoch *value*
ERROR 9204: Cannot promote or demote subcluster including the initiator node
ERROR 9543: Realignment will cause the subcluster having no control nodes
ERROR 9548: Dropping the nodes will result in subcluster *string* having no control nodes
ERROR 9609: Cannot drop this subcluster, because other subclusters depend on the control nodes in this subcluster
ERROR 9610: Cannot realign this subcluster, because other subclusters depend on the control nodes in this subcluster
ERROR 9620: Cannot drop nodes, because other subclusters depend on the control nodes in the dropping list
ERROR 9731: Cannot replicate properties for subcluster *string*, because other subclusters depend on the control nodes in this subcluster
ERROR 9907: A DDL statement interfered with this statement: A constraint has been disabled or dropped on table '*string*'
ERROR 10340: Another operation (like object replication) interfered with this CTAS statement

Warning messages

WARNING 4896: Table (*value*) has been dropped

Note

The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 55V02

This topic lists the messages associated with the SQLSTATE 55V02.

SQLSTATE 55V02 description

CANT_CHANGE_RUNTIME_PARAM

Messages associated with this SQLState

WARNING 4324: Parameter *string* will not take effect until database restart WARNING 7567: This Parameter has been disabled and will not be set

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 55V03

This topic lists the messages associated with the SQLSTATE 55V03.

SQLSTATE 55V03 description

LOCK_NOT_AVAILABLE

Messages associated with this SQLState

ERROR 5156: Unavailable: *string* - Locking failure: *string* ERROR 5157: Unavailable: [Txn %#llx] *string* - *string* error *string*

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 55V04

This topic lists the messages associated with the SQLSTATE 55V04.

SQLSTATE 55V04 description

TM_MARKER_NOT_AVAILABLE

Messages associated with this SQLState

ERROR 2082: A *string* operation is already in progress on projection *string.string* [container *value* txnid *value* session *string*] ERROR 2083: A *string* operation is already in progress on projection *string.string* [txnid *value* session *string*]

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 57014

This topic lists the messages associated with the SQLSTATE 57014.

SQLSTATE 57014 description

QUERY_CANCELED

Messages associated with this SQLState

Cancel messages

CANCEL 4278: Operation canceled

Error messages

ERROR 2246: Audit canceled
ERROR 2279: Bulk Import canceled
ERROR 2325: Canceled (in *string*)
ERROR 2326: Canceled: *string* - Locking canceled: *string*
ERROR 2327: Canceled: [Txn %#llx] *string* - *string string*
ERROR 2576: Catchup recovery interrupted
ERROR 2704: Connection canceled
ERROR 2996: DBDesigner canceled by user
ERROR 3286: evaluate_delete_performance canceled
ERROR 3319: Execution canceled (compile)
ERROR 3320: Execution canceled (prepare)
ERROR 3321: Execution canceled (start)
ERROR 3322: Execution canceled by operator
ERROR 3323: Execution got unlucky!
ERROR 3324: Execution intentionally failed
ERROR 3326: Execution time exceeded run time cap of *string*
ERROR 3515: import_catalog_objects canceled
ERROR 4114: No super projection available for analyze_statistics
ERROR 4143: Node failure in *string*
ERROR 4278: Operation canceled
ERROR 4287: Operator intervention on *string*
ERROR 4380: Plan canceled prior to execute call
ERROR 4439: Processing aborted by peer on *string*
ERROR 4496: Query canceled while waiting for resources
ERROR 4520: Read failed in FileColumnReader: *string string*
ERROR 4787: Statement abandoned due to subsequent DDL
ERROR 4789: Statement is canceled
ERROR 4843: Subsession interrupted
ERROR 5757: build_flextable_view canceled
ERROR 5787: compute_flextable_keys canceled
ERROR 5915: Hcatalog webservices query canceled
ERROR 5953: materialize_flextable_columns canceled
ERROR 6292: Internal query raised exception during ALTER TABLE ADD CONSTRAINT
ERROR 6389: Rebalance task *string* canceled
ERROR 6535: Table Owner lock canceled
ERROR 7278: StopExecution BeforePlan
ERROR 7279: StopExecution CompilePlan
ERROR 7280: StopExecution ExecutePlan
ERROR 7281: StopExecution InitPlan
ERROR 7282: StopExecution Plan
ERROR 7283: StopExecution PreparePlan
ERROR 7284: StopExecution SerializePlan
ERROR 7457: Mergeout canceled
ERROR 8467: Uploader was interrupted due to user cancel or query failure
ERROR 8895: Canceled during globbing
ERROR 8912: FileColumnReader: unable to open position index [*string*]: *string*
ERROR 9652: Receive on *string*: query has been canceled
ERROR 9766: RPC: peer cancelled response
ERROR 9767: RPC: peer cancelled response: *string*
ERROR 9927: TransferSender: send failed to node [*string*]. Transport failure
ERROR 9941: StopExecution PreparePlanSections
ERROR 10473: Query exceeded maximum runtime
ERROR 10488: RPC[*string*]: could not get file [*string*] from *string*. The remote probably went down
ERROR 10563: Client is unresponsive; the query has been canceled
ERROR 10642: *string* was interrupted due to user cancel or query failure

FATAL 2605: Client canceled session *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 57015

This topic lists the messages associated with the SQLSTATE 57015.

SQLSTATE 57015 description

SLOW_DELETE

Messages associated with this SQLState

ERROR 5822: Detected slow delete for projection *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 57V01

This topic lists the messages associated with the SQLSTATE 57V01.

SQLSTATE 57V01 description

ADMIN_SHUTDOWN

Messages associated with this SQLState

ERROR 3556: Initiating node is down ERROR 4150: Node status is not UP ERROR 8258: Initiating node is down

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 57V03

This topic lists the messages associated with the SQLSTATE 57V03.

SQLSTATE 57V03 description

CANNOT_CONNECT_NOW

Messages associated with this SQLState

Error messages

ERROR 2863: Could not fork UDx zygote process, *string*
ERROR 2929: Couldn't create new UDx side process, failed to get UDx side process info from zygote: *string*
ERROR 3363: Failed to connect to side process, *string*
ERROR 3364: Failed to connect to UDx zygote, *string*
ERROR 3366: Failed to create new UDx side process, couldn't connect to it: *string*
ERROR 4720: Session manager cannot add an external session - disabled
ERROR 5699: Cannot find java binary: neither the Linux environment variable JAVA_HOME nor Vertica config parameter JavaBinaryForUDx is set
ERROR 5702: Couldn't create new UDx side process: *string*
ERROR 5803: Couldn't create new UDx side process, failed to set locale information: *string*
ERROR 6712: Couldn't cancel the external procedure: *string*
ERROR 6715: Couldn't execute the external procedure: *string*
ERROR 9529: Couldn't create new UDx side process for language *string*. See details in log files written in UDxLogs/

Fatal messages

FATAL 4149: Node startup/recovery in progress. Not yet ready to accept connections
FATAL 4748: Shutdown in progress. No longer accepting connections
FATAL 5785: Cluster Status Request by *string*:*string*
FATAL 6361: Node is an active STANDBY node that cannot accept connections
FATAL 6881: Non-superuser is not allowed to log into a DB started in UNSAFE mode

Warning messages

WARNING 2937: Couldn't set TCP_NODELAY option, might get latency in RPC message delivery: *string*
WARNING 6707: Could not fork UDx zygote process: *string*
WARNING 10579: OpenSSL key generation for the zygote failed: [*string*]

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 57V04

This topic lists the messages associated with the SQLSTATE 57V04.

SQLSTATE 57V04 description

DML_COMMIT_DURING_SHUTDOWN

Messages associated with this SQLState

ERROR 6523: Cannot commit DML while a node is shutting down
ERROR 8651: Cannot commit transaction modifying data while a node is shutting down

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 58030

This topic lists the messages associated with the SQLSTATE 58030.

SQLSTATE 58030 description

IO_ERROR

Messages associated with this SQLState

Error messages

ERROR 2005: *string*
ERROR 2024: *string* Error occurred during BZIP decompression. BZIP error code: *value*
ERROR 2026: *string* Error occurred during ZLIB decompression. ZLIB error code: *value*, Message: *string*
ERROR 2432: Cannot get LibraryPath from node: *string*
ERROR 2433: Cannot get MD5 checksum from node: *string*
ERROR 2600: Checksums do not match (computed=0x*value*, fromdisk=0x*value*) discarding checkpoint!
ERROR 2674: ColumnAccessBase open error
ERROR 3197: Error deserializing snapshot info from file *string*
ERROR 3255: Error reading from file *string*
ERROR 3303: Exception during measurement deserialization
ERROR 3305: Exception during Stats deserialization:*string*
ERROR 3408: File size on disk does not match catalog for *string*
ERROR 3412: FileColumnReader: Get block *string* @ *value* error
ERROR 3550: Info file *string* does not exist
ERROR 3796: IO_ERROR writing data file [*string*]
ERROR 4364: Performance measurement of [*string*] failed
ERROR 4518: Read error when expanding glob: *string*
ERROR 4632: RowAccessBase open error
ERROR 5124: Unable to close catalog file [*string*]
ERROR 5126: Unable to create catalog file [*string*]
ERROR 5131: Unable to drop catalog file [*string*]
ERROR 5141: Unable to open file [*string*]
ERROR 5152: Unable to write catalog file [*string*]
ERROR 5153: Unable to write checksum to catalog file [*string*]
ERROR 5154: Unable to write object to catalog file [*string*]
ERROR 5887: Failed to mount file system *value*: *string*
ERROR 5901: Filesystem does not pass basic test: *string*
ERROR 5902: Filesystem does not pass basic test: I/O data differ
ERROR 6082: Unable to open spread conf file *string* for writing
ERROR 6118: *string* Error occured during LZO decompression (compressed data violation).LZO error code: *value*
ERROR 6258: Exception during file writer deserialization: *string*
ERROR 6259: Exception during file writer serialization: *string*
ERROR 6260: Exception during snapshot deserialization: *string*
ERROR 6261: Exception during snapshot serialization: *string*
ERROR 6262: Exception during storage container deserialization: *string*
ERROR 6263: Exception during storage container serialization: *string*
ERROR 6527: Filesystem does not support snapshot
ERROR 6528: Filesystem failed to restore snapshot
ERROR 6550: Cannot open catalog source file *string*
ERROR 6746: Duplicate storage location id: *value*
ERROR 6762: Error manifest format
ERROR 6776: Failed to glob [*string*] because of error: *string*
ERROR 6808: Improperly ordered or duplicate storage ids: *string*, *string*
ERROR 6830: Invalid section for storage locations
ERROR 6860: Malformed object line: *string*
ERROR 6861: Malformed storage location line: *string*
ERROR 6984: tar_append_file: *string*: *value*
ERROR 6985: tar_append_tree failed:*value*. Real dir:*string*; save dir: *string*
ERROR 6987: tar_extract_all failed:*value*. Extract path:*string*
ERROR 6988: tar_open failed:*value*. Path:*string*
ERROR 7178: Error loading from all sources

ERROR 7718: Exception on closing file: *string*
ERROR 7720: Exception on flushing file: *string*
ERROR 7721: Exception on opening file: *string*
ERROR 7722: Exception on reading file: *string*
ERROR 7723: Exception on resizing file: *string*
ERROR 7724: Exception on writing to file: *string*
ERROR 7728: No place to store chunk files
ERROR 7869: No files match when expanding glob: [*string*]
ERROR 8025: Exception when open blob file: *string*
ERROR 8067: Cannot open file *string* for tiered catalog printer
ERROR 8085: Empty filename specified
ERROR 8340: Unable to open transaction log file [*string*]
ERROR 8350: Error syncing missing transaction logs to [*string*] following node startup: *string*
ERROR 8351: Error syncing transaction logs to [*string*]: *string*
ERROR 8352: Failed to copy checkpoint to [*string*] for commitid=*value* at global catalog version *value*: *string*
ERROR 8355: Failed to write cluster configuration file for catalog version *value*: *string*
ERROR 8368: Cluster configuration file [*string*] is empty
ERROR 8370: Error loading remote catalog: *string*
ERROR 8374: Unable to find cluster configuration file [*string*]
ERROR 8380: Txn log [*string*] synced from [*string*] is larger: (*value* remotely vs *value* locally)
ERROR 8381: Txn log [*string*] was only partially synced to [*string*]: (*value* remotely vs *value* locally)
ERROR 8458: Cannot write file: export_log.json
ERROR 8471: Error writing catalog diffs from *string*: *string*
ERROR 8472: Error writing catalog diffs to *string*: *string*
ERROR 8493: UploadFileTask::File *string* not found for uploading
ERROR 8503: Cannot get a reply from node: *string*
ERROR 8539: Exception during DTRosFileInfo serialization: *string*
ERROR 8540: Exception during RosMessage serialization: *string*
ERROR 8541: Exception during sending DTContainers (temp name = *string*)
ERROR 8542: Exception during sending EOF : *string*
ERROR 8543: FileColumnWriter::Exception during file name sending : *string*
ERROR 8551: StorageBundleWriter: Exception during file name sending : *string*
ERROR 8552: StorageBundleWriter: Exception during flush bundled data: *string*
ERROR 8553: StorageBundleWriter: Exception during footer sending : *string*
ERROR 8578: Exception during file name deserialization: *string*
ERROR 8579: Exception during ROS message deserialization: *string*
ERROR 8819: No Library tar file in *string*
ERROR 8831: Cannot append to the memory report file: *string*
ERROR 8990: Temp path is not set when opening S3 file operator for file *string*
ERROR 9208: Cannot create directory for blob [*string*] at [*string*]. Reason: *string*
ERROR 9421: Temp path is not set when opening webhdfs file operator for file *string*
ERROR 9447: Exception during creating remote node direcotry on [*string*]: *string*
ERROR 9448: Failed to copy checkpoint (commitid *value*, version *value*) to [*string*]: *string*
ERROR 9449: Failed to copy txn logs to remote directory *string*: *string*
ERROR 9521: Cannot write import log to file [*string*]
ERROR 9742: File size changed during read: read *value*, initial size *value*
ERROR 9888: Temp path is not set when opening AzureBlob file operator for file *string*

ERROR 9928: TS: cannot get data from stream
ERROR 10130: Error renaming temp directory to '*string*', it already exists.
Please remove it or specify another directory
ERROR 10131: Exception during externalExport_CheckNode: *string*
ERROR 10132: Exception during externalExport_Cleanup: *string*
ERROR 10133: Exception during externalExport_FinalizeInitiator: *string*
ERROR 10134: Exception during externalExport_FinalizeNode: *string*
ERROR 10135: Exception during externalExport_SetupInitiator: *string*
ERROR 10291: Target directory [*string*] exists. Please remove it or specify another directory
ERROR 10307: Error creating target directory '*string*', it already exists.
Please remove it or specify another directory
ERROR 10308: Error creating temp directory '*string*', it already exists.
Please retry your export
ERROR 10481: readTask returned failure with no additional exception information available

Warning messages

WARNING 2899: Could not remove file or directory [*string*]: *value*
WARNING 3478: getnameinfo_all() failed: *string*
WARNING 8341: Failed to create communal storage location: *string*
WARNING 8352: Failed to copy checkpoint to [*string*] for commitid=*value* at global catalog version *value*: *string*
WARNING 8353: Failed to remove *value* old checkpoints from *string*: *string*
WARNING 8354: Failed to remove old txn logs older than *value* from *string*: *string*
WARNING 8868: tar_close failed: *value*
WARNING 8994: Unable to stat file *string*: *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE 58V01

This topic lists the messages associated with the SQLSTATE 58V01.

SQLSTATE 58V01 description

UNDEFINED_FILE

Messages associated with this SQLState

ERROR 3664: Invalid filename. Input filename is an empty string
ERROR 8817: Depends can specify files and directories only. [*string*] is not a valid file or directory
ERROR 9507: Could not read cluster configuration file from *string*
ERROR 9770: TS: Cannot find file *string* for sending

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE V0001

This topic lists the messages associated with the SQLSTATE V0001.

SQLSTATE V0001 description

PLVSQL_ERROR

Messages associated with this SQLState

ERROR 10147: *string* string already defined
ERROR 10180: Errcode already defined
ERROR 10192: GET STACKED DIAGNOSTICS must be called from inside an EXCEPTION block
ERROR 10228: Must use PERFORM for SQL statements that return results
ERROR 10229: Object not found; should have been caught in static validation
ERROR 10258: Invalid exception condition name "*string*"
ERROR 10337: Type "*string*" is not yet supported for stored procedures
ERROR 10436: Cannot substitute PL/vSQL cursor into SQL by value

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE V0005

This topic lists the messages associated with the SQLSTATE V0005.

SQLSTATE V0005 description

ASSERT_FAILURE

Messages associated with this SQLState

ERROR 2005: *string*

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE V1001

This topic lists the messages associated with the SQLSTATE V1001.

SQLSTATE V1001 description

LOST_CONNECTIVITY

Messages associated with this SQLState

Error messages

ERROR 2709: Connection to spread closed
ERROR 4048: NetworkReceive: Decompression failed
ERROR 4054: NetworkSend on *string*: failed to open connection to node *string* (*string*)
ERROR 4140: Node *string* was not successfully added to the cluster
ERROR 4142: Node failure during execution
ERROR 4533: Receive: Decompression failed
ERROR 4534: Receive on *string*: Message receipt from *string* failed [*string*]
ERROR 4541: ReceiveFiles on *string*: Unexpected end of stream from *string* [*string*]
ERROR 4547: RecvFiles on *string*: Open failed on node [*string*] (*string*)
ERROR 4572: RemoteSend: Open failed on node [*string*], IPAddr is [*string*], port is [*value*] (*string*)
ERROR 4683: Send: Connection not open [*string* tag:*value* plan *value*]
ERROR 4684: Send: Open failed on node [*string*] (*string*)
ERROR 4689: SendFiles on *string*: Open failed on node [*string*] (*string*)
ERROR 5579: Failure in send on socket *string*: *string*
ERROR 5624: NetworkReceive on *string*: failed to open connection to node *string* (*string*)
ERROR 5625: NetworkReceive on *string*: Message receipt from *string* failed: *string*
ERROR 5658: Send on *string*: Open failed on node [*string*] (Address lookup for *string*(*string*) failed)
ERROR 7116: Receive on *string*: open failed for node *string* (*string*) --- has query been cancelled?
ERROR 8537: DataTargetProxy on *string*: handle is canceled
ERROR 8538: DataTargetProxy on *string*: Unexpected end of stream from *string* [*string*]
ERROR 9020: DataTargetProxy: Unexpected end of stream from *string*
ERROR 9061: Network address lookup for [*string*] failed
ERROR 9446: Exception : Database migration hasn't been completed on node *string*
ERROR 9667: TransferReceiver: Open failed on node [*string*] (*string*)
ERROR 9669: TransferSendRequest: Open failed on node [*string*] (*string*)
ERROR 9670: TransferSendRequest: send failed to node [*string*]. Reason unknown
ERROR 9671: TransferSendRequest: send failed to node [*string*]. Reason: *string*
ERROR 9672: TransferService is shutting down
ERROR 9673: TransferService: node *string* was down on transfer start
ERROR 9674: TransferService: open failed on node [*string*]
ERROR 9750: Cannot send to node *string*(*string*) due to invalid id (expected *string*)
ERROR 9769: TransferService: node [*string*]: cannot find node [*string*]

Rollbacktxn messages

```
ROLLBACKTXN 4236: One or more nodes did not open a data connection to
                  this node.  This may indicate a network
                  configuration problem.  Check that the private
                  interfaces used for communication among the cluster
                  hosts reside in the same subnet and are returned
                  first by host address lookup
```

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE V1002

This topic lists the messages associated with the SQLSTATE V1002.

SQLSTATE V1002 description

K_SAFETY_VIOLATION

Messages associated with this SQLState

Error messages

ERROR 2406: Cannot drop *value* nodes from a *value* node cluster with *value* nodes
down - cluster would appear partitioned and database would
shutdown. Bring some nodes up and try again
ERROR 4477: Projection KSAFE *value* can not be met with only *value* nodes
ERROR 4478: Projection KSAFE override *value* cannot be less than current
system K-safe value *value*
ERROR 9096: Cannot *string* Subcluster "*string*". The cluster would appear
partitioned and the database would shutdown
ERROR 9097: Cannot demote Subcluster "*string*". The operation would result
in loss of shard coverage
ERROR 9098: Cannot drop *value* primary nodes. The cluster would appear
partitioned and the database would shutdown
ERROR 9101: Cannot support K=*value* on only *value value*
ERROR 9108: Dropping node would leave one or more shards with no
active subscribers
ERROR 9125: The cluster must keep at least one primary node at all
times
ERROR 9378: System is not k-safe. Cannot recover node

Notice messages

NOTICE 2520: Cannot shutdown unsafe cluster with this command

Warning messages

WARNING 2957: Current system KSAFE level is not fault tolerant
WARNING 8657: KSAFE > 0 for temporary table unsupported in Eon mode,
no buddy projections created

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE V1003

This topic lists the messages associated with the SQLSTATE V1003.

SQLSTATE V1003 description

CLUSTER_CHANGE

Messages associated with this SQLState

ERROR 2094: A node has come UP which missed ALTER COLUMN check
ERROR 2095: A node has come UP which missed drop partition keys check
ERROR 2096: A node has come UP which missed partitioning check
ERROR 2097: A node has entered the cluster since the session started
ERROR 2098: A node has entered the cluster since the session was
started
ERROR 2099: A node has entered/left the database cluster
ERROR 5312: Up node set changed during restore
ERROR 5523: A node has come UP which missed ADD COLUMN statement
ERROR 6876: No nodes up!
ERROR 7214: Node types changed during restore
ERROR 8480: A node has come UP which missed ADD COLUMN O LOCK
ERROR 10138: Invalid config section: valid values are [*string*]
ERROR 10551: *string* is not supported in read-only mode

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE V2000

This topic lists the messages associated with the SQLSTATE V2000.

SQLSTATE V2000 description

AUTH_FAILED

Messages associated with this SQLState

ERROR 3493: GSS error: *string*. Error details: (*string*/*string*)
ERROR 3718: Invalid old password
ERROR 6635: An error occurred during GSS authentication
ERROR 6636: An error occurred during kerberos authentication

> **Note**
> The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE V2001

This topic lists the messages associated with the SQLSTATE V2001.

SQLSTATE V2001 description

LICENSE_ISSUE

Messages associated with this SQLState

Error messages

ERROR 2005: *string*
ERROR 2382: Cannot create another node. The current license permits *value* node(s) and the database catalog already contains *value* node(s)
ERROR 3248: Error parsing license end date
ERROR 4943: The Enterprise Edition is installed. You cannot downgrade from the Enterprise Edition to the Community Edition
ERROR 5517: Your Vertica license is invalid or has expired
ERROR 6164: Cannot alter STANDBY node. The current license permits *value* node(s) and the database catalog already contains *value* node(s)
ERROR 6549: Cannot install new license to the database. New license permits *value* node(s) but the database catalog already contains *value* node(s)
ERROR 7018: Unable to audit license: database lacks an active license!
ERROR 8263: Invalid combination of input licenses
ERROR 8268: No license found
ERROR 8277: Unable to overwrite existing licenses
ERROR 8450: Cannot drop the license. The remaining licenses permit *value* node(s) and the database catalog contains *value* node(s)
ERROR 8625: Unsupported licensable feature
ERROR 8724: All of your Vertica licenses have expired
ERROR 8733: Node count exceeded license allowed node count limit
ERROR 9397: None of the licenses in this file can be installed [*string*]*string*

Notice messages

NOTICE 8723: Vertica license *string* is in its grace period; grace period expires in *value string*

Warning messages

WARNING 3863: License issue: *string*
WARNING 8724: All of your Vertica licenses have expired
WARNING 9164: Found *value* node(s) in the database catalog but only *value* node(s) are permitted by the current license

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE VC001

This topic lists the messages associated with the SQLSTATE VC001.

SQLSTATE VC001 description

CONFIG_FILE_ERROR

Messages associated with this SQLState

Error messages

ERROR 10357: Secure location to store the intermediate-results cannot be empty

Fatal messages

FATAL 2450: Cannot load configuration from *string*

Warning messages

WARNING 3833: Kerberos keytab file must be owned by the database user, and have no permissions for "group" or "other"
WARNING 4951: The Kerberos keytab file is either empty or too small in size to be valid

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE VD001

This topic lists the messages associated with the SQLSTATE VD001.

SQLSTATE VD001 description

DESIGNER_FUNCTION_ERROR

Messages associated with this SQLState

Error messages

ERROR 2010: *string* cannot be NULL
ERROR 2012: *string* clause does not exist in the query
ERROR 2204: Anchor table of projection *string* is a Session scoped table
ERROR 2205: Anchor table of projection *string* is a System table
ERROR 2211: API *string* not available in old DBD engine
ERROR 2212: API cannot take query input file and query string, only one can be set
ERROR 2328: Cannot *string* as design was created already
ERROR 2336: Cannot add another Comprehensive design to deployment *string*
ERROR 2337: Cannot add design projections in extend catalog type deployment *string* in workspace *string*
ERROR 2338: Cannot add design tables to design *string* because there are populated designs
ERROR 2339: Cannot add design to deployment *string* because design *string* has not been populated
ERROR 2369: Cannot clear design tables from design *string* because there are populated designs

are populated designs
ERROR 2375: Cannot compute projections to be dropped for only incremental designs deployment
ERROR 2394: Cannot design encoding for Projection *string* as it does not have any AUTO encoded columns
ERROR 2395: Cannot design encoding for Projection *string* as it is not SAFE -- Create its buddies
ERROR 2396: Cannot design/deploy for virtual system schema *string*
ERROR 2423: Cannot execute deployment when there are non-up-to-date safe projections for table *string*
ERROR 2456: Cannot load queries as design was populated already
ERROR 2463: Cannot output design projections because design is not available
ERROR 2464: Cannot output query because query id is invalid
ERROR 2471: Cannot populate drop projections in extend catalog type deployment *string* in workspace *string*
ERROR 2477: Cannot refresh projections for table *value* as it was dropped
ERROR 2480: Cannot remove any design table from design *string* because there are populated designs
ERROR 2485: Cannot remove workspace *string* because it does not exist
ERROR 2492: Cannot retrieve design tables for design *string* in workspace *string*
ERROR 2493: Cannot retrieve information for design *string* in workspace *string*
ERROR 2507: Cannot set k-safety when design *string* has been populated
ERROR 2657: Column '*string*' does not exist in Table *string*
ERROR 2658: Column '*string*' is duplicated in the column list
ERROR 3053: Deployment *string* already exists in workspace *string*
ERROR 3054: Deployment got canceled
ERROR 3056: Deployment ksafety should be equal or greater than design ksafety. Deployment ksafety is *value* and design ksafety is *value*
ERROR 3057: Deployment name cannot be NULL
ERROR 3058: Deployment Projections status is set to Error
ERROR 3060: Design *string* already exists
ERROR 3061: Design *string* already exists for workspace *string*
ERROR 3063: Design *string* has already been added to deployment *string*
ERROR 3064: Design *string* has not been populated in workspace *string* so projection cannot be added
ERROR 3065: Design *string* hasn't been populated
ERROR 3066: Design *string* in workspace *string* is not available
ERROR 3067: Design *string* is already populated
ERROR 3068: Design *string* is populated, remove design first (designer_remove_design)
ERROR 3071: Design name cannot have more than *value* characters
ERROR 3072: Design name may contain only alphanumeric or underscore characters
ERROR 3073: Design did not complete successfully, so deployment did not start
ERROR 3074: Design K-safety should be 0
ERROR 3077: Design name cannot have character '.'
ERROR 3079: Optimization objective cannot be NULL
ERROR 3080: Design query with design_query_id *value* does not exist
ERROR 3082: Design *string* does not exist
ERROR 3088: design_override_type *string* for table *string* already exists
ERROR 3100: Did not find any projections to design encodings for
ERROR 3101: Did not find design projections for projection ids given
ERROR 3102: Did not find design projections for tablePattern *string*
ERROR 3104: Did not find design tables to remove
ERROR 3105: Did not find projection id *value* in deployment *string* in workspace *string*

ERROR 3107: Did not find rows in deployment table for deployment *string* in workspace *string*
ERROR 3108: Did not find rows in designs table for workspace *string*
ERROR 3140: Dropping design without getting design projections, API call is of no use
ERROR 3166: Empty design name is not allowed
ERROR 3188: Error after projection refresh: *string*
ERROR 3194: Error creating workspace: Invalid workspace name
ERROR 3195: Error deleting deployment status table
ERROR 3202: Error during deployment while setting ksafety before deployment starts
ERROR 3203: Error during design: *string*
ERROR 3205: Error during drop projections: *string*
ERROR 3208: Error during projection creation: *string*
ERROR 3214: Error during remove design *string*
ERROR 3215: Error during rename projections: *string*
ERROR 3241: Error opening query input file [*string*]
ERROR 3250: Error querying deployment projections statements table
ERROR 3251: Error querying deployment projections table
ERROR 3252: Error querying design projections table for design *string* in workspace *string*
ERROR 3253: Error querying: *string*
ERROR 3266: Error status for projections to add for table *string*
ERROR 3267: Error status for projections to drop for table *string*
ERROR 3268: Error updating deployment projections table
ERROR 3270: Error while loading statistics into design tables for design *string*
ERROR 3277: Error writing to [*string*]
ERROR 3356: External table *string* is not a design table
ERROR 3358: Failed during select mark_design_ksafe(*value*)
ERROR 3415: Filename cannot be NULL
ERROR 3480: Given design *string* does not exist
ERROR 3543: Incremental design needs a query or an input query file to be set
ERROR 3649: Invalid Deploy Operation string *string*
ERROR 3650: Invalid deploy status string *string*
ERROR 3740: Invalid query input file [*string*]
ERROR 3795: Invalid design creator name
ERROR 3824: K cannot be *value* (maximum allowed is *value*)
ERROR 3825: K must be equal to or greater than *value*, cannot reduce current k-safety level
ERROR 3827: K-safety cannot be NULL
ERROR 3867: List of projections cannot be NULL
ERROR 3898: mark_design_ksafe(*value*) failed; some projections may not be k-safe
ERROR 4031: Namespace for LOCAL temporary tables cannot be used to add design tables
ERROR 4057: New ksafety cannot be less than 0
ERROR 4078: No deployment data in *string.string*
ERROR 4080: No drop entries found for deployment *string* in workspace *string*
ERROR 4099: No projections found for the projection ids string *string*
ERROR 4117: No tables found in schema *string*
ERROR 4118: No tables found in the table pattern given
ERROR 4119: No tables to design projections for
ERROR 4235: One of the design tables no longer exist
ERROR 4314: override_type *string* is invalid
ERROR 4460: Projection *string* does not exist
ERROR 4461: Projection *string* does not exist
ERROR 4466: Projection *string* to be refreshed was dropped
ERROR 4475: Projection id cannot be NULL

ERROR 4476: Projection id list cannot be NULL
ERROR 4479: Projection name cannot be NULL
ERROR 4497: Query Id cannot be NULL
ERROR 4498: Query referencing EPOCH column is not supported
ERROR 4503: Query table *string* does not exist
ERROR 4504: Query table contains multiple entries with qid = *value*
ERROR 4505: Query weight must be a positive number
ERROR 4525: Rebalance data cannot proceed when there are non-up-to-date projections in the catalog
ERROR 4526: Rebalance data failed during select mark_design_ksafe(*value*)
ERROR 4651: Schema *string* does not exist
ERROR 4652: Schema *string* is not a designer created schema, so it cannot be dropped
ERROR 4655: Schema name cannot be NULL
ERROR 4721: Session scoped table *string* is not a design table
ERROR 4783: Start deploy: deploy is already running on this node
ERROR 4866: System table *string* is not a design table
ERROR 4874: Systems tables within system schema *string* cannot be added as design tables
ERROR 4885: Table *string* does not exist
ERROR 4886: Table *string* does not exist anymore in the catalog
ERROR 4890: Table *string* is not a design table
ERROR 4902: Table name cannot be NULL
ERROR 4907: Table pattern cannot be NULL
ERROR 4920: Terminated after SO enum. See log for the content of the SOs
ERROR 4942: The design table entry with table name *string*.*string* is corrupted, as that table has been renamed in the Vertica catalog
ERROR 4976: There are *value* nodes. Deployment K = *value* is not possible
ERROR 4977: There are no projections to add in deployment *string* for workspace *string* so no projections can be dropped
ERROR 4980: There is 1 node. Deployment K = *value* is not possible
ERROR 4981: There is more than one design *string* in workspace *string*
ERROR 4983: There is no design tables system table in workspace *string*
ERROR 4995: This query is not supported in DBDesigner
ERROR 5363: User *string* does not have privileges to access design table: *string*
ERROR 5364: User *string* does not have privileges to access table: *string*
ERROR 5480: Workspace *string* cannot be a virtual system schema
ERROR 5481: Workspace *string* does not exist
ERROR 5482: Design *string* is configured for extend_catalog so no designs can be computed
ERROR 5483: Design *string* is configured for extend_catalog so remove drops is not supported
ERROR 5484: Design *string* is configured for extend_catalog so there are no design tables
ERROR 5485: Design *string* is configured for extend_catalog, there are no design tables
ERROR 5486: Workspace cannot be NULL
ERROR 5487: Design name cannot be NULL
ERROR 5564: Deployment Parallelism cannot be less than zero
ERROR 5565: Deployment parallelism cannot be NULL
ERROR 5573: Error generating results set
ERROR 5575: Error querying designs table
ERROR 5588: Fenced mode false is not supported for *string* functions
ERROR 5589: Fenced mode is not supported for SQL functions
ERROR 5591: Hurryup parameter cannot be NULL
ERROR 5597: Invalid input query: '*string*'
ERROR 5747: analyzeStats flag cannot be NULL
ERROR 5773: Cannot output deployment script because design is not

available
ERROR 5792: continueAfterError flag cannot be NULL
ERROR 5817: Deploy flag cannot be NULL
ERROR 5855: Did not find any tables to analyze correlations on
ERROR 5857: dropDesignAndCtx flag cannot be NULL
ERROR 5858: dropProjs flag cannot be NULL
ERROR 5866: Error while analyzing correlations for design table *string*.*string*
ERROR 5867: Error while analyzing count distincts for design table *string*.*string*
ERROR 5868: Error while analyzing count distincts on correlation sample for design table *string*.*string*
ERROR 5869: Error while analyzing segmentation skew for design table *string*.*string*
ERROR 5871: Error while loading or analyzing correlations in design tables for design *string*
ERROR 5907: Force option cannot be NULL
ERROR 5908: forceRecomputation flag cannot be NULL
ERROR 5919: Input cannot be NULL
ERROR 5938: isAdminUser flag cannot be NULL
ERROR 5939: K-safety of incremental designs must match the current system k-safety (which is *value*)
ERROR 5979: onlyScript flag cannot be NULL
ERROR 5980: outputScript flag cannot be NULL
ERROR 6049: The mode of analyzing correlations cannot be NULL
ERROR 6050: The mode of analyzing correlations is invalid
ERROR 6223: Design K-safety should be in [0,*value*] range
ERROR 6247: Error during deployment: no projections found for deployment *string*
ERROR 6489: Error querying v_catalog.projections table
ERROR 7191: Projection with expressions *string* is not supported for encoding design
ERROR 7379: Cannot generate unique name for Database Designer schema
ERROR 8810: No deployment data
ERROR 9106: Design interval cannot be NULL

Notice messages

NOTICE 4075: No data rebalancing required
NOTICE 4076: No data rebalancing required for replicated projections
NOTICE 4077: No data rebalancing required for segmented projections
NOTICE 4132: No work for rebalance data for replicated projections
NOTICE 4133: No work for rebalance data for segmented projections
NOTICE 4134: No work to be done for deployment/rebalance data -- No projections to add or drop

Warning messages

WARNING 2202: Anchor table for projection *string* does not exist, so it cannot be added to deployment
WARNING 2304: Can only load *value string* under the *string* design type
WARNING 2454: Cannot load invalid query: *string*
WARNING 3081: Design Query with design_query_id *string* does not exist
WARNING 3087: design_override_type *string* for query (design_query_id *value*) already exists
WARNING 3089: design_override_type *string* for table *string* does not exist
WARNING 3103: Did not find design tables to add
WARNING 3106: Did not find projections for design *string* in workspace *string*
WARNING 3489: Group-by override *value* on query *value* cannot be satisfied
WARNING 3574: INSERT query without SELECT is not supported: *string*
WARNING 3817: Join override *value* on query *value* cannot be satisfied
WARNING 4311: Override (design_override_id *value*) is ignored because the table *string* is no longer a design table
WARNING 4312: Override (design_override_id *value*) is ignored because the table does not exist
WARNING 4313: override_type *string* for query (design_query_id *value*) does not exist
WARNING 4499: Query referencing local temporary table *string* is not supported: *string*
WARNING 4500: Query referencing projection *string* is not supported: *string*
WARNING 4501: Query without referencing any catalog table is not supported: *string*
WARNING 4819: Subqueries in UPDATE/DELETE is not supported: *string*
WARNING 4888: Table *string* has no statistics or data. As a result, the proposed projections on this table may be suboptimal
WARNING 4891: Table *string* is not a design table, referenced in query (qid=*value*): *string*
WARNING 4991: This invalid query cannot be loaded: *string*
WARNING 4994: This non-SELECT query is not supported: *string*
WARNING 5390: User has insufficient privileges on table *string*
WARNING 5650: Query without referencing any design tables is not supported: *string*
WARNING 5657: Segmentation type of the projection *string* is not supported for encoding design, skipping
WARNING 5694: Weight for query_text '*string*' is '*value*'. Only positive weight values are accepted
WARNING 5870: Error while dropping existing correlations in design table *string*
WARNING 6040: Table *string* has no correlations
WARNING 6041: Table *string* has no data. As a result, no correlations were analyzed on this table
WARNING 6042: Table *string* has no statistics or data. As a result, no correlations were analyzed on this table
WARNING 6043: Table *string* has no statistics or data. As a result, no correlations were read into this table
WARNING 6096: User has insufficient privileges on table *string.string*
WARNING 7192: Projection with expressions *string* is not supported for encoding design, skipping
WARNING 8487: No new projections recommended for deployment by DBD

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE VP000

This topic lists the messages associated with the SQLSTATE VP000.

SQLSTATE VP000 description

USER_PROC_ERROR

Messages associated with this SQLState

ERROR 2059: *string* with specified name and parameters does not exist: *string*
ERROR 3354: External procedures directory has not been set
ERROR 3355: External procedures have not been installed
ERROR 3465: Function cannot be moved into a system schema
ERROR 4323: Parameter type is not valid for an external procedure: *string*
ERROR 4373: Phase *value* of multi-phase transform function marked prepass
ERROR 5232: Unrecognized identifier: *string*
ERROR 5368: User Defined Aggregates do not support fenced execution mode
ERROR 5372: User Defined Function type not found
ERROR 5374: User Defined Scalar Function *string* is giving bad numeric precision *value*, the maximum is *value*
ERROR 5375: User Defined Scalar Function *string* is giving bad string length *value*, the minimum is 1
ERROR 5376: User Defined Scalar Function *string* is giving typmod of precision *value*, larger than the max precision *value*
ERROR 5377: User Defined Scalar Function *string* provided non-zero precision (*value*) for Interval Year To Month
ERROR 5378: User Defined Scalar Function *string* provided precision *value*, larger than the maximum precision *value*
ERROR 5379: User Defined Scalar Function *string* provided range for Day To Second, but the function's return type is Interval Year To Month
ERROR 5380: User Defined Scalar Function *string* provided range for Year To Month, but the function's return type is Interval Day To Second
ERROR 5684: User Defined Extension cannot be created in a system schema
ERROR 6051: The schema has been dropped
ERROR 6576: Schema does not exist: *string*
ERROR 6580: Stemmer UDx *string* must have VARCHAR or LONG VARCHAR parameter type
ERROR 6581: Stemmer UDx *string* must have VARCHAR or LONG VARCHAR return type
ERROR 6642: Argument types must be specified for stemmer "*string*"
ERROR 6869: Multi-phase transform function must have atleast one phase
ERROR 6961: Stemmer UDx *string* must return a single argument
ERROR 7010: Tokenizer required to create text index
ERROR 7015: UDSF or SQL Function with specified name and parameters does not exist: *string*
ERROR 7082: "*string*" is an incorrect EarlyMaterialize directive
ERROR 7128: Tokenizer "*string*" with specified argument types could not be found
ERROR 7129: Tokenizer UDx *string* must be polymorphic or have a single input field of CHAR, VARCHAR, LONG VARCHAR, VARBINARY, LONG VARBINARY, or USER DEFINED argument type
ERROR 7139: View cannot be moved into a system schema
ERROR 7582: User Defined Transforms with cursors do not support fenced execution mode
ERROR 9950: Can't have more than one parameter with the same name: *string*
ERROR 9962: Parameters must be named
ERROR 10438: DO is only compatible with the language 'plvsql'
ERROR 10496: ALTER PROCEDURE does not support external procedures
ERROR 10524: Procedure *string* with specified parameters does not exist
ERROR 10525: Procedure *string* with the specified arguments does not exist
ERROR 10526: Procedure cannot be moved into a system schema

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE VP001

This topic lists the messages associated with the SQLSTATE VP001.

SQLSTATE VP001 description

USER_PROC_EXEC_ERROR

Messages associated with this SQLState

Error messages

ERROR 2837: Could not create pipe for user procedure execution, errno=*value*
ERROR 2853: Could not execute user procedure: fork error
ERROR 2858: Could not find function definition
ERROR 2861: Could not find running procedure for *string*, proc ID=[*value*]
ERROR 3007: DDL statement interfered with this statement
ERROR 3399: Failure in UDx RPC call *string*(): *string*
ERROR 4424: Procedure execution error: exit status=*value*
ERROR 4425: Procedure execution error: procedure killed by signal (*value*)
ERROR 4538: Received message with unexpected type *string*
ERROR 5205: Unknown error killing procedure *string*
ERROR 5395: User procedure execution failed
ERROR 5398: User-defined Analytic Function *string* produced fewer output rows than input rows
ERROR 5399: User-defined Scalar Function *string* outputted a timezone (*value*) not in allowed range (*value*, *value*)
ERROR 5400: User-defined Scalar Function *string* produced fewer output rows (*value*) than input rows (*value*)
ERROR 5430: Vertica process is not allowed to kill procedure *string*
ERROR 5580: Failure sending parameters block because the *value* parameters require *value* bytes, which exceeds the maximum size of *value* bytes
ERROR 5604: Invalid procedure file: [*string*]
ERROR 5638: Procedure file [*string*] cannot be owned by root
ERROR 5639: Procedure file [*string*] must be executable by vertica user (dbAdmin)
ERROR 5640: Procedure file [*string*] must be owned by specified procedure user
ERROR 5641: Procedure file [*string*] must enable set UID attribute
ERROR 5656: Root cannot execute external procedure
ERROR 5683: User '*string*' not found on node
ERROR 5861: Error calling *string*() in User Function *string* at [*string*:*value*], error code: *value*, message: *string*
ERROR 5863: Error during setting up function, message: *string*
ERROR 6085: Unexpected exception calling *string*() User Function in *string*
ERROR 6087: Unexpected exception thrown by UDFileSystem at [*string*:*value*], error code: *value*, message: *string*
ERROR 6668: Can't access [*string*]: No filesystem is mapped to scheme *string*
ERROR 6713: Couldn't cancel the user procedure, *string*
ERROR 6783: Filesystem does not support glob *string*
ERROR 6859: makeConnection: send_msg(*value*) did not succeed: *string*
ERROR 7097: Failure in UDx RPC call *string*()
ERROR 7112: Procedure reported: *string*
ERROR 7685: User-defined Scalar Function *string* produced more output rows (*value*) than input rows (*value*)
ERROR 7981: Cannot reserve memory from the JVM resource pool
ERROR 8092: Failure in UDx RPC call *string*() in User Defined Object [*string*]: *string*
ERROR 8468: Error calling *string*() in User Function *string*, message: *string*
ERROR 9754: Could not find filesystem for path *string*
ERROR 9939: Received hi bound with size *value* when data type has max size *value*
ERROR 9940: Received lo bound with size *value* when data type has max size *value*
ERROR 9964: Type conversion function *value* cannot be done within collection coercion
ERROR 10297: User procedure exceeded recursion limit of *value*

Warning messages

WARNING 2005: *string*
WARNING 2376: Cannot connect to UDx side process (pid = *value*) during cancel: *string*
WARNING 3223: Error in calling *string*() for User Defined Function *string* at [*string*:*value*], error code: *value*, message: *string*
WARNING 3224: Error in calling *string*() for User Defined Scalar Function *string* at [*string*:*value*], error code: *value*, message: *string*
WARNING 3398: Failure in UDx RPC call *string*() (pid = *value*): *string*
WARNING 3399: Failure in UDx RPC call *string*(): *string*
WARNING 5170: Unexpected exception from in calling *string*() for User Defined Scalar Function *string*
WARNING 5171: Unexpected exception in calling *string*() in User Defined Function *string*
WARNING 6086: Unexpected exception calling destroyUDxFenced()
WARNING 6756: Error in calling destructor for UDFilter function at [*string*:*value*], error code: *value*, message: *string*
WARNING 6757: Error in calling destructor for UDParser function at [*string*:*value*], error code: *value*, message: *string*
WARNING 6758: Error in calling destructor for UDSource function at [*string*:*value*], error code: *value*, message: *string*
WARNING 6759: Error in calling ~*string*() for User Defined Function *string* at [*string*:*value*], error code: *value*, message: *string*
WARNING 6760: Error in calling ~*string*() for User Defined Scalar Function *string* at [*string*:*value*], error code: *value*, message: *string*
WARNING 6859: makeConnection: send_msg(*value*) did not succeed: *string*
WARNING 7025: Unexpected exception in calling destructor in UDFilter function
WARNING 7026: Unexpected exception in calling destructor in UDParser function
WARNING 7027: Unexpected exception in calling destructor in UDSource function
WARNING 7028: Unexpected exception in calling ~*string*() for User Defined Scalar Function *string*
WARNING 7029: Unexpected exception in calling ~*string*() in User Defined Function *string*
WARNING 7712: Error during cleanup for User Defined Function *string*: *string*
WARNING 8607: Error in calling destructor for UDChunker function at [*string*:*value*], error code: *value*, message: *string*
WARNING 8624: Unexpected exception in calling destructor in UDChunker function

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE VX001

This topic lists the messages associated with the SQLSTATE VX001.

SQLSTATE VX001 description

INTERNAL_ERROR

Messages associated with this SQLState

Error messages

ERROR 2025: *string* Error occurred during BZIP initialization. BZIP error code: *value*
ERROR 2027: *string* Error occurred during ZLIB initialization. ZLIB error code: *value*, Message: *string*
ERROR 2111: Active queue cleared while running
ERROR 2405: Cannot do boundary analysis on type *value*

ERROR 2405: Cannot do boundary analysis on type *value*
ERROR 2616: Cluster recovery failed, try again
ERROR 2928: Couldn't check this session's state
ERROR 3099: Did not find a variable
ERROR 3211: Error during recovery running *string* queries, cannot continue:
string (at *string*:*value*)
ERROR 3212: Error during recovery running *string*: *string* (at *string*:*value*)
ERROR 3220: Error generating query for: *string*
ERROR 3292: Event apply failed
ERROR 3483: Got unexpected error code from spread: *value*, *string*
ERROR 3818: JOIN qualifications to not refer to the correct
relation(s)
ERROR 3969: More than one variable found
ERROR 4342: Password encryption failed
ERROR 4372: pg_analyze_and_rewrite for View query failed
ERROR 4514: Raw parse of View query string failed
ERROR 4545: Recovery Error: Cannot get projections on local node
ERROR 5236: Unrecognized node type *value*
ERROR 5237: Unrecognized node type *value* in postprocess conditions
ERROR 5238: Unrecognized node type: *value*
ERROR 5540: Cannot find buddy projections for collecting row counts,
min and max
ERROR 5679: Unrecognized order by expression
ERROR 5680: Unrecognized select column list
ERROR 5695: With query is not a Select Statement
ERROR 5719: Path Sampling failed. Try a different random seed for the
pathSampling hint
ERROR 5802: Could not stop all dirty transactions[txnId = *string*]
ERROR 5865: Error while analyzing approximate count distincts on table
string.*string*
ERROR 6062: Too Many User defined types
ERROR 6248: Error occured during LZO decompression: LZO checksum error
on compressed data, possibly due to file corruption
ERROR 6249: Error occured during LZO decompression: LZO checksum error
on uncompressed data, possibly due to file corruption
ERROR 6250: Error occured during LZO decompression: LZO expected
destination length larger than BLOCK_SIZE, possibly need
to recompile lzop, or set --blocksize to a larger value
ERROR 6251: Error occured during LZO decompression: LZO expected
destination length larger than MAX_BLOCK_SIZE, possibly
due to file corruption
ERROR 6252: Error occured during LZO decompression: LZO expected
source length is wrong, possibly due to file corruption
ERROR 6253: Error occured during LZO header processing: expecting more
than *value* bytes, possibly file corrupted
ERROR 6254: Error occured during LZO header processing: return code
value, possibly due to file corruption
ERROR 6429: The sending password for "*string*", encryption algorithm *string*
does not match the effective server configured encryption
algorithm *string*
ERROR 6440: Trying to change password for "*string*", but password
encryption algorithm does not match, server configured *string*,
client send in *string*
ERROR 6468: Wrong Password Security Algorithm *string*
ERROR 6482: Failed to parse Access Policies for table "*string*" [*string*]
ERROR 6678: Cannot extract relations in the query
ERROR 6738: DML is running while collecting dirty txns
ERROR 6768: Error retrieving Group ROS [*value*] of ROS [*value*]
ERROR 6771: Fail to get table recovery status when node is not
INITIALIZING/RECOVERING/READY/UP
ERROR 6775: Failed to generate an annotated query: *string*

ERROR 6779: Failed to recover all tables, would retry!
ERROR 6780: Failed to recover node, shutting down...
ERROR 6820: Input query cannot be deparsed
ERROR 6821: Input query is not supported
ERROR 6893: Output annotated query is not supported
ERROR 6895: Output stream failed to initialize
ERROR 6927: Query not ready to write to export file
ERROR 6930: Recovery Error: Cannot get projections of table *string*
ERROR 6942: ROS [*value*] is in a bundle without a storageId
ERROR 6995: The content of the input query saved previously changed
ERROR 7032: Unexpected segmentation for constraint projection
ERROR 7274: Recovery Error: projection recovery start epoch is behind AHM in *string* phase. Has AHM been advanced during recovery?
ERROR 7493: OpenSSL RAND_bytes returns error; *value*
ERROR 7645: Configured password type *string* not admissable under FIPS
ERROR 7656: Effective password type *string* not admissable under FIPS
ERROR 7661: MD5 not permitted in FIPS mode
ERROR 7669: Trying to change password for "*string*", but MD5 hash algorithm not permitted
ERROR 7855: Found SAL corruption
ERROR 7873: Property *string* is in *string*, not *string*
ERROR 7874: Property *string* not found in *string*
ERROR 8387: Cluster membership change interfered with an operation to initialize tables for recovering node *string*
ERROR 8404: Cannot initialize necessary map for recovery
ERROR 8485: Input query is not supported for EXPLAIN ANNOTATED
ERROR 8492: Uploader caught unknown exception: *string*
ERROR 8758: StorageContainer (ROS) *value* starts at epoch *value*, end at epoch *value*, straddle truncate epoch *value*
ERROR 8813: *string* Error occurred during ZSTD decompression. ZSTD error code: *value*: *string*
ERROR 8814: *string* Error occurred during ZSTD initialization (*value*): *string*
ERROR 8980: Revoke statements do not support use of 'EXTEND' keyword
ERROR 9040: Cannot add new sort order to an existing projection
ERROR 9510: Invalid use of TOPK projections. Try disabling DoQueryRewriteForLAPs
ERROR 9542: Error checking support for Webhdfs TRUNCATE operation: *string*
ERROR 9682: Spread Encryption failed (*value* bytes)
ERROR 9929: TS: transfer error: *string*
ERROR 9977: Unexpected dest receiver type *value*
ERROR 10000: Partition ranged projection not qualified, retry with normal projection
ERROR 10118: Corrupted diag option *value*
ERROR 10119: Corrupted diag type *value*
ERROR 10121: Corrupted fetch direction type *value*
ERROR 10122: Corrupted fetch type *value*
ERROR 10123: Corrupted raise level *value*
ERROR 10124: Corrupted raise option *value*
ERROR 10125: Corrupted type kind *value*
ERROR 10167: Corrupted assign type *value*
ERROR 10168: Corrupted cond type *value*
ERROR 10169: Corrupted control flow type *value*
ERROR 10170: Corrupted empty behavior *value*
ERROR 10171: Corrupted expression or statement behavior *value*
ERROR 10172: Corrupted raise level type *value*
ERROR 10173: Corrupted raise option type *value*
ERROR 10577: Raw parse of Select * From query failed
ERROR 10626: JOIN qualifications do not refer to the correct relation(s)

Fatal messages

FATAL 2866: Could not get current working directory: *value*
FATAL 3419: findMySession: no session for thread id 0x*value*
FATAL 3501: Holding Catalog locks when returning control to client
FATAL 3502: Holding Catalog snapshot when returning control to client
FATAL 4936: The cluster performed ASR without this node, but there is
a local drop partition event without a global match

Warning messages

WARNING 3245: Error parsing *string*
WARNING 5526: Already have a ready_recv *string*, ignoring
WARNING 6428: The password for "*string*", encryption algorithm *string* does not
match the effective server configured encryption
algorithm *string*, please expire the password to reset
WARNING 7668: The server password *string* is not allowed on FIPS systems;
please have DB admin correct this

Note
The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE VX002

This topic lists the messages associated with the SQLSTATE VX002.

SQLSTATE VX002 description

DATA_CORRUPTED

Messages associated with this SQLState

ERROR 2940: CRC Check Failure Details:

File Name: *string*

File Offset: *value*

Compressed size in file: *value*

Memory Address of Read Buffer: *value*

Pointer to Compressed Data: *value*

Memory Contents:

string

ERROR 3030: Delete: could not find a data row to delete (data integrity violation?)

ERROR 3218: Error finalizing ROS DataTarget

ERROR 3219: Error flushing data file [*string*]

ERROR 3409: FileColumnReader: block *string* @ *value* 's CRC *value* doesn't match record *value*

ERROR 3410: FileColumnReader: Decompression error in *string* at offset *value*

ERROR 4762: Sort Order Violation:

Row Position: *value*

Column Index: *value*

Last Row: *string*

This Row: *string*

ERROR 5704: Delete (Join): could not find a data row to delete (data integrity violation?)

ERROR 6767: Error reading from orc parser input stream [*string*]: file shorter than expected, read to *value*, requested to *value*

ERROR 7767: Error reading from parquet parser input stream [*string*]: file shorter than expected, read to *value*, requested to *value*

ERROR 7859: Block[*value*] has unknown type. (size: *value*, type: *value*, pad: *value*, count: *value*, position: *value*)

ERROR 7860: Block[*value*]->count is not consistent with other blocks. (*value* != *value*)

ERROR 10589: Error reading from avro parser input stream [*string*]: file shorter than expected, read to *value*, requested to *value*

Note

The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE VX003

This topic lists the messages associated with the SQLSTATE VX003.

SQLSTATE VX003 description

INDEX_CORRUPTED

Messages associated with this SQLState

ERROR 3544: Index corruption. *string*: *string*

Note

The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Messages associated with SQLSTATE VX004

This topic lists the messages associated with the SQLSTATE VX004.

SQLSTATE VX004 description

PLAN_TO_SQL_INTERNAL_EROR

Messages associated with this SQLState

ERROR 8643: Optimizer-generated annotated query has unexpected error. Please report to Vertica

Note

The Vertica User Community contains knowledge base articles, blogs, and forum posts that may help you resolve these errors.

Glossary

The Vertica Glossary defines terms that are common and specific to Vertica.

In this section

- Access rank
- Administration host
- Administration tools
- Agent
- Anchor table
- Ancient history mark (AHM)
- Apportioned load
- Authentication service (AS)
- Bitstring
- Buddy projection
- Bulk loading
- C-store
- Cardinality
- Catalog
- Catalog path
- Cell
- Cluster
- Columnar tables
- Communal storage
- Control node
- Cooperative parse
- Correlated columns
- Critical node
- Current epoch
- Data collector
- Data path
- Database
- Database Designer
- Database superuser
- Default subcluster
- DELTAVAL (delta encoding)
- Depot
- Depot warming
- Derived column
- Dimension table
- Directed query

- Encoding
- Enterprise Mode
- Eon Mode
- Epoch
- Epoch map
- Executor node
- Expression
- External procedure
- Event series
- Flexible tables
- Flattening (subqueries and views)
- Full backup
- GENERAL pool
- Grant
- Grouped ROS
- Hash segmentation
- Historical data
- Historical query
- Immutable (invariant) functions
- Initiator node
- K-safety
- Last epoch
- Last good epoch (LGE)
- Live aggregate projection
- Locale
- Logical schema
- Location label
- Management Console
- MC super (superuser administrator)
- MC-managed database
- Mergeout
- Meta-functions
- Metadata
- Mutual mode
- Node
- Non-repeatable read
- NUL
- NULL
- OID
- Parallel load
- Path (quality plan)
- Partitioning
- Phantom read
- Physical schema
- Physical schema design
- Primary column
- Primary node
- Primary subcluster
- Primary subscriber
- Projection
- Projection set
- Query cluster level
- Query optimizer
- Quorum
- Recovery
- Referential integrity
- Refresh (projections)
- Resegmentation
- Resource pool
- Resource manager
- Role

- Rollback
- ROS (Read optimized store)
- Savepoint
- Secondary subcluster
- Segmentation
- Server mode
- Session
- Shard
- Shard coverage
- Snapshot isolation
- Spread
- Sort-merge join
- Stable functions
- Statement
- Storage location
- Storage policy
- Strict
- Subcluster
- Superprojection
- Temp location
- Temp space
- Top-k projection
- Transaction
- Tuple mover (TM)
- UDx
- Up-to-date (projection)
- User-defined SQL function
- View
- Volatile functions
- vsql
- Window (analytic)
- Working data size
- Workload analyzer

Access rank

Determines the speed at which a column can be accessed. Columns are stored on disk from the highest ranking to the lowest ranking in which the highest ranking columns are placed on the fastest disks and the lowest ranking columns are placed on the slowest disks.

Administration host

The host on which the Vertica rpm package was manually installed. Always run the Administration Tools on this host if possible.

Administration tools

Vertica Administration Tools provides a graphical user interface for managing a Vertica database. With this tool, you can perform various tasks, including:

- View the state of the database cluster.
- Create a database.
- Start a database.
- Stop a database.
- Run Database Designer.
- Connect to a database using vsql.

Agent

A daemon process that runs on each Vertica cluster node. The agent is used by certain clients, such as Management Console, to administer Vertica.

Agents monitor Vertica database clusters and communicate with their clients to provide the following functionality:

- Provide local access, command, and control over database instances on a given node, using functionality similar to Administration tools
- Report log-level data from the Administration Tools and Vertica log files
- Cache details from long-running jobs—such as create/start/stop database operations—that you can view through your browser
- Track changes to data-collection and monitoring utilities and communicate updates to clients
- Specifically for MC, communicate between all cluster nodes and MC through a webhook subscription, which automates information sharing and reports on cluster-specific issues like node state, alerts, events, and so on

The agent runs on port 5444, which must be accessible to agent clients.

Anchor table

Database table that is the source for data in a projection . The anchor table and its projection must be in the same schema. The privileges to create, access, or alter a projection are based on the anchor tables that the projection references, as well as the schemas that contain them.

In the following simple statement, public data is the anchor table:

```
=> CREATE PROJECTION publicdataproj AS (SELECT * FROM publicdata);
```

Ancient history mark (AHM)

Also known as AHM, the ancient history mark is the oldest epoch whose data is accessible to historical queries . Any data that precedes the AHM is eligible to be purged .

For detailed information, see Understanding Vertica Epochs in the Vertica Knowledge Base .

Apportioned load

An *apportioned load* is a divisible load, such that you can load a single data file on more than one node. Apportioned load divides the load at planning time, based on available nodes and cores on each node. Each load starts at a different offset, requiring a parser that supports apportioning. Some of the parsers built into Vertica support apportioned load. Using the SDK, you can write parsers that perform apportioned loads.

Authentication service (AS)

A service that usually runs on the same host as the Kerberos Key Distribution Center (KDC). The AS issues tickets for a requested service. The tickets are in turn given to users for access to the service. The AS answers requests from clients that do not send credentials with a request. The AS is generally used to gain access to the ticket-granting service (TGS) by issuing a ticket-granting ticket (TGT).

Bitstring

A sequence of bits.

Buddy projection

Required for K-safety. Two projections are considered to be buddies if they contain the same columns and have the same hash segmentation, using different node ordering.

For more information, see:

- High availability with projections
- Designing segmented projections for K-safety
- GET_PROJECTION_STATUS

Bulk loading

A process of loading large amounts of data, such as an initial load of historic data.

C-store

A research project at MIT, Brandeis, Brown, and the University of Massachusetts (Boston), on which Vertica is based.

Cardinality

Refers to the number of unique values for a given column in a relational table:

- *High cardinality* : Refers to columns containing values that are highly unique, such as a customer ID or an employee e-mail address. For example, in the Vertica VMart schema , the employee_dimension table contains an employee_key column. This column contains values that uniquely identify each employee. Since the values in this column are unique and could be numerous, the column's cardinality type is referred to as high cardinality.
- *Normal cardinality* : Refers to columns containing values that are less unique, such as job titles and street addresses. An example of a normal-cardinality column would be job_title or employee_first_name in the employee_dimension table, where many employees could share the same job title or same first name.
- *Low cardinality:* Refers to a low number of unique values, relative to the overall number of records in a table. For example, in the employee_dimension table, the column called employee_gender would contain two unique values: 'Male' or 'Female'. Since there are only two values possible in this column, cardinality is low.

Catalog

A set of files that contains information (metadata) about the objects—such as nodes, tables, constraints, and projections—in a database. Vertica maintains a catalog on each node in the cluster.

Catalog path

A storage location used to store the database catalog.

Cell

A cell is space in a database object allocated for a single piece of data. For a list of supported data types that can populate a cell, see Data types .

Cluster

The concept of Cluster in the Vertica Analytics Platform is a collection of hosts with the Vertica software packages (RPM or DEB) that are in one admin tools domain. You can access and manage a cluster from one admintools initiator host.

Columnar tables

Vertica database tables consisting of structured data columns. The term differentiates these tables from Flex (or Flexible) tables, which minimally contain one column of unstructured, or semi-structured data. Flex tables can also have structured data columns, but they are not required. Compare with Flexible tables .

Communal storage

The shared storage location containing an Eon Mode database's data. The data within the communal storage is the canonical copy of the data—Vertica does not consider data as being committed until it has been written to communal storage.

The communal data storage location is based on an object store, such as an S3 bucket when Vertica runs in the AWS cloud, or a PureStorage FlashBlade.

Control node

A node that connects to the Spread service to send and receive cluster-wide broadcast messages. In databases where the large cluster feature is disabled, all nodes are control nodes. In databases where the large cluster feature is enabled, a subset of nodes are control nodes. Vertica assigns non-control nodes to a control node. These dependent nodes rely on their control node to relay broadcast messages to them. See Large Cluster for more information.

Cooperative parse

A *cooperative parse* occurs when a single node uses multiple threads to parse data for loading. Cooperative parse divides a load at execution time, based on how threads are scheduled, if the parser supports cooperation. Some of the parsers built into Vertica support cooperative parse. Using the SDK, you can write parsers that perform cooperative parse.

Correlated columns

Two columns are correlated if the value of one column is related to the value of the other column. For example, state name and country name columns are strongly correlated because the city name usually, but perhaps not always, identifies the state name. The city of Conshohoken is uniquely associated with Pennsylvania, whereas the city of Boston exists in Georgia, Indiana, Kentucky, New York, Virginia, and Massachusetts. In this case, city name is strongly correlated with state name.

Critical node

A critical node is a node whose failure would cause the database to become unsafe and force a shutdown. Nodes can become critical for the following reasons:

- A node has the only copy of a particular projection.
- Fewer than half of your nodes are active. In an Eon Mode database, at least half of your primary nodes must be up to maintain data integrity.

The V_MONITOR.CRITICAL_NODES system table lists the critical nodes, if any, in your cluster.

For more information, see K-safety in an Enterprise Mode database for Enterprise Mode databases, and Data integrity and high availability in an Eon Mode database for Eon Mode databases.

Current epoch

The epoch into which data (COPY, INSERT, UPDATE, and DELETE operations) is currently being written.

Data collector

A utility that collects and retains database monitoring information. For details, see Data collector utility .

Data path

A storage location that contains actual database data files.

Database

A cluster of nodes that, when active, can perform distributed data storage and SQL statement execution through administrative, interactive, and programmatic user interfaces.

Database Designer

A tool that analyzes a logical schema definition, sample queries, and sample data, and creates a physical schema (projections) in the form of a SQL script that you deploy automatically or manually. The script creates a minimal set of superprojections to ensure K-safety, and, optionally, non-superprojections. In most cases, the projections created by the Database Designer provide excellent query performance within physical constraints.

The Database Designer can create two distinct design types. The design you choose depends on what you are trying to accomplish:

- Comprehensive
- Incremental

You can also create custom designs if the Database Designer does not meet your needs.

For detailed information, see Creating a database design .

Database superuser

The automatically-created database user who has the same name as the Linux database administrator account and who can bypass all GRANT/REVOKE authorization, or any user that has been granted the PSEUDOSUPERUSER role. Do not confuse the concept of a database superuser with Linux superuser (root) privilege. A database superuser cannot have Linux superuser privilege.

Default subcluster

The default subcluster is the subcluster Vertica adds new nodes to if you do not specify a subcluster to contain the new nodes. Your database can only have a single default subcluster. You can make a subcluster the default using the ALTER SUBCLUSTER statement.

DELTAVAL (delta encoding)

Stores only the differences between sequential data values instead of the values themselves.

Depot

A cache of data maintained by the nodes in an Eon Mode database to limit reads from communal storage . Retrieving data from communal storage often has high latency and potentially limited bandwidth, especially in cloud environments. Each node in the database caches the data it reads from and writes to the communal storage. When processing a query, they first check this local cache for the data they need. If the data is cached locally, the nodes use the cached version instead of retrieving the data from communal storage.

Depot warming

On startup, the depots of new nodes are empty, while the depots of restarted nodes often contain stale data that must be refreshed. When depot warming is enabled, a node that is undergoing startup preemptively loads its depot with frequently queried and pinned data. When the node completes startup and begins to execute queries, its depot already contains much of the data it needs to process those queries. This reduces the need to fetch data from communal storage, and expedites query performance accordingly.

> **Note**
> Fetching data to a warming depot can delay node startup.

For details, see Managing depot caching .

Derived column

A column whose values are calculated by an expression at load time. The expression is specified within the COPY statement, and the column exists in the target database.

Dimension table

Sometimes called a lookup or reference table, a dimension table is one of a set of companion tables to a large (fact/anchor) table in a star schema. It contains the PRIMARY KEY column corresponding to the join columns in fact tables. For example, a business might use a dimension table to contain item codes and descriptions.

Dimension tables can be connected to other dimension tables to form a hierarchy of dimensions in a snowflake schema.

Directed query

A saved set of instructions that direct the optimizer to generate a query plan for a given query. The query plan consists of SQL annotated with hints. A directed query pairs two components:

- *Input query* : A query that triggers use of this directed query when it is active.
- *Annotated query* : A SQL statement with embedded optimizer hints . The annotated query is used by the optimizer in creating a query plan for the specified input query.

Encoding

The process of converting data into a standard format. In Vertica, encoded data can be processed directly, while compressed data cannot. Vertica uses a number of different encoding strategies, depending on column data type, table cardinality, and sort order.

Enterprise Mode

A database mode that optimizes your database for analytic speed. This is the "original" mode of the Vertica database—the only mode it supported before Eon Mode was introduced. In this mode, data is stored by the database nodes. This proximity allows nodes to operate on locally stored data for most queries, reducing query times. An Enterprise Mode database is harder to scale up and down than an Eon Mode database, which is optimized for scalability.

Eon Mode

Eon Mode is the database mode that optimizes your database for scalability. This mode separates data storage from computing resources. By separating the two, you can add or remove nodes from your cluster to adjust growing or shrinking analytic workloads. Scaling the cluster size does not affect running queries. Eon Mode is especially well-suited for cloud environments. See Eon Mode concepts for more information.

Epoch

A logical unit of time in which a single change is made to data in your Vertica database.

For detailed information, see Epochs .

Epoch map

A catalog object that provides mapping between time and epochs. Specifically, an epoch map contains a list of epoch numbers and their associated timestamps.

Executor node

Any node that participates in executing a specific SQL statement. The initiator node can, and usually does, also function as an executor node.

Expression

An expression is anything that can represent a single column in a SELECT statement.

For example, the following are all expressions:

```
1+2
foo(3)
x
```

External procedure

A procedure external to Vertica that you create, maintain, and store on the server.

Event series

Tables with a time column, most typically a timestamp data type.

Flexible tables

Vertica database tables that minimally contain two columns:

__identity__ : A real column with an incrementing IDENTITY value for partitioning and sorting. Used if no other columns serve this purpose.

__raw__ : A real LONG VARBINARY column containing unstructured, or semi-structured data.

You can create Flex tables with additional real columns, but they are not required. Compare with Columnar tables .

Flattening (subqueries and views)

Occurs when a subquery or named view is internally rewritten so the subquery is combined with the outer query block. The result sets of the original and flattened queries are exactly the same, but the flattened query usually benefits from significant performance improvements.

Full backup

Consists of copying each catalog and all data files (ROS containers) on each node, as well as the complete /opt/vertica/config directory.

GENERAL pool

A special built-in pool that represents the total amount of RAM available to the resource manager for use by queries. Other pools can borrow memory from the GENERAL pool. See also Built-in pools .

Grant

Vertica defines GRANT in two ways:

1. Grant a user privileges to access database objects using any GRANT statement , except GRANT (Authentication).
2. Associate a user-defined authentication method with a user through GRANT (Authentication) . This operation differs from GRANT <privileges> as authentication methods are “associated” with a user or role and privileges are “granted” to a user or role.

Examples

Grant Access Privileges to a User

This example shows how to grant a user, Joe, privileges to access the online_sales schema:

```
=> GRANT USAGE ON SCHEMA online_sales TO Joe;
```

Associate an Authentication Method with a User

This example shows how to associate the v_ldap authentication method to user jsmith:

```
=> GRANT AUTHENTICATION v_ldap TO jsmith;
```

Grouped ROS

A grouped ROS is a highly-optimized, read-oriented physical storage structure organized by projection. A grouped ROS makes heavy use of compression and indexing. Unlike a ROS, a grouped ROS stores data for two or more grouped columns in one disk file.

Hash segmentation

Specifies how to segment projection data for distribution across all cluster nodes. You can specify segmentation for a table and a projection. If a table definition specifies segmentation, Vertica uses it for that table's auto-projections .

It is strongly recommended that you use Vertica's built-in HASH function, which distributes data evenly across the cluster, and facilitates optimal query execution.

For more detailed information, see Segmented projections .

Historical data

Refers to any data in memory or physical storage other than the current epoch. Historical data includes all COPY, INSERT, UPDATE, and DELETE operations, including deleted rows. This allows Vertica to run a historical query on data written up to and including the epoch representing the specified date and time.

Historical query

A query that retrieves data from a snapshot of the database, taken at a specific timestamp or epoch. For details, see Historical queries .

Immutable (invariant) functions

When run with a given set of arguments, immutable functions always produce the same result, regardless of environment or session settings such as locale.

Initiator node

In the context of a client connection, the initiator node is the node associated with the specific host to which the connection was made. The initiator node can, and usually does, also function as an executor node, and is generally where the most descriptive log messages reside.

K-safety

K-safety sets fault tolerance for the database cluster, where K can be set to 0, 1, or 2. The value of K specifies how many copies Vertica creates of Segmented projections data. If K-safety for a database is set to 1 or 2, Vertica creates K+1 instances, or *buddies* , of each projection segment. Vertica distributes these buddies across the database cluster, such that projection data is protected in the event of node failure. If any node fails, the database can continue to process queries so long as buddies of data on the failed node remain available elsewhere on the cluster.

For more information, see Designing for K-safety .

Last epoch

The last epoch is the current epoch minus one. SELECT statements under READ COMMITTED isolation read from the last epoch.

Last good epoch (LGE)

A term used in manual recovery, LGE (Last Good Epoch) refers to the most recent epoch that can be recovered.

Live aggregate projection

A live aggregate projection contains columns with values that are aggregated from columns in its anchor table. When you load data into the table, Vertica aggregates the data before loading it into the live aggregate projection. On subsequent loads—for example, through INSERT or COPY —Vertica recalculates aggregations with the new data and updates the projection.

Locale

Locale specifies the user's language, country, and any special variant preferences, such as collation. Vertica uses locale to determine the behavior of certain string functions. Locale also determines the collation for various SQL commands that require ordering and comparison, such as aggregate GROUP BY and ORDER BY clauses, joins, and the analytic ORDER BY clause.

The default locale for a Vertica database is en_US@collation=binary (English US). You can define a new default locale that is used for all sessions on the database. You can also override the locale for individual sessions. However, projections are always collated using the default en_US@collation=binary collation, regardless of the session collation. Any locale-specific collation is applied at query time.

If you set the locale to null, Vertica sets the locale to en_US_POSIX . You can set the locale back to the default locale and collation by issuing the vsql meta-command \locale . For example:

Note

```
=> set locale to '';
INFO 2567:  Canonical locale: 'en_US_POSIX'
Standard collation: 'LEN'
English (United States, Computer)
SET
=> \locale en_US@collation=binary;
INFO 2567:  Canonical locale: 'en_US'
Standard collation: 'LEN_KBINARY'
English (United States)
=>  \locale
en_US@collation-binary;
```

You can set locale through ODBC , JDBC , and ADO.net .

Logical schema

Consists of a set of tables and referential integrity constraints in a Vertica database. The objects in the logical schema are visible to SQL users. The logical schema does not include projections, which make up the physical schema.

Location label

A label assigned to a storage location. Location labels identify the location so you can create object storage policies. The labeled location you use in a storage policy becomes the default storage location for the object.

Management Console

A database management tool that provides a unified view of your Vertica database and lets you monitor multiple clusters from a single point of access. See Management Console .

MC super (superuser administrator)

Called Super on the MC interface, the MC super is the Linux user account that gets created when you configure MC . See Configuring Management Console and Configuration privileges .

MC-managed database

A Vertica database that an MC SUPER or ADMIN user creates on MC or imports into the MC interface, along with the database cluster. When MC users are granted database privileges, their privileges are defined though the MC itself and pertain only to databases managed on the MC. See Users, roles, and privileges .

Mergeout

Mergeout is a Tuple Mover process that consolidates ROS containers and purges deleted records.

Meta-functions

Used to query or change the internal state of Vertica and are not part of the SQL standard.

Meta-functions can be used in a top-level SELECT statement only, and the statement cannot contain other clauses such as FROM or WHERE.

Metadata

Data that describes the data in a database, such as data type, compression, constraints, and so forth, for each column in the tables. Metadata is stored in the Vertica catalog.

Mutual mode

When a database is configured for TLS/SSL security in *mutual mode* , incoming client requests verify the certificate of the server, and the server also requires that each client present a certificate and private key so that the server can verify the client.

Node

A host configured to run an instance of Vertica. It is a member of the database cluster. For a database to have the ability to recover from the failure of a node requires a database K-safety value of at least 1 (3+ nodes).

Non-repeatable read

Occurs in a READ COMMITTED isolation level when two identical queries in the same transaction produce different results. This occurs when another transaction commits changes that alter the results of the query after the first query has completed and before the second query has begun.

NUL

Represents a character whose ASCII/Unicode code is zero, sometimes qualified "ASCII NUL".

NULL

Means *no value* , and is true of a field (column) or constant but not of a character.

OID

An object identifier (OID) is an identifier used to name an object. Structurally, an OID consists of a node in a hierarchically-assigned namespace, formally defined using the International Telecommunication Union-Telecommunication's (ITU-T) Abstract Syntax Notation standard (ASN.1). Vertica database objects are always assigned an OID. The OID can be used directly in some statements, such as HAS_TABLE_PRIVILEGE .

Source: Wikipedia

Parallel load

Parallel load is the process of loading data on any available node in the cluster (not necessarily the local node). Use this approach in combination with wildcards (such as *.dat) to load data files in parallel in a distributed manner. See COPY ON ANY NODE in Parameters .

Path (quality plan)

The execution strategy of the Vertica cost-based query optimizer, denoting a sub operation in the query plan.

Partitioning

Specifies how data is organized within individual nodes. Partitioning attempts to introduce hot spots within the node, providing a convenient way to drop data and reclaim the disk space.

> **Note**
> Partitioning and segmentation are terms often used interchangeably for other databases, but they have completely separate goals in Vertica regarding data localization. See Partitioning and segmentation in the Administrator's Guide for details.

Phantom read

Occurs in a READ COMMITTED isolation level when two identical queries in the same transaction produce different collections of rows. This occurs because a table lock is not acquired on SELECT during the initial query.

Physical schema

Consists of a set of projections used to store data on disk. The projections in the physical schema are based on the objects in the logical schema.

Physical schema design

A usable K=1 design based on an analysis of the sample data files and queries (if available). The physical schema design contains segmented (fact table) superprojections and replicated (dimension table) superprojections.

Primary column

A column that is loaded from raw data, and not derived from an expression.

Primary node

In Eon Mode, a primary node is a node that is a member of a primary subcluster . Primary nodes are the only nodes in the database that Vertica considers when determining whether the database is able to maintain data integrity. Vertica requires that more than half of the primary nodes in the database be up. Also, all shards must have at least one primary node as a subscriber. If either of these conditions are not met, Vertica shuts the database down to prevent potential data corruption.

Primary subcluster

In Eon Mode, a primary subcluster is a type of subcluster that is intended to form the core of your database. Vertica only considers the nodes in primary subclusters when determining the K-safety of your database. Your database can remain running as long as half the nodes in your primary subclusters are up, and all shards have at least one node in a primary subcluster as a subscriber. See Subcluster Types and Elastic Scaling for more information.

Primary subscriber

In an Eon Mode database, each shard has a primary subscriber node. This node is responsible for running Tuple Mover processes that maintain the data in the shard.

Projection

Optimized collections of table columns that provide physical storage for data. A projection can contain some or all of the columns of one or more tables.

For more information about Vertica projections, see Projections .

Projection set

A group of buddy projections that are safe for a given level of K-safety. When K=1, there are two buddies in a set; when K=2, there are three buddies. The Database Designer assigns all projections in a projection set the same base name so they can be identified as a group.

A projection must be part of a projection set before it is refreshed. Once a projection set is created (by creating buddies), the set is refreshed in a single transaction.

Query cluster level

Determines the number of sets used to group similar queries. The query cluster level can be any integer from one (1) to the number of queries to be included in the physical schema design.

Queries are generally grouped based on the columns they access and the way in which they are used. The following work loads typically use different types of queries and are placed in different query clusters: drill downs, large aggregations, and large joins. For example, if a reporting tool and dashboard both access the same database, the reporting tool is likely to use a drill down to access a subset of data and the dashboard is likely to use a large aggregation to look across a large range of data. In this case, there would be at least two (2) query clusters.

Query optimizer

The component that evaluates different strategies for running a query and picks the best one.

Quorum

Your database has a minimum number of nodes that must be up and part of the database cluster for the database to continue operating. When this minimum requirement is met, your database has a quorum of nodes.

The number of nodes that make up a quorum depends on the database mode:

- In Enterprise Mode, over half the nodes (50% of total nodes + 1) must be up.
- In Eon Mode, over half of the primary nodes in the cluster must be up.

The database can lose quorum if too many nodes go down. The database's reaction to losing quorum also depends on the database's mode:

- An Enterprise Mode database that loses quorum shuts down to prevent potential data corruption.
- An Eon Mode database that loses quorum goes into read-only mode to prevent potential data corruption. In this mode, you can usually execute queries, but DDL and DML statements fail with an error.

For more on database availability, see K-safety in an Enterprise Mode database and Data integrity and high availability in an Eon Mode database .

Recovery

Vertica can restore the database to a fully functional state after one or more nodes in the system experiences a software- or hardware-related failure. Vertica recovers nodes by querying replicas of the data stored on other nodes. For example, a hardware failure can cause a node to lose database objects or to miss changes made to the database (INSERTs, UPDATEs, and so on) while offline. When the node comes back online, queries other nodes in the cluster to recover lost objects and catch up with database changes.

Referential integrity

Consists of a set of constraints (logical schema objects) that define primary key and foreign key columns.

- Each small table must have a PRIMARY KEY constraint.
- The large table must contain columns that can be used to join the large table to smaller tables.

- Outer join queries produce the same results as the corresponding inner join query if there is a FOREIGN KEY constraint on the outer table. Note that the inner table of the outer join query must always have a PRIMARY KEY constraint on its join columns.

Refresh (projections)

Ensures that all projections on a node are up-to-date (can participate in query execution). This process could take a long time, depending on how much data is in the table(s).

For more information, see Refreshing projections .

Resegmentation

A process that Vertica performs automatically during query execution that distributes the rows of an existing projection or intermediate relation evenly to each node in the cluster. At the end of resegmentation, every row from the input relation is on exactly one node. Vertica resegments data when the input does not have the segmentation required to compute the requested result efficiently and correctly.

Resource pool

A resource pool comprises a pre-allocated subset of the system resources, with an associated queue. A resource pool is created using the CREATE RESOURCE POOL command as described in the SQL Reference Manual.

Resource manager

In a single-user environment, the system can devote all resources to a single query and get the most efficient execution for that one query. However, in a environment where several concurrent queries are expected to run at once, there is tension between providing each query the maximum amount of resources (thereby getting fastest run time for that query) and serving multiple queries simultaneously with a reasonable run time. The Resource Manager provides options and controls for resolving this tension, while ensuring that every query eventually gets serviced and that true system limits are respected at all times.

Role

A role groups together a set of privileges that can be assigned to a user or another role. You can use roles to quickly grant or revoke privileges on multiple tables, schemas, functions or other database entities to one or more users with a single command.

Vertica's implementation of roles conforms to the SQL Standard T331 for basic roles.

Rollback

Transaction rollbacks restore a database to an earlier state by discarding changes made by that transaction. Statement-level rollbacks discard only the changes initiated by the reverted statements. Transaction-level rollbacks discard all changes made by the transaction.

With a ROLLBACK statement, you can explicitly roll back to a named savepoint within the transaction, or discard the entire transaction. Vertica can also initiate automatic rollbacks in two cases:

- An individual statement returns an ERROR message. In this case, Vertica rolls back the statement.
- DDL errors, systemic failures, dead locks, and resource constraints return a ROLLBACK message. In this case, Vertica rolls back the entire transaction.

Explicit and automatic rollbacks always release any locks that the transaction holds.

ROS (Read optimized store)

Read Optimized Store (ROS) is a highly optimized, read-oriented, disk storage structure, organized by projection. ROS makes heavy use of compression and indexing.

Savepoint

A *savepoint* is a special marker inside a transaction that allows commands that execute after the savepoint to be rolled back. The transaction is restored to the state that preceded the savepoint.

Vertica supports two types of savepoints:

- An *implicit savepoint* is automatically established after each successful command within a transaction. This savepoint is used to roll back the next statement if it returns an error. A transaction maintains one implicit savepoint, which it rolls forward with each successful command. Implicit savepoints are available to Vertica only and cannot be referenced directly.
- *Named savepoints are* labeled markers within a transaction that you set through SAVEPOINT statements. A named savepoint can later be referenced in the same transaction through RELEASE SAVEPOINT , which destroys it, and ROLLBACK TO SAVEPOINT , which rolls back all operations that followed the savepoint. Named savepoints can be especially useful in nested transactions: a nested transaction that begins with a savepoint can be rolled back entirely, if necessary.

Secondary subcluster

A secondary subcluster is a type of subcluster that is easy to start and shutdown on demand. In Eon Mode, Vertica does not consider the nodes in a secondary subcluster when determining whether the database has shard coverage or a quorum of nodes. You can stop or remove secondary subclusters without causing the database to go into read-only mode. See Subcluster Types and Elastic Scaling for more information.

Segmentation

Defines how physical data storage (projections) is stored in a database cluster using the CREATE PROJECTION statement. The goal is to distribute data evenly across multiple nodes in the database so that all nodes can participate in query execution.

> **Note**
> Partitioning and segmentation are terms often used interchangeably for other databases, but they have completely separate goals in Vertica regarding data localization. See Partitioning and segmentation in the Administrator's Guide for details.

Server mode

When a database is configured for TLS/SSL security in *server mode,* incoming client requests do verify the certificate of the server, but the server does not verify the clients. In Vertica, the server is the Vertica database server, and the client is any Vertica user who logs into the database. The Vertica client user may log in to the database directly in a command window, or may connect to the Vertica database server via the Management Console running in a browser.

Session

An occurrence of a user interacting with a database through the use of SQL statements. You can start a session using vsql or a JDBC application. In Vertica, the scope of a session is the same as that of a connection. Connecting to the database starts a session, and exiting ends it.

Session-scoped data is preserved beyond the lifetime of a single transaction. Terminating a session truncates a table and deletes all rows.

Shard

A subset of the data and associated metadata stored in an Eon Mode database. Shards are how Vertica divides the work of processing queries, data loads, and data maintenance between the nodes in a subcluster. Nodes **subscribe** to one or more shards. When processing a query, each node is responsible for processing the data in the shard or shards to which it subscribes.

You set the initial number of shards for the database during database creation. You can change the number of shards with the RESHARD_DATABASE function.

Shards have a single **primary subscriber** in the database cluster. The primary subscriber node maintains the data in the shard by running Tuple Mover processes on it.

Shard coverage

In an Eon Mode database, nodes in a subcluster subscribe to one or more shards in communal storage. Each node handles the data in the shard or shards it subscribes to when processing queries. When a subcluster has at least one node subscribed to each shard, it has shard coverage. It is able to process queries because it has access to all of the data in the database. In a K-safe database, each subcluster has at least two nodes subscribing to each node. If the subcluster loses too many nodes in rapid succession, it may no longer have a node subscribing to each shard. In this case, the subcluster loses shard coverage and cannot process queries.

The database cluster as a whole must have at least one primary node subscribed to each shard in communal storage. This state is called having primary shard coverage. Primary nodes coordinate the maintenance of data in communal storage. If the cluster loses primary nodes to the point that it does not have at least one subscribed to each shard, it loses primary shard coverage. In this case, the database goes into read-only mode because it is unsafe to alter the data in communal storage when one or more shards do not have a primary node subscriber.

See Data integrity and high availability in an Eon Mode database .

Snapshot isolation

An historical query that gets data from the latest epoch (AT LATEST EPOCH). For details, see Historical queries .

Spread

An open source toolkit used in Vertica to provide a high performance messaging service that is resilient to network faults. Spread daemons start automatically when a database starts up for the first time, and the spread process runs on control nodes in the cluster.

Sort-merge join

If both inputs are pre-sorted, merge joins do not have to do any pre-processing. Vertica uses the term sort-merge join to refer to the case when one of the inputs must be sorted prior to the merge join. Vertica sorts the inner input side but only if the outer input side is already sorted on the join keys.

Stable functions

When run with a given set of arguments, stable functions produce the same result within a single query or scan operation. However, a stable function can produce different results when issued under different environments or at different times, such as change of locale and time zone—for example, SYSDATE .

See also Immutable (invariant) functions .

Statement

A line of SQL, excluding the semicolon. For example, the following are all statements:

```
SELECT * FROM foo WHERE bar = 6
CREATE TABLE t1 (i INT);
INSERT INTO t1 VALUES(3)
```

Storage location

A directory path used by Vertica to store catalog, actual data files, and temporary data files. You can also create storage locations for users, and then grant one or more users access to the storage. You can also create a storage location with a location label, for use in storage policies.

Storage policy

A database object you create to associate a labeled location as the default storage location for the object. Database object can be a database, schema, table, or min- and max- ranges of a partition.

Strict

Indicates that a function always returns null when any of its input arguments is null.

Subcluster

A subset of a cluster in an Eon Mode database. Subclusters isolate workloads to a group of nodes, letting you separate tasks such as load and query.

Subclusters come in two types:

- **Primary subclusters** count when Vertica determines whether the database can continue to run safely. Your database shuts down if less than half the primary nodes in your database are up, or does not have a primary node as a subscriber. Primary subclusters are best used for data loading and DDE workloads.

- **Secondary subclusters** do not count towards maintaining data integrity. You can shut them down without impacting the stability of the database. Secondary subclusters are best used for query workloads. They cannot commit transactions directly, so they are less efficient at performing data loads and DDE statements.

Superprojection

A projection that includes all columns in an anchor table. Vertica uses superprojections to ensure support for all queries and other DML operations.

Under certain conditions, Vertica automatically creates a table's superprojection immediately on table creation. Vertica also creates a superprojection when you first load data into that table, if none already exists. CREATE PROJECTION can create a superprojection if it specifies to include all table columns. A table can have multiple superprojections.

For more information, see Creating projections .

Temp location

A storage location used as temp space by Vertica.

Temp space

Disk space temporarily occupied by temporary files created by certain query execution operations, such as hash joins and sorts, in the case when they have to spill to disk. Such operations might also be encountered during queries, recovery, refreshing projections, and so on. If a temp location is provided to the database, Vertica uses it as temp space.

Top-k projection

A projection that configures the projection data for a Top-K query. A Top-K query is one that retrieves the top *k* rows from a group of tuples.

A Top-K projection is a type of live aggregate projection .

Transaction

One or more operations that are executed as a unit of work. At the user level, transactions occur in the current session by a user or script running one or more SQL statements. When you commit a transaction, any changes you make to data in tables using INSERT , DELETE , UPDATE , MERGE , and COPY during the transaction become permanent. If you roll back the transaction, all changes made to table data are undone.

Vertica supports Atomicity, Consistency, Isolation, and Durability (ACID) for SQL transactions.

Tuple mover (TM)

The Tuple Mover manages ROS data storage. On mergeout , it combines small ROS containers into larger ones and purges deleted data.

UDx

User-defined extensions (UDxs) are functions contained in external libraries that are developed in C++, Python, Java, or R using the Vertica SDK. The external libraries are defined in the Vertica catalog using the CREATE LIBRARY statement. They are best suited for analytic operations that are difficult to perform in SQL, or that need to be performed frequently enough that their speed is a major concern.

Up-to-date (projection)

A projection is up-to-date (or up to date) if it is eligible to participate in query execution. Projections on empty tables are up-to-date upon creation. If the table has data loaded already, newly created projections are marked not up-to-date until refreshed. If a projection is refreshed while a node is down, that projection can be marked up-to-date even though it is missing data on one of the nodes. This is because the node will build the data during the recovery process before participating in queries.

User-defined SQL function

User-defined SQL functions let you define and store commonly-used SQL expressions as a function. User-defined SQL functions are useful for executing complex queries and combining Vertica built-in functions. You simply call the function name you assigned in your query.

A user-defined SQL function can be used anywhere in a query where an ordinary SQL expression can be used, except in a table partition clause or the projection segmentation clause.

View

A named logical relation specified by an associated query that can be accessed similarly to a table in the FROM clause of a SQL statement. The results of the query are not stored but obtained on the fly when the SQL referencing the view is executed.

Volatile functions

Regardless of their arguments or environment, volatile functions can return a different result with each invocation—for example, UUID_GENERATE .

vsql

vsql is a character-based, interactive, front-end utility that lets you type SQL statements and see the results. It also provides a number of meta-commands and various shell-like features that facilitate writing scripts and automating a variety of tasks.

For more information, see Installing the vsql client and the more general topic, Using vsql .

Window (analytic)

An analytic function's OVER clause specifies how to partition, sort, and frame function input with respect to the current row. The input data is the result set that the query returns after it evaluates FROM , WHERE , GROUP BY , and HAVING clauses.

Working data size

The amount of data that most of your queries operate on. In most Vertica databases, you store more data than you regularly query. For example, suppose you are storing sales data for a corporation. You may keep many years of sales data in your database. However, the majority of queries you run may only query the last year of data to determine current sales trends. In that case, your working data size is the amount of sales data you load into the database in a year.

Knowing your working data size is important when configuring an Eon Mode database. You want the total size of the depots in all of the nodes a subcluster to be large enough to fit the working data size. Note that the working data size may vary per subcluster. Some subclusters may only work on recently-loaded data, while others do many queries on a larger data set.

Workload analyzer

An advisor tool that analyzes system information held in SQL system tables (monitoring APIs) and returns a set of tuning recommendations. See Analyzing workloads in the Administrator's Guide for details.

Copyright notice

Open Text Corporation Open Text Corporation, and its licensors. All rights reserved.

Open Text Corporation

55 Blue Sky Drive, Suite 102

Burlington, MA 01803

Phone: +1 617 386 4400

E-Mail: contactvertica@microfocus.com

Web site: http://www.vertica.com

The software described in this copyright notice is furnished under a license and may be used or copied only in accordance with the terms of such license. Open Text Corporation software contains proprietary information, as well as trade secrets of Open Text Corporation, and is protected under international copyright law. Reproduction, adaptation, or translation, in whole or in part, by any means — graphic, electronic or mechanical, including

photocopying, recording, taping, or storage in an information retrieval system — of any part of this work covered by copyright is prohibited without prior written permission of the copyright owner, except as allowed under the copyright laws.

This product or products depicted herein may be protected by one or more U.S. or international patents or pending patents.

Trademarks

Vertica™, the Vertica Analytics Platform™, and FlexStore™ are trademarks of Open Text Corporation.

Adobe®, Acrobat®, and Acrobat® Reader® are registered trademarks of Adobe Systems Incorporated.

AMD™ is a trademark of Advanced Micro Devices, Inc., in the United States and other countries.

DataDirect® and DataDirect Connect® are registered trademarks of Progress Software Corporation in the U.S. and other countries.

Fedora™ is a trademark of Red Hat, Inc.

Intel® is a registered trademark of Intel.

Linux® is a registered trademark of Linus Torvalds.

Microsoft® is a registered trademark of Microsoft Corporation.

Novell® is a registered trademark and SUSE™ is a trademark of Novell, Inc., in the United States and other countries.

Oracle® is a registered trademark of Oracle Corporation.

Red Hat® is a registered trademark of Red Hat, Inc.

VMware® is a registered trademark or trademark of VMware, Inc., in the United States and/or other jurisdictions. Other products mentioned may be trademarks or registered trademarks of their respective companies.

Information on third-party software used in Vertica, including details on open-source software, is available in the LICENSES file in the installation directory (/opt/vertica by default).